PROGRAMMER TO PROGRAMMER™

C# Data Security Handbook

C#
数据安全手册

Matthew MacDonald
Erik Johansson 著
崔伟 毛尧飞 译

清华大学出版社

PROGRAMMER TO PROGRAMMER™

C# Data Security Handbook

.NET Framework 提供数据安全功能的命名空间中实现的一些密码技术从本质上来说是坚不可摧的。导致机密数据暴露给攻击者的一些缺陷，绝大部分都是由应用程序实现中的错误引起的。因此仅仅向用户介绍如何使用.NET 中的类是不够的，我们还需要指导他们如何创建可靠的应用程序。

本书将介绍在.NET 中使用对称和非对称密码术的基本要点，然后讨论如何在下列几个方面实际运用这些技术：

- 安全数据交换——对于现代的电子商务应用来说，安全的在线通信是至关重要的。我们将讨论如何在应用程序中使用 SSL、TLS、安全的远程通信以及实现密码系统
- 安全数据存储——这一点实际上更具挑战性，我们将讨论如何在数据库中以及在使用 Windows 安全存储 API 的 NT 文件系统中安全地存储数据
- 数据完整性——我们将看一下在必须验证数据有无改变的情况下，散列代码和签名所扮演的角色
- 公钥体系结构——对于任何安全应用程序来说，密钥的管理实际上是最为关键的。本书将讨论如何进行密钥管理

本书目的在于使那些对密码术了解甚少(或是完全不了解)的C#开发人员能够从容地实现自己的安全应用程序。

本书读者对象

本书适用于那些正在开发分布式应用程序或是希望以一种安全的方式来存储数据的 C# 开发人员。

ISBN 7-302-06791-0
9 787302 067917 >
定价：35.00 元

Recommended Computer Book Categories: Programming; ASP.NET, .NET

TP 2Y

wrox direct

C#数据安全手册

Matthew MacDonald
Erik Johansson
著
崔 伟 毛尧飞 译

清华大学出版社
北 京

内 容 简 介

本书主要讲述了如何使用.NET 支持的加密技术。首先介绍了.NET 支持的密码术，接着介绍散列和签名，讨论了如何通过加密技术来保护通信数据和一些长期保存的数据，重点讨论了密钥和证书管理，此外还列出了一些好的和不好的做法。本书最后通过一个完整的例子演示了前面所讲的技术。

本书适用于创建分布式应用程序或需要安全地存储数据的 C#开发人员，旨在帮助 C#开发人员创建他们自己的安全应用程序。

EISBN：1-86100-801-5

北京市版权局著作权合同登记号 01-2002-6525

图书在版编目(CIP)数据

C#数据安全手册/(美) 麦克唐纳，(美) 约翰逊著；崔伟，毛尧飞译. 一北京：清华大学出版社，2003
书名原文：C# Data Security Handbook
ISBN 7-302-06791-0

Ⅰ.C… Ⅱ.①麦… ②约… ③崔… ④毛… Ⅲ.C 语言—程序设计—手册 Ⅳ.TP312-62

中国版本图书馆 CIP 数据核字(2003) 第 054526 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社 总 机：010-62770175　　**客户服务**：010-62776969

组稿编辑：曹　康
文稿编辑：徐燕萍
封面设计：康　博
版式设计：康　博
印 刷 者：北京密云胶印厂
发 行 者：新华书店总店北京发行所
开　　本：185×260　**印张**：17　**字数**：435 千字
版　　次：2003 年 7 月第 1 版　2003 年 7 月第 1 次印刷
书　　号：ISBN 7-302-06791-0/TP・5052
印　　数：1～4000
定　　价：35.00 元

前　言

“如果要实现加密系统，要么就出色完成，要么就干脆不要实现”。密码术并不能确保应用程序一定是安全的。如果要免受攻击者的入侵，必须了解潜在的弱点。虽然在.NET Framework 安全命名空间中实现的大部分加密算法基本是不会被攻破的，但导致数据暴露的大部分缺陷是由应用程序实现中存在的错误引起的。

尽管密码术背后的实际算法比较复杂，但没有必要详细了解它们。最重要的是正确使用实现这些算法的.NET 类。因此，本书将介绍如何充分利用.NET Framework 的加密支持，重点是实际问题和良好的编码习惯。

本书读者对象

本书对象是创建分布式应用程序或需要以安全的方式存储数据的 C# .NET 开发人员。本书旨在使那些了解甚少或不具备加密知识的 C#开发人员可以放心地实现他们自己的安全应用程序。

本书主要内容

本书介绍了在.NET Framework 和 Windows 平台中使用对称和非对称的加密技术，并说明如何实际使用这些技术。

- 第 1 章——密码术简介

本章通过介绍在新的分布式计算环境下需要考虑的威胁，说明了使用密码术背后的动机。本章介绍了.NET 支持的各种密码术的类型和用法。

- 第 2 章——.NET 密码术

在本章中，首先介绍了.NET 的加密服务。我们将对类模型作详细研究，了解它如何与流和加密转换一起工作，如何用新算法和实现来扩展它。

- 第 3 章—— 数据的完整性——散列码和签名

从本章中将学到如何应用散列技术确保数据不被篡改。我们将讨论基本的散列法、密钥散列、数字签名和 XML 签名标准。

- 第 4 章——保护长期保存的数据

本章讨论了如何保护在存储器中存储了很长时间的数据。同时，将看到如何将加密的数据存储到文件、XML 文档和数据库中。还将介绍对操作系统功能的支持，如加密文件系统(EFS)，以及这些技术在安全应用程序中的作用。

- 第 5 章——保护通信数据

在本章中，重点讨论了如何使用密码术和身份验证确保分布式系统中组件间通信的安全。将讨论内置到传输格式(例如可能使用 SSL)中的自动(透明的)加密，使用.NET Framework 的加密类在将部分数据发送到消息中之前有选择性地进行加密。

- 第 6 章——密钥和证书管理

密钥管理是密码术中最为重要的内容，但通常不易于理解。本章将讨论如何将密钥存储在 Windows 中，如何使用.NET 环境访问它们。还将讨论数字证书背后的概念，证书到底是什么，证书的作用和如何管理证书。

- 第 7 章——密码术——最佳和最坏做法

要创建安全代码，开发人员不仅需要了解本书介绍的加密原理和概念，还需要深刻理解细节。本章介绍了一些开发人员需要养成的良好做法和应该避免的不好的做法。

- 第 8 章——设计安全的应用程序

本章介绍了一个综合运用前面所讲技术的完整例子。它演示了一个“虚拟硬盘”的 Web 服务，允许用户安全地存储和获取远程服务器上的所有类型的数据。

- 附录 A——传输层安全(TLS)
- 附录 B——生成安全的随机数

使用本书的条件

本节列出了运行本书中的示例代码所需要的软件。

- .NET Framework SDK——这显然是开发和运行本书中的 C#代码所需要的。尽管不是绝对需要，我们还是推荐使用 Visual Studio .NET，特别是在处理 Windows Forms 应用程序时。
- SQL Server 2000 —— 本书中存储加密和散列的数据的示例使用 SQL Server 2000，原则上，这些几乎适合于所有的数据库。
- Microsoft Windows 2000 (或更高版本)—— 许多加密支持只可以在 Windows 更高版本中获得。为了充分利用本书，建议您至少使用 Windows 2000 Professional(带有高度加密包，http://www.microsoft.com/windows2000/downloads/recommended/encryption/)，不过 Windows XP Professional(或在将来的.NET 服务器中)更好。

.NET Framework 只为 Windows 上的证书存储提供了有限的支持。在第 6 章中，使用了另外两个库，它们为处理证书提供了额外的支持。

- CAPICOM —— 它为大部分 Windows 加密 API 提供了一个简单的 COM 接口。可从 http://www.microsoft.com/downloads/search.asp?下载 CAPICOM 2.0。运用关键字搜索，把 CAPICOM 作为关键字。
- Web Service Enhancements for .NET —— 这个.NET 库主要针对的是 Web 服务开发人员，但可以利用它所提供的额外的证书支持；可从 http://msdn.microsoft.com/webservices/building/wse/中下载它。

目　　录

第1章　密码术简介

在发送客户机中的电子邮件时，您是否想过邮件将发往何处？没想过？并不是只有您才把消息发送到指定接收者的技术问题都留给软件搞定，不过个中缘由还是值得探讨。电子邮件实际上并不像我们绝大多数人认为的那么可靠或值得信赖。电子邮件用户之间的往来消息可以被中途截获并阅读，就像截获和看别人的明信片那么简单。不仅如此，而且还可以在发送和接收邮件的双方察觉不到的情况下对邮件进行修改、替换或伪造邮件。现在通过网络进行这种危害他人的事情并不是件小事，被损坏的服务器、DNS 攻击甚至由于在合法的管理机构工作的人(如系统管理员)的滥用权力，都可能会破坏网络的安全。因此，如果想开展私人业务或确保所接收到的电子邮件确实是那些要发邮件给您的人，并且在传输过程中没有被篡改过，那么我们就不能依赖电子邮件来为我们提供便捷服务。所有这些都是数据安全(data security)问题。在密码术晦涩难懂的数学知识中，我们可以找到数据安全的解决之道。

虽然电子邮件消息发送中普遍存在数据安全风险，但是存储在您所信任的公司内部网中的敏感信息何尝又不是如此。通过内部文件系统窃取保密信息并转卖给竞争对手的案件举不胜举。人们耗费大量时间担心遭受网络外部的攻击，殊不知自己工作中存在着大量的安全漏洞(或成功攻击，这得取决于自己怎么看待)。知道这些后，就应该考虑内部可访问信息的安全问题，以及如何防止外部攻击。

密码术并不是包治百病的灵丹妙药。除非应用得当，否则所设定的保护措施只不过是让人产生安全的错觉而已。许多软件和编程环境中都有便捷的工具和服务可以处理实现密码系统的许多关键部分。不过，我们不能只运行程序或调用 API，也不要指望它可以像魔杖一样可以解决我们的数据安全问题。密码术是一门复杂的科学，我们需要认真对待。.NET Framework 提供了确保数据安全的几乎所有工具，当然本书讲解如何充分利用这些工具 — 我们将介绍如何才能使用这些工具创建提供真正安全的系统。

1.1　加密技术简史

加密技术的应用已有几千年的历史。在数字时代以前，就出现了对文字进行加密处理。现在则可以通过数字密码术对采用二进制数字表示的任何信息进行加密处理，不过在较早的历史时期就开始使用了双密钥加密技术。要加密的消息就是明文(plaintext)，加密后的消息则称为密文(ciphertext)。

早期的加密技术都以字符为基础(如替换或换位密码)或是以词为基础(如编码)。替换密码就是使用密文字符替换明文字符(例如，A 与 G 对应，B 与 K 对应等)，换位密码实质上就是根据事先定义好的规则打乱明文字符的顺序。编码则以整个词/短语为基础，这要求编码手册或相似的工具将编码转换成可以理解的句子。编码只定义符号与某一组字词之间的映射关系，因此在

某个领域之外通常就不适用，这也不属于常规加密技术。

在后来的几个世纪里，密码术的基本原则的演变很缓慢，到了 15 世纪，意大利的 Leon Battista Alberti 出版了第一本有关多字母密码的书后，密码术才得到大大改进。要破解多字母密码算法，比替换和换位密码要难得多，所以美国内战期间仍然在使用这种密码。

20 世界早期严谨的密码术研究和数论发现了很理想的密码系统。这种密码系统被称为一次性密码本(one-time pad)，它真的无懈可击。采用与要加密的文本具有相同长度的完全随机的符号序列(密码本)，就可以使用密码本中相应的符号对文本中每个符号进行加密处理。不使用密码本就根本无法破解加密后的文本。完成对消息的加密和解密处理后就销毁密码本——密码本仅用一次，它也就因此而得名。

与几乎所有数学意义上可以完美无缺的方法一样，它也存在着缺陷。这个解决方案在大多数密码应用程序中都行不通。因为发送者和接收消息的人都必须能访问完全相同的密码本，且这个密码本与消息要完全等长。问题就在于首先要保证所发送的消息与密码本都是保密的——如果我们不能传送消息，那又怎么能传送密码本？虽说如此，不过也可以将一次性密码本用于某些特定的对安全性要求高的应用程序。事先将许多密码本的两个副本分发给发送和接收消息的双方，每个密码本使用一次，用后就立即销毁。

开始时一般通过心算以及笔和纸来处理所有这些基于字符的密码算法。到第二次世界大战(1939~1945)时，开始使用电动机器处理复杂的多字母密码算法，甚至相当简单的消息传输都可以使用复杂的密码系统。其中最为著名的就是德国的 Enigma 使用马达进行加密和解密处理。

破解这种密码算法的工作直接导致了计算机科学和数字加密技术的发展。有了数字系统，密码员就将他们的注意力由加密字符转移到了加密二进制数据上来——换言之就是对数字进行加密处理，并且可以运用许多过去与加密范式基本就不相关的数学思想处理数据加密工作。同时，许多现有的基于字符的技术也有直接对应的数字技术，其中包括一次性密码本。

1976 年发生了两件重大事件，这年也因此成为数学密码术发展史的关键年份。首先，IBM 和 US NSA (美国国家安全局)发明了数据加密标准(DES)算法，至今仍在广泛应用——虽然现在不再认为它是最安全的算法，不过却将它视为基准。第二个事件就是 Whitfield Diffie 和 Martin Hellman 发表论文“密码学的新发展方向(New Directions in Cryptography)”，该论文向世人介绍了非对称密码术，也称之为公钥密码术(PKC)。现代数字加密技术就从这个时候开始，这些技术是现有技术的基础。

1.2　密码术的基础知识

虽然现代密码术以数学为基础，但是利用它时未必要理解它背后的复杂数论方法体系。实际上，这部分根本就不会深入探讨任何数学运算，只是讨论密码术语及简单应用。

古代研究密码术专门用于保密，如果要想与其他任何人沟通，这本身就不能充分保密。当然，这已有史为鉴，现代密码术则能实现：

- 保密性——使信息处于保密状态
- 完整性——知道信息未被篡改
- 身份验证——知道信息的来源和要发送的目的地

- 不可否认性——知道信息一旦发送出去就不能撤销或拒绝承认自己发送过

因此，密码术并不仅仅向没有正确密钥的人展示无用的数据，它还可用于确定此前是否有人更改过数据，实体确实有必要的证书来访问相应的信息，以及用于证明实体实际上执行了某个操作。所有这些功能就组成了一个极为有用的工具箱，可以在商务中应用它确保一切事情都如所预期的那样进行。

1.2.1　定义

密码术已应用多年，最初都是秘密地且只在相当封闭的团队中使用。和计算机科学一样，密码术中充斥着术语。这里需要讨论一些基本定义，这些定义即使在需要密码术的最简单应用程序中也会遇到。稍后将对它进行详细说明，下面只是将它们列出来，以便学习下面几节的内容。

- 明文(plaintext)——加密算法处理的所有数据在处理前都称为明文。如果它真是由纯文本、图象或其他内容组成也没有关系。
- 密文(ciphertext)——明文是加密算法处理前的数据，而密文就是加密处理后生成的结果。
- 加密(encryption)——将数据(明文)变成一堆杂乱无章的数据(密文)的过程。该过程也称为译成密码(enciphering)。
- 解密(decryption)——加密过程的逆过程。它将密文再次恢复成明文。该过程也称为回译密码(deciphering)。
- 密码算法(cipher)——可以加密和解密数据的密码算法。由于密码算法可以加密和解密，所以也称为双向或可逆算法。
- 密钥(key)——密钥是由许多密码算法用于控制结果的参数。两种不同的密钥对同一明文进行处理会生成不同的密文。
- 消息摘要(message digest)——由很小的大小固定的字节来表示任何大小的明文。生成这种消息摘要的算法称为消息摘要算法。消息摘要算法不可逆，是单向的。
- 散列(hash)——消息摘要的另一个名称。散列算法也称为散列函数。生成散列就是进行散列算法。

另外两个术语值得深入讨论：加密协议和密码分析学。

1. 加密协议

加密协议是一种严格的过程定义，它描述在某种情况下应该如何应用密码术。如果双方同意采用密码术交换数据，那么他们就必须就协议达成一致。协议规定各方在数据交互过程的各阶段都必须进行什么操作，以便协议可以继续实施。因此，其中一方必须对消息进行加密处理，或生成随机数字，或将消息发送给另一方。

为了便于讲述协议，密码术著作中使用了大量传统的角色，他们可以帮助我们演示加密的各种情形：

- Alice——第一个参与者，一般用户
- Bob——第二个参与者，也是一般用户
- Eve——偷听两位参与者之间发送的信息

- Mallory——怀有恶意的主动攻击者
- Trent——被信任的仲裁者

Alice、Bob、Eve、Mallory 和 Trent 在密码术著作中个性鲜明，在本书中他们也扮演各自的角色。

2. 密码分析学

由于密码术用于保密信息(维护信息的完整性、对某人进行身份验证或确保不可否认)，当然会有人想阅读这些保密信息(更改信息、模仿某人或声明其没有做他们做过的事、或声称其做过其并没有做过的事)。发现如何破解加密系统的科学就是密码分析学(cryptanalysis)。解码专家试图通过分析有效部分如密文和明文的组成部分，破解密文或破解加密算法，以便可以恢复明文或密钥(可以最终获得密文的明文)。解密专家也会分析协议和算法找出可能被参与者利用的弱点，以便可以获取他们不应该获取的信息或将错误的信息发送给另一个人。

密码分析学是密码术的关键分支——它可以确保发现并摒弃那些较脆弱的算法和协议。密码应用程序中使用的算法和协议是大量密码分析工作的结果，它们都是经得起考验的。只有通过持续的密码分析才能声明某个密码系统是安全的。

现代密码术的基本原则之一就是只有想保密的信息能得到保密才能说密码系统是安全的，即攻击者完全访问其实现方案的所有详细信息、所采用的协议和算法以及参与者之间交换的所有消息之后也不会泄露秘密。安全性不能含糊；使用自定义协议和自己制定的算法也不能保证安全，这只是我们对这些信息保密而已。相反，我们只能使用经过严格的密码分析的公开协议和算法，这样才能确保受益于那些显然比我们更有天赋的解密专家。

1.2.2 加密功能

下面将介绍前面提到过的加密功能：

- 保密性
- 完整性
- 身份验证
- 不可否认性

1.3 保密性

许多情况下保密性都很重要。如果要存储不想让其他人看到的数据，或将数据传输给特定的人或一组人并且防止其他人看到，则需要采取措施确保信息处于保密状态。下面将剖析必需使信息保密的情形。

1.3.1 信息交换

在交换信息的情形下，Alice 撰写并发送一封电子邮件给 Bob，如图 1-1。

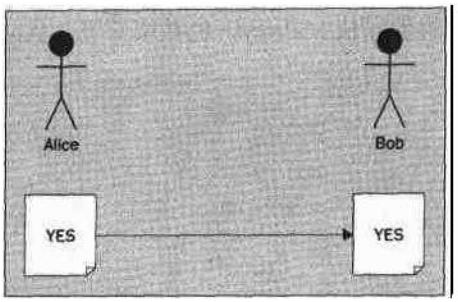

图　1-1

(1) Alice 创建一条文本消息

(2) Alice 将消息发送给 Bob

(3) Bob 阅读消息

在这种情况下消息发送给 Bob，在发送的过程中没有其他人阅读或篡改过消息。

由于电子邮件只不过是纯文本文件(将二进制附件编码成文本)，获取它的任何人都可以阅读或对消息进行解码，所以存在几种不同的安全隐患。

- 其他非指定人员可能阅读消息内容
- 不怀好意的任何人都可以更改电子邮件的任何部分，如发件人地址、收件人地址列表、内容、时间戳等。
- 接收者可能不是您认为的人
- 发送者可能不是您认为的人

要想阅读通过全球不同的网络间传输的消息，其方法不只一种，虽然它有时不是我们听说的那么简单，但是那些不怀好意或只是出于好奇的人确实可以阅读未保护的电子邮件。这里 Alice 将与前面相同的一个消息发送给 Bob，不过这次 Eve 会采用被动攻击监听这条消息，如图 1-2 所示。

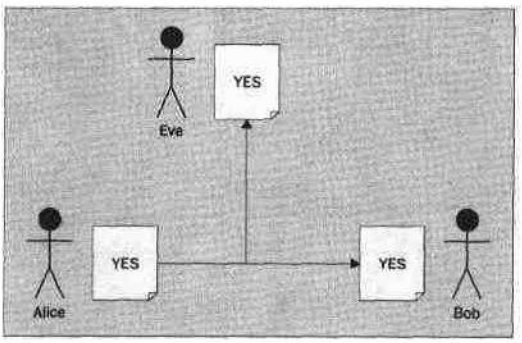

图　1-2

(1) Alice 创建一个文本消息

(2) Alice 将消息发送给 Bob

(3) Eve 阅读该消息

(4) Bob 阅读该消息

即使消息在发送途中没有被更改，只是被 Eve 监听了消息内容，是否会发生比较严重的后果取决于消息本身。

现在介绍一个新的心怀恶意的家伙 Mallory，他截取并更改 Alice 发给 Bob 的消息，如图 1-3。

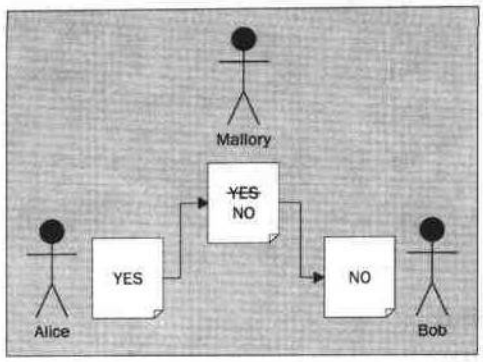

图　1-3

(1) Alice 创建一个文本消息

(2) Alice 将消息发送给 Bob

(3) Mallory 截取并更改消息内容

(4) Bob 阅读已更改过的消息

Mallory 不仅阅读了原始信息，而且还在将它转给 Bob 之前对消息内容加以修改。

显然，防止信息不泄露给未经授权的人的方法就是对信息进行加密处理。加密过程就是将最初的原文变成明显毫无意义的密文。这种过程的核心就是加密算法或密码算法(cryptographic algorithm 或 cipher)。密码算法必须是可逆的，以便可以将密文恢复成明文，相反的过程称为解密。图 1-4 演示典型密码算法的工作原理。

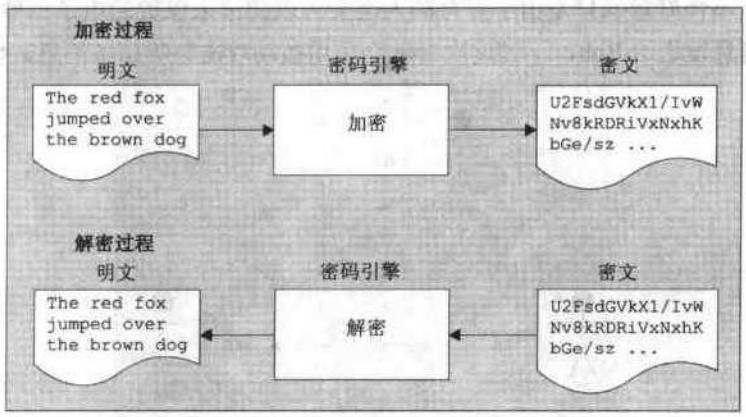

图　1-4

显然，尽管密文看起来没有意义，但是它对密码算法引擎必须是有意义的。密码算法引擎必须知道将明文转换成特殊密文消息的算法，利用该算法也可以知道如何将密文转换回原来的明文。密文与明文的关系需要保密，以便使明文处于保密状态。

信息交换协议

下面将它应用于信息交换中，看看 Alice 和 Bob 是否可以使用它通信，而不必担心 Eve 或 Mallory 会监听消息。这个简单示例将显示如何制定和分析加密协议。

在该实例中，Alice 和 Bob 使用简单的加密协议：

(1) Alice 使用达成一致的密码算法对明文消息进行加密处理

(2) Alice 将消息发送给 Bob

(3) Bob 使用相同的达成协议的密码算法对密文进行解密处理

如果 Eve 和 Mallory 干预消息发送，情况又会如何？首先，让我们来看看这个偷听消息的情形，如图 1-5 所示。

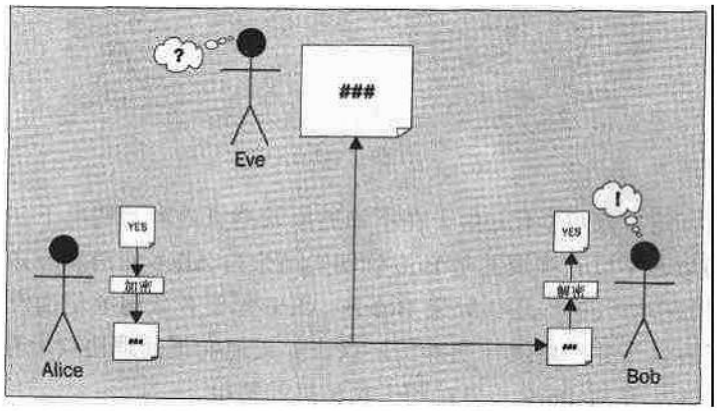

图　1-5

从图 1-5 中可以看到，由于对消息进行了加密处理，所以 Eve 不能解密发送给 Bob 的消息，她也不知道如何解密这个消息。另一方面，Bob 却知道如何解密这个消息，因此他可以阅读这个消息。因此，密码术显然可以保护两方或多方之间的信息交换免受被动攻击。下面将介绍如何使用保密性来使信息免受主动攻击，如图 1-6。

尽管作了改进，但是保密性还是不能防止密文免遭篡改。不过，它却起到一定的保护作用，由于 Mallory 不能生成解密后对 Bob 来说具有意义的明文的密文，除非 Mallory 知道 Alice 对明文进行加密处理的方法。

