Articles » General Programming » Cryptography & Security » Cryptography

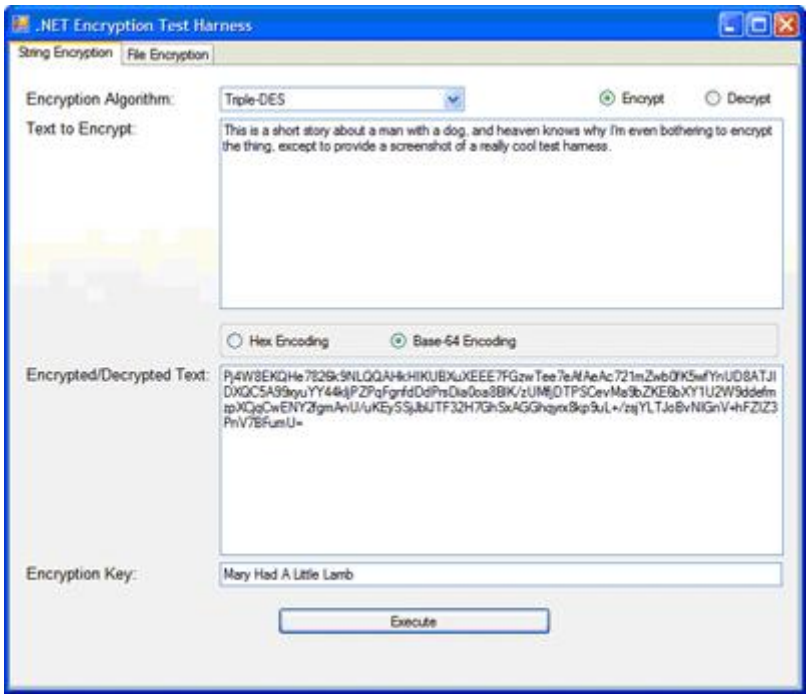
Cryptography 101 for the .NET Framework

Toby Emden, 22 Dec 2006 CPOL

★★★★★ 4.83 (92 votes)

An introduction to key cryptographic concepts supported by the .NET Framework.

Download source - 62.1 Kb



Introduction

Cryptography is one of the least sexy topics in computing. The mere word invariably causes eyes to glaze over, heads to spin, marriages to break up, and otherwise stable individuals to become alcoholics. Even though most people have a high-level understanding of what cryptography means, the inner workings of modern encryption algorithms are a mystery to all but the most advanced mathematicians among us. In fact, the quickest way to scare off annoying persons at a party is to ask them where they stand on the Triple-DES versus AES debate. Trust me on that one.

In the bad old days, we were often required to roll our own cryptographic routines if we needed to protect data. This required detailed knowledge of the underlying algorithms, not to mention copious supplies of caffeine and aspirin.

Thankfully, the .NET Framework now provides a series of classes that abstract the complexity of these algorithms for us. In this article, I will present a brief overview of cryptography, discuss some of the algorithms supported by .NET 2.0, and provide some source code demonstrating how to implement them.

What is Cryptography?

Cryptography is the ancient art of encoding a message so that it cannot be read by an unauthorized party. In its simplest form, a cryptographic cipher could involve encoding a message by substituting one character with another. If both the creator and the recipient of the enciphered message have an identical list of substitute characters (known as a "key"), the message can easily be enciphered and deciphered. This methodology is known as a substitution cipher, and was being used long before anybody ever dreamed of electricity, never mind computers.

Of course, a message is only safe as long as the key itself does not fall into the wrong hands. For this reason, the German military, in 1919, attempted to solve this problem by instituting the use of a "one-time pad"—a key that is only used once. Typically, a one-time key is derived from some peripheral factor known to both parties, such as a sequential message number, or the date on which the message was sent. In other words, the recipient may have a book full of keys, and has to know exactly which one to use based on information that is not necessarily included in the message itself. Of course, anybody who has a copy of the book and is able to intercept the message can simply try every key until one of them works, but key obfuscation is entirely another issue with which to get rid of annoying people at parties.

