# **.Net中的加密解密**

## **引言**

在一些比较重要的应用场景中，通过网络传递数据需要进行加密以保证安全。本文将简单地介绍了加密解密的一些概念，以及相关的数字签名、证书，最后介绍了如何在.NET中对数据进行对称加密和解密。

## **加密和解密**

说到加密，可能大家最熟悉的就是MD5了，记得几年前我刚开始接触Web编程的时候，研究的一个ASP论坛程序，它的用户密码就是采用的MD5进行加密。MD5实际上只是一种散列运算，或者可以称为单向的加密，即是说无法根据密文（加密后的数据），推导出明文（原数据）。而我们下面要说明的，是在加密后可以进行解密、还原数据的。对于欲进行加密的对象，有的人称为消息，有的人称为数据，有的人称为信息，为了避免混淆，在本文后面部分，我统一将其称为**消息**。那么加密是什么呢？加密是通过对消息进行编码，建立一种安全的交流方式，使得只有你和你所期望的接收者能够理解。

那么怎么样才能叫安全呢？消息在接收方和发送方进行安全传递，一般要满足下面三个要点：

1. 消息的发送方能够确定消息只有预期的接收方可以解密(不保证第三方无法获得，但保证第三方无法解密)。
2. 消息的接收方可以确定消息是由谁发送的（消息的接收方可以确定消息的发送方）。
3. 消息的接收方可以确定消息在途中没有被篡改过（必须确认消息的完整性）。

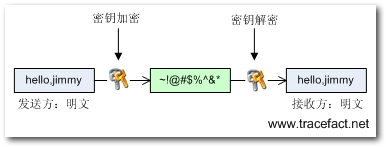
加密通常分为两种方式：对称加密和非对称加密，接下来我们先看看对称加密。

### **对称加密**

对称加密的思路非常简单，就是含有一个称为**密钥**的东西，在消息发送前使用密钥对消息进行加密，在对方收到消息之后，使用**相同的**密钥进行解密。根据密钥来产生加密后的消息（密文）的这一加工过程，由**加密算法**来完成**，**加密算法通常是公开的。它的流程如下：

1. 发送方使用密钥对消息进行加密。
2. 接收方使用同样的密钥对消息进行解密。

可以使用下面一副图来表示：



对称加密存在这样两个问题：

1. 虽然可以通过密钥来保证消息安全地进行传递，但是如何确保密钥安全地进行传递？因为发送者和接收者总有一次初始的通信，用来传递密钥，此时的安全如何保证？
2. 接收者虽然可以根据密钥来解密消息，但因为存在上面的问题，消息有可能是由第三方（非法获得密钥）发来的，而接收方无法辨别。

为了解决上面两个问题，就需要介绍一下非对称加密。

### **非对称加密**

非对称加密的**接收者和发送者**都持有两个密钥，一个是对外公开的，称为**公钥**，一个是自行保管的，称为**私钥**。非对称加密的规则是**由某人A的公钥加密的消息，只能由A的私钥进行解密；由A的私钥加密的消息只能由A的公钥解密。**此时我们可以得出接收方、发送方有两个公钥两个私钥一共四个密钥，我们先看看两种简单的方式，这两种方式都是只使用两个密钥。

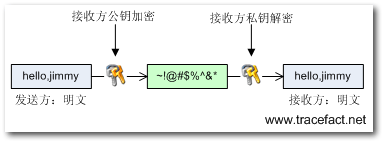
第一种模式只使用接收方的公钥和私钥，称为加密模式。

#### **加密模式**

在加密模式中，由消息的**接收方**发布公钥，持有私钥。比如发送方要发送消息“hello,jimmy”到接收方，它的步骤是：

1. 发送方使用接收者的公钥进行加密消息，然后发送。
2. 接收方使用自己的私钥对消息进行解密。

可以使用下面一幅图来描述：



在这种模式下，如果第三方截获了发送者发出的消息，因为他没有接收者的私钥，所以这个消息对他来说毫无意义。可见，它能够满足本文最开始提出的消息安全传递的要点一：**消息的发送方能够确定消息只有预期的接收方可以解密(不保证第三方无法获得，但保证第三方无法解密)**。

除此以外，因为接收方的公钥是公开的，任何人都可以使用这个公钥来加密消息并发往接收者，而接收者无法对消息进行判别，无法知道是由谁发送来的。所以，它不满足我们开始提出的消息安全传递的要点二：**消息的接收方可以确定消息是由谁发送的（消息的接收方可以确定消息的发送方）。**