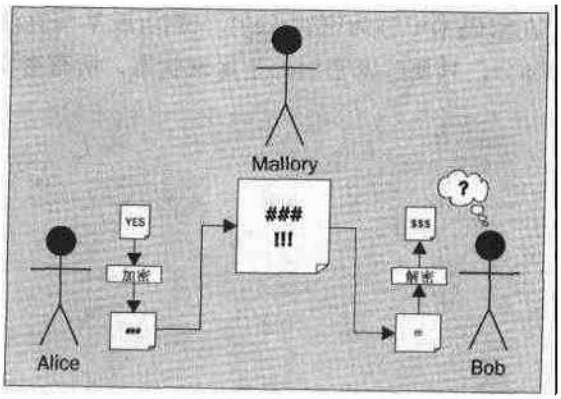

图　1-6

Alice 和 Bob 通过这种方式对确保数据安全达成协议是很重要的，否则尽管 Mallory 不能阅读来自 Alice 的原始消息，他却可以伪造发送给 Bob 的消息，如图 1-7。

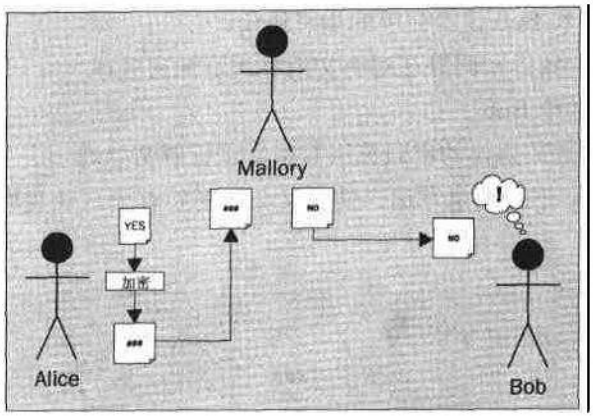

图 1-7

这个特殊示例可能有点戏剧化，由于在不知道原始消息内容的情况下，Mallory 不太可能伪造发送给 Bob 的消息，不过它强调的是 Bob 确切知道来自 Alice 的消息应该加密的重要性。

这个示例强调加密协议的作用。如果 Bob 和 Alice 想要有效地通信而没有 Mallory 的干扰，那么他们必须达成协议，并严格遵守明确的协议。如果他们正处理协议上存在的分歧或不能履行部分协议，则应该发出 Mallory 一直试图干预消息的警报。

1.3.2 密钥简介

密码算法将明文加密成密文并将密文恢复成明文，不过如果每次都采用相同的方式对明文进行加密，这也是不安全的。这样的密码算法很快就会失效，因为任何访问密码的人都可以对密文进行解密。请记住，我们要使用已公布的算法，而不是依赖于安全性仍然不清楚的算法。选择用于执行加密和解密的机制不应该成为协议的保密内容。如果上个示例中 Eve 或 Mallory 想要找出 Alice 和 Bob 使用的算法(经验说明找出算法只是个时间问题，如果 Alice 和 Bob 持续使用相同的算法，经过足够长的时间就可以找出算法)，他们就可能成功地阅读和替换消息。

对于生成无人可解密的密码算法，即使他们知道密码算法机制，也需要用户定义的参数才能加密和解密，这种参数在密码术中称为密钥(key)。密钥(或某些算法中的密钥)是除算法本身之外密码算法中最重要的部分，只要解密密钥处于保密状态，所有密文就是保密的。

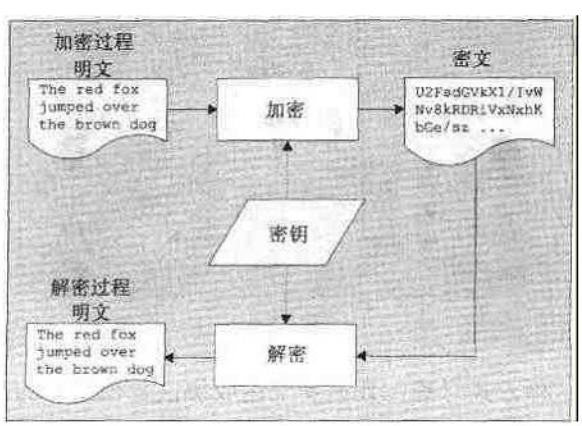

图 1-8

密钥就是加密和解密过程中的输入，使用不同密钥，同一个密码算法在每次使用时就会产生不同的效果。使用它就可以采用比较健壮的算法，改变密钥并使密钥保密，却不用担心 Eve 或 Mallory 会知道他们使用的算法。

在将消息从 Alice 发送到 Bob 的情形中使用密钥对消息进行加密处理，如图 1-9。

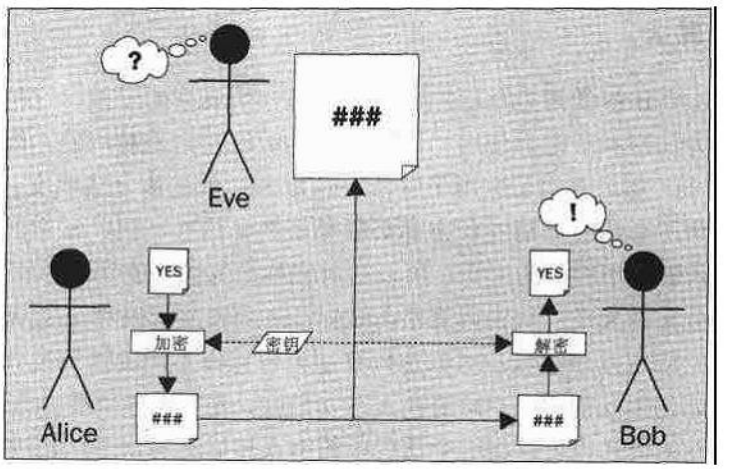

图　1-9

从该示例可以清楚地看到，Eve 不能阅读这个消息，因为她没有密钥，不过也让我们来看看相关的情形：Mallory 即使没有密钥，他也想通过篡改消息来欺骗 Bob。

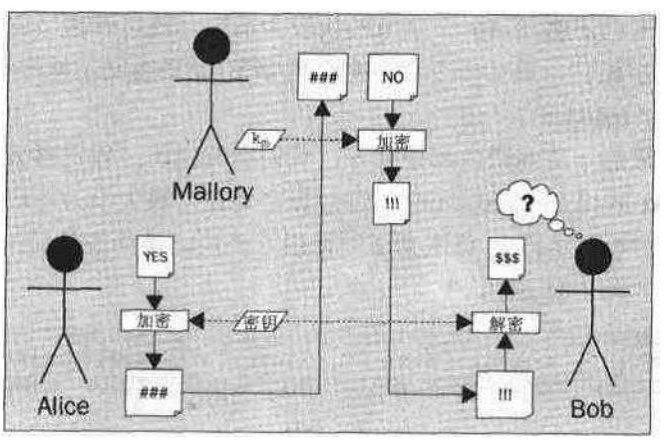

图　1-10

Mallory 不能阅读来自 Alice 的消息，如果他试图伪造发送给 Bob 的加密消息，就必须使用另一个密钥而不是 Alice 和 Bob 使用的那个密钥。如果 Bob 对 Mallory 发来的密文进行解密，解密的结果就是没有意义的垃圾。

在这里 Mallory 所做的事就没有什么意义，由于他并没有愚弄到 Bob。如果他指望干扰 Alice 和 Bob 之间的通信，也可以向 Bob 发送任何随机数据。

现在需要注意的是 Alice 和 Bob 都需要使用密钥，如前面所述，他们不能对外泄露密钥，以确保不泄露消息。这有点类似于小鸡和蛋的问题，如果 Eve 或 Mallory 没有密钥，他们就不能发送消息，那么他们不能发送消息又如何能发送密钥呢？毕竟，密钥也是消息。稍后本章将

介绍 Alice 和 Bob 在不将密钥放在消息中发送的情况下，也可以获取相同的密钥。在某些情况下也会看到这种情况，Alice 和 Bob 不需要有相同的密钥——只需要两个紧密相关的密钥。

单密钥系统也称为对称密码算法。双密钥系统就是非对称密码算法。下面将介绍如何应用这两种基本的加密工具。

1.3.3　对称密码算法

对由 Alice 发送给 Bob 的消息进行加密处理就是一个典型的示例，这种情况下很适合采用对称密码算法。因为加密和解密都使用相同的密钥，所以说它是对称的，即如果密钥 k 用于将明文加密成 NajDs78a，那么密钥 k 也用于解密 NajDs78a，使之恢复成明文。对称密码也称为秘密或会话密钥，因为所有参与通信的各方都必须秘密共享这些密钥，常常通过某种会话进行通信。由于加密和解密都采用相同的密钥，所以有时也称之为同一密钥和单密钥。

对称密码算法分为两大类：块密码算法(block cipher)和流密码算法(stream cipher)，它们分别用于稍有所不同的应用程序领域。对称密码算法也可以在不同的密码模式下使用，在什么密码模式下运行由确切的用法而定。不同的模式在安全性、效率和容错能力方面存在差别。例如，提高密码算法安全性的模式也可以将一个迭代中的错误传递给另一个迭代过程。

1. 块加密算法

块加密算法一次处理一个明文或密文块。对数据进行批量加密处理则可以应用复杂的密码术——块越大，安全性则越高。不过，块越大，采用的数学计算就越复杂，加密和解密过程速度就越慢。现代密码术已证明 8~16 个字节的块较为适中，既可管理又比较安全。密文块的大小总应该与明文块大小相同，最后一个块中剩余的字节(如果明文不能按块大小平均分割)，加密之前就必须将其补齐(padded)。

为了演示如何将最后一块的空白部分补齐，我们举例说明：假定要采用块加密算法对明文“This is a sample text to illustrate padding”进行加密处理，块大小为每块 8 个字节。由于明文长度为 43 个字节(使用 8 位字符)，算法则必须使用 6 个块，最终在最后一块中余下 5 个字节，这个块没有被填满，如图 1-11 所示。

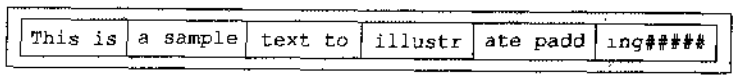

图　1-11

这里最后 5 个字节就使用#字符补齐。.NET 中的密码库包括自动补齐功能，不过应该注意在最坏的情况下，不充分的填充会帮助破坏者恢复密钥。

由于将明文分成许多块，所以有好几种方法可以处理这些块以便生成密文(可以将类似的相反技术用于解密)。它们就是块加密算法模式。

ECB——电子编码本

ECB 模式下的运算很简单。加密使用密钥作为惟一的其他输入对每个明文块进行处理，生成密文块。解密则是执行与此相反的过程。如果明文由可以预见的样式如纯文本组成，分析这种模式就相当容易(从解密专家的角度，相对于其他模式而言)，因为总是使用指定的同一个密

钥将每个明文块转换成相同的密文块。也就是说，如果在同一个消息中相同的字符块出现两次，或使用相同密钥加密的两个消息中出现相同的字符块，都会出现相同的密文块。这就可以向解密专家提供了消息结构的线索，这种线索有助于他们分析基本的明文，从而会大大缩短他们发现密钥的过程。有助于解密专家的典型示例就是信件和备忘录中的通用称呼，如 Hi 或 Hello，或消息的明确位置中的任何静态数据块，如电子邮件的题头、信笺抬头或脚注，如图 1-12 所示。因此，ECB 模式在随机明文的基础上生成的密文比根据结构严谨或面向字符的文本生成的密文要安全得多。

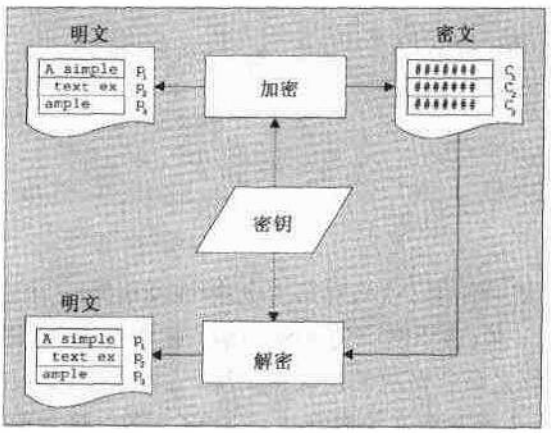

图　1-12

CBC—— 密码块链接

该模式解决这样的问题：通过将相邻块合并到一起由 ECB 对相匹配的块进行统一转换时会引发的问题。也就是说，任何特定块的密文最终取决于前面所有块的密文，即同一个消息中出现两次的明文块、或在两个不同的消息中出现相同的明文块而其前面的明文却不相同，这样就会生成两个不同的密文块。这样就产生了一个问题：前面经常提到消息的题头通常都是一样的，也就是说，相同的明文块可能出现在不同的消息中，并且前面的明文块排列顺序也相同。

也就是说，题头完全相同的消息最终还是可能采用相同的块顺序加密成密文。使用初始化向量(IV，Initialization Vector)就可以克服这个问题。IV 的用途就是通过充当加密第一个块的输入来增加密文的安全性。每个消息使用不同的 IV 就可以使用通用的题头，而不会由于加密前会根据 IV 对它们进行修改而危及消息安全。IV 不必保密，因此它可以与没有加密的密文一起发送。对于密码术了解不深的人通常会觉得很奇怪，不过它确实如此—— 不应该将 IV 当作另一个密钥，它只是增加安全的随机值而不包含任何数据资料。

那么，CBC 又是如何工作的呢？用 IV 对第一个明文块进行 XOR(异或)运算，稍后进行加密。用第一个密文块对第二个块执行 XOR 运算，然后加密，以此类推。加密之前，使用上一个密文块对下一个块执行 XOR 运算。加密前，对明文块和 IV 或上一个密文块执行 XOR 运算，也就是说，IV 对最终的密文的影响和明文同样多。更改相同明文的 IV 所得到的密文就不同。

这种模式很适合用于标准的明文，如文档、图像和电子表格，因为 IV 和从前一次迭代中得到的反馈使安全性得到增强。

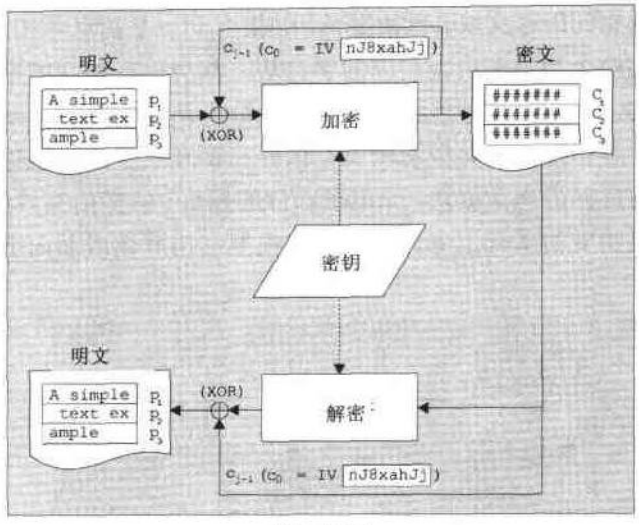

图　1-13

图 1-13 说明加密之前如何使用 IV(只对第一个块)或反馈(前一个密文块)对每个明文块执行 XOR 运算。其中 c_0 表示 IV 值(该示例中为 nJ8xahJj)，使用第一个明文块对 IV 值执行 XOR 运算，c_{j-1} 表示用于第一个块之后所有块的反馈密文块。

解密接收 IV ，用第一个解密密文块对它执行 XOR 运算，恢复第一个明文块。用前一个密文块对所有后面的密文块执行 XOR 运算，恢复明文的其余部分。

既然知道如何以与将 IV 值用于第一个块的相同方式重用每个密文块，这很显然不必防止别人知道它。不过，同样必须保护所有密文块。

2. 流密码算法

如果要处理的数据已在内存中或在大型缓冲区中，使用块密码算法效率最高。它们可以将缓冲区划分成块，并依次对这些块进行处理。不过，有时并没有获得所有要立即加密的数据，这些数据可能正在通过网络传人，或要等待用户输入下一个字符。在这种情况下，则不必一次就处理成块的数据。为此，可以使用流密码算法。

从道理上讲，真正的流密码算法没有块密码算法中的密钥，它们使用运行密钥(running-key)或密钥流(keystream)生成器。密钥流生成器可以输出随机位的恒定流，可以使用明文中的一位对随机位执行 XOR 操作。其结果就是密文的一位。要解密密文使之恢复成明文，则需要一个统一的密钥流，使用密钥流中的位对密文中的每个位执行 XOR 运算。流密码算法的任务就是生成位流，该位流显然是随机的且不可预见，但流必须完全可复制，以便解密密文。

通过在使用密钥生成密钥流的模式下执行块密码算法，就可以获取流密码算法属性。有两个这样的模式：CFB 和 OFB。

CFB—— 密码反馈

这种运算模式与 CBC 相似，不过我们可以使用它减少每次块操作中处理明文的位数。这些经过简化的块称为“单元”或“部分”，实际上它可以只有一个位。不过，仍然有必要对每个

部分执行完整块大小的转换操作，因此与 ECB 或 CBC 模式相比，该系统的结果就是降低了性能。该模式也称为自我同步的流密码算法。

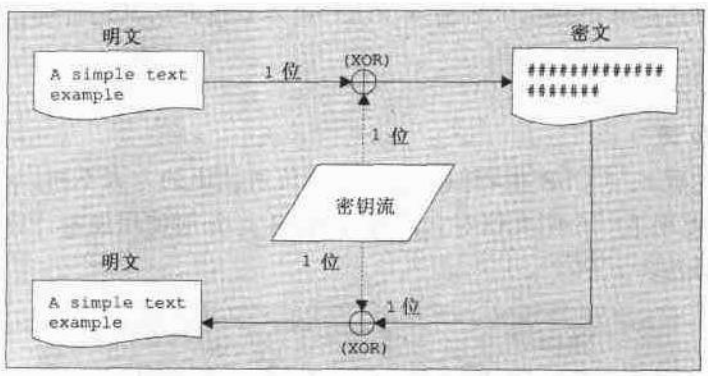

图　1-14

图 1-15 看起来让人有点发懵，不过它并没有像表面上看起来这么让人难懂。该模式实际上对一个位进行操作，不过由于对每个位都执行整个块加密的操作，所以不太实用，它或多或少使密码比在 ECB 或 CBC 模式中的速度要慢一些，其影响大小与块的大小相关(以位为单位)。

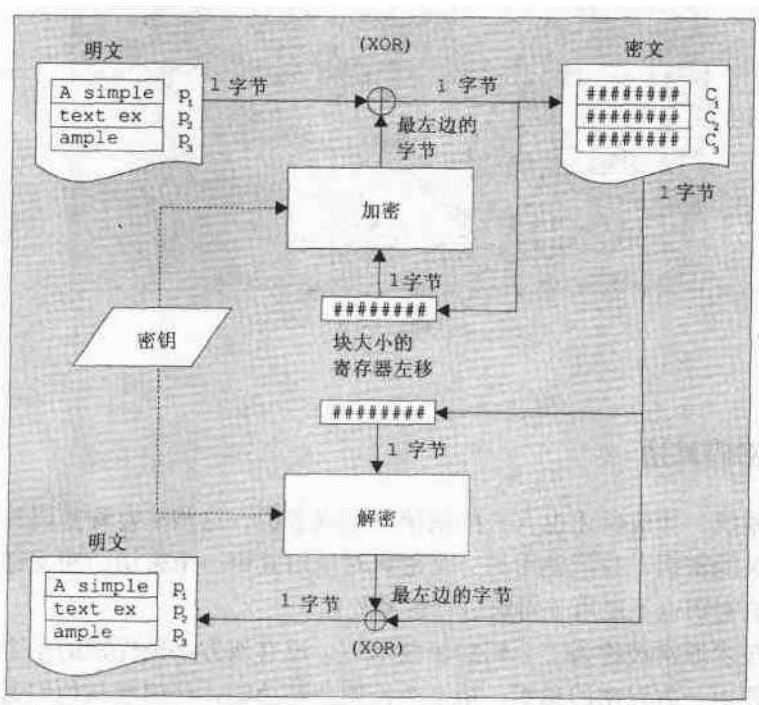

图　1-15

上面的示例使用 8 字节的密码块大小和 1 字节的 CFB 单元大小，也就是说将采用 8 个字节大小的寄存器，因为寄存器的大小等于块大小。密码使用 IV 初始化寄存器，与 CBC 模式(该模式也不必保护)中一样，然后每次处理一个字节。加密寄存器并使用明文字节对加密结果最左侧的字节执行 XOR 运算。然后，在进行下次加密前，将密文字节左移到寄存器中。

解密采用另一套工作方式，它也是首先采用 IV(与加密时使用的值相同)对寄存器进行初始化处理，然后每次处理一个字节。

这种模式下运行密码算法可以很容易地在用户键入数据时对其进行加密处理，用户每次按键输入一个字符。

OFB——输出反馈

这种流密码算法模式与 CFB 很相似，它与反馈位所使用的方式不同。在 CFB 使用密文反馈的地方，OFB 在使用下一个明文位对它们进行 XOR 运算前使用加密计算的反馈位。该模式也称为同步流密码算法。

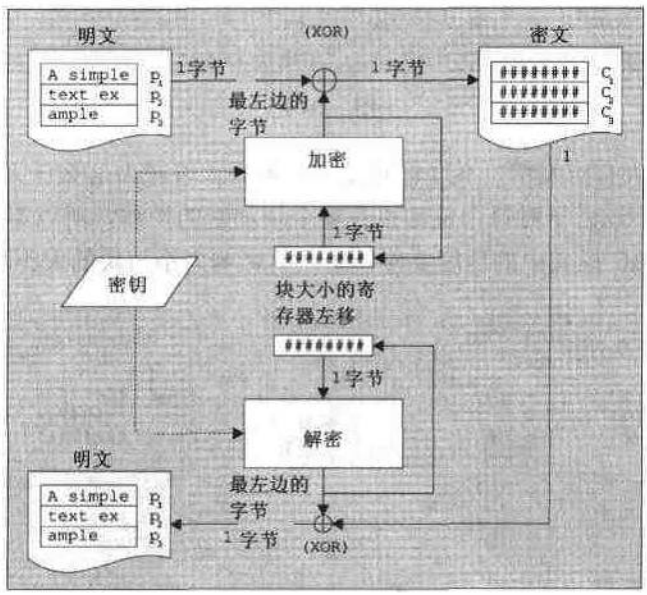

图　1-16

1.3.4　非对称密码算法

对于单密钥系统，如前所述也有一些系统使用双密钥。这些非对称密码算法使用两个不同但在数学上却相关的密钥，只能使用另一个密钥对使用其中一个密钥的明文进行解密处理。即使用于加密数据的密钥也不能用于对它进行解密处理。

非对称加密技术通常被称为“公钥加密技术”。没有双方必须保密的单个密钥，公钥加密技术只有一个必须由一方保密的私钥，第二个密钥就是公钥，任何想与他们通信的任何人都可以随意使用它。

该机制可以在从未接触过的情况下通过 Internet 与实体进行安全通信，这就是安全 Web 通信所采用的技术，该项技术可进行电子商务。

说明：

使用非对称算法，就可以使用用于加密数据的其中一个密钥对数据进行解密．不过，通常

公钥用于加密，私钥用于解密。

非对称加密和解密

非对称密码算法与对称密码算法稍有不同。对称密码算法将密钥和数据主要当作排成行的位处理，非对称密码算法对数字执行数学运算(如乘法、求模运算)，因此必须将二进制数据分成一系列二进制数。这些二进制数用作非对称加密技术的数据块。为了获取可以一起使用的数字，数据块的长度必需与密钥的长度相等。如果密钥有 1024 位(128 个字节)，明文和密文数据块就会是 128 个字节。下面的示例采用 32 个字节=256 位作为密钥长度，它比多数非对称密码算法使用的密钥都要短得多。

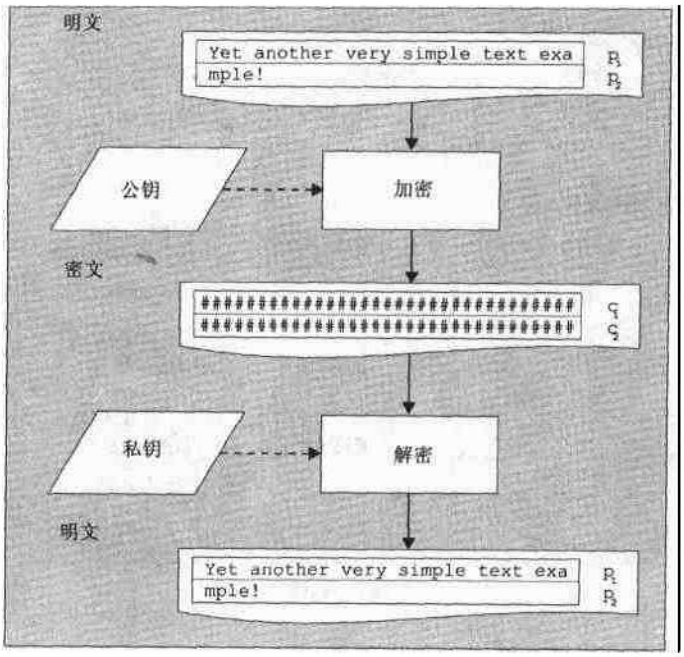

图　1-17

使用公钥加密数据是安全的，因为即使心怀恶意的偷听者知道公钥，也不能用于解密数据。只有使用私钥(由一个用户保密)才能解密消息。因此，Bob 可以使用 Alice 的公钥加密消息，这样加密消息是安全的，只有 Alice 才能阅读该消息，Eve 或 Mallory 甚至 Bob 本人都不能阅读。