There is a strong argument to suggest that the evolution of modern computing has been partly driven by the need of governments and intelligence agencies to create, intercept, and decode enciphered messages. During World War II, both sides made heavy use of cryptography. The most famous cryptography tale of all involved a mechanical rotor machine called Enigma, which the Germans invented for the generation of secure messages. Enigma was a tremendously advanced machine for its time, but a group of Polish, English, and French mathematicians managed to break its code. Breaking the Enigma code was one of the most closely held secrets of the war, and helped to ensure Hitler's defeat. As a matter of interest, one member of the Enigma-cracking cryptology team was British mathematician Alan Turing, who went on to become the father of modern computing.

With the advent of the microchip and steady advances in computing power, encryption algorithms became increasingly sophisticated, but so did the tools to crack them, and now the perpetual cat-and-mouse game between cryptographers and crackers has become a fact of life.

You see, I could spend hours talking about this subject. Which is exactly why I am such a big hit at parties!

Types of Algorithms

Broadly speaking, we will be dealing with three types of algorithms.

1. Symmetric Encryption

This is the most common and straightforward type of encryption. Both the creator and the recipient of a message share a secret key that they use to encipher and decipher the message. However, if the key is compromised, so is the integrity of the message.

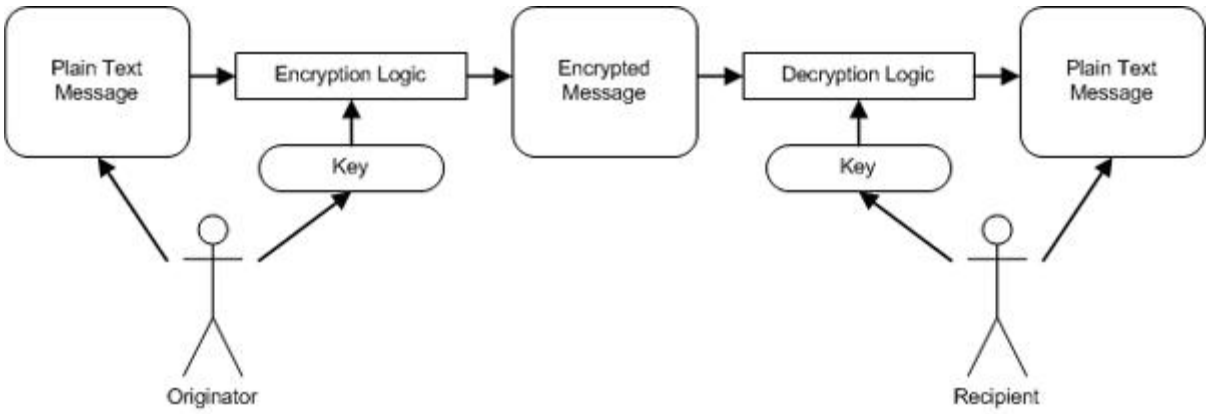


Diagram 1. Symmetric Encryption

Common sense suggests that a simple plain-text key is vulnerable to dictionary attacks. One way of avoiding this vulnerability is to use a hashed version of the key to encrypt and decrypt the message. We will discuss password-based key generation later on.

There are two kinds of symmetric algorithms; block ciphers and stream ciphers. A block cipher will take, for example, a 256-bit block of plain text and output a 256-bit block of encrypted text. The cipher works on blocks of a fixed length, usually 64 or 128 bits at a time, depending on the algorithm. If the unencrypted message is greater than the required length, the algorithm will

break it down into 64 or 128-bit chunks and XOR each chunk with the preceding chunk.

There is an obvious snag to this approach. If each chunk is XORed with the previous chunk, then what will the first chunk be XORed with? Welcome to the world of initialization vectors. No, this is not a narrative device for a Star Trek movie. An initialization vector, commonly known as an IV, is an arbitrary chunk of bytes that is used to XOR the first chunk of bytes in the unencrypted message. You will see this technique being used in my source code later on.

The .NET Framework natively supports popular symmetric key algorithms such as AES, RC2, DES, and 3-DES.

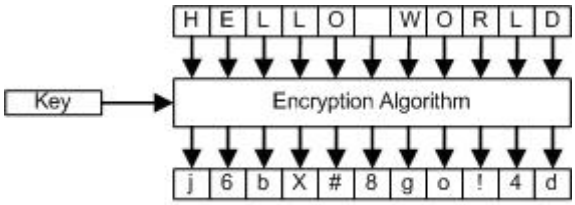


Diagram 2. Block Cipher

A stream cipher, on the other hand, generates a pseudorandom "keystream", similar in concept to the one-time pads used by intelligence officers during World War II. A stream cipher algorithm works on small chunks of bits of indeterminate length, XORing them with bits from the keystream instead of with previous chunks of the message.