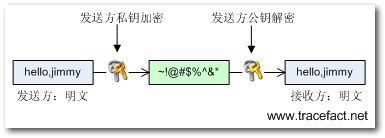
这个问题可以在下面的认证模式中得到解决。

#### **认证模式**

在认证模式中，由消息的**发送方**发布公钥，持有私钥。比如发送者要发送消息“Welcome to Tracefact.net”到接收者，它的步骤是：

1. 发送者使用自己的私钥对消息进行加密，然后发送。
2. 接收者使用发送者的公钥对消息进行解密。

可以用下面一副图来表述：



在这种模式下，假如发送方叫做Ken，接收方叫做Matthew，因为Matthew只能使用Ken的公钥对消息进行解密，而无法使用Molly、Sandy或者任何其他人公钥对消息进行解密，所以他一定能够确定消息是由Ken发送来的。因此，这个模式满足了前面提出的消息安全传递的要点二。

与此同时，因为Ken的公钥是公开的，任何截获了该消息的第三方都能够使用Ken的公钥来对消息进行解密，换言之，消息现在是**不安全的**。因此，与加密模式正好相反，它无法满足前面提出的消息安全传递的要点一。

而不管是采用加密模式还是认证模式，都没有解决加密解密中的要点三：接收方必须能够确认消息没有被改动过。为了解决这个问题，又引入了数字签名。

## **数字签名**

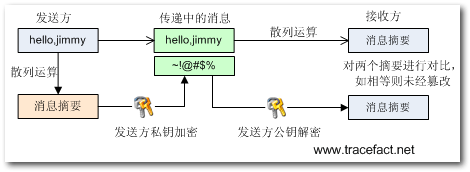
### **基本实现**

数字签名实际上就是上面非对称加密时的认证模式，只不过做了一点点的改进，加入了散列算法。大家比较熟悉的散列算法可能就是MD5了，很多开源论坛都采用了这个算法。散列算法有三个特点：一是不可逆的，由结果无法推算出原数据；二是原数据哪怕是一丁点儿的变化，都会使散列值产生巨大的变化；三是不论多么大或者多么少的数据，总会产生固定长度的散列值（常见的为32位64位）。产生的散列值通常称为消息的**摘要**（digest）。

那么如何通过引入散列函数来保证数据的完整性呢？也就是接收方能够确认消息确实是由发送方发来的，而没有在中途被修改过。具体的过程如下：

1. 发送方将想要进行传递的消息进行一个散列运算，得到消息摘要。
2. 发送方使用自己的私钥对摘要进行加密，将消息和加密后的摘要发送给接收方。
3. 接收方使用发送方的公钥对消息和消息摘要进行解密(确认了发送方)。
4. 接收方对收到的消息进行散列运算，得到一个消息摘要。
5. 接收方将上一步获得的消息摘要与发送方发来的消息摘要进行对比。如果相同，说明消息没有被改动过；如果不同，说明消息已经被篡改。

这个过程可以用下面的一副图来表述：



我们可以看出，数字签名通过引入散列算法，将非对称加密的认证模式又加强了一步，确保了消息的完整性。除此以外，注意到上面的非对称加密算法，**只是对消息摘要进行了加密，而没有对消息本身进行加密**。非对称加密是一个非常耗时的操作，由于只对消息摘要加密，使得运算量大幅减少，所以这样能够显著地提高程序的执行速度。同时，它依然没有确保消息不被第三方截获到，不仅如此，因为此时消息是以明文进行传递，第三方甚至不需要发送方的公钥，就可以直接查看消息。

为了解决这样的问题，只需要将非对称加密的认证模式、加密模式以及消息摘要进行一个结合就可以了，这也就是下面的高级模式。

### **高级实现**

由于这个过程比上面稍微复杂了一些，我们将其分为发送方和接收方两部分来看。先看看**发送方**需要执行的步骤：

1. 将消息进行散列运算，得到消息摘要。
2. 使用自己的私钥对消息摘要加密(认证模式：确保了接收方能够确认自己)。
3. 使用接收方的公钥对消息进行加密(加密模式：确保了消息只能由期望的接收方解密)。
4. 发送消息和消息摘要。

接下来我们看一下接收方所执行的步骤：

1. 使用发送方的公钥对消息摘要进行解密(确认了消息是由谁发送的)。
2. 使用自己的私钥对消息进行解密(安全地获得了实际应获得的信息)。
3. 将消息进行散列运算，获得消息摘要。
4. 将上一步获得的消息摘要 和 第一步解密的消息摘要进行对比(确认了消息是否被篡改)。

可以看到，通过上面这种方式，使用了接收方、发送方全部的四个密钥，再配合使用消息摘要，使得前面提出的安全传递的所有三个条件全都满足了。那么是不是这种方法就是最好的呢？不是的，因为我们已经说过了，非对称加密是一种很耗时的操作，所以这个方案是很低效的。实际上，我们可以通过它来解决对称加密中的密钥传递问题，如果你已经忘记了可以翻到前面再看一看，也就是说，我们可以使用这里的高级实现方式来进行对称加密中密钥的传递，对于之后实际的数据传递，采用对称加密方式来完成，因为此时已经是安全的了。