1.4　完整性

有时确定发布的信息从头到尾都没有更改过比较重要，如合同或软件模块。在加密术语中这就是数据完整性(data integrity)，它通常用作辅助手段确保信息安全。某些情况下，如软件模块中，其用途就是保证自部署甚至创建以来软件模块的内容就没被修改过。.NET 本身广泛使用基于非对称加密技术的密码系统，以确保强名程序集自创建以后就没有被修改过。

提供数据完整性信息的加密算法称为散列或消息摘要算法，应用它们的过程与加密过程相似，主要区别就是加密过程为双向可逆过程，而散列或消息摘要却是单向的。散列或消息摘要

计算的产品或结果就是散列或消息摘要，它采用比较小的大小固定的形式表示明文，对于某个明文其消息摘要总是相同。

从本质上看，散列提供单向加密。不能从散列重新创建明文，却可以通过比较散列来验证两个明文是否相同。

为了利于加密，每个散列值都必须相对惟一，从某种意义上讲，要编写散列值相同而意思不同的明文极为困难。

图 1-18 中，Alice 编写消息，计算出散列，并将消息和散列发送给 Bob。Bob 也根据消息计算散列，并使用他从 Alice 那里获得的消息来计算散列。如果散列相匹配，他就知道该消息完整无缺。

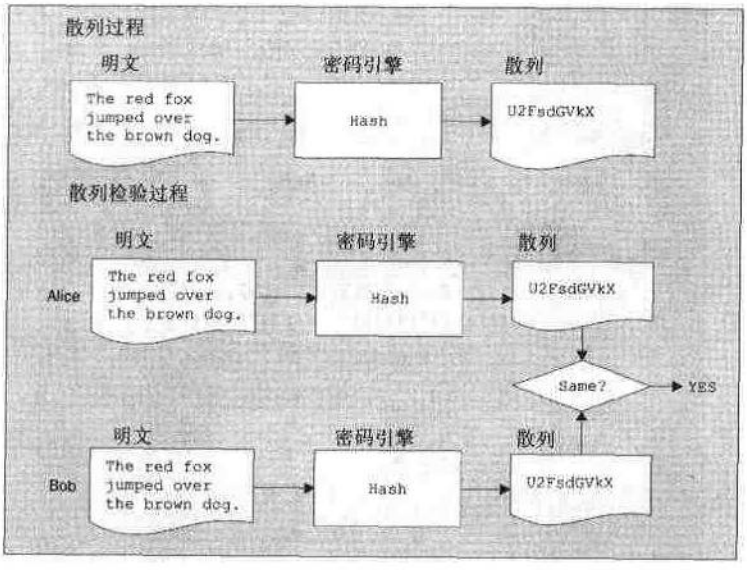

图　1-18

下面将介绍在基本的信息交换时如何应用散列。

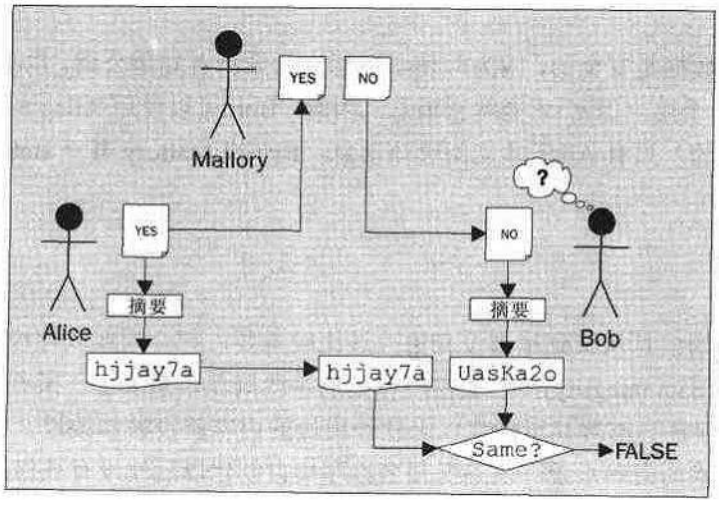

图　1-19

图 1-19 中 Mallory 不知道为什么不能获得随消息发送给 Bob 的摘要。Bob 必须有消息摘要才能通过比较消息摘要来验证消息的完整性。由于 Mallory 只截获并篡改了消息，通过消息来计算消息摘要，并将结果与 Alice 发送来的摘要比较就可以很容易地知道消息已被修改过。不过，Mallory 也可能获取消息摘要，然后也可以伪造摘要。

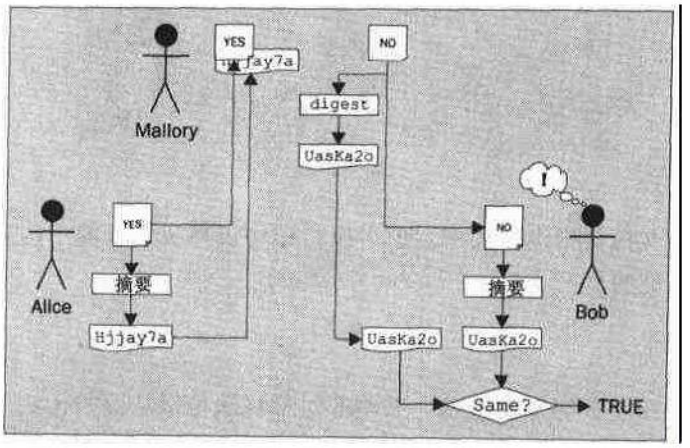

图　1-20

图 1-20 中 Mallory 截获了消息和摘要值。然后，他伪造新消息并确定该消息的新摘要，并发送给 Bob。由于摘要是根据伪造的消息生成的，所以 Bob 不能确定消息是伪造的。

因此，我们需要找到一种方法来确保散列安全，确保散列不能被复制。有两项技术可以使用，它们是消息认证码(Message Authentication Code)和数字签名(Digital Signature)。稍后将介绍它们。

散列对于存储希望能够与其他数据进行比较却不能阅读的数据也很有用。典型示例就是口令文件。我们不需要将口令以纯文本形式存储到数据库中，而是可以存储口令散列，在用户输入口令时再计算用户口令的散列，检查散列是否与口令文件中的散列相匹配。如果两个散列相匹配，那么用户输入的口令就是正确的。

1.4.1　MAC——消息认证码

MAC 就是对称加密技术，它提供安全的消息摘要格式。MAC 的计算方法与散列计算方法一样，不过要添加加密密钥以保护散列值不会被篡改。使用该技术就可以信任数据的完整性，甚至通过 Internet 传输也可以信任。MAC 技术在 Internet 传输层安全协议如 TLS 和 SSL 中起着重要作用，因为它与其他技术如数字签名相比，性能更为优越。

如果不知道接收者希望看到的用于对 MAC 进行编码的密钥，就不可能伪造 MAC，Mallory 就不可能伪造发送给 Bob 的消息和 MAC。图 1-21 就是信息交换的情形，Mallory 伪造发送给 Bob 的消息。

这种情况与加密消息后 Mallory 试图使用自己的密钥伪造消息的情形相似，如果 Bob 遵守与 Alice 达成的协议，这就没有任何意义，因为 Bob 可以轻易发现 MAC 并没有验证消息。不过，我们需要共享 Alice 和 Bob 之间使用的密钥，他们则需要确定 Mallory 没有获得该密钥。

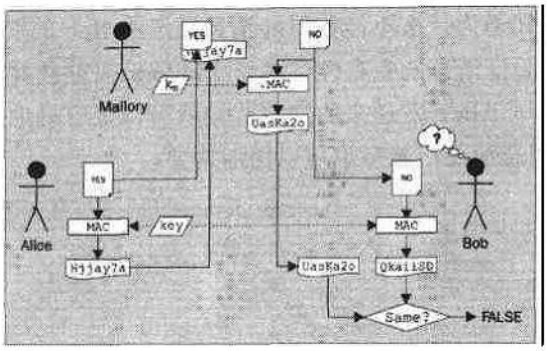

图　1-21

请注意，这里的文档数据没有加密。您可以将该协议与我们的简单信息交换协议结合起来使用，并交换经过验证的加密文档。

1.4.2　签名

本章前面简要介绍了公钥加密技术。某些非对称算法可以从两种密钥中任意选择一种进行加密和解密。不过，接下来还讨论了在我们持有私钥的情况下如何使用公钥(包括 Eve 和 Mallory)。在这种情况下，使用私钥进行加密会有什么益处呢？由于公钥可以对采用私钥加密的消息进行解密，Eve 和 Mallory 可以解密这样的消息。因此在这种方案中使用私钥就没有什么用处。那该在什么情况下使用呢？

我们不能使用该技术保密数据，不过可以使用它证明只有一个人——私钥持有者——可以对消息进行加密处理，使用与私钥相匹配的公钥就可以对该消息进行解密处理。它的功能很强大。因此，我们通常不将这些操作看作加密和解密，而是签名和确认。

数字签名如何证明某些明文数据就是采用某个密钥签名的呢？既然称之为签名，指的就是可以很容易地附加到要签名数据上的小部分数据。要执行签名，必须执行下面的过程：

(1) 通过安全消息摘要算法如 SHA-1 运行数据

(2) 使用私钥在消息摘要上签名

如图 1-22 所示，采用这种方式，必须使用与密钥等长的签名结尾(假定消息摘要结果比密钥短)。由消息摘要算法的强度、非对称密码算法和密钥长度来定义签名的技术强度。

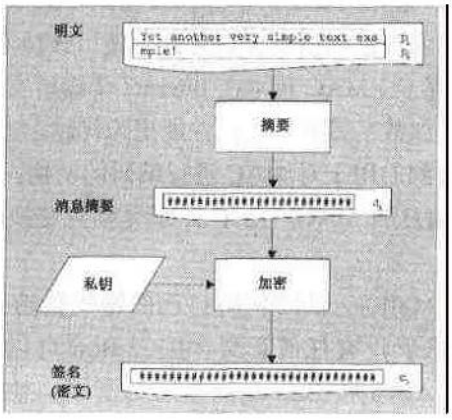

图　1-22

签名确认

签名的确认与签名的生成一样简单，如图 1-23 所示。需要与签名时处于相同状态的原始消息、公钥和签名的副本。

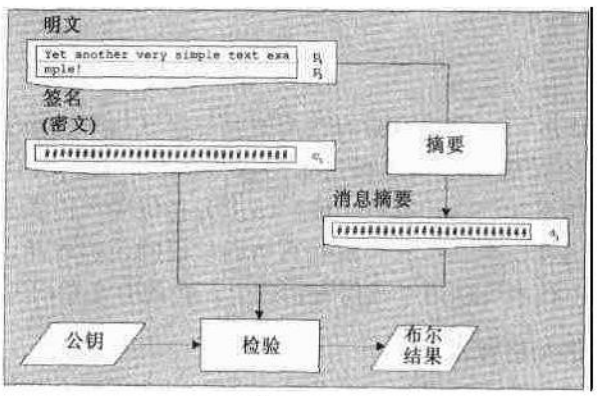

图　1-23

因此，使用该技术就可以为消息生成不能伪造的安全消息摘要，Alice 就不必与 Bob 共享私钥。要确认其签名，Bob 只需要 Alice 的公钥即可。Mallory 不能伪造消息摘要是因为他没有 Alice 的私钥。

1.4.3　安全保存协议

下面将介绍一种稍有些不同的加密方案——不过这种也很常见。它用于安全地处理那些存储在其他用户可以访问的存储介质中的数据。这是相当简单的应用，其中两个对称密钥 ks 和 ki 用于保护该示例中称为 p 的信息(明文)。我们应该使用两个不同的密钥来提高数据安全性。即使其中一个密钥受到破坏，仍然可以监测主动攻击情况。

1. 对数据进行加密处理

某些密码算法引擎允许在一个通道中同时运行 MAC 和加密，不过该示例表明您必须进行其中一种操作以确保数据安全，如图 1-24 所示。

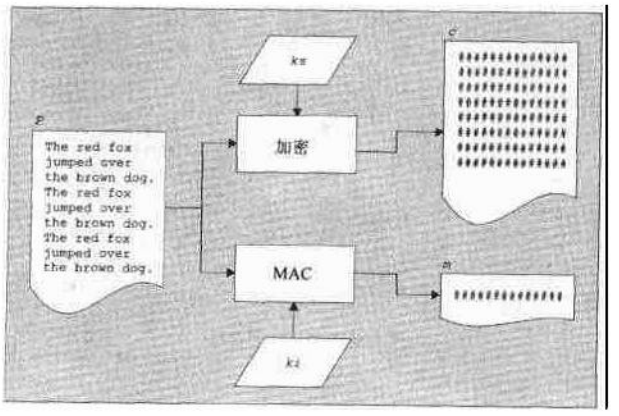

图　1-24

(1) 生成保证私密性的密钥(ks)

(2) 生成保证完整性的密钥(ki)

(3) 使用 ki 根据明文生成 MAC(m)

(4) 使用 ks 对明文 p 进行加密处理生成密文

(5) 确保密钥 ks 和 ki 的安全

2. 确保密钥安全

您可能已注意到，该示例中比较脆弱的就是如何确保这些密钥的安全。初次使用密码术时常常会出这样的错误，不过确定密钥是很安全的也很重要，否则所有的保护都会失效。包括Windows在内的多数操作系统都在计算机上或网络上为每位用户提供了安全的存储区用于存储密钥，利用操作系统的内置安全确保只有那些有访问权限的用户才能访问那些密钥。不过，在操作系统之外也有两三项技术可以用于确保密钥安全，在此有必要研究一下。

PBE——基于口令的加密技术

尽管没有一种完美的方法可以确保密钥安全，但是口令或口令短语都可以用于保护信息，采用哪种方式视情况而定。有许多加密技术可以根据人们可以记住的密码生成好的密钥。不过，解密专家认为密码公开的同时受 PBE 保护的数据就会遭到破坏。有一些技术可以使猜密码变得更为困难，对于优秀的密码专家而言破解生成整个密码的密钥只是一个时间问题。

PKE——公钥加密

确保密钥安全的更好方法就是使用公钥加密方案。使用该方案至少需要一组非对称密钥，即公钥和私钥。公钥用于对要想加以保护的私钥进行加密处理，私钥则用于解密密钥。采用这种方式只是不必有一组非对称密钥，也不必担心将受 PKE 保护的对称密钥存储在哪个地方，因为它们总是安全的。

说明：

也可以考虑将私钥存储到可移动媒介上，如软盘、智能卡，或者将这些媒介存储到物理安全的地方。

3. 对数据进行解密处理

首先，必须单独使用 PBE 或结合 PKE 和 PBE 恢复对称密钥，如图 1-25 所示。

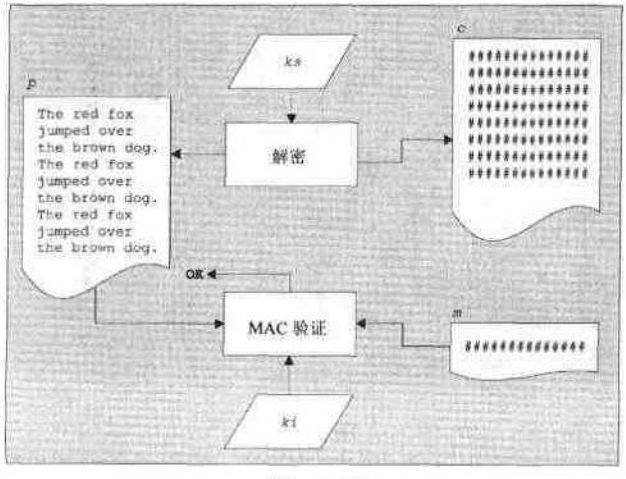

图　1-25

(1) 恢复密钥 ks 和 ki

(2) 使用 ks 对密文进行解密处理生成明文 p

(3) 使用 p 和 ki 确认 MAC

1.5　身份验证

不管技术如何，所有验证的原则都是相同的。请求方、用户、计算机等都必须提供正确的证书才能访问所请求的信息。证书就像 IP 地址或密码一样简单，但是也可以与生物测量设备配合读取请求访问者虹膜的数字证书一样深奥。

基本情形就是，Alice 和 Bob 通常能确定与他们通信的人是谁，因为只有他们才有密钥。我们没有说明他们如何才能获得相同的私钥，必须假定他们在某个时候已见过面并且就使用的密钥已达成协议。

如果 Alice 和 Bob 不曾谋面，或 Bob 在与 Eve 的闲聊中偶然说出了他与 Alice 的密钥，那么他们就必须换一个新密钥，接下来他们就会遇到一个新问题。他们如何才能就密钥达成协议？Alice 可以将密钥邮寄给 Bob，但 Bob 如何知道所收到的密钥不是 Mallory 在搞鬼，或 Eve 没有在密钥传输过程中截获该密钥，从而可以继续监视他与 Alice 之间的对话。

即使在讲公钥的示例中，我们也可以假定 Alice 不知如何将她手里的公钥副本提供给 Bob，但是 Bob 如何知道 Alice 的公钥并不是 Mallory 的呢？

这里就有一个身份验证的问题，这就需要受信任的第三方 Trent 加入到我们的讨论中。

关于 Alice 和 Bob 如何在 Eve 不知情的情况下就私钥达成协议，或双方必须进行面谈的问题，下面将通过交换密钥的过程来说明。

密钥交换协议

这里将介绍几种加密协议，下面将讲解其中两种协议——一种使用对称加密技术，另一种使用非对称加密技术。

1. 使用对称加密技术进行密钥交换

该协议包括信任的第三方 Trent，他持有用于保护信息的对称密钥。此时假定 Alice 和 Bob 分别与 Trent 共享一个私钥(KA 和 KB)。

(1) Alice 要求 Trent 提供一个密钥用于保护与 Bob 之间的通信。

(2) Trent 生成一个新的对称密钥 K 并使用 K_A 对 K 的副本进行加密处理以便生成 $E_{KA}K$，使用 K_B 加密 K 的另一个副本生成 $E_{KB}K$。将加密后的密钥 $E_{KA}K$ 和 $E_{KB}K$ 分发给 Alice 和 Bob。

(3) Alice 使用 K_A 对 $E_{KA}K$ 进行解密获得 K。

(4) Bob 使用 K_B 对 $E_{KB}K$ 进行解密获得 K。

(5) Alice 和 Bob 使用 K 安全地通信。

该协议有一点不好，即 Trent 必须获得双方的信任。Mallory 不知道如何攻破 Trent 的防线，他就必须破坏 Trent 作为信任的仲裁者进行的所有通信。如果不知道何故 Trent 不可用，那么所

有通信安全都会被有效禁止，如遭到 DoS 攻击。

Mallory 或 Eve 都不能监听或篡改 Alice 和 Bob 之间的通信。对这个基本协议的攻击也会存在，稍后将详细介绍。

在较大的网络中处理这样的协议会很困难，因为所有客户都需要有单独的私钥用于与 Trent 通信。

2. 使用非对称加密技术进行密钥交换

这个协议的可扩展性更强，它不需要前面分发给 Alice 和 Bob 的私钥，却需要 Trent 的公钥。其操作步骤如下：

(1) Alice 生成公钥/私钥对。

(2) Alice 与 Trent 会面，并给他一份自己的公钥副本。

(3) Trent 在 Alice 的公钥上注释说明他知道该公钥属于 Alice 的文字。

(4) Trent 将签名后的密钥返回给 Alice。

(5) Alice 将签名后的密钥发送给 Bob。

(6) Bob 确认使用 Trent 的公钥作的签名。Bob 信任 Trent，因此他相信密钥是 Alice 的。

(7) Bob 生成一个私钥。

(8) Bob 使用 Alice 的公钥加密密钥并将它发送给 Alice。

(9) Alice 使用她个人的密钥加密该私钥。

(10) Bob 和 Alice 现在共享相同的私钥并可以使用它进行通信。

实际应用中，诸如 Verisign 这样的公司扮演 Trent 这样的角色，如果您可以证明自己的真实身份，这样的公司就可以签发一个公钥(创建我们所知的认证)。Verisignt 和其他认证机构大量发放公钥，也称为根证书，使用根证书可以确认任何出示由他们签发公钥的人的身份。

这些情况都表明 Alice 和 Bob 如何才能建立常用的密钥，并能信任与他们共享密钥的人。下面将介绍另外几个身份验证的情形，主要讲 Alice 如何向计算机而不是 Bob 证明她的身份。

3. 使用单向函数验证身份

(1) Alice 将她的密码发送给其他人即主机。

(2) 主机根据密码计算散列值。

(3) 主机将散列值与其口令数据库进行比较。

这个协议比较简单，在目前的 WWW 浏览器中运行良好，不过由于 Alice 的证书采用明文发送，所以不是安全的。必须使用安全信道如 HTTPS (SSL/TLS)辅助这种加密协议。

4. 使用公钥密码术进行身份验证

该协议进行强有力的身份验证，不过也需要在客户端进行计算操作，常用操作系统中通常没有这种计算，也就是说客户机上必须安装一种特殊的软件来执行这种计算。其操作步骤如下：

(1) 主机将随机值发送给 Alice。

(2) Alice 使用她的私钥在这个随机值上签字生成一个签名。

(3) Alice 将签名发送给主机。

(4) 主机根据随机值和 Alice 的公钥检查签名的真实性。

这个协议提及另一个密钥加密的概念——随机数。前面对协议作了大量讨论，其中双方中必有一方生成密钥，对于这种的操作，我们也需要从密码学的角度看作是随机的数字。如果在加密应用程序中使用普通的随机数，就会在密码系统中出现一些容易受到攻击的地方，因为解密专家可以攻击这种由自己生成随机数的机制，最终就可以预见到下一个随机数会是什么。强加密的随机数生成器专门用于杜绝这种易受攻击的缺点。

1.6　不可否认性

身份验证就是证明某人的访问权限，不可否认性则是证明某人确实执行过某个操作或就某个合同达成过协议等。有时很难区分二者之间的差别，不可否认性所采用的技术就可以用于验证某人的身份。技术方面的差别没有在指定用法和它们所应用的环境中那么多。这是加密协议在其中很重要的一个典型示例。

不可否认性用于帮助解决数字协议方面的纠纷。例如，Alice 通过电子邮件向 Bob 许诺付$1000 购买名为 C# Data Security Handbook 的书的第一版，但是交货后她却拒绝付款。Bob 持有的有关协议的惟一证据就是 Alice 发来的电子邮件，但这在法庭上未必可以作为证据。Alice 可以声称 Bob 只是用写字板的副本形式生成了电子邮件。如果根据不可否认的加密协议使用数字签名在电子邮件上签名，Alice 就很难在付款问题上抵赖。

1.6.1　不可否认案例

在这个案例中，Bob 将小汽车卖给 Alice，如果其中一个人不诚实，Bob 或 Alice 都可能遇到问题。首先，看看理想的情况应该如何(不需要加密技术)，如图 1-26 所示。

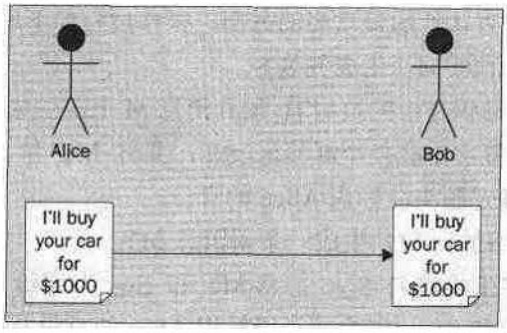

图　1-26

(1) Alice 向 Bob 发送一则消息称她出价$1000 买下其小汽车。

(2) Bob 阅读该消息后就将小汽车提交给 Alice。

(3) Alice 向 Bob 支付$1000。

理想的情况应该如此，Alice 和 Bob 两人都很诚实，并且双方都履行了协议。

不过，也有可能 Alice 或 Bob 其中一方会欺骗另一方，这就得看事情的进展如何了。如果 Bob 要求在交货前付款，他可以轻易从 Alice 手中拿到钱，然后声称没有拿到钱，因此不必将小汽车给她。如果 Bob 付款前向 Alice 提交小汽车，Alice 可以说已将钱付给 Bob 了并拒绝付款。

由记录 Alice 和 Bob 之间任何协议内容的第三方参与就可以解决这个问题，这与公证人一样，如图 1-27 所示。

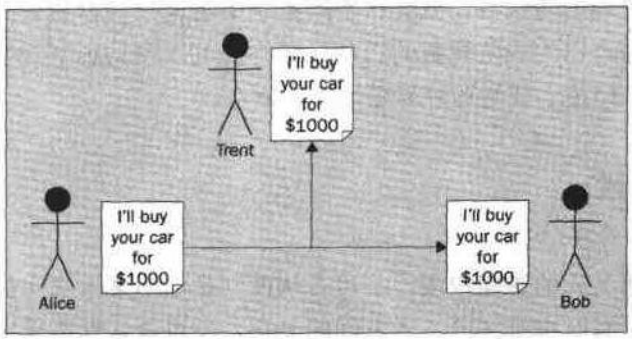

图　1-27

(1) Alice 向 Bob 发送一则消息称她出价$1000 买下其小汽车。

(2) Alice 将同一则消息发送给可信的第三方 Trent。

(3) Bob 阅读该消息后就将小汽车提交给 Alice。

(4) Alice 向 Bob 支付$1000。

现在 Alice 或 Bob 是否诚实与否都无关紧要。Trent 手中握有证据！

由于记录了 Alice 和 Bob 之间所有协议和交易的第三方的参与，使得他们任何一方都很难否认自己的承诺或动作。

1.6.2　不可否认协议

通用的协议实现了不可否认性的最重要的方面，操作过程如下所示：

(1) Alice 编写她签名的消息 M，生成签名 S_M。

(2) Alice 将签名 S_M 和标识身份的页眉 H 加到消息 M 上，生成一条新消息 M1。

(3) Alice 在消息 M1 上签字生成一个新签名 S_{M1}，并将 M1 和 S_{M1} 发送给 Trent。

(4) Trent 检验 S_{M1} 并确定题头能标识 Alice 的身份。

(5) Trent 将时间戳 T 添加到 M1 生成一条新消息 M2。

(6) Trent 在 M2 上签字生成签名 S_{M2}，并将 M2 和 S_{M2} 发送给 Alice 和 Bob。

(7) Bob 检验 Trent 的签名 S_{M2}，确定题头能标识 Alice 的身份并检验 Alice 的签名 S_{M1}。

(8) Alice 检验 Trent 的签名 S_{M2}，其内容就是她的原始内容，时间戳 T 也是正确的。

1.7　算法

最后将简要介绍现在广泛使用的算法。这里只简要介绍它们，不过更多相关信息及其他算法可以在 Internet 上查询。建议您时刻关注安全方面的消息，这方面的技术发展得很快。