From a security perspective, stream ciphers generally perform much faster, and are less resource intensive than block ciphers, but are far more vulnerable to attack.

All of the symmetric providers natively supported by the .NET Framework are block ciphers. For some reason, the most popular stream cipher, RC4, is not included in the Framework, although there is a very good open-source RC4 library written in C# that can be downloaded from Sourceforge.net.

2. Asymmetric Encryption

With a symmetric cipher, both parties share a common key. Asymmetric encryption, on the other hand, requires two separate keys that are mathematically related. One of the keys is shared by both parties, and can be made public. This is known, appropriately, as a public key. The other key is kept secret by one of the two parties, and is therefore called a private key. The combination of public and private key is described, amazingly enough, as a "key pair". Sometimes, even encryption terminology makes sense.

Consider the following example. Bob wants to send a secure message to Nancy. He encrypts the message using Nancy's public key. This means it must be decrypted using Nancy's private key, which only she knows. The combination of Nancy's public key and private key constitutes her key pair.

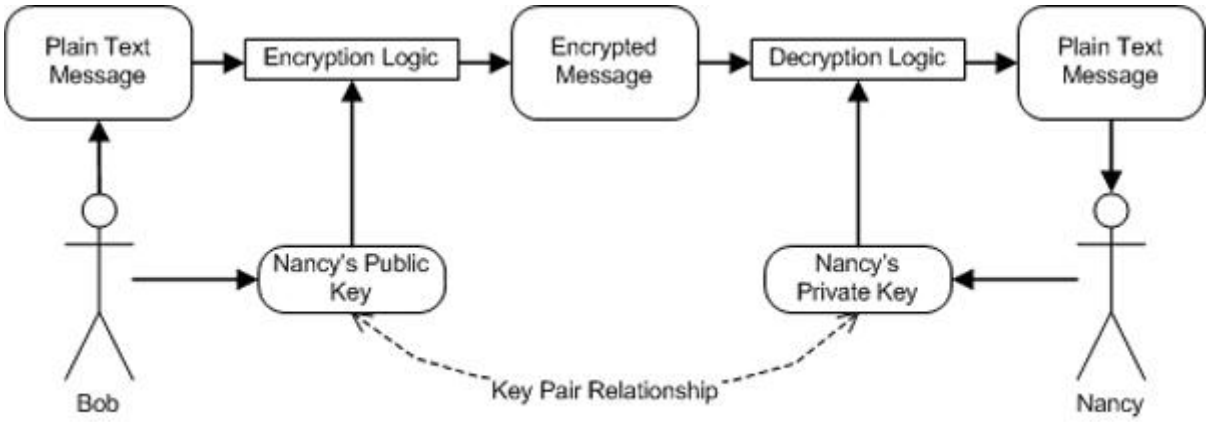


Diagram 3. Asymmetric Encryption

Conversely, it is also possible for Bob to encipher his message using his private key and have Nancy decipher it using his public key. This is a less desirable approach from a security perspective, as an attacker could intercept the enciphered message and, knowing that Bob was the creator, decipher it using his public key, which is... um... public.

Therefore, it is always preferable to have the creator of a message encipher it using the recipient's public key, and have the recipient decipher using her private key.

The two main asymmetric algorithms supported by .NET are RSA and DSA, of which RSA is by far the most commonly used.

The advantage of asymmetric encryption is that it does not require both parties to share a key. The disadvantage is that it incurs a significant performance overhead, and is therefore recommended for use only with short messages.

3. One-Way Hashing

As the name implies, a one-way hash is non-reversible. Hashes are generally used for information validation.

For instance, imagine that you have a database populated with user passwords. You may not want to store them in plain text, but you still need a way of authenticating a user who enters her credentials into a login form. So, you store the password in hashed format. When the user enters her password in plain text, you can hash it and compare the value to the hashed password stored in the database.