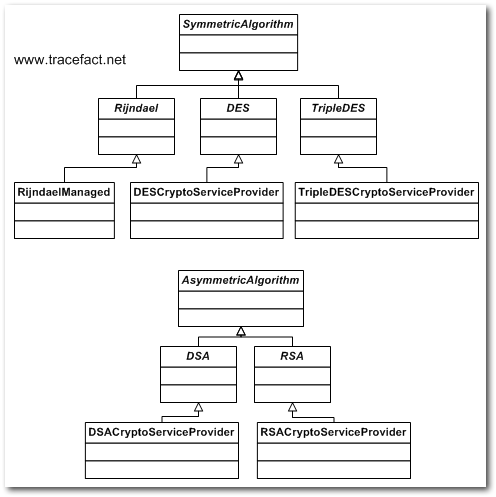
### **证书机制**

与数字签名相关的一个概念就是证书机制了，证书是用来做什么呢？在上面的各种模式中，我们一直使用了这样一个假设，就是接收方或者发送方所持有的、对方的公钥总是正确的（确实是对方公布的）。而实际上除非对方手把手将公钥交给我们，否则如果不采取措施，双方在网络中传递公钥时，一样有可能被篡改。那么怎样解决这个问题呢？这时就需要证书机制了：可以引入一个**公正的**第三方，当某一方想要发布公钥时，它将自身的身份信息及公钥提交给这个第三方，第三方对其身份进行证实，如果没有问题，则将其信息和公钥打包成为**证书（Certificate)**。而这个公正的第三方，就是常说的**证书颁发机构（Certificate Authority）**。当我们需要获取公钥时，只需要获得其证书，然后从中提取出公钥就可以了。

## **.NET中加密解密的支持**

### **对称加密和解密**

相信通过前面几页的叙述，大家已经明白了加密解密、数字签名的基本原理，下面我们看一下在.NET中是如何来支持加密解密的。正如上面我们所进行的分类，.NET中也提供了两组类用于加密解密，一组为对称加密，一组为非对称加密，如下图所示：



上面的类按照名称还可以分为两组，一组后缀为“CryptoServiceProvider”的，是对于底层Windows API的包装类，一组后缀为“Managed”，是在.NET中全新编写的类。现在假设我们以TripleDES作为算法，那么加密的流程如下：

1. 先创建一个TripleDESCryptoServiceProvider的实例，实例名比如叫provider。
2. 在provider上指定密钥和IV，也就是它的Key属性和IV属性。这里简单解释一下IV（initialization vector），如果一个字符串（或者数据）加密之前很多部分是重复的比如ABCABCABC，那么加密之后尽管字符串是乱码，但相关部分也是重复的。为了解决这个问题，就引入了IV，当使用它以后，加密之后即使是重复的也被打乱了。对于特定算法，密钥和IV的值可以随意指定，但长度是固定，通常密钥为128位或196位，IV为64位。密钥和IV都是byte[]类型，因此，如果使用Encoding类来将字符串转换为byte[]，那么编码方式就很重要，因为UTF8是变长编码，所以对于中文和英文，需要特别注意byte[]的长度问题。
3. 如果是加密，在provider上调用CreateEncryptor()方法，创建一个ICryptoTransform类型的加密器对象；如果是解密，在provider上调用CreateDecryptor()方法，同样是创建一个ICryptoTransform类型的解密器对象。ICryptoTransform定义了加密转换的运算，.NET将在底层调用这个接口。
4. 因为流和byte[]是数据类型无关的一种数据结构，可以保存和传输任何形式的数据，区别只是byte[]是一个静态的概念而流是一个动态的概念。因此，.NET采用了流的方式进行加密和解密，我们可以想到有两个流，一个是明文流，含有加密前的数据；一个是密文流，含有加密后的数据。**那么就必然有一个中介者，将明文流转换为密文流；或者将密文流转换为明文流。**.NET中执行这个操作的中介者也是一个流类型，叫做CryptoStream。它的构造函数如下，共有三个参数：

public CryptoStream(Stream stream, ICryptoTransform transform, CryptoStreamMode mode)

1. 当加密时，stream为密文流（注意此时密文流还没有包含数据，仅仅是一个空流）；ICryptoTransform是第3步创建的加密器，包含着加密的算法；CryptoStreamMode枚举为Write，意思是将流经CryptoStream的明文流写入到密文流中。最后，从密文流中获得加密后的数据。
2. 当解密时，stream为密文流（此时密文流含有数据）；ICryptoTransform是第3步创建的解密器，包含着解密的算法；CryptoStreamMode枚举为Read，意思是将密文流中的数据读出到byte[]数组中，进而再由byte[]转换为明文流、明文字符串。