1.7.1　加密软件出口问题

在 2000 年 1 月以前，美国广泛限制加密软件的出口，根本就不允许将使用不容易破解的算法编写的程序出口到其他国家。由于现在广泛使用的绝大多数算法都来自美国，因此这是一个大问题。现在这方面的限制已放松了许多，允许出口到除下面几个禁止向其出口的目的地外的所有国家：古巴、伊朗、伊拉克、利比亚、北朝鲜、苏丹、叙利亚、塞尔维亚和阿富汗由塔利班控制的地区。

如果想使用美国发明的算法，就可以将软件分布到除上面提到的那些目的地以外的任何国家。不过，您自己的国家完全可能不会接受您加密的作法。法国就是这样的国家，对于加密他们有非常严格的内部规定，因此您必须搞清楚要使用加密软件的每个最终目的地的规章制度，与软件是否来自美国无关。

1.7.2　对称密码算法

现在使用广泛的一些对称密钥密码算法如下所示，不过还有其他许多密码算法。要变得流行，密码算法必须经历或长或短的时间考验。密码算法使用的时间越长且没有遭到任何破坏，它的使用就会越广泛。

- DES——数据加密标准

  1970 年由 IBM 发明。尽管这种密码目前使用得最为广泛，但是现在它已不再十分安全。据说现在这种密码算法已遭到过成功攻击，因此在需要高度安全的地方不应该使用该算法。

- 三重 DES——数据加密标准

  通过使用 3 个密钥来使用 DES 算法 3 次(也称之为 3DES 或 EDE)。第一个密钥加密明文，第二个密钥解密由第一个密钥生成的密文，第三个密钥加密由第二个密钥解密的结果(因此命名为 EDE)，这样就可以有效地将密钥长度增加为原来的 3 倍。现在还没听说成功攻击过该密码算法，对于需要高度安全的加密可以使用它。

- Rijndael——AES(高级加密标准)

  由 Joan Daemen 和 Vincent Rijmen 开发。2002 年 5 月，它作为 AES 程序的候选者提交并成功胜出。该密码算法赢得 AES 的冠军自然已成为众人关注的焦点，它已在越来越多的加密库中得到实现。对于现在大部分对称加密应用程序来说，自然应该选择它。

- Blowfish——Blowfish

  1993 由 Bruce Schneier 发明。开发它供人免费使用来替代 DES。只要根据所推荐的指导原则生成密钥，这种密码算法是很优秀的(实际应用中进行 15 层或更高层次的加密处理)。

- RC2——Rivest Cipher 2

  由 Ron Rivest 发明，其商标为 RSADSI (RSA 数据安全)。设计用于替代 DES，不过使用 128 位及更多位的长密钥要安全得多。

- RC4——Rivest Cipher 4

  最初于 1994 年匿名发布，由 RSADSI 发明并用 RSADSI 注册商标。这是一种流密码算

法，已被其他算法取代，如 3DES-CBC。

1.7.3 非对称密码算法

和对称算法一样，某些非对称算法已比其他算法的应用更为广泛。最为广泛使用的一些非对称算法如下所示：

- DH——Diffie Hellman
  1976 年由斯坦福大学发明。它仍然是现在使用得相当广泛的密钥协议和密钥交换算法，如可以在 SSL 中实现。
- RSA——Rivest, Shamir, and Adelman
  1977 发明。这是现在最严格的数字签名应用程序中应用得非常广泛的公钥。RSA 密码算法用于 X.509 证书。
- DSS——数字签名标准 (DSA)
  1991 由 NSA(美国国家安全局)开发。开始开发为数字签名标准，后来标准化(在美国)为 DSS。现在仍然使用 DSA 的缩写形式。DSS 只能签名和检验，不能加密或解密数据。

1.7.4 消息摘要和散列

- MD2——消息摘要 2
  1989 年由 Ron Rivest 开发。已有成功攻击该算法的事件发生(这表明可以找到该算法的冲突，也就是除原始数据以外的有意义的明文生成同样的散列值)，它生成的散列值太小不足以防止攻击。下面的算法就可以替代它。
- MD4——消息摘要 4
  1990 年由 Ron Rivest 开发。这个摘要已被破解。在任何环境下都不要使用它。
- MD5——消息摘要 5
  1991 年由 Ron Rivest 开发。在许多情况下都可以使用该算法，不过现在可以使用更安全的算法。
- SHA-1——安全散列算法 1
  1995 年由 NSA(美国国家安全局)开发。这是现在使用最广泛的散列算法，TLS 协议和 SSL 中都实现了该算法。SHA-1 及其以前的版本也是数字签名的首选算法。

1.7.5 消息身份验证码

某些 MAC 实现方案使用口令或相似的东西而不是密钥，有些实现方案则结合口令和密钥使用。不过，现在差不多只有 MAC 被广泛使用。

- HMAC——基于散列的消息身份验证码
  1996 年由 Bellare、Canetti 和 Krawczyk 发布。该规范介绍了流行的散列算法和对称密码算法的所有组合形式。使用最为流行的组合就是 3DES-CBC 和 SHA-1。

1.8　小结

现在，(如果已将本章从头至尾读过一遍)，您是否觉得有点头晕？我们将要点总结如下。许多应用程序运用少数几个广泛接受的模式。通常可以使用下面两种模式，事实上，它们是基本模式：

长期数据加密：

- 生成两个私钥
- 使用其中一个密钥(MAC)确保数据的完整性
- 使用另一个密钥加密数据
- 确保密钥安全

短期数据加密：

- 生成两个私钥
- 安全地交换密钥(在这种情况下，实际使用证书)
- 使用其中一个密钥(MAC)确保数据完整性
- 使用另一个密钥加密数据
- 传输数据和 MAC

在发明新的协议前，应该先搜索一下广泛接受的协议。您可能会错误地认为手中掌握的加密技术已不能作任何改进，请注意，要掌握非常复杂的科学应该不会太轻松。

本书将采用这些模式及其他模式创建许多应用程序。

加密技术是一个范围很广泛的科学，就一章内容不能全面介绍它。应该学习一些基本原则和可以使用的算法，并了解在一些领域中对加密技术的实际应用。鼓励您学习更多加密技术的理论，并时刻关注这个学科的发展，这里并没有告诉您加密技术的发展前景如何。

第2章 .NET密码术

在.NET Windows 出现之前，使用密码术通常就是指使用非托管的 Win32 CryptoAPI 库，这对 C++程序员来说工作很繁杂，对于采用其他语言如 Visual Basic 处理的编码器则更不方便。和许多其他编程方面的工作一样，.NET 中密码术的运用使人感觉良好。开发人员可以使用一流的加密模型，将这些模型放在类库中并且它们具有很好的扩展性。不用调用非托管库中晦涩难懂的函数，现在可以使用一组功能齐全的加密技术类，这些类将那些繁杂的编码数据、计算散列和生成安全的随机数等工作都包装起来了。

本章首先介绍.NET 的加密服务。然后详细探讨类模型，并阐述它如何使用流和加密转换，以及如何使用新算法和实现方案扩展它。另外，还会列举基本示例，说明如何使用对称和非对称加密以防止数据泄露给他人，以及如何应用基本技术管理密钥信息。事实上，本章介绍了许多技术，并利用这些技术创建了实际的应用程序。

首先，简要介绍加密技术和核心命名空间。

2.1 .NET 密码术模型

.NET Framework 有下面 3 个密码术命名空间：

- System.Security.Cryptography 包含完成所有加密任务的核心类。
- System.Security.Cryptography.Xml 包含可结合使用 System.Security.Cryptography 类，对 XML 文档部分内容进行加密和签名的类。
- System.Security.Cryptography.X509Certificates 包含允许检索证书信息的类。

第一个命名空间 System.Security.Cryptography 是目前最重要的命名空间，由于它包含所有加密类型的基本功能，并且有创建散列代码(第 3 章将讨论)、生成安全随机数字(附录 B 将讨论)和交换密钥的其他类。

System.Security.Cryptography 命名空间中的核心加密类分为 3 层，如图 2-1 所示。第一层是一组抽象类，它们表示加密算法的类型，用于完成特定的加密任务。其中包括：

- AsymmetricAlgorithm(表示非对称加密)
- SymmetricAlgorithm(表示对称加密)
- HashAlgorithm(表示散列的生成和检验)

第二层包含表示特定加密算法的类。它们均由加密基类派生而来，不过它们也属于抽象类。例如，DES 算法类就表示 DES(Data Encrgption Standard，数据加密标准)算法，它由 SymmetricAlgorithm 派生而来。

第三层类就是一组加密实现方案(implementation)。每种实现类都由算法类派生而来。也就是说，特定的加密算法如 DES 可能有多个实现类。实际应用中，.NET Framework 则只提供一

个实现类(DESCryptoServiceProvider)，不过 Microsoft 或第三方供应商以后也可能创建性能更好的其他实现方案。

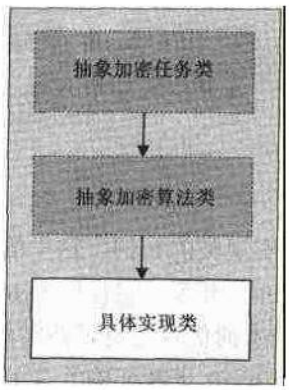

图 2-1

尽管一些.NET Framework 加密类是在托管的代码中实现的，绝大多数类都是 CryptoAPI 库上的瘦包装器(thin wrapper)。包装 CryptoAPI 函数的类在其类名称中有 CryptoServiceProvider(例如，DESCryptoServiceProvider)，而托管类一般在其名称中包含有 Managed(例如，RijndaelManaged)。从本质上讲，托管类在.NET 中执行任务时都是在 CLR 的监管下完成的，而非托管的 CSP 类则采用对 CryptoAPI 库的非托管的 P/Invoke 调用。这看起来有点受限制，实际上它是对现有技术的高效重用。CryptoAPI 的技术没有缺陷，但它的编程接口很不好用。

这种 3 层组织(如图 2-2)几乎可以不受任何限制地扩展。通过从现有算法类生成派生类就可以为现有加密技术类创建新的实现方案。例如，通过新建 DESManaged 类并从抽象类 DES 继承其功能，就可以创建完全采用托管代码实现 DES 算法的类。与此相似，通过添加它的抽象算法类(如与 DES 相似的，而.NET Framework 中却没提供的 CAST128)和具体的实现类(如 CAST128Managed)就可以增加对新加密算法的支持。

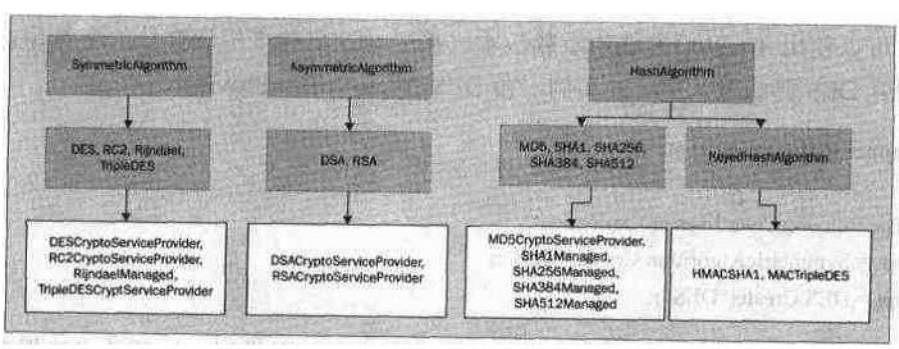

图 2-2

注意：

加密类是.NET 类库中的几个示例之一，它不遵守标准的命名规则。例如，可以找到 TripleDES 和 RSA 这样的类，但没有 TripleDes 和 Rsa 这样的类。

2.1.1 抽象类

抽象类实际上有两种用途。首先，它们定义实现加密时所需要支持的基本成员。不过，它们也有一些可以直接使用的功能，不需要通过 Create()静态方法创建类实例。采用该方法可以创建其中一个具体实现类的实例，而不需要知道如何实现它。

例如，可以使用下面的代码行：

```
DES crypt = DES.Create();
```

Create()静态方法返回默认 DES 实现类的实例。这里的类就是 DESCryptoServiceProvider。不过，上面的代码没有假定它就是如此。相反，它使用 SymmetricAlgorithm 和 DES 基本类中定义的基本成员来处理新实例。这种技术的优点是可以按常规方式编写代码，不用依赖于具体实现方案。最好的情况就是，如果 Microsoft 更新架构并且 DES 实现类发生变化，您编写的代码就会无缝地随之变化。如果您使用 CSP 类，这就特别有用，以后也可以在相应的托管类中重新编写它。

事实上，如果您愿意也可以在其中一个加密任务类中使用 Create()静态方法，就可以在更高的层次上运行。例如，采用下面的代码：

```
SymmetricAlgorithm crypt = SymmetricAlgorithm.Create();
```

这将创建任何被定义为默认对称算法的加密类的实例。这里它就不是 DES 而是 Rijndael。返回的对象就是 RijndaelManaged 实现类的实例。

您可以更改这些默认值，稍后将介绍如何配置.NET 加密系统。

说明：

使用抽象算法类按常规方法编写代码是很好的做法。这就可以知道使用的是哪种算法(及其局限)，而不用担心基本的实现方式。

最后，可以使用 Create()方法的重载版本，接受通过名称来标识算法的字符串。例如，下面的代码创建 DES 算法默认实现的实例，即 DESCryptoServiceProvider。

```
SymmetricAlgorithm crypt;

// The following two lines are equivalent.
crypt = SymmetricAlgorithm.Create("DES");
crypt = DES.Create("DES");
```

如果要在 application.config 文件中使用配置设置确定加密类型，这项技术就很有用。可以从该文件中读取要使用的加密算法并将它传递给 Create()方法。

2.1.2 CryptoConfig

上面的示例引出了几个有趣的问题。即在哪里定义算法名称和类之间的映射关系以及如何指定默认算法？运行时，Create()方法使用 CryptoConfig 类，这个类提供了一个静态帮助方法 CreateFromName()。CryptoConfig 类使用表 2-1 中列出的默认映射关系。

表 2-1

简 写 名	算 法 实 现
SHA	SHA1CryptoServiceProvider
SHA1	SHA1CryptoServiceProvider
MD5	MD5CryptoServiceProvider
SHA256	SHA256Managed
SHA-256	SHA256Managed
SHA384	SHA384Managed
SHA-384	SHA384Managed
SHA512	SHA512Managed
SHA-512	SHA512Managed
RSA	RSACryptoServiceProvider
DSA	DSACryptoServiceProvider
DES	DESCryptoServiceProvider
3DES	TripleDESCryptoServiceProvider
TripleDES	TripleDESCryptoServiceProvider
Triple DES	TripleDESCryptoServiceProvider
RC2	RC2CryptoServiceProvider
Rijndael	RijndaelManaged

可以使用这些名称中的任何一个名称来指定算法，或可以向字符串提供完全限定的类名称。例如，采用默认设置值，下面的语句就是等价的：

```
SymmetricAlgorithm crypt;

// The following two lines are equivalent.
crypt = SymmetricAlgorithm.Create("DES");
crypt = SymmetricAlgorithm.Create(
  "System.Security.Cryptography.DESCryptoServiceProvider");
```

当然，需要特别注意不要遇到强制转换错误。例如，上面的代码可以创建由 SymmetricAlgorithm 派生而来的任何实现。不过，下面的代码成功地创建实现类，如果试图将引用存储到 crypt 对象中就会抛出异常。

```
SymmetricAlgorithm crypt;

// This won't work because RSA is an asymmetric algorithm.
crypt = SymmetricAlgorithm.Create("RSA");
```

如果指定没有定义的字符串，就会收到 CryptographicUnexpectedOperationException 异常。

2.1.3 配置算法映射

通过更改计算机的 machine.config 文件(该文件通常在 C:\[WindowsDirectory]\Microsoft.NET\Framework\[Version]\CONFIG 目录中)，可以方便地修改每台计算机的 CryptoConfig 映射。从本质上讲，通过添加<cryptographySettings>部分重写默认设置就可以更改 CryptoConfig 映射。默认情况下不提供该元素，不过可以添加它，代码如下所示：

```
<configuration>
  <mscorlib>
    <cryptographySettings>
      <cryptoNameMapping>
        <cryptoClasses>
          <cryptoClass />
        </cryptoClasses>
        <nameEntry />
      </cryptoNameMapping>
    <cryptographySettings>
  </mscorlib>

  <!-- Other configuration settings go here. -->

</configuration>
```

在<cryptoClasses>部分中，可以添加其他<cryptoClass>元素标识要添加到密码系统中的新算法实现类。全局程序集缓存(Global Assembly Cache)中必须安装推荐的类。也可以将<nameEntry>元素添加到<cryptoNameMapping>部分，定义每个类的友好字符名称。

下面的配置设置定义一种新的密码类，用于 DES 算法的假设的托管实现，并为类指定友好名称 ManagedDES。设置值也可以标识类的强名及其程序集(这里命名为 ManagedDESAssembly.dll)。请注意强名总是包括有关程序集版本、文化和公钥的信息。使用 Windows Explorer 在 GAC 中查找程序集就可以检索这些信息。

```
<configuration>
  <mscorlib>
    <cryptographySettings>
      <cryptoNameMapping>
        <cryptoClasses>
          <cryptoClass ManagedDES="Wrox.Crypto.ManagedDES,
          ManagedDESAssembly, Culture='en',
          PublicKeyToken=a5d015c7d5a0b012, Version=1.0.0.0"/>
        </cryptoClasses>
        <nameEntry name="ManagedDES" class="ManagedDES"/>
      </cryptoNameMapping>
    </cryptographySettings>
  </mscorlib>
```

```
    <!-- Other configuration settings go here. -->

  </configuration>
```

请注意，这样的配置设置没有标识这个类由 DES 派生而来，尽管该类确实是这样的。

一旦这些修改来注册自定义的密码类，就可以使用 Create()或 CreateFromName()静态方法来创建它，代码如下所示：

```
SymmetricAlgorithm crypt = SymmetricAlgorithm.Create("ManagedDES");
```

如果要使用这个类替换 DESCryptoServiceProvider 作为新的默认值，就可以添加其他<nameEntry>元素。例如，<nameEntry>重写默认设置值：

```
<nameEntry name="DES" class="ManagedDES"/>
```

现在该代码返回自定义的 ManagedDES 类：

```
SymmetricAlgorithm crypt = SymmetricAlgorithm.Create("DES");
```

最后，可以添加这个<nameEntry>，将新算法设置为默认的对称算法：

```
<nameEntry name="System.Security.Cryptography.SymmetricAlgorithm"
           class="ManagedDES" />
```

现在该代码也返回自定义的 ManagedDES 类：

```
SymmetricAlgorithm crypt = SymmetricAlgorithm.Create();
```

请注意，这些配置设置值只适用于当前计算机。没有其他方法可以在应用程序文件中设定这些值(如果可以设定这些值，就允许程序获取计算机设置的安全设置，这就可能成为危险的安全漏洞)。一般情况下，不应该依赖于对这些默认值进行修改，因为不应该修改客户计算机上的 machine.config 文件。

2.1.4 高加密支持和 Windows

在深入探讨加密类前，有必要讨论对不同密钥长度的支持。由于过去美国限制加密技术出口，Microsoft 没有采用特定的(较大的)密钥，只有通过单独下载(理论上讲，它会强制保证用户不是限制使用这些加密技术的国家的公民)。这些规则限制最终还是放宽了，最近的操作系统和更新版本都支持这种大尺寸密钥。

所有托管类都支持使用不易破解(长度大)的密钥。不过，在 CryptoAPI 顶层实现的加密算法需要将它用作操作系统的组成部分。如果操作系统没有安装强大的加密技术，或者如果试图使用不支持的密钥大小创建类就会抛出异常。Windows XP 和 Windows .NET Server 支持不易破解的密钥长度，而不需要进行其他安装。不过，支持.NET 的操作系统不全如此：

- 对于 Windows 2000 用户，可以安装 Service Pack 2，它有高加密包(High Encryption Pack)。如果没有安装 Service Pack 2，就必须安装 Windows 2000 High Encryption Pack(从 http://www.microsoft.com/windows2000/downloads/recommended/encryption 下载)，或升级

到 Service Pack 2(从 http://www.microsoft.com/windows2000/downloads /recommended/SP2 下载)。

- 对于少数仍然安装 Windows NT 4.0 的用户，就可以发布标准或高加密版本的 Service Pack。如果已安装高加密服务包，则可以下载 Service Pack 6a 的高加密版本，可从 http://www.microsoft.com/ntserver/nts/downloads/recommended/SP6 下载。
- Windows ME 和 Windows 98 可以通过安装 Internet Explorer 5.5 来支持高加密功能，Internet Explorer 5.5 中包括高加密包。也可以安装与您的 Internet Explorer 版本相对应的高加密包，可以从 http://www.microsoft.com/windows/ie/download/128bit 下载。当然，这些平台只支持运行 .NET 应用程序，而不是开发这些应用程序，也就是说，在开发应用程序时，容易忘记.NET 应用程序最终要在这样的系统上运行。

2.2 构建块

要掌握加密类，就需要多了解基本信道技术的知识。.NET 使用基于流的体系结构来加密和解密，这就可以方便地对来自不同数据源的不同数据类型进行加密处理。使用这种体系结构也可以很容易地飞快连续执行多次加密操作，并且不受算法实际块大小的影响。要知道它是如何工作的，就需要看看下面两个类：IcryptoTransform 和 CryptoStream。

2.2.1 ICryptoTransform 接口

ICryptoTransform 接口由可以执行块加密转换的类实现。这可以是加密、解密、散列计算、基于 64 的编码/解码或格式化操作。ICryptoTransform 的重要信息就是它一次操作一个数据块。InputBlockSize 和 OutputBlockSize 属性定义这种块的大小(以字节计算)，而 TransformBlock()方法则执行实际任务。TransformFinalBlock()接收最后一块数据，并执行其他需要的操作(如使用空格将其补齐)。CanReuseTransform 属性标识 ICryptoTransform 实例是否可以在操作结束后通过调用 TransformFinalBlock()再次使用。

下面的代码片段创建 ICryptoTransform，它使用 DES 算法进行加密处理：

```
DES crypt = DES.Create();
ICryptoTransform transform = crypt.CreateEncryptor();

// You can now use the transform to encrypt a block of bytes.
```

从中可以看到，即使实际执行转换操作的加密函数截然不同，也采用相同的方式执行不同的加密任务。每种加密操作都需要在处理数据之前将这些数据划分成大小固定的数据块。

可以直接使用 ICryptoTransform 实例，不过多数情况下可以采用更为简便的方法，只将它传递给另一个类：CryptoStream。

2.2.2 CryptoStream 类

CryptoStream 包装普通流，并使用 ICryptoTransform 在后台执行操作。主要优点是

CryptoStream 采用缓冲访问，使用它就可以进行自动加密，而不用担心算法所需块的大小。CryptoStream 的另一个优点就是由于它包装了普通的.NET Stream 派生类，所以它可以方便地“处在”另一操作之上，如文件访问(通过 FileStream)、内存访问(通过 MemoryStream)、低级的网络调用(通过 NetworkStream)等。这些功能极为有用——例如，使用它们就可以很快地使用会话密钥加密网络通信、加密由用户提供的值并放入内存缓冲器中等。本书将介绍这些技术的应用。

创建 CryptoStream 需要 3 部分信息：基本流、模式和要使用的 ICryptoTransform。例如，下面的代码片段使用 DES 算法实现类创建 ICryptoTransform，然后结合使用已有流来创建 CryptoStream。

```
DES crypt = DES.Create();
ICryptoTransform transform = crypt.CreateEncryptor();

CryptoStream cs = new CryptoStream(fileStream, crypt,
  CryptoStreamMode.Write);

// Now use cs to write encrypted information to the file.
```

请注意，CryptoStream 可以采用下面两种模式之一：读模式或写模式，与 CryptoStreamMode 枚举的定义相同。在读模式下，在从底层数据流中获取数据时执行转换操作，如图 2-3 所示。