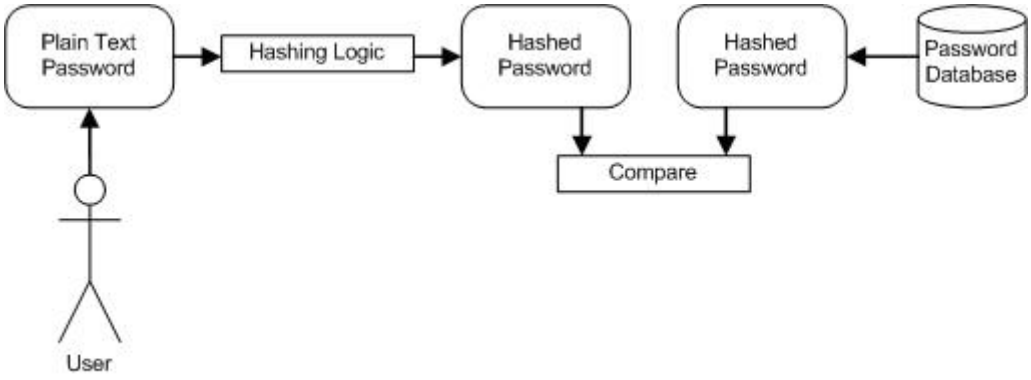


Diagram 4. One-Way Hash for Secure Password Storage

As you can see, there is no key involved in creating a hashed value. A hashing algorithm always generates the same value from a plain text input, but the original message can never be determined from a hash.

Another popular use case for hashing is to validate the authenticity of software downloads. After a file is downloaded, the user generates a hash of the file using an MD5 algorithm, and the hash is then compared to a publicly available value to ensure that the file has not been tampered with.

Cryptographic Support within the .NET Framework

Now that we have discussed the different types of ciphers, let us look at the algorithms supported by the .NET Framework. This list isn't entirely comprehensive, but it covers all the most popular providers.

Algorithm	Type	Block Bits
RC2	Block	64
DES	Block	64
3-DES	Block	192
AES (Rjindael)	Block	256
MD5	Hash	N/A
SHA-1	Hash	N/A
SHA-256	Hash	N/A
SHA-384	Hash	N/A
SHA-512	Hash	N/A
RSA	Asymmetric	384-16384

Weaknesses and Vulnerabilities

When deciding what kind of cipher to use in your application, you must carefully weigh the sensitivity of the data you wish to protect against the impact of performance degradation with more sophisticated encryption algorithms.

If security is your main priority, I would recommend using AES as a symmetric cipher and SHA-512 for hashing. While

asymmetric ciphers are more secure, they are also a huge drain on system resources, particularly if you are dealing with large messages. Therefore, the use of RSA should be limited to small messages only.

The CryptoHelper Class

To illustrate how simple it is to implement cryptography using .NET 2.0, I created the `CryptoHelper` class, which supports all of the major hashing and block cipher algorithms, as well as an implementation of the asymmetric RSA provider.