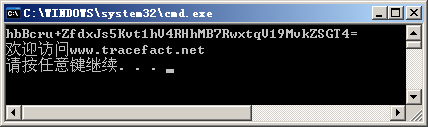
可见，CryptoStream总是接受密文流，并且根据CryptoStreamMode枚举的值来决定是将明文流写入到密文流（加密），还是将密文流读入到明文流中（解密）。下面是我编写的一个加密解密的Helper类：

// 对称加密帮助类  
public class CryptoHelper {  
  
    // 对称加密算法提供器  
    private ICryptoTransform encryptor;     // 加密器对象  
    private ICryptoTransform decryptor;     // 解密器对象  
    private const int BufferSize = 1024;  
  
    public CryptoHelper(string algorithmName, string key) {  
        SymmetricAlgorithm provider = SymmetricAlgorithm.Create(algorithmName);  
        provider.Key = Encoding.UTF8.GetBytes(key);  
        provider.IV = new byte[] { 0x12, 0x34, 0x56, 0x78, 0x90, 0xAB, 0xCD, 0xEF };  
  
        encryptor = provider.CreateEncryptor();  
        decryptor = provider.CreateDecryptor();  
    }  
  
    public CryptoHelper(string key) : this("TripleDES", key) { }  
  
    // 加密算法  
    public string Encrypt(string clearText) {  
        // 创建明文流  
        byte[] clearBuffer = Encoding.UTF8.GetBytes(clearText);  
        MemoryStream clearStream = new MemoryStream(clearBuffer);  
  
        // 创建空的密文流  
        MemoryStream encryptedStream = new MemoryStream();  
  
        CryptoStream cryptoStream =  
            new CryptoStream(encryptedStream, encryptor, CryptoStreamMode.Write);  
  
        // 将明文流写入到buffer中  
        // 将buffer中的数据写入到cryptoStream中  
        int bytesRead = 0;  
        byte[] buffer = new byte[BufferSize];  
        do {  
            bytesRead = clearStream.Read(buffer, 0, BufferSize);  
            cryptoStream.Write(buffer, 0, bytesRead);  
        } while (bytesRead > 0);  
  
        cryptoStream.FlushFinalBlock();  
  
        // 获取加密后的文本  
        buffer = encryptedStream.ToArray();  
        string encryptedText = Convert.ToBase64String(buffer);  
        return encryptedText;  
    }  
  
    // 解密算法  
    public string Decrypt(string encryptedText) {  
        byte[] encryptedBuffer = Convert.FromBase64String(encryptedText);  
        Stream encryptedStream = new MemoryStream(encryptedBuffer);  
  
        MemoryStream clearStream = new MemoryStream();  
        CryptoStream cryptoStream =  
            new CryptoStream(encryptedStream, decryptor, CryptoStreamMode.Read);  
  
        int bytesRead = 0;  
        byte[] buffer = new byte[BufferSize];  
  
        do {  
            bytesRead = cryptoStream.Read(buffer, 0, BufferSize);  
            clearStream.Write(buffer, 0, bytesRead);  
        } while (bytesRead > 0);  
  
        buffer = clearStream.GetBuffer();  
        string clearText =  
            Encoding.UTF8.GetString(buffer, 0, (int)clearStream.Length);  
  
        return clearText;  
    }  
  
    public static string Encrypt(string clearText, string key) {  
        CryptoHelper helper = new CryptoHelper(key);  
        return helper.Encrypt(clearText);  
    }  
  
    public static string Decrypt(string encryptedText, string key) {  
        CryptoHelper helper = new CryptoHelper(key);  
        return helper.Decrypt(encryptedText);  
    }  
}

我们可以对上面这个类进行一个简单的测试：

static void Main(string[] args) {  
    string key = "ABCDEFGHIJKLMNOP";  
    string clearText = "欢迎访问www.tracefact.net";  
  
    CryptoHelper helper = new CryptoHelper(key);  
      
    string encryptedText = helper.Encrypt(clearText);  
    Console.WriteLine(encryptedText);  
  
    clearText = CryptoHelper.Decrypt(encryptedText, key);  
    Console.WriteLine(clearText);  
}

应该可以看到下面的输出结果：



## **总结**

首先向大家表示歉意，我并没有写.NET中非对称加密的部分，因为我很少用到，所以这部分我并不是很熟悉，但是原理现在应该已经很清楚了，我想等到需要的时候再去学习如何来使用它们。到那个时候，我也会对这篇文章再次进行更新。通过这篇文章，相信大家对于加密、解密、数字签名等这些安全方面的概念已经有了一个初步的认识，同时也学习到了如何在.NET下进行对称加密。

<http://www.cnblogs.com/JimmyZhang/archive/2008/10/02/Cryptograph.html>