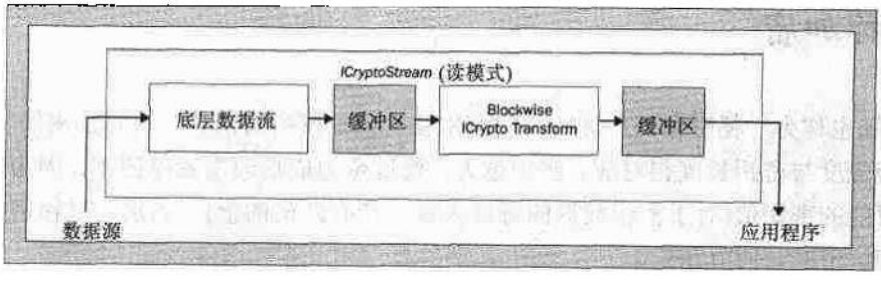

图 2-3

写模式下，在将数据写到底层数据流之前执行转换操作，如图 2-4 所示。

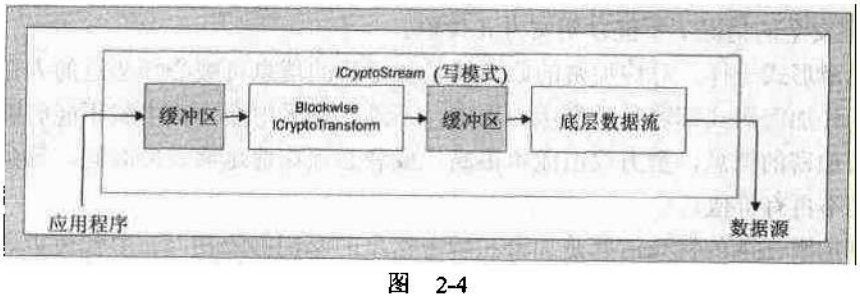

图 2-4

不能同时使用这两种模式创建即可读也可写的 CryptoStream。Seek()方法和 Position 属性用于移到流中的不同位置，CryptoStream 不支持它们，如果调用它就会抛出 NotSupportedException。不过，可以经常将这些成员用于底层数据流，稍后本章将演示操作过程。

说明：

从流读取信息或将数据写到流时，可以使用 CryptoStream。通常，读操作使用 CryptoStream 解密，而写操作则使用它加密数据。不过，这不是必然的。某些情况下也可以在写数据到流时执行解密操作，稍后本章将应用该项技术简化某些字节大小计算的操作。

不管在何处使用常规 Stream 对象都可以使用 CryptoStream 替代它，这是使用 CryptoStream 的好处。下面使用几个实际的示例进行详细介绍。

2.2.3 加密异常

.NET 密码术定义了两个异常类，用于表示可能发生的问题，这些问题涉及范围广泛。它们都没有定义任何新成员：

- CryptographicException 表示加密操作中的常规错误。其中一个常见错误就是试图为算法实现类的属性设置非法值(如指定不存在的密钥大小)。
- CryptographicUnexpectedOperationException 由 CryptographicException 扩展而来，使用频率也会少得多。它表示意外发生的问题，如果实际应用中错误地指定不存在的加密服务提供商，通常就会出现这样的问题。

当然，如果误用加密类，加密代码仍然会出现更大范围的.NET 基本错误。例如，如果试图引用带有使用错误基类类型的变量的算法实现类，就会出现 InvalidCastException 异常。

2.3 对称加密

对称加密也称为“秘密密钥”加密，它使用秘密值加密所有消息。对称加密的特点如下：

- 加密强度与密钥长度相对应。密钥越大，通过蛮力成功攻击就越困难，因为需要测试太多可能的密钥值(对于密钥数据的每位来说，都有两倍的值)。当然，对称密钥越长，加密和解密的时间就越长。
- 对称加密在数学方面要简单一些，运行速度也比非对称加密快得多。处理大量数据时最好选择它。
- 对称加密以双方都知道的共享秘密为基础。进行加密操作前必须知道这个值，在没有危及数据安全的情况下不能以明文方式传输。

和所有加密形式一样，对称加密的原则就是加密后的信息可被心怀恶意的人截获却不能理解其内容。各种加密形式都容易受到蛮力攻击，不过，不采用蛮力攻击就不能破解您所使用的对称加密类型加密的信息，蛮力攻击成本很高，或者必须花费足够长的时间，等到其破解信息时这些信息已不再有价值。

使用对称加密出现的特殊问题是加密和解密信息的密钥必须相同。显然可以选择使用这种共享秘密(如用户口令)，或必须采取其他措施用另外的方式发布这种秘密信息，具体采用哪种方式取决于应用程序。这里采用加密措施确保分布式通信的安全，通常结合使用对称加密和非对称加密，使用它们动态地生成保密信息，请参见第 5 章。

2.3.1 对称算法

表 2-2 列出了.NET Framework 提供的对称算法。大多数情况下，可以使用 Rijndael 作为默认的推荐算法。

表 2-2

算 法	抽象算法类	默认实现类	有效密钥大小(位)	默认密钥大小(位)
DES	DES	DESCryptoServiceProvider	64	64
Triple DES	TripleDES	TripleDESCryptoService Provider	128, 192	192
RC2	RC2	RC2CryptoServiceProvider	40–128	128
Rijndael	Rijndael	RijndaelManaged	128, 192, 256	256

DES、TripleDES 和 RC2 都使用 CryptoAPI 实现，这样对于超过 40 位的密钥，在 Windows 2000 或更老的版本上就需要使用高加密包。另外，还要注意 DES 和 TripleDES 的密钥长度包括奇偶位，计算加密强度不包括它。因此，具有 192 位密钥的 TripleDES 只使用 168 位，而 128 位的密钥则使用 112 位。DES 中，64 位的密钥只使用 56 位。按规定，任何没有超过 100 位的加密都应该视为相当脆弱的加密，也就是说在可以避免使用 DES 的地方就应该不使用它。有关这些算法相对长度的其他信息，请参阅专门的书籍或 Internet 上有关加密原理的资料(如 Bruce Schneier 的 Applied Cryptography(应用加密学)，ISBN 0-471-11709-9)。

2.3.2 SymmetricAlgorithm 类

表 2-2 中所有类都由 SymmetricAlgorithm 派生而来，因此它们支持下列几种属性：

- BlockSize 就是使用该算法在一次操作中可以加密或解密的数据基本单位。必须将较长的数据分解成连续的数据块，而较短的数据要使用其他位将其填满补齐。
- Key 是用于加密或解密数据的秘密值。用字节数组表示它。
- KeySize 是分配给秘密密钥的位数。
- IV 是初始化向量，也用字节数组表示。它提供加密过程的额外输入并提高安全性，稍后将介绍。初始化向量与密钥一样，加密者和解密者都必须知道。不过，它与密钥不同，它不需要保密，可以随消息一起传输。IV 的长度总是与数据块的大小相同。
- LegalKeySizes 和 LegalBlockSizes 返回 KeySizes 实例数组，表明对特定算法可以使用的密钥和数据块大小。

上述属性除 Key 和 IV 外，都是只读的。另外，SymmetricAlgorithm 类也提供了下列方法：

- Create()是一种用于生成默认算法实现类的静态方法，或提供指定算法的类实例。
- CreateEncryptor()和 CreateDecryptor()生成 ICryptoTransform 对象，可以使用该对象进行手工加密和解密字节数据，每次解密一个数据块。
- GenerateKey()和 GenerateIV()生成随机 Key 和 IV 值。通常不需要这些方法，由于创建算法实现类时会创建随机值。
- ValidKeySize()测试当前算法是否可以使用指定位长度作为密钥。

- Clear()释放所有资源，并从内存删除密钥信息。它等同于调用 Dispose()。

某些派生类如 DES 和 TripleDES 添加 IsWeakKey()和 IsSemiWeakKey()方法，使用它可以测试生成加密数据的密钥，加密数据比密电译文更简单。如果使用较弱的密钥对文档进行加密处理，然后再用相同弱密钥对它进行再次加密，结果生成未加密的原文档。例如，在 128 位模式下使用 TripleDES，如果前 64 位与后 64 位相同，那么它就容易被破解。没必要显式测试弱密钥，因为如果试图使用弱密钥就会自动收到 CryptographyException 异常。使用 GenerateKey()方法或初始类值随机生成的密钥将不是弱密钥。

说明：

使用加密或解密对象后就调用 Dispose()，是个不错的主意。所有算法实现类都会重写该方法，这样它就会自动调用 Clear()。Clear()重写存储密钥的内存的敏感部分，这样就可以保护内存免受内存转储(memory dump)攻击。

1. 链接(chaining)

所有 SymmetricAlgorithm 类都提供两个属性，我们还没有介绍这些类。它使用 CipherMode 枚举中的一个值，包括确定如何编码多个数据块的 Mode。

第 1 章讨论过，基于数据块的加密算法的局限之一就是相同的信息块加密后的数据块总是相同。可以利用这种行为来攻击加密后的数据，尤其是在对相同数据进行频繁编码的情况下更会是如此。要增强加密效果可以使用链接模式，它获取上一个数据块的值，并在处理后面的数据块时使用它。采用这种方式，后面的相同数据块的实例加密后的密文就不相同。

有 5 种不同的 CipherMode 值，并不是所有对称加密类都支持所有这些值。其中两个最为常用的值如下所示：

- CipherMode.CBC 使用 Cipher Block Chaining，它是绝大多数算法的默认链接模式，并且最安全。加密数据块之前，通过 bitwise exclusive OR 运算将它与前一个数据块的密文结合到一起。即使前一个数据块包含许多相似数据它也可以确保加密后的数据不相同。由于增加了额外步骤，所以处理时间会长一些。
- CipherMode.ECB 使用电子密码本(Electronic Codebook)并单独加密每个密码块。如果有数据块是一样的(在同一个消息中或使用相同密钥加密的不同消息)，都会转化成相同的加密数据块。该模式的安全性最低，尽管从理论上讲，可以使用它对消息中的部分数据进行解密，或在其中一部分遭到破坏后解密消息的剩余部分。它所需的处理工作稍微少一些，对大型消息的加密速度更快。

了解了链接后，也可以顺便了解一下初始化向量(IV)的作用。在 CBC 模式中，没有反馈值用于第一个数据块，因为此前还没有处理过数据块。使用初始化向量替代。IV 是链接对称加密的重要元素，不应该低估了它的作用。使用链接确定同一则消息中相同数据块是否出现两次，可以将它加密成不同的密文，使用 IV 就可以确定使用相同密钥加密的两个不同消息是否出现在同一个数据块中，可以将它加密成不同的密文数据块。如果总是使用同一个密钥和初始向量，那么消息开头部分数据就总是相同(例如，电子邮件中的 From 行)，加密后消息的开头部分的密文总是相同。IV 则可以使加密后的文本不相同。

显然，加密者和解密者都需要知道 IV。不过，没有必要将 IV 作为第二个密钥来处理——只需将 IV 添加在加密消息的前面就非常安全。使用它任何有密钥的人都可以正确解密消息，却不会向截获消息却没有密钥的有恶意的第三方提供任何其他信息。

2. 填充(padding)

绝大多数数据都不能平均分割成数据块。通常数据都不能将最后一个数据块填充满。这种情况下就必须将其填满补齐。如何填满这样的数据块则由 Padding 属性的值来确定，该属性可接受 PaddingMode 枚举的任何下列值：

- PaddingMode.None 根本就不用填满补齐。如果要编码没有填满的数据块，就会抛出异常。
- PaddingMode.PKCS7 使用 PKCS #7 填充系统。PKCS 是一组公共密钥加密标准，它列出了所有通信情况下的加密协议。这种方案使用一系列字节将数据块填满，每组字节就是要填充字节的总数。例如，如果需要填充 3 个字节，那么每个字节都设置为 03，加密前要添加到最后一块数据块的字符串就是 030303。至少要添加一个填充字节，因此消息中最后一个字节总是会告诉您要添加多少个填充字节。如果消息以整块数据块结束，就添加整个填充块。这是默认的填充模式。
- PaddingMode.Zeros 将所有添加字节都设置为 0，如果数据以 0 值结束，这就可能出错。例如，如果需要添加 3 个字节，要添加的字符串就是“00 00 00”。

PaddingMode.PKCS7 是默认值。使用其他值实际上没有什么好处。

2.3.3 检索密钥数据和默认值

下面的代码就是一个控制台应用程序，它创建 RijndaelManaged 算法实现类的实例，它显示默认字符串的信息。它也在密钥和初始向量中输出数字值，实例化类时就会自动生成该数字值。

```
// EncryptionDefaults.cs

using System;
using System.Security.Cryptography;

class EncryptionDefaults
{
  static void Main(string[] args)
  {
    // Create an instance of RijndaelManaged.
    Rijndael crypt = Rijndael.Create();
```

```
        // Display basic information about this algorithm.
        Console.WriteLine("Block size: {0}", crypt.BlockSize);
        foreach (KeySizes keySize in crypt.LegalKeySizes)
        {
          Console.WriteLine("Max key size: {0}", keySize.MaxSize);
          Console.WriteLine("Min key size: {0}", keySize.MinSize);
        }
        Console.WriteLine();

        // Display basic information about the defaults used.
        Console.WriteLine("Size of current key: {0}", crypt.KeySize);
        Console.WriteLine("Mode: {0}", crypt.Mode);
        Console.WriteLine("Padding: {0}", crypt.Padding);
        Console.WriteLine();

        // Display information about the
        // automatically generated Key and IV.
        Console.WriteLine("Key: {0}", BitConverter.ToString(crypt.Key));
        Console.WriteLine("IV: {0}", BitConverter.ToString(crypt.IV));
        Console.ReadLine();
      }
    }
```

请注意，为了将字节数组输出到控制台，使用 ByteConverter.ToString()静态方法。 它采用了用连字符号连接的两字符的十六进制值序列输出字节数组。

这段代码的输出与下面的输出相似：

```
Block size: 128
Max key size: 256
Min key size: 128

Size of current key: 256
Mode: CBC
Padding: PKCS7

Key:
A3-7E-E1-3F-35-0E-E1-A9-83-A5-62-AA-7A-AE-89-93-A7-33-49-FF-E6-AE-BF-8D-8D-20-8A-49-31-3A
-12-60
IV: F1-8A-01-FB-08-85-9A-A4-BE-45-28-56-03-42-F6-19
```

2.3.4　二进制数据和文本编码

我们一般对二进制数据进行加密。对于绝大部分数据则将这种数据放到字节数组中(包含未加密数据)，并将它作为另一个字节数组(有加密数组)进行检索。如果数据已是二进制格式，这

个过程就很简单——例如，从文件中读取较大图像。不过，读取字符串、整型和十进制数字等基本类型的情况又如何呢？在编码之前需要将这些类型转换成二进制表示形式。

其中一种方式就是使用 System.BitConverter 类，使用它将字节数组转换成字符串。它还提供了重载的 GetBytes()方法，该方法可以将数字类型转换成字节数组。

```
// Convert a number into a byte array.
int number = 102343;
bytes[] data = BitConverter.GetBytes(number);

// You can now write these bytes to a CryptoStream for encryption.
```

GetBytes()不能使用字符串数据。问题就是还有其他方法可以用二进制格式表示字符串，使用何种方法取决于所采用的文本编码。最常见的编码方式有：

- ASCII，采用 7 位字符串对每个字符进行编码。采用 ASCII 编码的数据不能包含扩展 Unicode 字符。
- Full Unicode (或 UTF-16)，它采用 16 位字符串表示每个字符。
- UTF-7 Unicode，使用 7 位表示普通 ASCII 字符，多个 7 位对表示扩展字符。通常不使用这种编码。
- UTF-8 Unicode，它使用 8 位表示普通 ASCII 字符，多个 8 位对表示扩展字符。推荐使用这种标准——这种紧凑格式支持扩展 Unicode 字符。.NET 类采用它作为默认标准。不过，如果文本主要由 ASCII 字符组成，使用它的效率最高。

对于用两个字节的数据表示一个字符的多字节 Unicode 格式，必须选择写两个字节的顺序。默认的 Unicode 编码使用 Little-Endian 字节顺序，单独的 Big-Endian Unicode 编码则采用其他方式对字节排序。

.NET 的 System.Text 命名空间中为每种编码提供一个类。所有编码类都提供 GetBytes()方法，用于将字符串转换成字节数组，GetString()方法则执行相反的操作。

```
// Convert a string into a byte array using UTF-8 encoding.
string text = "sample text";

System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
bytes[] data = enc.GetBytes(text);
```

通常执行加密操作时使用编写器和读取器就可在稍高的水平上工作，如 System.IO 命名空间中的 BinaryWriter 和 BinaryReader。它们将普通数据类型无缝地转换成二进制表示形式(默认情况下字符串使用 UTF-8 编码)。对多块数据进行写操作时，使用这些类则要方便一些，它们支持十进制数据类型的二进制编码(它不是通过 BitConverter 提供)。下面的例子中有应用该技术的示例。

2.3.5　加密和解密流

要执行简单的加密任务，只需要将前面讨论过的那些问题综合到一起，其中包括转换、流和加密算法。首先，考虑将字符串转换成二进制数据、再将它加密成简单的内存流，然后再解

密其代码。下面将详细剖析这段代码。

首先，创建加密对象和存储加密数据的 MemoryStream：

```
// EncryptionTest.cs

using System;
using System.IO;
using System.Security.Cryptography;

class EncryptionTest
{
  static void Main(string[] args)
  {
    // Create an instance of RijndaelManaged.
    Rijndael crypt = Rijndael.Create();

    // Create an in-memory stream.
    MemoryStream s = new MemoryStream();
```

接下来，必须生成 CryptoStream，它包装 MemoryStream 并执行加密操作。CryptoStream 只在只写模式下工作。

```
    // Create a cryptographic stream for encryption.
    CryptoStream csWrite = new CryptoStream(s,
crypt.CreateEncryptor(), CryptoStreamMode.Write);
```

说明：

CreateEncryptor()或 CreateDecryptor()创建的对象实现 ICryptoTransform 接口，并且是完全独立的。它们有加密或解密数据所需的所有信息。也就是说，如果修改原始算法实现类(例如更改密钥)，那么这些更改就不会对这些对象造成影响。

下一步，使用 StreamWriter 将文本编码成字节，并写到 CryptoStream。

```
    // Write a value to the stream (which will then be encrypted).
    StreamWriter w = new StreamWriter(csWrite);
    w.WriteLine("Here is a string with some secret data!");

    // Make sure the CryptoStream has everything.
    w.Flush();

    // Finish encoding the last block.
    csWrite.FlushFinalBlock();
```

这时采用加密格式将信息存储到 MemoryStream 中。作为演示，可以检索该数据，并转换成字符串并显示它。这种数据没有与真正的字符串相对应——与此相反，根据每个字节的值，它会包括扩展和不可打印的字符。

```
            // Now move the information out of the stream,
            // and into an array of bytes.
            Byte[] bytes = new Byte[s.Length];
            s.Position = 0;
            s.Read(bytes, 0, (int)s.Length);

            // Display this encoded information.
            System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
            Console.Write("Encrypted data: ");
            Console.WriteLine(enc.GetString(bytes));
```

最后，可以使用另一个 CryptoStream 解密该信息，这在读模式中。由 StreamReader 包装 CryptoStream，StreamReader 将解密后的二进制数据转换成字符串。

```
            // Now decipher the information in the memory stream.
            s.Position = 0;
            CryptoStream csRead = new CryptoStream(s,
              crypt.CreateDecryptor(), CryptoStreamMode.Read);

            StreamReader r = new StreamReader(csRead);
            string deciphered = r.ReadToEnd();

            // Display the decoded information.
            Console.Write("Decrypted data: ");
            Console.WriteLine(deciphered);

            Console.ReadLine();
        }
    }
```

这个简单的控制台应用程序的输出与下面的输出相似：

```
Encrypted data: ?♂P-EdK►↕∟g?←5AyOGx▼←◄v^wx,↓JXV§
Decrypted data: Here is a string with some secret data!
```

当然，每次运行程序输出都不相同，在运行时会动态生成密钥和初始化向量。不过，您可以添加代码手工将 Key 和 IV 属性设置为固定值。

请注意，在您自己的代码中可以使用下列几种最佳方法：

- CryptoStream 的 Write()方法只接受字节。为了避免必须手工将数据转换成字节数组以及编码问题，使用 StreamWriter 类即可。这个类会使用 UTF8 编码自动地对字符串进行编码，除非另有说明。
- 一旦将信息写到 CryptoStream，就在 StreamWriter()方法中调用 Flush()，以确保所有缓冲信息都已写到 CryptoStream(也可以将 StreamWriter.AutoFlush 属性设置为 True)。
- 在输出编写器中的内容后，调用 CryptoStream 的 FlushFinalBlock()以确保数据的最后一块已填满并编码。

- 如果需要重用流，则不关闭它。可以使用底层数据流的 Position 属性移到另一个位置，但请记住不能使用 CryptoStream 的 Position 属性。

如果调用 Flush()或 FlushFinalBlock()失败，在某些情况下会丢失信息，这主要取决于算法的数据块大小、要加密的数据量和要使用的缓存技术。

2.3.6 将数据加密到文件

为了创建更为实用的示例，则以下面的示例为例加以说明，该示例加密数据并将数据存储到文件。应用程序由一个 Windows 窗体组成(simpleFileReader.cs)，它有两个文本框和两个按钮，如图 2-5 所示。

图 2-5

将 RijndaelManaged 加密类的实例存储为窗体成员变量。这是必需的，因为加密和解密数据都要使用同一个实例，除非要采用其他步骤保存 Key 和 IV。

```
Rijndael crypt = Rijndael.Create();
```

单击 Write 按钮，就会将文本框中的数据存储到文件中，同时在写模式中使用 CryptoStream 加密数据。

```
private void cmdWriteFile_Click(object sender, System.EventArgs e)
{
  // Create the encryption transform for this algorithm.
  ICryptoTransform transform = crypt.CreateEncryptor();

  // Open a file for writing to.
  FileStream fs = new FileStream("c:\\testfile.bin",
    FileMode.Create);

  // Create a cryptographic stream.
  CryptoStream cs = new CryptoStream(fs, transform,
    CryptoStreamMode.Write);
```

```
    // Create a text writer.
    StreamWriter w = new StreamWriter(cs);
    w.Write(txt.Text);

    w.Flush();
    cs.FlushFinalBlock();

    w.Close();
}
```

单击 Read 按钮，就会检索文本框中的数据，并在读模式下使用 CryptoStream 对它进行解密。在消息框中显示检索到的数据，然后输入到文本框中。在这种情况下，将保持这两种操作有相同的 crypt 对象，因此加密和解密都使用相同的密钥。

```
private void cmdReadFile_Click(object sender, System.EventArgs e)
{
    // Create the encryption transform for this algorithm.
    ICryptoTransform transform = crypt.CreateDecryptor();

    // Open a file for reading from.
    FileStream fs = new FileStream("c:\\testfile.bin", FileMode.Open);

    // Create a cryptographic stream.
    CryptoStream cs = new CryptoStream(fs, transform,
      CryptoStreamMode.Read);

    // Create a text reader.
    StreamReader r = new StreamReader(cs);
    string text = r.ReadToEnd();
    r.Close();

    MessageBox.Show("Retrieved: " + text);
    txt.Text = text;
}
```

2.3.7　生成密钥和使用 Salt

此前假定要使用密钥和实例化类时默认创建的初始化向量。当然，情况不总是如此。有时会使用另一个值，如用户口令。不幸的是，由于下列原因而很少使用这种信息作为密钥：

- 信息通常太短。
- 它使用有限的字符集(字母和数字)而不是可能完整的字节值。
- 它经常使用普通字或名称，这样就容易遭受字典式攻击(攻击者通过尝试字典中的每个字来解密消息)。

其中一个解决方案就是使用 PasswordDeriveBytes 类，在加密服务提供商的帮助下用口令生

成进行很好加密的密钥。当然，为了确保它确实是强密钥，最好也使用 salt 值，它是作用与初始化向量相似的随机字节值。如果使用非零 salt 值，创建用于解密数据的密钥时也必须提供该值。不需要对它保密，确保相同的口令生成不同密钥。与 IV 相似，它可以与加密数据一起传递。

说明：

采用这种格式的派生口令仍然易受字典式攻击，由于相同的口令和 salt 总是可以得到相同的字节序列(假定攻击者访问相同的 PasswordDeriveBytes 代码) —— 不过攻击者必须生成每个可能口令的派生字节，而不是使用预先计算好的字典。在任何情况下，从加密技术的角度看派生密钥都不容易破解，使用该密钥生成的密文则不容易受到频繁的分析攻击以及类似的攻击。

下面的示例使用 PasswordDeriveBytes 类和 RNGCryptoServiceProvider 类生成随机 salt 值。附录 B 将介绍与 RNGCryptoServiceProvider 有关的更多知识。现在了解它使用该技术生成随机数是很重要的，它生成几乎不可能预见的数字序列(与 System.Random 类不同)。

```
Rijndael crypt = Rijndael.Create();

byte[] salt = new byte[8];
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
rng.GetBytes(salt);

PasswordDeriveBytes pdb = new PasswordDeriveBytes(
  "OpenSesame", salt);

byte[] key = pdb.GetBytes(16);
crypt.Key = key;
crypt.IV = new byte[16];
```

我们调用 GetBytes()，根据提供的口令和 salt 中派生出必要长度的密钥。

2.3.8 Base64 转换

CryptoStream 可以包装任何由 Stream 派生的对象，其中包括另一个 CryptoStream。采用这种方式就可以对同一组数据进行多次加密(它未必会提高加密强度，甚至有可能降低其强度)。链接 CryptoStream 实例更为有用的原因就是为了使用加密技术命名空间中的 ToBase64Transform 和 FromBase64Transform。

默认情况下，在加密数据时，得到的字节数组就使用任何有效的值(0~256)。其结果就是，不能将加密数据安全地嵌入到 HTML 页面或 XML 文档(因为它可能包含特殊的字符)，或与 Web 服务通信时将它作为 SOAP 消息的组成部分。为了解决这个问题，可以进行 Base64 变换。

Base64 编码中，每个 3 字节的序列——24 位——分成 6 位的 4 块。每个块都是 64 个值中的一个，每个值都与{A~Z, a~z, 0~9, +, /}中的可打印字符相对应。然后，将这些字符转化成 4 字节序列。每个 Base64 编码的字节都是 64 个可能值之一，所有值都与可以安全显示、通过电子邮件发送的或包括在 XML 数据中的字符相对应。它的不足就是 Base64 编码的数据大小增加

了 33%，由于 3 个源字节就成了 4 个输出字节。

在加密的情况下，可以在对消息加密之后应用 Base64 编码，因此可以删除与特殊字符相对应的任何字节。下面的示例使用简单文件加密应用程序。在这种情况下，必须添加两个 CryptoStream，但顺序相反(因为我们将它们添加到了最终的输出 FileStream 中)，如图 2-6 所示。