In .NET 2.0, symmetric providers all extend the `SymmetricAlgorithm` base class. If you know the key size for each provider, it is possible to create generic encryption and decryption methods, which is exactly what I have done here by casting an instance of `SymmetricAlgorithm` to the specific implementation of a cryptographic service provider:

```
Private Shared Function SymmetricEncrypt(ByVal Provider As SymmetricAlgorithm, _
        ByVal plainText As Byte(), ByVal key As String, _
        ByVal keySize As Integer) As Byte()
    'All symmetric algorithms inherit from the SymmetricAlgorithm
    ' base class, to which we can cast from the original
    ' crypto service provider
    Dim ivBytes As Byte() = Nothing
    Select Case keySize / 8
    'Determine which initialization vector to use
        Case 8
            ivBytes = IV_8
        Case 16
            ivBytes = IV_16
        Case 24
            ivBytes = IV_24
        Case 32
            ivBytes = IV_32
        Case Else
            'TODO: Throw an error because an invalid key
            '      length has been passed
    End Select

    Provider.KeySize = keySize


    'Generate a secure key based
    'on the original password by using SALT
    Dim keyStream As Byte() = DerivePassword(key, keySize / 8)


    'Initialize our encryptor object
    Dim trans As ICryptoTransform = _
              Provider.CreateEncryptor(keyStream, ivBytes)


    'Perform the encryption on the textStream byte array
    Dim result As Byte() = trans.TransformFinalBlock(plainText, 0, _
                           plainText.GetLength(0))


    'Release cryptographic resources
    Provider.Clear()
    trans.Dispose()

    Return result
End Function


Private Shared Function SymmetricDecrypt(ByVal Provider As SymmetricAlgorithm, _
        ByVal encText As String, ByVal key As String, _
        ByVal keySize As Integer) As Byte()
    'All symmetric algorithms inherit from the SymmetricAlgorithm base class,
    'to which we can cast from the original crypto service provider
```

```
    Dim ivBytes As Byte() = Nothing
    Select Case keySize / 8
    'Determine which initialization vector to use
        Case 8
            ivBytes = IV_8
        Case 16
            ivBytes = IV_16
        Case 24
            ivBytes = IV_24
        Case 32
            ivBytes = IV_32
        Case Else
            'TODO: Throw an error because an invalid key length has been passed
    End Select

    'Generate a secure key based on the original password by using SALT
    Dim keyStream As Byte() = DerivePassword(key, keySize / 8)

    'Convert our hex-encoded cipher text to a byte array
    Dim textStream As Byte() = HexToBytes(encText)
    Provider.KeySize = keySize

    'Initialize our decryptor object
    Dim trans As ICryptoTransform = Provider.CreateDecryptor(keyStream, ivBytes)

    'Initialize the result stream
    Dim result() As Byte = Nothing

    Try
        'Perform the decryption on the textStream byte array
        result = trans.TransformFinalBlock(textStream, 0, textStream.GetLength(0))
    Catch ex As Exception
        Throw New _
         System.Security.Cryptography.CryptographicException("The following" & _
         " exception occurred during decryption: " & ex.Message)
    Finally
        'Release cryptographic resources
        Provider.Clear()
        trans.Dispose()
    End Try

    Return result
End Function
```

Any of the TripleDES, DES, RC2, or Rjindael cryptographic service providers can be cast to an instance of `SymmetricAlgorithm`.

You may also notice that I have used a method called `DerivePassword`. This takes an unsecured, plain-text password, and transforms it into a secure key, as follows:

```
Private Shared Function DerivePassword(ByVal originalPassword As String, _
        ByVal passwordLength As Integer) As Byte()
    Dim derivedBytes As New Rfc2898DeriveBytes(originalPassword, SALT_BYTES, 5)
    Return derivedBytes.GetBytes(passwordLength)
End Function
```

The `Rfc2898DeriveBytes` method generates a secure key by taking our original plain text key, applying a salt value (in this case, an arbitrary byte array), and specifying the number of iterations for the generation method. Obviously, the more iterations, the safer. I chose five because, well, it seemed a good a number as any.

This is the simplest way to implement password-based key generation, which we discussed a long time ago in a paragraph far, far away.

Hashes are even simpler to implement in .NET, since they require neither a key nor an initialization vector. As with symmetric algorithms, the implementation of each hashing algorithm is derived from a base class, in this case, `HashingAlgorithm`. This allows us to generate a SHA1, SHA256, SHA384, SHA512, or MD5 hash, using just three lines of code:

```
Private Shared Function ComputeHash(ByVal Provider As HashAlgorithm, _
                                    ByVal plainText As String) As Byte()
    'All hashing mechanisms inherit from the HashAlgorithm base class
    'so we can use that to cast the crypto service provider
    Dim hash As Byte() = Provider.ComputeHash(UTF8.GetBytes(plainText))
    Provider.Clear()
    Return hash
End Function
```

Finally, we have asymmetric algorithms, and this is where things get messier. Like any block cipher, the RSA algorithm works on chunks of bytes, but unlike with symmetric block ciphers, the .NET implementation does not handle this for you. If you try to encrypt or decrypt a chunk of bytes longer than what the algorithm expects, you will get a nasty exception thrown in your face... and boy, does that hurt!

Therefore, we have to handle these chunks ourselves. Without going into too much detail as to the reasons why, RSA works on 128-byte chunks of data. When encrypting, the maximum we can pass to the algorithm is 12 bytes less than the modular. This amounts to 58 Unicode characters, since each Unicode character represents two bytes and (58 * 2) + 12 = 128.

The same rule applies when decrypting, although the length of an RSA-enciphered stream is always divisible by 128, which makes life a little easier, since we don't have to worry about the modular.