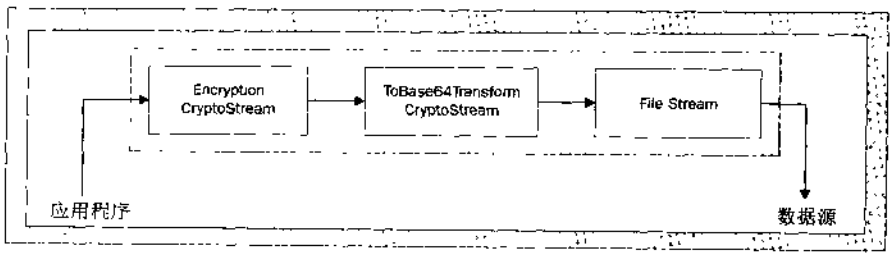

图 2-6

其代码如下所示：

```
private void cmdWriteFile_Click(object sender, System.EventArgs e)
{
  // Open a file for writing to.
  FileStream fs = new FileStream("c:\\testfile.txt",
FileMode.Create);

  // Create a stream for converting to Base64 encoding.
  ICryptoTransform transformEncode = new ToBase64Transform();
  CryptoStream csEncode = new CryptoStream(fs, transformEncode,
    CryptoStreamMode.Write);

  // Create a stream for encryption.
  ICryptoTransform transformEncrypt = crypt.CreateEncryptor();
  CryptoStream csEncrypt = new CryptoStream(csEncode,
    transformEncrypt, CryptoStreamMode.Write);

  // Create a text writer.
  StreamWriter w = new StreamWriter(csEncrypt);
  w.Write(txt.Text);

  w.Flush();
  csEncrypt.FlushFinalBlock();

  w.Close();
}
```

结果文档如下所示：

```
IuCP6bMgihL2B21i21AWT1U2+ABmJtB7q+3SYQovHNv6WVxomEo92w4XbS2u3bxM=
```

获取数据时，就必须将信息转换回其原有的表示形式，然后再解密。

```
private void cmdReadFile_Click(object sender, System.EventArgs e)
{
  // Open a file for reading from.
  FileStream fs = new FileStream("c:\\testfile.txt", FileMode.Open);

  // Create a stream for converting from Base64 encoding.
  ICryptoTransform transformDecode = new FromBase64Transform();
  CryptoStream csDecode = new CryptoStream(fs, transformDecode,
    CryptoStreamMode.Read);

  // Create a cryptographic stream.
  ICryptoTransform transformDecrypt = crypt.CreateDecryptor();
  CryptoStream csDecrypt = new CryptoStream(csDecode,
    transformDecrypt, CryptoStreamMode.Read);

  // Create a text reader.
  StreamReader r = new StreamReader(csDecrypt);
  string text = r.ReadToEnd();
  r.Close();

  MessageBox.Show("Retrieved: " + text);
  txt.Text = text;
}
```

2.4　非对称加密

非对称加密使用包括私钥和公钥的密钥对，第 1 章已讨论过。私钥受到很好的保护，而所有人都可以获取公钥。这些密钥在数学上是相联系的，这样使用一个密钥加密的数据都可以使用其他密钥解密。因此，可使用接收者的公钥加密消息，确保只有发送者才能阅读消息。事实上，一旦加密了消息，即使是发送者也不能解密它，因为发送者没有接收者的私钥。

现在可以使用私钥或公钥将数据译成密码，然后使用另一个密钥译解密码，如果使用私钥将数据译成密码，它并不是真正对数据进行了加密处理，只是隐藏而已，任何人使用公钥都可以解译它，按定义是任何人都可以进行解译。另一方面，使用公钥将数据译成密码确实显示隐藏的数据，因此被称作“加密”。通常不将使用私钥将数据译成密码的操作称为“加密”，也不将使用公钥解译密码称为“解密”——这些操作对数字签名和认证都很重要，在这些过程中称之为“签名”和“检验”。这种消息完整性和检验的思想将在第 3 章中讲述。

本节谈论“加密”时，所指的是使用公钥将数据译成密码，“解密”是指使用私钥解译密码。

非对称密钥加密的特点如下：

- 使用对称密钥加密时，加密强度与密钥长度相对应。不过，不能直接比较对称和非对称加密中密钥的大小。
- 非对称加密的数学含义很复杂。(要使用的非对称算法由这样的情况决定：两个素数相乘得到一个新数的计算比较容易，但是对于大数，要分解其所得的结果来找出最初的素数对却很困难。)由于这个原因，非对称密钥加密的速度大概要比对称加密慢 1000 倍。

使用非对称加密的特殊问题就是它的执行速度比较慢。处理大量数据非对称密钥加密就不太实用(例如加密的图像)。通常结合非对称加密，通过协商确定随机秘密值(称为会话密钥)，然后就可以将其用于速度更快的对称加密操作。第 5 章将描述该过程。这样就充分利用了两者的优点，由于使用较慢却安全的非对称处理，在通信开始就对使用的密钥达成一致，然后使用速度较快的基于数据块的对称密码算法对大部分会话进行加密处理。

2.4.1 非对称算法

表 2-3 列出了.NET Framework 提供的非对称算法。默认算法是 RSA，DSA 只适用于创建和确认数字签名，而不适用于加密数据。

表 2-3

算　法	抽象算法类	默认实现类	有效密钥大小(位)	默认密钥大小(位)
RSA	RSA	RSACryptoServiceProvider	364~16384 (以 8 位递增)	1024
DSA	DSA	DSACryptoServiceProvider	364 ~512 (以 64 位递增)	1024

RSA 和 DSA 由 CryptoAPI 实现，因此在 Windows 2000 或早期版本上使用较大的密钥需要高加密包。下面将注意到非对称加密的密钥要大得多。这是因为非对称加密中密钥的作用与对称密码中所起的作用非常不同。对称密码中等于密钥大小的任何数字都可以作为密钥。非对称加密技术中，只有具有某种数学特性的数字才能作为密钥，如果这些数字非常巨大，攻击者就很难破解。因此需要许多位用于存储较大数。不过，攻击者不必试用每个数字进行蛮力攻击，他们只试用那些可能的密钥。从密码术的角度看，1024 位 RSA 密钥(.NET 中默认的密钥大小)大致相当于 75 位的对称密钥。如果可能的话，可以考虑使用较大的密钥，不过也要仔细考虑性能问题。较大的密钥生成和使用所占用的时间要长得多。

2.4.2 AsymmetricAlgorithm 类

表 2-3 中的类由 AsymmetricAlgorithm 派生而来，这样它就支持几个常见的属性，其中包括 KeySize 和 LegalKeySizes。另外，可以使用 Create()这样的静态方法生成默认的非对称算法实现类和用于导出和检索密钥信息的 FromXmlString()和 ToXmlString()。不过，一般情况下不能通过 AsymmetricAlgorithm 类的方法进行加密或签名检验。由于并不是所有算法都支持该功能，所以是在抽象算法类中而不是 AsymmetricAlgorithm 类中实现。不过，.NET 没有提供任何包含该功能的接口或基类，因此使用非对称算法编程要求编写的普通代码比使用对称算法编写的少。

说明：

.NET Framework 只提供一个可以用于加密和解密的 AsymmetricAlgorithm 类：RSA 及其实现 RSACryptoServiceProvider。因此，下面的示例重点说明这些类成员。

另一点不同就是非对称算法不支持 CryptoStream 模型，前面几节讨论了该模型。造成这种差别的部分原因是非对称算法不是设计为可以“立刻”由其他 Stream 实例使用，由于它们的运行速度太慢。这样就需要自己进行处理：将字符串转换成字节数组，编码或解码之前再将字节数组分成大小合适的数据块。这就使得执行非对称加密比对称加密更费事。

2.4.3 加密数据

RSA 基类定义两种方法 EncryptValue()和 DecryptValue()，所有实现类都必须支持它们。不过，很少直接调用这些方法，因为它们采用不进行填充的原始计算。而是使用 RSACryptoServiceProvider 实现类的 Encrypt()和 Decrypt()方法。这些方法加密或解密字节数组中的数据。

下面的代码使用 UTF-8 编码将数据字符串转换成字节数组，再加密数据，然后将它写到文件。请注意，由于在实现类中定义 Encrypt()和 Decrypt()方法，所以最容易的方法就是在加密代码的开始部分直接创建 RSACryptoServiceProvider 实现类的实例。

```
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider();

// Convert text to a byte array.
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
byte[] bytes = enc.GetBytes("Secret string");

// Encrypt data.
bytes = crypt.Encrypt(bytes, false);

// Write it to a file.
FileStream fs = new FileStream("c:\\testfile.bin",
  FileMode.Create);
fs.Write(bytes, 0, bytes.Length);

fs.Flush();
fs.Close();
```

传递给 Encrypt()方法的 false 参数表明不使用 OAEP(Optimal Asymmetric Encryption Padding)填充。只有 Windows XP 及以上的操作系统才支持 OAEP 填充。

检索数据与它一样简单，假定使用相同的 RSACryptoServiceProvider 实例：

```
// Read data from a file.
FileStream fs = new FileStream("c:\\testfile.bin",
  FileMode.Open);
byte[] bytes = new byte[fs.Length];
```

```
fs.Read(bytes, 0, bytes.Length);
fs.Close();

// Decrypt data.
bytes = crypt.Decrypt(bytes, false);

// Convert byte array to text.
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
Console.WriteLine(enc.GetString(bytes));
```

如果处理字符串数据，确定使用相同的编码对象对数据进行编码和解码极为重要。如果使用 UTF8Encoding 类对数据进行编码而使用 ASCIIEncoding 进行解码，就不能成功地获得原始信息。

注意：

非对称加密使用公钥和私钥组成的密钥对。如果使用 Encrypt()方法，就使用公钥对数据进行加密处理，使用 Decrypt()方法，则需要使用私钥。

例如，前面演示的加密文件的编写器/读取器测试实用程序。要将它配合非对称加密使用，则需要在加密之前将数据分成块。允许使用的块大小由加密级别决定——使用高加密软件包，每次就可以加密 16 字节，否则只能加密 5 字节。这段代码首先检索密钥大小。如果密钥大小为 1024 位，就可以使用高加密并采用长度为 16 字节的数据块。

编码代码的文件如下所示。最重要的行已突出显示。

```
private void cmdWriteFile_Click(object sender, System.EventArgs e)
{
  // Convert text to a byte array.
  System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
  byte[] bytes = enc.GetBytes(txt.Text);

  // Determine the optimum block size for encryption.
  int blockSize = 0;
  if (crypt.KeySize == 1024)
  {
    // High encryption capabilities are in place.
    blockSize = 16;
  }
  else
  {
    // High encryption capabilities are not in place.
    blockSize = 5;
  }

  // Create the file.
```

```
FileStream fs = new FileStream("c:\\testfile.bin",
  FileMode.Create);

// Move through the data one block at a time.
byte[] rawBlock, encryptedBlock;
for (int i = 0; i < bytes.Length; i += blockSize)
{
  if ((bytes.Length - i) > blockSize)
  {
    rawBlock = new byte[blockSize];
  }
  else
  {
    rawBlock = new byte[bytes.Length - i];
  }

  // Copy a block of data.
  Buffer.BlockCopy(bytes, i, rawBlock, 0, rawBlock.Length);

  // Encrypt the block of data.
  encryptedBlock = crypt.Encrypt(rawBlock, false);

  // Write the block of data.
  fs.Write(encryptedBlock, 0, encryptedBlock.Length);
}

// Clean up.
fs.Flush();
fs.Close();
}
```

Buffer.BlockCopy()静态方法用于将源数组的每个块复制到新数组 rawBlock 中。如果遇到最后一个数据块，rawBlock 数组就会根据块长度相应地缩短，这是因为 Encrypt()方法会自动执行必需的填充操作。

如果读取文件，就必须注意每次只能解密一个数据块。在这种情况下，数据块的大小由所采用算法的密钥大小决定。KeySize 属性提供这种信息(以位为单位)，将该数除以 8 就能得到字节数。例如，这里 KeySize 就是 1024 位(或 128 字节)。因此，必须以 128 字节长的数据块为单位解密数据。

将数据分成块进行解密，就需要一种方法重新组织它，以便可以将它转换回其最初的字符串格式。处理这种任务最为容易的方法就是使用另一个流。该示例中代码将解密后的数据写到内存流中，然后在将其转换成字符串时读取完整的信息。

```
private void cmdReadFile_Click(object sender, System.EventArgs e)
{
```

```
// Read encrypted data from file.
FileStream fs = new FileStream("c:\\testfile.bin", FileMode.Open);
byte[] bytes = new byte[fs.Length];
fs.Read(bytes, 0, (int)fs.Length);
fs.Close();

// Create the memory stream where the decrypted data
// will be stored.
MemoryStream ms = new MemoryStream();

// Determine the block size for decrypting.
int keySize = crypt.KeySize / 8;

// Move through the data one block at a time.
byte[] decryptedBlock, rawBlock;
for (int i = 0; i < bytes.Length; i += keySize)
{
   if ((bytes.Length - i) > keySize)
   {
      rawBlock = new byte[keySize];
   }
   else
   {
      rawBlock = new byte[bytes.Length - i];
   }

   // Copy a block of data.
   Buffer.BlockCopy(bytes, i, rawBlock, 0, rawBlock.Length);

   // Decrypt a block of data.
   decryptedBlock = crypt.Decrypt(rawBlock, false);

   // Write the decrypted data to the in-memory stream.
   ms.Write(decryptedBlock, 0, decryptedBlock.Length);
}

// Read the in-memory information into a byte array.
ms.Position = 0;
byte[] decoded = new byte[ms.Length];
ms.Read(decoded, 0, (int)ms.Length);

// Convert byte array to text.
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
string text = enc.GetString(decoded, 0, decoded.Length);
```

```
    // Display the result.
    MessageBox.Show("Retrieved: " + text);
    txt.Text = text;
}
```

2.4.4　以 XML 格式导入和导出密钥

与对称算法类不同，非对称算法不提供可以用于检索密钥信息的任何公钥属性。不过，如果需要存储密钥供以后使用(最理想的就是在某种安全存储区中或密钥管理系统中)，就可以将密钥信息作为 XML 文档进行检索。最好的是这种 XML 文档提供了足够的信息，供以后重新构建相同的密钥。

下面的代码使用 ToXmlString()方法输出密钥对的完整密钥信息。

```
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider();

// Display XML source for this key.
Console.WriteLine(crypt.ToXmlString(true));
```

所生成的信息如下所示(简写形式)。每个值都是二进制数的 Base64 编码。采用一种方式生成数字就可以获得公钥。通过不同的方式组合这些数字就可以获得私钥。值得注意的是该格式与在第 3 章中介绍的 XML 签名标准(以及 XML 加密和 XKMS 标准)中所使用的格式相匹配。

```
<RSAKeyValue>
  <Modulus>xTnyhC93bO...</Modulus>
  <Exponent>AQAB</Exponent>
  <P>7yoEyMvp5B63eud3...</P>
  <Q>0xwqy2Leud3GDUQ...</Q>
  <DP>dyzX/3rk421dbh...</DP>
  <DQ>MUdH7gX8O6K7Em...</DQ>
  <InverseQ>f+kt/KTGt...</InverseQ>
  <D>BU/DNUtELSgvqJkb...</D>
</RSAKeyValue>
```

使用这种 XML 文档就可以创建相同的密钥：

```
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider();

// Get XML source for this key.
string key = crypt.ToXmlString(true);

// Destroy the current key.
crypt.Clear();

// Reload the key.
crypt = new RSACryptoServiceProvider();
crypt.FromXmlString(key);
```

请记住，如果导出密钥对的内容，就需要确保在任何时候 XML 文档都是保密的。如果第三方获得了该文档，就会知道您的私钥，也就可以容易地攻破您的加密系统。在极端环境下，即使将密钥存储到字符串这样单纯的任务也容易遭到攻击，心怀恶意的用户可以访问您的服务器，引发错误并进行内存转储操作。

说明：

不可能完全防止出现这样的问题，不过将密钥信息存储到字符串变量中就可以解决该问题，尤其是 CLR 可以自由重新分配内存，在您认为已清除或重写很久后也可以将这些敏感信读取出来。

上面的示例中，生成了 XML 信息并且 includePrivateParameters 参数值为 True。也就是说完整地导出私钥和公钥的信息。不过，如果提供 false 值，就会收到较短的文档，如下所示：

```
<RSAKeyValue>
  <Modulus>xTnyhC93bO...</Modulus>
  <Exponent>AQAB</Exponent>
</RSAKeyValue>
```

该文档的信息只够生成密钥的公钥部分，因此不需要确保它的安全。使用该信息，另一个用户就可能创建 RSACryptoServiceProvider 对象，并使用它加密数据。由于它们没有私钥，所以只有您才能解密该信息。它们不能使用自己的 RSACryptoServiceProvider 解密数据，因为它没有私钥。

2.4.5 使用参数导入和输出密钥

有另外一种方法可以管理密钥信息。即使用 ExportParameters()和 ImportParameters()方法将信息导出到特定的参数对象。在使用 RSA 算法的情况下，该参数对象就是 RSAParameters 类，而在使用 DSA 的情况下，它就是 DSAParameters 类。这些对象展示的字段与以前的文档中所示的 XML 元素相对应(如 Modulus、Exponent 等)。每个提供者都可以定义自己的具体参数对象，它们不实现公共接口或从公共基类派生类。

下面的示例将完整的密钥信息存在于 RSAParameters 实例中，用于重新创建密钥。

```
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider();

// Get the key information.
RSAParameters param = crypt.ExportParameters(true);

// Destroy the current key.
crypt.Clear();

// Reload the key.
crypt = new RSACryptoServiceProvider();
crypt.ImportParameters(param);
```

下面的示例只使用公钥信息：

```
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider();

// Get the key information.
RSAParameters param = crypt.ExportParameters(false);

// Destroy the current key.
crypt.Clear();

// Reload the key.
crypt = new RSACryptoServiceProvider();
crypt.ImportParameters(param);
```

说明：

在 CryptoAPI 库中，可以创建密钥容器，这些容器不允许导出密钥信息。如果要在.NET 中使用该技术，就需要对 CryptoAPI 库进行非托管的 P/Invoke 调用。

如果要从安装的证书中获得参数信息，该怎么办呢？不过，.NET Framework 目前还没有这项功能。System.Security.X509Certificates 命名空间包括创建证书、从磁盘载入它们所必需的功能，但不能访问计算机专用的存储区。不过，Microsoft.Web.Services.SecurityX509 命名空间中已由 WSE(Web Services Enhancements for .NET)增添了这项功能。

使用 WSE 就可以使用下列代码从本地证书存储中获得信息，并使用它创建 RSACryptoServiceProvider。只有在证书有个人证书存储(Personal Certificate Store)中的密钥时，它才有效。

```
// Open the private certificate store of the current user.
X509CertificateStore store = X509CertificateStore.CurrentUserStore(
  X509CertificateStore.MyStore);
store.OpenRead();

// Read first certificates.
X509Certificate c = (X509Certificate)store.Certificates[0];

RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
rsa.ImportParameters(c1.Key.ExportParameters(true));
```

2.4.6　使用容器存储 CSP 密钥

CryptoAPI 将密钥存储到密钥容器(key container)中。密钥容器是计算机特有的密钥数据库的组成部分，密钥数据库用于存储密钥对。密钥对归特定用户所有，或者是共享计算存储的一部分。如果使用 RSACryptoServiceProvider 这样的类，就不用担心密钥容器，因为程序会自动请求并销毁它。不过，在某些情况下，您可能想创建特定的密钥容器。

使用 CspParameter 类，就可以创建指定的 CryptoAPI 存储容器。采用这种方式可以生成随

机密钥，然后访问相同的密钥容器就可以重用它。

```
// Set a name for the key container used to store the RSA key pair.
CspParameters cp = new CspParameters();
cp.KeyContainerName = "MyKeyContainerName";

// Create the key in this container.
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider(cp);

// Display the key pair that is in use.
Console.WriteLine(crypt.ToXmlString(true));
```

如果多次运行该应用程序，就会发现下次使用相同的容器创建 RSACryptoServiceProvider 时，就可以获得第一次创建的密钥。这对于长期使用密钥很有用，不过同一台计算机上的其他应用程序也可以潜在地使用该密钥。这样其安全风险就取决于环境。

为了确保密钥没有保存在容器中，可以添加下面的代码语句：

```
crypt.PersistKeyInCsp = false;
```

现在如果调用算法实现类的 Clear()或 Dispose()方法就可以将密钥信息随密钥容器一起销毁。

请记住，这项技术不适用于在托管代码(如 Rijndael)中实现的算法。CryptoAPI 系统中每个 CSP 都有密钥数据库，密钥数据库存储长期使用的加密密钥，所以它可以运行。密钥数据库被分成密钥容器，这些密钥容器包含特定客户的密钥对。

CryptoAPI 使用文件系统来存储密钥容器。这些容器在安装 Windows 的驱动器上的 Documents 和 settings 目录中。交互式用户的密钥存储在<username>\Application Data\Microsoft\Crypto\RSA\<user SID>这样的目录中，非交互式用户(如 ASP.NET 应用程序)的密钥存储在 All Users\Application Data\Microsoft\Crypto\RSA\Machinekeys 目录中。如果使用的是 Windows NT，就将它们存储在注册表中，而不是文件系统中。最后，应用程序的用户就可以确定应用程序中所使用的密钥是否有适当的权限。这是编程中的基本事实—— 程序代表用户，不能使用用户不知如何确定的秘密数据。

2.4.7 CSP 和 ASP.NET

如果要将到目前为止所有介绍的代码移到 Web 服务或 ASP.NET Web 页面中，就会遇到一些小麻烦。问题就是在 ASP.NET 内部使用该应用程序时，就不能被指定为交互用户。没有当前用户的配置文件，因此 Windows 操作系统不能检索默认的密钥容器。如果您正在使用依赖于 CryptoAPI 的算法并动态地生成新密钥，就会出现这样的问题。

为了克服该问题，就需要使用存储在本地计算机密钥存储中的信息。为了通知这种变化，就使用 CspParameters 对象，并修改其 Flags 属性。Flags 属性可以接受两个值中的一个：UseDefaultKeyContainer (默认值)或 UseMachineKeyStore。

```
CspParameters cp = new CspParameters();
cp.Flags = CspProviderFlags.UseMachineKeyStore;
```

```
RSACryptoServiceProvider crypt = New RSACryptoServiceProvider(cp);
```

下面就是私有帮助函数示例，可以将其添加到 ASP.NET Web 页面或 Web 服务。它动态地创建密钥，然后将其存储到应用程序状态中。它也根据需要设置 CspParameters 标记。

```
private RSACryptoServiceProvider GetKeyFromState()
{
  RSACryptoServiceProvider crypt;

  // Check if the key has been created yet.
  // This ensures that the key is only created once.
  if (Application("Key") == null)
  {
    // Create a key for RSA encryption.
    CspParameters cp = new CspParameters();
    cp.Flags = CspProviderFlags.UseMachineKeyStore;
    crypt = new RSACryptoServiceProvider(cp);

    // Store the key in the server memory.
    Application("Key") = crypt;
  }
  else
  {
    crypt = (RSACryptoServiceProvider)Application("Key");
  }
  return crypt;
}
```

请记住，CspParameters 对象只用于将其他信息传递到 CryptoAPI。不应该将它与算法特有的参数对象搞混了，算法特有的参数对象用于导出密钥信息。

2.5 加密强度

经常遇到的问题就是“加密强度如何”？通常很难对此作回答。使用加密算法的最主要弱点就是错误而不是算法本身存在的局限。例如，没有以安全方式存储密钥(或终端用户透露太多信息)、所有通信都重用同一个密钥，以及初始化向量都是比较明显的经常被忽略的安全漏洞。

要在没有这些错误的帮助下解译加密消息，黑客一般都会求助于下列两种方法中的一种。其中一种攻击方法就是找出加密数据中重复的模式。如前所述，可以使用 CBC 链接和不断变化的非零初始化向量对此加以防范。另一种常见的攻击形式就是通过试用每个可能的值进行蛮力攻击。防范这种攻击的惟一方法就是尽可能使用密钥最大(如 256 或 192 位而不是 128 位)的强度最大的算法(例如，TripleDES 或 Rijndael，而不是 DES)。不过，专家对测试每种可能的密钥和初始化向量组合要花费的时间没有定论。因为如果使用专用硬件和一台费用高昂(六位数)的计算机，破解 DES 加密的消息可能只需要几个小时。不过，如果您要将敏感信息发送给很可能

必须应用这些专用资源的人，那因使用更强的算法所带来的性能损耗还是可以容许的。

2.6 小结

本章学习.NET 密码术模型如何使用抽象类、转换和流。另外，还学习了如何使用对称和非对称算法进行简单的加密和解密任务。

从本章学到的密钥方面的一些知识如下：

- 密码术中广泛使用密钥长度为 128 位及以上的对称算法。Rijndael 算法代表加密技术领域中加密技术的当前状态。
- 对称算法处理大小固定的数据块中的数据。
- 密码块链接(CBC)提供了最安全的方式将对称算法应用于一系列数据块。
- 要添加 CBC 的安全性，就必须提供初始化向量(IV)。
- .NET CryptoStream 提供简单接口，通过对称加密处理器读取和写二进制数据，并提供缓冲，因此不需要担心数据块大小和是否填满最后一个数据块。
- 用字节数组形式对二进制数据进行加密运算。可以将数据转换成字节数组并使用 BitConverter 和 System.Text.Encoding 类将其转换回来。
- 加密后的二进制数据可以使用 Base64 编码表示为可打印文本字符。
- 非对称密码算法使用的密钥比对称密码的密钥要大得多，因为它们所依赖的数学基础不同。默认的算法是 RSA，默认的密钥大小为 1024 位。
- .NET 非对称密码算法不使用 CryptoStream。我们必须将自己的数据划分成块，然后根据块重新构建它。对于加密而言，如果支持高加密，就必须采用 5 个字节或 16 个字节长度的数据块。对于解密而言，则必须使用与密钥长度相同的块(1024 位密钥则采用 128 个字节的块长度)。

第3章　数据的完整性——散列码和签名

数据加密技术可以保护信息，但它不能阻止恶意用户对信息的篡改。例如，在用户通过Internet发送消息时就很容易受到恶意用户的攻击。如果使用加密技术，那么该消息的接收者相对来说可以确信没有其他人读取过该消息，但不能确保它就是发送者最初发送的那条消息。它可能已被修改过。通常情况下，当对消息解密时，这种改变会导致明显的数据错乱。有时后果会更加严重，尤其是在您期望得到的原二进制数据或者整个新消息都被替代时。要想克服这种情况，我们需要一种简单有效的方法对消息的完整性(有时是可靠性)进行验证。

这在密码术中是由散列码实现的。从本质上看，散列码是一种强加密的校验和。这里所说的强加密表明在两个文档中几乎不可能发现相同的散列码。在进行消息交换时，发送者计算一个散列，并将其添加到消息中。接收者根据接收的文档重新计算散列码，并验证它是否与发送者的散列码相符。当然，散列码还具有多种不使用密码术的应用，像惟一标识对象——但是在这种情况下，不需要强加密的散列算法。

运用散列算法，过程会变得更加复杂。您可能需要混合使用散列和一种单独的加密形式来确保是否有第三方截取消息，它们不能创建一个新的消息并生成与之相符的散列码。但是，基本原则仍然没有改变：散列码允许用户验证数据是否被修改，而加密技术可以使数据对任何人保密。

当然，保护通信只是使用散列码的一个示例。散列码还可以用在以下几种任务中：

- 验证存储在磁盘中的数据是否已在应用程序外部被修改。
- 验证可执行应用程序是否由可信的组织创建(利用.NET对公钥签名和强名的支持)。
- 验证可执行程序自部署之后完全没有被修改。
- 验证全部或者部分XML文档未被修改。
- 在不保存原始密码数据的情况下验证密码(用户只需验证散列即可)。
- 为文档或者应用程序文件创建一个短的、惟一的标识符。

本章将介绍如何应用这些散列技术。我们将对基本的散列、密钥散列、数字签名以及XML签名标准进行深入的探讨。

3.1　散列算法

散列算法可以使用任意数量的数据(作为一个大的二进制信息块)，并利用它生成该信息惟一的较小散列码。下面的伪代码可以表示该过程。

```
Hash = HashAlgorithmFunction(Data)
```

所有加密的散列算法必须是：

- 单向的——如您所想，根据散列重新创建文档是完全不可能的，因为散列不包括文档中的所有信息。但是，也同样不能通过检查散列来计算与信息相关的任何事情。
- 抵抗冲突 ——表明不可能通过计算创建出生成相同散列的其他文档。需要散列的文档数比 20 个字节的(散列码的一般长度)散列多。但是，20 个字节的散列数实际上比人们创建的文档数多得多。如果把散列值以正确的方式赋给文档，那么从统计学上看，可以保证这里不会发生任何冲突，因为可能的散列数实在太多了。

这种行为在多数情况下都是很重要的。例如，您不希望客户创建一个已经更改但仍然具有相同散列码的合同。

由于这些方面的原因，散列有时被作为强加密的校验和。利用校验和(类似于经常在文件系统中使用的循环冗余校验(CRC，Cyclical Redundancy Check))，创建出生成相同校验和的其他文档就相对容易许多。就散列码而言，这几乎不可能实现。

3.1.1　.NET 散列类

散列算法类使用的 3 层继承模型与第 2 章介绍的模型相同。.NET Framework 支持的所有散列算法都继承自 HashAlgorithm 抽象类。表 3-1 列出了所有支持的算法(除密钥散列算法之外，它将在本章后一部分内容中介绍)。

表 3-1

算　法	抽象散列算法类	实　现　类	散列长度(位)
MD5	MD5	MD5CryptoServiceProvider	128
SHA-1	SHA1	SHA1CryptoServiceProvider	160
		SHA1Managed	160
SHA-256	SHA256	SHA256Managed	256
SHA-384	SHA384	SHA384Managed	384
SHA-512	SHA512	SHA512Managed	512

注意：

SHA-1 算法是当前在 .NET Framework 中惟一使用的算法，它既有可以托管实现又可以 CryptoAPI 实现。 CryptoAPI 实现 (SHA1CryptoServiceProvider)是系统默认的，但是如果需要的话也可以手工使用托管实现 (SHA1Managed) 。

所有散列算法都可以使用 SHA(安全散列算法)或者 MD5(消息摘要 5)。对于大部分散列算法来说，散列的长度越长，散列码抵抗蛮力攻击的能力就越强(发现有相同散列的文档就更加困难)。一般来说，散列强度利用密钥的长度来映射加密强度，该长度是散列算法长度的一半。换句话说，SHA-256 提供的抵抗蛮力攻击的能力大约和 Rijndael 算法用一个 128 位密钥提供的抵抗能力相同，SHA-384 映射一个 192 位的密钥，而 SHA-512 映射一个 256 位的密钥。

散列值重复的可能性与散列长度有关。如果一个散列有 256 位，它就可能有 2^{256} 种不同的散列码。例如，您也不希望在查看了大约 10^{76} 个文档之后才发现一个重复的。对照一下吧，请

牢记，据估计宇宙中也只有 10^{66} 种原子。

HashAlgorithm 类

HashAlgorithm 抽象类可以定义具体算法实现类支持的成员。KeyedHashAlgorithm 抽象类是从 HashAlgorithm 类中派生的，并且添加了一个 Key 属性，稍后对其进行介绍。

HashAlgorithm 类提供了一个可以生成默认实现类的 Create()方法。由于我们使用的是对称加密技术，所以应该利用该方法并使代码更加普通。例如，下面的代码语句创建了一个默认的 MD5 实现类(即 MD5CryptoServiceProvider)。

```
HashAlgorithm hash = MD5.Create();
```

此外，该代码还为默认的散列算法创建了一个默认实现的实例，即 SHA1CryptoServiceProvider：

```
HashAlgorithm hash = HashAlgorithm.Create();
```

HashAlgorithm 类还定义了一种可以把最后一次生成的散列从内存中删除的 Clear()方法，(这也可以通过 Dispose()方法来间接调用)和一种为给定的数据缓冲创建散列的 ComputeHash()方法。最后，HashAlgorithm 类还可以实现 ICryptoTransform 实例，这就意味着它可以当作 CryptoStream 的一部分来包装其他 Stream 派生对象。

3.1.2 计算散列值

计算散列值和调用 ComputeHash()方法一样简单。ComputeHash()方法被重载为两种非常有用的方法，它允许用户通过指定一个偏移量为字节数组的一部分或者一个流计算散列值。下面的代码是一个简单的创建散列算法实例并显示散列长度的控制台应用程序。那么它将为一个较小的二进制数据(data)数组创建一个散列值并指出最终的散列值结果。

```
// HashTest.cs

using System;
using System.Security.Cryptography;

public class HashTest
  {
    static void Main(string[] args)
    {
    HashAlgorithm hash = HashAlgorithm.Create();

    // Display hash algorithm information.
    Console.Write("Hash size: ");
    Console.WriteLine(hash.HashSize.ToString() + " bits");
    Console.Write("Hash size: ");
    Console.WriteLine((hash.HashSize/8).ToString() + " bytes");
    Console.WriteLine();
```

```
        // Data to be hashed
        byte[] data = {200, 34, 12, 14, 210, 199, 172, 77};
        // Create hash
        byte[] hashBytes = hash.ComputeHash(data);

        // Display the hash.
        Console.Write("Hash: " + BitConverter.ToString(hashBytes));
        Console.Read();
    }
}
```

该程序的输出结果如下：

```
Hash size: 160 bits
Hash size: 20 bytes

Hash: 6F-F3-AD-EC-31-4A-63-C6-80-95-62-B9-C6-16-22-64-F4-03-91-26
```

您会注意到散列长度是固定的。例如，当使用 SHA-1 算法(就像我们在该示例中使用的一样)时，不管源数据多大或者多小，结果生成的散列始终是 20 个字节。

注意：

但是，出于安全方面的原因，建议源数据的长度至少应该等于散列长度。这样就可以阻止由于计算所有可能的短消息的散列码而发起的攻击。

有趣的是，最后一次计算的散列总可以在实现类的 Hash 属性中使用。但是，您可以利用下面等价的代码对前一示例中的散列进行迭代：

```
foreach (byte hashByte in hash.Hash)
{
  ...
}
```

如果多次运行这个测试应用程序，除非修改了源数据，否则接收到的散列都相同。为了便于理解这一点，可以考虑下面的示例。该示例利用一个测试程序来验证两个相同的字符串。我们可以利用 SHA1CryptoServiceProvider 的个别实例独立创建散列。然后，代码对散列数据进行迭代，并测试每个字节的值是否相等。因为散列是相同的，所以最终的结果是相匹配的。

注意，不能使用 Equals()方法来比较两个数组。该方法可以测试引用等价性。如果两个对象都存储在内存中的相同位置上——换句话说，即如果两个变量实际都指向同一个数组，它会返回 true。

```
// HashTest2.cs

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

```
class HashTest2
{
  static void Main(string[] args)
  {
    HashAlgorithm hashA = HashAlgorithm.Create();
    HashAlgorithm hashB = HashAlgorithm.Create();

    // Create two identical strings
    string sourceString = "This is a test string";
    string matchString = "This is a test string";

    System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();

    // Create a hash for the first string
    byte[] sourceBytes = enc.GetBytes(sourceString);
    byte[] sourceHash = hashA.ComputeHash(sourceBytes);

    // Create a hash for the second string
    byte[] matchBytes = enc.GetBytes(matchString);
    byte[] matchHash = hashB.ComputeHash(matchBytes);

    // Test for value equality.
    bool match = true;
    for (int i = 0; i < sourceHash.Length; i++)
    {
      if (sourceHash[i] != matchHash[i])
      {
        match = false;
      }
    }

    if (match)
    {
      Console.WriteLine("The hashes are identical");
    }
    else
    {
      Console.WriteLine("The hashes do not match");
    }

    Console.ReadLine();
  }
}
```

该示例的输出结果是：

```
The hashes are identical
```

还应该通过把散列转换为一个共同的表示形式(例如字符串)，然后比较两个字符串来测试散列是否相匹配。上面给出的方法是最常使用的。

说明：

为了使用该方法，必须确定您所使用的编码方法和把字符串转换为字节时使用的方法相同。这里没有简单的方法来强制实施这个规定，或者可以发现由于文本编码方法的类型不同而造或的散列差异。

创建了散列码之后应该如何处理它取决于应用程序本身。我们将在后面“保存散列和签名”这一小节中进行介绍。

3.1.3　散列流中的数据

ComputeHash()方法还可以提供一种可以接收流的重载形式。在这种情况下，ComputeHash()方法可以从当前位置读取完整的流，并对它所读取的全部数据计算散列。

```
// HashStream.cs

using System;
using System.IO;
using System.Security.Cryptography;

class HashStream
{
  static void Main(string[] args)
  {
    Console.WriteLine("Enter A File Name:");
    String fileName = Console.ReadLine();
    FileStream fs = new FileStream(fileName, FileMode.Open);
    // Calculate the hash.
    HashAlgorithm hash = HashAlgorithm.Create();
    Byte[] hashBytes = hash.ComputeHash(fs);
    fs.Close();

// Display the hash data
    Console.Write("Hash: ");
    Console.Write(BitConverter.ToString(hashBytes));
    Console.Read();
  }
}
```

如果希望在该操作之后利用流，就需要把流的位置重新设置为开始时的位置(在该示例中，

需要打开文件)。

散列 CryptoStream 中的数据

因为 HashAlgorithm 类可以实现 ICryptoTransform 接口，所以还可以将它和 CryptoStream 实例一起使用。在同时执行其他加密操作时该方法非常有用——例如，如果对文档和它的散列码加密。HashAlgorithm 起着传递转换的作用；实际上它不能修改数据。但是，只要对 CryptoStream 调用 FlushFinalBlock()，就可以获得对所有通过流的数据计算后的散列。注意用来散列的 CryptoStream 必须处在写模式下，否则不能正确地计算散列。这样就使一些任务更加难以实现。如果要对一个只读流计算散列值，就必须把该信息读入内存的缓冲器中。如果需要为大量的数据创建散列码，这显然是不合适的。

说明：

在 CryptoStream 中使用 HashAlgorithm 时，CryptoStream 必须处在写模式下。如果试着在读模式下使用(例如，在检索文件信息时计算一个散列)，则不会抛出异常——但散列值是无效的。实际上，它可能包含与内存中一个随机位置相对应的散列数据。

下面代码说明的是在读取文件时利用 CryptoStream 执行散列的一个控制台应用程序。

```
// HashStream2.cs

using System;
using System.IO;
using System.Security.Cryptography;

class HashStream2
{
  static void Main(string[] args)

  {
    // Open the file
    Console.Write("Enter A File Name: ");
    string fileName = Console.ReadLine();
    FileStream fs = new FileStream(fileName, FileMode.Open);

    /*
      Create the hashing stream (must map to a writable stream,
      like a memory stream)
    */
    HashAlgorithm hash = HashAlgorithm.Create();
    MemoryStream ms = new MemoryStream();
    CryptoStream cs = new CryptoStream(ms, hash,
                                       CryptoStreamMode.Write);

    // Display the data
```

```
        Console.Write("Data: ");
        byte fileByte;
        do
        {
          fileByte = (byte)fs.ReadByte();
          cs.WriteByte(fileByte);
          Console.Write(fileByte.ToString() + " ");
        } while (fs.Position != fs.Length);
        cs.FlushFinalBlock();
        fs.Close();

    // Display the hash data (note that ComputeHash() is not used)
        byte[] hashBytes = hash.Hash;
        Console.WriteLine();
        Console.Write("Hash: ");
        Console.Write(BitConverter.ToString(hashBytes));
        Console.Read();
      }
    }
```

3.2　加 salt 值的散列

前面我们已经提到过，如果被散列的数据比较短而且可以预测，有人就能够对字典中每个字符串的可能值进行散列并把它作为查找表来寻找密码值。即使计算该查找表所花费的时间很长，这也不是什么大问题，因为我们只需执行一次这样的操作。

以散列形式保存密码在许多系统中都是很常见的。因此如果有人设法攻击系统并窃取密码文件，您不能泄漏所有用户的密码。遗憾的是，该文件将受到那些已经创建了一系列可能密码的散列值的用户攻击。

可以击败预编译字典攻击的一种方法是使用 salted 散列。在保存密码时许多操作系统(包括 Unix)都采用这种办法。下面将对进行散列和加 salt 值的方法做以解释。从本质上看，它的工作原理如下。当创建一个散列时，需要采取下列步骤：

(1) 把数据(比如口令)转换为一个字节数组。

(2) 为密码添加随机的 salt 值，并对其散列。

(3) 把散列值和原始的 salt 值存储在一起。

把一种不可推测的、强加密的随机数字作为 salt 值是非常重要的。有关的详细内容可以参考附录 B。基本规则是散列值是利用随机字节序列(即“salt”)生成的。在密码表中，每个密码都具有不同的 salt 值。这就意味着由于数据都是随机构成的，所以攻击者不得不对文件中的每个散列运行字典攻击，而不是利用预编译的查找表。第 4 章将介绍一个利用这种技术保存口令的示例。

这项技术提供了额外的安全性，但不能提高目前处理器的运行速度。它最多可以减慢攻击者的速度，并为您提供时间来警告那些口令受到威胁的用户。

3.3 加密的散列码

散列码允许用户检验数据是否被修改。但是，散列码有一个明显的不足，在分布式通信过程中表现得尤为突出。问题在于恶意用户可以把原来的消息替换为错误消息并为其生成相应的散列码。然后接收消息的应用程序对消息进行测试，发现散列码是正确的，而且根本不知道该消息不是原始的消息。这就是我们在第 1 章介绍的中间人(man-in-the-middle)攻击方式。

有 3 种方法可以解决该问题，而且所有方法都要混合使用散列和加密技术。

- 对整个消息加密，包括消息正文和它的散列。根据实际情况，可以使用对称加密，也可以使用非对称加密。另外，还可以利用下面的技术只对散列加密。
- 使用密钥散列函数，把散列法和对称加密技术结合在一起。在创建散列时，密钥散列函数把密钥和消息结合在一起使用。因为恶意用户没有密钥，所以他们不能为新消息生成散列。
- 使用数字签名，把散列法和对称加密技术结合在一起使用。从本质上看，可以计算散列，然后利用私钥对其加密。因为恶意用户没有私钥，所以他们不能生成伪造的散列。

如果希望消息不被窃听和篡改，可以使用第 1 种方法。如果只需要保证消息不被篡改，但可以窃听，用户可以选用其他两种方法。要根据实际情况选择使用密钥散列函数还是使用数字签名。如果有适当的值作为共享的密钥使用，应该使用密钥散列函数，尽管数字签名可以保证任何人都能够通过公钥来验证消息。

3.3.1 散列并加密文件

同时进行散列操作和加密操作的最简单的方法是利用一个单独的 CryptoStream 对象进行加密，并直接利用 HashAlgorithm 类计算散列。最重要的是要记住必须对散列加密。实现该操作的最简单方法是先执行散列，然后对整个结果文档进行加密。如果对文档加密，就会生成一个散列，散列码是不受保护的，攻击者仍然可以用错误的消息替换原来的消息(即使消息在另一端不能被解密)。

基本过程是先读取文件，把加密的文件数据写入一个新的流中，并且添加一个散列码，这部分代码可以在文件 EncryptHash.cs 中找到。首先我们来创建两个文件流，一个进行读操作，一个进行写操作：

```
// EncryptHash.cs
// ...
// Open the input file.
FileStream fsInput = new FileStream("c:\\customers.xml",
  FileMode.Open);

// Open the output file.
FileStream fsOutput = new FileStream("c:\\customers.enc",
  FileMode.Create);
```

下一步，必须创建两个 CryptoStream 对象。第一个 CryptoStream 对象可以包装输出文件，

并执行加密操作。第二个则包装正在加密的流并计算散列。

```
// Create the encrypting stream (in write mode).
Rijndael crypt = Rijndael.Create();
CryptoStream csEncrypt = new CryptoStream(fsOutput,
  crypt.CreateEncryptor(), CryptoStreamMode.Write);

// Create the hash stream (in write mode).
HashAlgorithm hash = HashAlgorithm.Create();
CryptoStream csHash = new CryptoStream(csEncrypt, hash,
  CryptoStreamMode.Write);
```

创建了这些流之后，现在来编写数据。在这种情况下，我们每次只能传递一个数据块。这样可以提供较好的性能，而且如果需要的话，还可以在某种程度上改进计数器。

```
byte[] bytes = new byte[1024];
int bytesRead = 0;
do
{
  // Read a block of 1K.
  bytesRead = fsInput.Read(bytes, 0, 1024);

  // Write the block (encrypted).
  csEncrypt.Write(bytes, 0, bytesRead);
} while (bytesRead > 0);
```

在编写了数据之后，我们要计算散列并将其添加到文件中。

```
fsInput.Position = 0;
hash.ComputeHash(fsInput);
fsInput.Close();
csEncrypt.Write(hash.Hash, 0, hash.Hash.Length);

csEncrypt.FlushFinalBlock();
fsOutput.Close();
```

如果颠倒操作步骤的顺序，就很难以实现，因为需要手动计算字节的偏移量，分开散列并对其进行验证。如果不谨慎，还会因为出现大量字节复制逻辑而结束操作，这样的效率必定非常低下而且难以维护。遗憾的是，.NET Framework 不能为这些任务提供更高级的抽象。

为了简化操作，下面的代码利用 MemoryStream 把从文件中读取的数据块拼凑在一起。另一种方法可以通过检验加密文件的长度，并利用加密算法的代码块长度计算出对应解密数据的长度，然后再减去散列的长度，从而创建出长度适宜的字节数组。这种代码不仅更容易出现错误，而且如果想修改加密逻辑以便使用不同的散列长度时，还可能要更新代码。为了避免出现这种问题，我们需要保存一些与散列算法和散列相关的信息，这与处理数字签名非常相似。

```
// Open the input file
FileStream fsInput2 = new FileStream("c:\\customers.enc",
                                     FileMode.Open);

// Create the decrypting stream (in read mode).
CryptoStream csInput = new CryptoStream(fsInput2,
   crypt.CreateDecryptor(), CryptoStreamMode.Read);

// A memory stream makes it easy to piece together the data
MemoryStream ms = new MemoryStream();

do
{
   // Read a block of 1K.
   bytesRead = csInput.Read(bytes, 0, 1024);

   // Write the block (decrypted), into memory.
   ms.Write(bytes, 0, bytesRead);
} while (bytesRead > 0);
fsInput2.Close();

// Copy the file data into a byte array
byte[] fileBytes = new byte[ms.Length - hash.HashSize/8];
ms.Position = 0;
ms.Read(fileBytes, 0, fileBytes.Length);

// Copy the hash data into a byte array
byte[] hashBytes = new byte[hash.HashSize/8];
ms.Read(hashBytes, 0, hash.HashSize/8);

// Calculate the hash of the decrypted data
hash.ComputeHash(fileBytes, 0, fileBytes.Length);

// Check if the hashes check out
if (CompareHash(hashBytes, hash.Hash))
{
   System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
   Console.WriteLine(enc.GetString(fileBytes));
}
else
{
   Console.WriteLine("The file has been tampered with");
}
Console.Read();
```

在这种情况下，我们还可以利用一个名为 CompareHash()的私有帮助函数：

```
// Tests two byte arrays for value equality.
private static bool CompareHash(byte[] hashA, byte[] hashB)
{
  // Verify the array sizes match.
  if (hashA.Length != hashB.Length)
  {
    throw new ArgumentException("Hashes must be same length");
  }

  // Verify that each individual byte matches.
  bool match = true;
  for (int i = 0; i < hashA.Length; i++)
  {
    if (hashA[i] != hashB[i])
    {
      match = false;
    }
  }
  return match;
}
```

该程序作为一个示例是非常合适的，但是您在生产系统中实现这种散列方法之前可能要三思。良好的编程规则要求尽可能少地编写加密代码，而且始终要有密码术专家对其评审。原因很简单，因为很容易就犯需要花时间发现的低级错误。常见的问题包括：

- 用加密数据代替源数据计算字节的长度会导致出错，反之亦然。
- 试图在一个只读流上执行散列，或者错误地分出多个 CryptoStream。这样当用户不小心在加密文件的最后编写一个未加密的散列值时，就会出现问题。
- 在创建散列和验证散列时，错误地计算散列长度或者使用不同的散列长度。
- 当散列数据在文件末尾时认为它在文件的开头(从理论上讲，可以使用一个类似数据库的安全日录来记录散列，但这是无法实现的)。

所有这些细小的问题都会产生严重的后果。即使只修改源数据中的一个字节，散列就完全不同(从技术上看，每位都有 50/50 的概率发生改变)。如果改变了要散列的数据，就会产生严重的后果。例如，一个利用散列码对串行化数据的某一部分进行验证的应用程序。如果修改这个应用程序来散列附加信息，就会破坏它和所有现存文档的兼容性。

如果需要对同一文件中的信息散列并加密，可以考虑开发自己的类似于 CompareHash()方法的帮助方法，自动完成需要处理原字节的一些低级操作。从理论上讲，这些帮助函数可以包含在一个专用的组件中，它封装了与从字符串到二进制转换、散列长度和验证过程相关的所有信息，以及在将散列集中到一个流中时，需要将它从它的数据中分离出来的功能。

3.3.2 密钥散列算法

.NET Framework 提供了两种密钥散列算法，如下表 3-2 所示。这两种算法都是从 KeyedHashAlgorithm 类中派生的，该类添加了一个 Key 属性来保存那些用来创建散列的密码值。

表 3-2

算　法	抽 象 类	默认的实现类	散列长度(位)	键　长
HMAC-SHA1	KeyedHashAlgorithm	HMACSHA1	160	64 (建议)
MAC-3DES-CBC	KeyedHashAlgorithm	MACTripleDES	64	24

HMAC-SHA1 把 HMAC(基于散列的消息验证代码)散列加密标准和 SHA1 散列法结合在一起。MAC-3DES-CBC 可以利用密码分组链接(Cipher Block Chaining，CBC)模式使用 TripleDES 算法对整个消息加密，因此每个块都会影响下一个块的加密，这样就只保留最后一个块不受影响。它可以自动用 0 填充并把初始向量全部设为 0。HMAC-SHA1 是系统默认的密钥散列算法，因此在调用 KeyedHashAlgorithm.Create 时，可以创建一个 HMACSHA1 类的实例。该方法通过增大其长度以提供更高的安全性，这是首选方法，除非消息的长度非常重要。

理解密钥散列算法的最简单方法是考虑前面单独对两个字符串进行散列的示例。如果用 KeyedHashAlgorithm 代替 HashAlgorithm，由于自动生成的密钥不同，所以散列也就不相等：

```
// HashTest3.cs

...