Of course, being an asymmetric algorithm, RSA also worries about public and private keys. There are many ways to implement key pairs, and that subject alone could cover several articles. I chose the simplest approach for this exercise, auto-generating a key pair using the framework's default options. The key pair is then saved to disk, where it can be reused:

```
Private Shared Sub ValidateRSAKeys()
    If Not File.Exists(KEY_PRIVATE) OrElse Not File.Exists(KEY_PUBLIC) Then
        'Dim rsa As New RSACryptoServiceProvider
        Dim key As RSA = RSA.Create
        key.KeySize = KeySize.RSA
        Dim privateKey As String = key.ToXmlString(True)
        Dim publicKey As String = key.ToXmlString(False)
        Dim privateFile As StreamWriter = File.CreateText(KEY_PRIVATE)
        privateFile.Write(privateKey)
        privateFile.Close()
        privateFile.Dispose()
        Dim publicFile As StreamWriter = File.CreateText(KEY_PUBLIC)
        publicFile.Write(publicKey)
        publicFile.Close()
        publicFile.Dispose()
    End If
End Sub
```

The key, if you'll pardon the very bad pun, is the `ToXmlString` method, which generates a public and/or private key. This method accepts a `Boolean`, which specifies whether or not to generate a private key along with the public key.

Now that we have our key pair, here is the implementation of the asymmetric encryption and decryption routines for RSA:

```
Private Shared Function RSAEncrypt(ByVal plainText As Byte()) As Byte()
    'Make sure that the public and private key exists
    ValidateRSAKeys()
    Dim publicKey As String = GetTextFromFile(KEY_PUBLIC)
    Dim privateKey As String = GetTextFromFile(KEY_PRIVATE)

    'The RSA algorithm works on individual blocks of unencoded bytes.
    ' In this case, the maximum is 58 bytes. Therefore, we are required
    ' to break up the text into blocks and encrypt each one individually.
    'Each encrypted block will give us an output of 128 bytes.
    'If we do not break up the blocks in this manner, we will throw
    'a "key not valid for use in specified state" exception

    'Get the size of the final block
    Dim lastBlockLength As Integer = plainText.Length Mod RSA_BLOCKSIZE
    Dim blockCount As Integer = Math.Floor(plainText.Length / RSA_BLOCKSIZE)
    Dim hasLastBlock As Boolean = False
    If Not lastBlockLength.Equals(0) Then
```

```
        'We need to create a final block for the remaining characters
        blockCount += 1
        hasLastBlock = True
    End If

    'Initialize the result buffer
    Dim result() As Byte = New Byte() {}

    'Initialize the RSA Service Provider with the public key
    Dim Provider As New RSACryptoServiceProvider(KeySize.RSA)
    Provider.FromXmlString(publicKey)

    'Break the text into blocks and work on each block individually
    For blockIndex As Integer = 0 To blockCount - 1
        Dim thisBlockLength As Integer

        'If this is the last block and we have a remainder,
        'then set the length accordingly
        If blockCount.Equals(blockIndex + 1) AndAlso hasLastBlock Then
            thisBlockLength = lastBlockLength
        Else
            thisBlockLength = RSA_BLOCKSIZE
        End If
        Dim startChar As Integer = blockIndex * RSA_BLOCKSIZE

        'Define the block that we will be working on
        Dim currentBlock(thisBlockLength - 1) As Byte
        Array.Copy(plainText, startChar, currentBlock, 0, thisBlockLength)

        'Encrypt the current block and append it to the result stream
        Dim encryptedBlock() As Byte = Provider.Encrypt(currentBlock, False)
        Dim originalResultLength As Integer = result.Length
        Array.Resize(result, originalResultLength + encryptedBlock.Length)
        encryptedBlock.CopyTo(result, originalResultLength)
    Next

    'Release any resources held by the RSA Service Provider
    Provider.Clear()

    Return result
End Function

Private Shared Function RSADecrypt(ByVal encText As String) As Byte()
    'Make sure that the public and private key exists
    ValidateRSAKeys()
    Dim publicKey As String = GetTextFromFile(KEY_PUBLIC)
    Dim privateKey As String = GetTextFromFile(KEY_PRIVATE)

    'When we encrypt a string using RSA, it works on individual blocks of up to
    '58 bytes. Each block generates an output of 128 encrypted bytes.
    'Therefore, to decrypt the message, we need to break the encrypted
    'stream into individual chunks of 128 bytes and decrypt them individually
    'Determine how many bytes are in the encrypted stream.
    'The input is in hex format, so we have to divide it by 2
    Dim maxBytes As Integer = encText.Length / 2

    'Ensure that the length of the encrypted stream is divisible by 128
    If Not (maxBytes Mod RSA_DECRYPTBLOCKSIZE).Equals(0) Then
       Throw New _
        System.Security.Cryptography.CryptographicException("Encrypted" & _
        " text is an invalid length")
       Return Nothing
    End If

    'Calculate the number of blocks we will have to work on
    Dim blockCount As Integer = maxBytes / RSA_DECRYPTBLOCKSIZE

    'Initialize the result buffer
    Dim result() As Byte = New Byte() {}
```

```
    'Initialize the RSA Service Provider
    Dim Provider As New RSACryptoServiceProvider(KeySize.RSA)
    Provider.FromXmlString(privateKey)

    'Iterate through each block and decrypt it
    For blockIndex As Integer = 0 To blockCount - 1
       'Get the current block to work on
       Dim currentBlockHex = encText.Substring(blockIndex * _
                             (RSA_DECRYPTBLOCKSIZE * 2), _
                              RSA_DECRYPTBLOCKSIZE * 2)
       Dim currentBlockBytes As Byte() = HexToBytes(currentBlockHex)

       'Decrypt the current block and append it to the result stream
       Dim currentBlockDecrypted() As Byte = _
                  Provider.Decrypt(currentBlockBytes, False)
       Dim originalResultLength As Integer = result.Length
       Array.Resize(result, originalResultLength + _
                            currentBlockDecrypted.Length)
       currentBlockDecrypted.CopyTo(result, originalResultLength)
   Next

   'Release all resources held by the RSA service provider
   Provider.Clear()

    Return result
End Function
```

Using the CryptoHelper Class

I have created `CryptoHelper` as a static class that can encrypt/decrypt either strings or files using the symmetric or asymmetric algorithms we have discussed, or generate a hash using the SHA/MD5 algorithms. The interface is as follows:

Properties:

`String Key()`	The encryption/decryption key
`Algorithm EncryptionAlgorithm()`	The algorithm to use for encryption and decryption
`EncodingType Encoding()`	The format in which content is returned after encryption, or provided for decryption. This will be either Hexadecimal or Base-64
`String Content()`	Encrypted content to be retrieved after an encryption event, or provided for a decryption event
`Boolean IsHashAlgorithm()`	`True` if the selected algorithm is a one-way hash
`CryptographicException CryptoException()`	Contains the `CryptographicException` object generated if a decryption event fails

Methods:

`Boolean EncryptString (String content)`	Encrypts a string specified in the "`content`" parameter and stores the result in the `Content()` property. Returns `True` if successful.
`Boolean DecryptString()`	Decrypts the encrypted value stored in the `Content()` property and stores the plain text string in the `Content()` property. Returns `True` if successful.
`Boolean GenerateHash(String content)`	Hashes a string specified in the "`content`" parameter and stores the result in the `Content()` property. Returns `True` if successful.

`Boolean EncryptFile(String filename, String target)`	Encrypts the file specified in "`filename`" and stores the enciphered content to the file specified by "`target`"
`Boolean DecryptFile(String filename, String target)`	Decrypts the file specified in "`filename`" and stores the deciphered content to the file specified by "`target`"

Here are a few use cases of how to use the `CryptoHelper` object:

To encrypt a string:

```
Crypto.EncryptionAlgorithm = Crypto.Algorithm.Rijndael
Crypto.Encoding = Crypto.EncodingType.BASE_64
Crypto.Key = "This is @ key and IT 1s strong"
If Crypto.EncryptString("This is the string I want to encrypt") Then
    MessageBox.Show("The encrypted text is: " & Crypto.Content)
Else
    MessageBox.Show(Crypto.CryptoException.Message)
End If
Crypto.Clear()
```

To decrypt a string:

```
Crypto.EncryptionAlgorithm = Crypto.Algorithm.Rijndael
Crypto.Encoding = Crypto.EncodingType.BASE_64
Crypto.Key = "This is @ key and IT 1s strong"
Crypto.Content = encryptedString
If Crypto.DecryptString Then
    MessageBox.Show("The decrypted string is " & Crypto.Content)
Else
    MessageBox.Show(Crypto.CryptoException.Message)
End If
Crypto.Clear()
```

To generate a hash:

```
Crypto.EncryptionAlgorithm = Crypto.Algorithm.SHA512
Crypto.Encoding = Crypto.EncodingType.HEX
If Crypto.GenerateHash("This is my password") Then
    MessageBox.Show("Hashed password is " & Crypto.Content)
Else
    MessageBox.Show(Crypto.CryptoException.Message)
End If
Crypto.Clear()
```

To encrypt a file:

```
Crypto.EncryptionAlgorithm = Crypto.Algorithm.RSA
Crypto.Encoding = Crypto.EncodingType.HEX
Crypto.Key = "This is @ key and IT 1s strong"
If Crypto.EncryptFile("c:\MyTextFile.txt", _
                      "c:\MyEncryptedFile.txt") Then
    MessageBox.Show("File Encrypted")
Else
    MessageBox.Show(Crypto.CryptoException.Message)
End If
Crypto.Clear()
```

To decrypt a file:

```
Crypto.EncryptionAlgorithm = Crypto.Algorithm.RSA
Crypto.Encoding = Crypto.EncodingType.HEX
Crypto.Key = "This is @ key and IT 1s strong"
If Crypto.DecryptFile("c:\MyEncryptedFile.txt", _
                      "c:\MyTextFile.txt") Then
    MessageBox.Show("File Decrypted")
Else
```

```
    MessageBox.Show(Crypto.CryptoException.Message)
End If
Crypto.Clear()
```

Conclusions

We have barely scratched the surface of Cryptography in this article, but thanks to the abstraction provided by .NET, the `CryptoHelper` class will suffice for about 95% of any developer's cryptographic needs.

There is much more you can do with the cryptographic providers in the .NET Framework. The intention of this article was to provide an introduction to the world of Cryptography and remove some of the mystery surrounding it. Gone are the days when building an encryption routine involved weeks of studying specific algorithms and then coding them in C or, worse still, Assembly. The cryptographic providers in the .NET Framework make encryption and decryption a relatively trivial undertaking.

In future articles, I will discuss more advanced cryptographic techniques. In the meantime, I have to go and bore some people at a party by telling them some Cryptography jokes. *Three symmetric algorithms walked into a bar...*

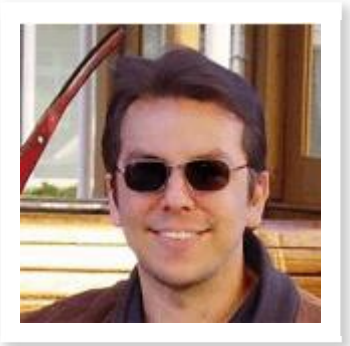
History

- Version 1.0: August 22, 2006

License

Share

About the Author



Toby Emden

Architect

United States

Toby Emden has been coding since the early days of 8-bit home computers, when developers had to write pure assembly to get anything worthwhile done. As a mere ankle-biter, his first coding projects included several games written for the ZX Spectrum in pure Z80 assembly code. Nowadays, his languages of choice are C++, C# and Java.

A member of the IEEE Computer Society and Worldwide Institute of Software Architects, Toby has spent ten years as an I.T. security executive, delivering enterprise security solutions for Fortune 100 organizations.

When not boring people at parties with jokes about cryptography and polymorphism, he enjoys writing, traveling and spending quality time with his wife and three cats. He still hasn't figured out how to cure cancer with .NET, but figures world peace can be solved with a Java wrapper class.

You may also be interested in...

Cryptography in .NET

It Takes 2 to Tango – and deliver a great solution for COBOL developers

Cryptography in .NET (part 1)

SAPrefs - Netscape-like Preferences Dialog

Cryptography

Window Tabs (WndTabs) Add-In for DevStudio

Comments and Discussions

50 messages have been posted for this article Visit **http://www.codeproject.com/Articles/15280/Cryptography-for-the-NET-Framework** to post and view comments on this article, or click **here** to get a print view with messages.