KeyedHashAlgorithm hashA = KeyedHashAlgorithm.Create();
KeyedHashAlgorithm hashB = KeyedHashAlgorithm.Create();

// Create two identical strings.
string sourceString = "This is a test string";
string matchString = "This is a test string";

System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();

// Create a hash for the first string.
byte[] sourceBytes = enc.GetBytes(sourceString);
byte[] sourceHash = hashA.ComputeHash(sourceBytes);

// Create a hash for the second string.
byte[] matchBytes = enc.GetBytes(matchString);
byte[] matchHash = hashB.ComputeHash(matchBytes);

// Test for value equality.
```

```
bool match = true;
for (int i = 0; i < sourceHash.Length; i++)
{
  if (sourceHash[i] != matchHash[i])
  {
    match = false;
  }
}
```

要想使该示例正常工作，只需保证每个 KeyedHashAlgorithm 实例都使用相同的密钥即可。在该示例中，只需在创建所有实例之后，添加下面的代码：

```
hashA.Key = hashB.Key;
```

当然，在实际操作中，该密钥可以从一些安全密钥存储(可能是一个数据库)中获得。

3.3.3　数字签名

数字签名是指已经利用非对称算法加密的散列码。它的优点是用户可以在不进行密钥交换的前提下验证数字签名，因为这里只需要公钥。如果需要的话，公钥甚至可以添加在已签名消息的后面，但是接收者必须认真检验公钥是否与所期望的发送者的公钥相匹配。

有许多方法都可以创建数字签名，但它们都依赖于 DSACryptoServiceProvider 类或者 RSACryptoServiceProvider 类(我们在上一章中已经介绍了这两个类)。上一章中讲到，直接创建具体实现类是最简单的，因为它们为加密和解密提供了在 DSA 和 RSA 抽象类中不能利用的方法。但都适用于数字签名。

第一个示例使用的是 DSACryptoServiceProvider，它提供了一种 SignData()方法和一种 VerifyData()方法。我们首先要创建 DSACryptoServiceProvider 的两个实例，而且每个都具有不同的密钥信息。第一个实例为一个字节数组签名，第二个实例用来对其进行验证。

```
// DigitalSig.cs
// ...
// Create two signature objects
DSACryptoServiceProvider dsaA = new DSACryptoServiceProvider();
DSACryptoServiceProvider dsaB = new DSACryptoServiceProvider();

// Create some data.
string sourceString = "This is a test string";
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
byte[] sourceBytes = enc.GetBytes(sourceString);

// Create a hash for the data, and sign it
byte[] signedHash = dsaA.SignData(sourceBytes, 0,
  sourceBytes.Length);

// Export the public key data from the signer
```

```
string publicKey = dsaA.ToXmlString(false);
//Import public key data
dsaB.FromXmlString(publicKey);

if (dsaB.VerifyData(sourceBytes, signedHash))
{
  Console.WriteLine("Signature authenticated");
}
else
{
  Console.WriteLine("Invalid signature.");
}
Console.Read();
```

注意 dsaB 对象是利用 dsA 的公钥创建的。ToXmlString()方法可以从正在签名的对象中导出密钥信息，而 FromXmlString()方法可以把它导入到正在验证的对象中。注意 false 参数可以确定只导出公钥。不过这也是验证签名所必需的。

当然，在所有步骤中都可以使用同一个 DSACryptoServiceProvider 对象，但这在实际操作中是不可能的。通常情况下，对签名的验证发生在真正签名后的某段时间内，也可能在另外一台计算机上或者在其他应用程序中。在这种情况下，要通过获得公钥(可能来自数字证书中)来创建 DSACryptoServiceProvider；详细信息可以参考第 6 章有关托管密钥的内容。

DSACryptoServiceProvider 依赖于 SHA1 算法。在调用 SignData()时，该方法会生成一个 SHA1 散列，然后再利用 DSA 对其加密。用户可以通过使用 SHA1CryptoServiceProvider 类创建散列码来手动执行该任务。但是，使用 VerifyData()方法更简单而且更加通用。它还可以执行比较散列字节这样冗长的操作。

SignData()方法还提供了两种重载，允许用户为字节数组的一部分或者流创建一个签名(从当前位置开始，一直读取到末尾)。

说明：

数字签名仅仅是经过加密的散列码。您仍然需要将数据和签名一起发送。用户在验证数字签名之前仍然需要把它和数据分开(根据签名的长度)。

1. 指定散列算法

RSACryptoServiceProvider 包括的方法和 SignData()方法和 VerifyData()方法的版本略微不同。这些方法可以接收一个附加参数，该参数可以指定应该使用哪种散列算法。但是，用户只能使用 CryptoAPI 支持的算法。例如，下面的代码是利用 MD5 散列法重写的代码。

```
// DigitalSig2.cs
// ...
// Create two signature objects
RSACryptoServiceProvider dsaA = new RSACryptoServiceProvider();
RSACryptoServiceProvider dsaB = new RSACryptoServiceProvider();
```

```
// Create some data
string sourceString = "This is a test string";
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
byte[] sourceBytes = enc.GetBytes(sourceString);

// Create a hash for the data, and sign it
byte[] sourceHash = dsaA.SignData(sourceBytes, 0, sourceBytes.Length,
  MD5.Create());

// Copy the public key
string signature = dsaA.ToXmlString(false);
dsaB.FromXmlString(signature);

// Verify hash
Console.WriteLine(dsaB.VerifyData(sourceBytes, MD5.Create(),
  sourceHash));
```

2. 散列后的签名

这里有另外一种和使用数字签名具有相同意义的方法：利用 AsymmetricSignatureFormatter 类的 CreateSignature()和 VerifySignature()方法。这些方法都可以对散列值签名，因此用户首先需要利用个人喜欢的算法执行散列。下面的示例利用 SHA1CryptoServiceProvider 类显式调用 ComputeHash()方法。

```
// DigitalSig3.cs
// ...
// Create some data.
string sourceString = "This is a test string";
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
byte[] sourceBytes = enc.GetBytes(sourceString);

// Create asymmetric algorithm class and linked signature formatter.
DSACryptoServiceProvider dsaA = new DSACryptoServiceProvider();
DSASignatureFormatter formatter = new DSASignatureFormatter(dsaA);

// Create a hash algorithm class.
SHA1 sha = SHA1.Create();

// Create a signature.
byte[] signedHash = formatter.CreateSignature(
  sha.ComputeHash(sourceBytes));
```

该过程差不多和利用 DSASignatureDeformatter 的相反过程相同。最主要的是您需要导入签名。在这种情况下：

```
DSACryptoServiceProvider dsaB = new DSACryptoServiceProvider();
string signature = dsaA.ToXmlString(false);
dsaB.FromXmlString(signature);
DSASignatureDeformatter deformatter =
  new DSASignatureDeformatter(dsaB);

// Verify the signature.
Console.WriteLine(deformatter.VerifySignature(
sha.ComputeHash(sourceBytes), signedHash));
Console.Read();
```

使用这种方法并没有什么太多的好处，但是它允许开发人员创建附加的格式程序(或者使用第三方厂商或.NET Framework 将来提供的版本)。

说明：

DSA 算法可以生成并利用一个随机数字，这就表明具有相同密钥的相同数据的签名不是相同的。还需要注意的是 DSA 与 RSA 的对等用法相比，在计算方面更精确(大约 10-40 倍)。

3.3.4 保存散列和签名

创建散列码之后应该如何处理它取决于应用程序本身，但是只有两种基本选项：

- 把它保存在一个隔离的(安全的)位置上——如果用户希望使用未加密的纯散列值，该选项是非常有用的，但是需要保证文件-散列对不能被替换。该选项另一有用之处是作为登录系统的一部分。

  在许多利用密钥散列或者数字签名的情况下可能会更好。
- 和数据保存在一起——如果要求具备任何级别的安全性，该选项就不适合纯散列。但是对于密钥的散列和数字签名来说，它是最佳的选择。
  - 把散列预先添加/追加到消息或者文件中。该操作非常简单但是不太灵活。当在较后点上提取散列时，需要知道散列的长度和它处在文件的哪一端。
  - 通常情况下，在提取散列时，只需提取前/后 X 个字节。因此该方法假定用户知道散列的长度，并假定在添加散列时，文件编码的方法和散列编码的方法相同。
  - 把散列包装在文件中。该操作虽然比较复杂，却很灵活，因为它可以简单地把有关散列法所用类型的信息添加到文件中，使应用程序支持多种类型的算法。PGP 就使用这种把被签消息的签名放置到定义符之内的封装方法。

```
To: <someone@wrox.com>
From: <someoneelse@somewhere.com>
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

This is a PGP signed message

-----BEGIN PGP SIGNATURE-----
```

```
Version: PGPfreeware 6.5.3 for non-commercial use <http://www.pgp.com>

iQA/AwUBPey8XgvjnxKFgR95EQKWfQCgnmPWjTQtiHeyUior30LIgWvfZs8An3mCZiYUqtAsVM
khq2lwkzrOAwOG=6Iy7
-----END PGP SIGNATURE-----
```

- 将其保存在一个单独的文件中。如果正在散列一个被其他应用程序使用的文件或者消息，修改文件的格式可能并不是一个办法。例如，如果要对一个可执行程序 MyProgramme.exe 签名，就不能把签名添加到文件中，因为这样的话程序将不能运行。但是可以把签名保存在一个名为 MyPrograme.sig 的文件中。
- 如果散列一个可串行化的对象，在反串行化对象时，可以把散列值保存在一个自定义的字段中，并且对其进行验证。

所有这些方法都限制了互操作性而且都不标准。为了解决这些问题，可以定义一个基于 XML 的标准格式来处理数字签名。

3.4　XML 签名

您可以利用目前所学的技术对 XML 文档进行加密(就像加密其他数据一样)，并为其生成一个散列码。如果希望立刻对一个完整的 XML 文档加密，使用这种方法是非常合适的。所得的格式不再是 XML，而且用户可以随意使用自己的系统来确定从前端或后端添加散列码或数字签名。

如果以一种标准的、平台独立的方式处理 XML 数据，情况就会有所改变。所面临的问题包括：

- 如何对部分 XML 文档编码，而且以这种方式编码生成的文档仍然是有效的 XML 吗？
- 如何确定用一种兼容的方法表示 XML，以便散列码的验证操作可以在多个平台上利用多个 XML 解析器成功完成？
- 如何把不同的数字签名应用于未加密 XML 文档的不同部分？而且，用户可以把数字签名或者散列值以一种兼容的、灵活的而且易于获取的方法存储在 XML 文档的什么地方？
- 应该先执行加密操作或者签名吗？牢记，由于签名涉及到私钥，所以不需要手工加密签名。

遗憾的是，这些答案在如今并不都可用。解决这些问题要用一种更加细密的方法进行加密和数字签名。World Wide Web 协会提供了许多建议或者规范草案，包括 XML 加密语法和处理(http://www.w3.org/TR/xmlenc-core)、XML 签名(http://www.w3.org/TR/xmldsig-core)及 XML 签名的解密转换(http://www.w3.org/2001/04/decrypt#)。在这三种规范当中，惟一一个在.NET Framework 中通过 System.Security.Cryptography.Xml 命名空间集成的就是 XML 签名。本章剩余内容将重点介绍这些类型。

3.4.1　XML 标准

有关 XML 签名的一个关键问题是在计算签名之前确定 XML 的形式。本章前面已经讲过，改变源文档中的一个值就会明显改变该数据的散列或者签名。这就意味着 XML 的灵活性实际

上成为一个严重的问题。例如，考虑下面两个文档：

```
<parentNode>
  <childNode></childNode>
</parentNode>

<parentNode>
  <childNode />
</parentNode>
```

这两个 XML InfoSets 是等价的，但是标记却不同。因此，这两个文档的数据签名也不相同。如果第一个文档通过不同的 XML 解析器(可能是在不同平台上使用的解析器)转换为第二个文档，就会出现问题。接收者将不能再对原始签名进行验证。更糟糕的是，即使只添加一个毫无意义的空白，也会出现相同的问题。

要想克服这种问题，就必须使用标准算法。标准是一个把多种等价形式分解为一种单一的、标准形式的过程。在该示例中，我们要把 XML 文档转换为一种标准格式。这样的一种标准可以是标准的 XML 规范(http://www.w3.org/TR/xml-c14n)，它还可以用于联合 XML 签名。如果把标准算法应用于前一示例的两个文档，它们都会被格式化以使用额外的</childNode>标记。.NET Framework 提供了一种标准的 XML 规范的实现，在使用 System.Security.Cryptography.Xml 命名空间中的类型时可以自动应用该实现。

3.4.2 XML 签名规范

XML 签名规范定义了一种<Signature>元素，它包含签名数据以及与签名有关的其他信息。该信息包括例如签名使用的类型、对已签名数据的引用以及验证签名所需的公钥信息(可选)等详细资料。<Signature>元素的基本结构如下：

```
<Signature>
  <SignedInfo>
    <CanonicalizationMethod>
      <SignatureMethod>
        (<Reference URI="">
          (<Transforms>)?
          <DigestMethod>
          <DigestValue>
        </Reference>)+
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo>)?
  (<Object>)*
</Signature>
```

在该示例中，“？”表示 0 次或 1 次出现，“＋”表示一次或多次出现，“*”表示 0 次或多次出现。圆括号指出这些特定字符应用的元素。如表 3-3 所示。

表 3-3

元　素	说　明
SignedInfo	包括签名、如何计算签名以及计算签名的数据的有关信息
CanonicalizationMethod	指定标准化 XML 时所使用的算法，因此逻辑上等价的 XML 在物理上是相同的。这就是标准 XML 算法
SignatureMethod	指定签署 XML 时使用的算法。XML 签名规范定义了两种算法：DSA 和利用 SHA-1 散列法的 RSA(也就是 PKCS1)
Reference	标识被签名的数据。可以引用嵌在<Signature>元素内部的数据、不在<Signature>元素中但在同一个文档中的信息，或者单独文件中的信息
Transforms	指定在进行散列之前所应用的转换(例如Base64 转换)，和与签名类型(包装、分离或者嵌入)相关的信息。多个转换元素的排放顺序是非常重要的——它将确定进行转换的顺序
DigestMethod	指定在签名 XML 之前进行散列所用的算法(摘要和散列同义)
DigestValue	指定计算的散列。加密该散列以创建签名
KeyInfo	指定验证签名时所用公钥的相关信息，例如算法，甚至还可能包含密钥的公共部分。该元素是可选的
Object	如果正在使用嵌入式签名，该元素包括已签署的数据

说明：

指定 XML 文件中的公钥信息时会有安全隐患。从理论上讲，用户可以替换一个新的签名和一个新的公钥，而且文档也可以成功地验证。这就是典型的中间人攻击；惟一的防御措施就是验证公钥是否来自期望的发送者。

用户可以利用签名来签署 XML 文档中的单个元素，或者整个文档。<Reference>元素表示被签署的元素。它可以是指定其他文件的 URL，也可以指定当前部分文档的引用。如果已签署的数据被嵌入到签名中，它将在<Object>元素内找到。用户还可以在一个文档内使用多种签名。

当首次创建 XML 签名规范时，需要认真讨论有关为文档添加签名的最好方法。但是，由于不同的应用程序可能需要不同的方法，XML 签名规范支持 3 种不同的可能性，如图 3-1 所示。

- 分开的签名—— 在这种情况下，<Signature>元素被存储在不同的文件中，而且<Reference>元素利用 URI 来引用已签署的文档。
- 封装签名—— 在这种情况下，已签署的数据被嵌入到<Signature>元素的<Object>元素内。
- 被封装的签名—— 在这种情况下，<Signature> 元素被插入到包括它所签信息的 XML 文档中。<Reference>元素利用一个 XPath 表达式表示已经被签署的元素。在.NET 中不直接支持被封装的签名，但它们仍然可以利用一些其他方法来创建。

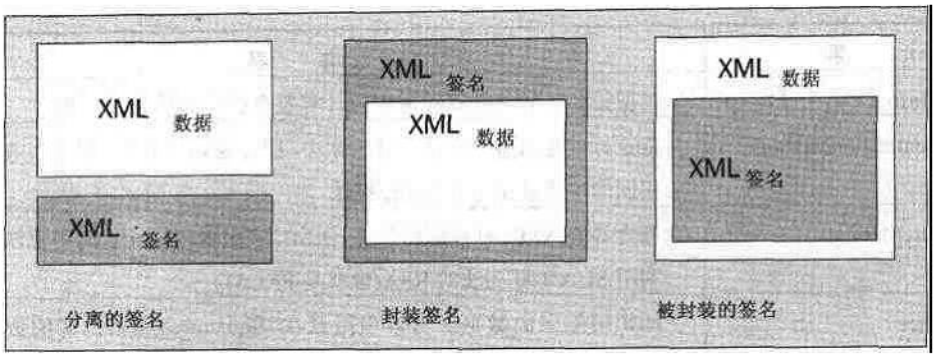

图　3-1

3.4.3　.NET 对已签署 XML 的支持

.NET Framework 在 System.Security.Cryptography.Xml 命名空间中提供了签署 XML 所需的全部支持。如果忽略了.NET 类库引用，就会发出警告："This XML model should not be used for general purposes"在其他文档规范中，Microsoft 公司鼓励使用该命名空间。在以后发布的版本中警告可能会有所改变。

System.Security.Cryptography.Xml 命名空间包括：

- 一组包装<Signature>元素内的 XML 元素的对象。它们包括 Signature、SignedInfo、Reference、KeyInfo 和 DataObject。虽然可以利用 System.Xml 命名空间中的类型创建这些元素，但需要采取附加的措施指定正确的命名空间和一些外部详细资料，例如所使用的加密技术、散列法和标准算法。System.Security.Cryptography.Xml 类型可以自动执行该任务。
- SignedXml 类通过它的 ComputeSignature()和 CheckSignature()方法提供了签署 XML 数据和验证 XML 签名的功能。

该命名空间中还包括其他一些代表不同转换类型的类，本书不再详细介绍。

为了验证 XML 签名，我们将创建一个 Windows Form 应用程序示例，它允许创建一个分离的签名、被封装的签名或封装签名。已签署的 XML 文档被写入磁盘，同时显示在屏幕上的文本框中，如图 3-2 所示。最好的是验证代码适合于所有类型的签名—— 不需要任何修改和条件逻辑。

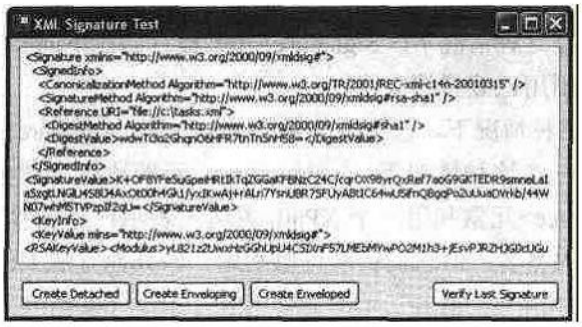

图　3-2

将要签署的 XML 数据保存在下面这个非常简单的文档(tasks.xml)中。它定义了一系列从用户到某种服务器端应用程序的任务请求。在应用程序中，如果数字签名被成功验证(而且发出请求的用户具有请求任务所需的特权)，那么服务器端的组件就只能执行任务。当然，数字签名也是不可否认的——换句话说，已签署的文档可以证明指定用户发出了给定请求。

```
<Requests>
 <Request>
  <TaskCode>130</TaskCode>
  <User>23</User>
  <ConfirmCode>A</ConfirmCode>
 </Request>
</Requests>
```

在继续进行下面的操作之前，我们还应该导入下面 3 个命名空间：

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

如果在项目中还没有引用 System.Security.dll 和 System.Xml.dll 程序集的话，还需要添加对它们的引用。

1. 创建一个分离的签名

分离的签名是一种类型最简单的签名，因为它不要求用户处理包含数据的 XML 文档。用户只需把<Reference>元素的 URI 特性设为引用文件的 URI 即可。例如，下面两种 URI 都是正确的：

```
http://MyWeb/MyXmlDocuments/MyDoc.xml
file://S:\MyServer\MyXmlDocuments/MyDoc.xml
```

当.NET 计算签名时，它将自动检测 URI 中指定的文档。如果没有找到该文档，或者不能理解 URI 的格式，那么它就会生成一个 XmlException。

```
private void cmdDetached_Click(object sender, System.EventArgs e)
{
  // Create the XML signature.
  SignedXml signedXml = new SignedXml();

  // Add a reference to an external file.
  Reference reference = new Reference();
  reference.Uri = @"file://" + Application.StartupPath +
   @"\tasks.xml";
  signedXml.AddReference(reference);

  // Use the RSA algorithm for encryption.
  RSA crypt = RSA.Create();
```

```
    // Add the key details to the signature.
    signedXml.SigningKey = crypt;
    KeyInfo keyInfo = new KeyInfo();
    RSAKeyValue rsaKey = new RSAKeyValue(crypt)
    keyInfo.AddClause(rsaKey);
    signedXml.KeyInfo = keyInfo;

    // Calculate the signature.
    signedXml.ComputeSignature();

    // Transfer the signature into an empty XML document.
    XmlElement xmlSignature = signedXml.GetXml();
    XmlDocument doc = new XmlDocument();

    // The signature must be imported to ensure the namespace is
    // preserved correctly.
    XmlNode node = doc.ImportNode(xmlSignature, true);
    doc.AppendChild(node);

    // Save the signature document.
    doc.Save(Application.StartupPath + "\\tasks_sign.xml");

    // Display the signature.
    txtSig.Text = xmlSignature.OuterXml;
}
```

在该示例中，生成的签名文档(保存为 tasks_sign.xml)的简写格式如下，

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">

 <SignedInfo>
  <CanonicalizationMethod
   Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
  <SignatureMethod
   Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
  <Reference URI="file://c:\DataSecurity\Chapter03\tasks.xml">
   <DigestMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
   <DigestValue>wdwTi3o2GhqnO6HFR7tnTnSnH58=</DigestValue>
  </Reference>
 </SignedInfo>

 <SignatureValue>FZPhN40aE49pfayPFz...</SignatureValue>

 <KeyInfo>
```

```
    <KeyValue xmlns="http://www.w3.org/2000/09/xmldsig#">
     <RSAKeyValue>
      <Modulus>021jHOZcGm0+E3J3...</Modulus>
      <Exponent>AQAB</Exponent>
     </RSAKeyValue>
    </KeyValue>
   </KeyInfo>

  </Signature>
```

请注意，保存公钥信息所使用的 XML 格式与调用 RSA.ToXmlString(false)方法时使用的 XML 格式相同。

2. 创建一个封装签名

封装签名是一种直接将已签署数据保存在签名中的签名。这种类型的签名利用 SignedXml 类实现起来相对比较容易。只需执行下面 3 个附加的步骤即可：

(1) 创建一个单独的 XmlDocument 实例来保存源 XML 文档。

(2) 创建一个 DataObject，并利用它包装源文档中希望签署的数据。

(3) 利用 SignedXml.AddObject()方法把 DataObject 插入到签名中。

除此之外，还需要利用 Reference.Uri 属性标识数据。在该示例中，用户可以使用#运算符指定后面添加了元素 ID 的元素是在签名内发现的。用户还可以利用该值设置 Reference.Uri 属性和 DataObject.ID 属性。实际上，所使用的元素名称是什么并不重要，只要在两个地方一致就可以了。在下面的示例中，我们使用的元素名称是 TaskData。

```
private void cmdEnveloping_Click(object sender, System.EventArgs e)
{
  // Load the XML data that must be signed
  XmlDocument doc = new XmlDocument();
  doc.Load(Application.StartupPath + "\\tasks.xml");

  // Create a data object to hold the data to sign
  System.Security.Cryptography.Xml.DataObject dataObject =
   new System.Security.Cryptography.Xml.DataObject();
  dataObject.Data = doc.ChildNodes;
  dataObject.Id = "#TaskData";

  // Create the XML signature
  SignedXml signedXml = new SignedXml();

  // Add the data object to the signature
  signedXml.AddObject(dataObject);

  // Create a reference that identifies the part of the document
  // you want to sign
```

```
    Reference reference = new Reference();
    reference.Uri = "#TaskData";
    signedXml.AddReference(reference);

    // Use the RSA algorithm for encryption
    RSA crypt = RSA.Create();

    // Add the key details to the signature
    signedXml.SigningKey = crypt;
    KeyInfo keyInfo = new KeyInfo();
    keyInfo.AddClause(new RSAKeyValue(crypt));
    signedXml.KeyInfo = keyInfo;

    // Calculate the signature.
    signedXml.ComputeSignature();

    // Insert the XML signature into the document
    XmlElement xmlSignature = signedXml.GetXml();
    doc = new XmlDocument();
    XmlNode node = doc.ImportNode(xmlSignature, true);
    doc.AppendChild(node);

    // Save the XML document with the enveloping signature
    doc.Save(Application.StartupPath + "\\tasks_sign.xml");

    // Display the full document, with signature
    txtSig.Text = doc.OuterXml;
}
```

注意 DataObject 类的名称必须用 System.Security.Cryptography.Xml 命名空间完全限定，因为在 System.Windows.Forms 命名空间中还有另外一个名为 DataObject 的类。

生成的签名文档如下：

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">

 <SignedInfo>
  <CanonicalizationMethod
   Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
  <SignatureMethod
   Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
  <Reference URI="#TaskData">
   <DigestMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
   <DigestValue>wdwTi3o2GhqnO6HFR7tnTnSnH58=</DigestValue>
  </Reference>
```

```
</SignedInfo>

<SignatureValue>FZPhN40aE49pfayPFz...</SignatureValue>

<KeyInfo>
 <KeyValue xmlns="http://www.w3.org/2000/09/xmldsig#">
  <RSAKeyValue>
   <Modulus>021jHOZcGm0+E3J3...</Modulus>
   <Exponent>AQAB</Exponent>
  </RSAKeyValue>
 </KeyValue>
</KeyInfo>

<Object Id="#TaskData">
 <Requests xmlns="">
  <Request>
   <TaskCode>130</TaskCode>
   <User>23</User>
   <ConfirmCode>A</ConfirmCode>
  </Request>
 </Requests>
</Object>

</Signature>
```

注意在该示例中，整个 XML 文档都被签名。用户还可以修改 DataObject.Data 属性对文档的一部分进行签名。例如，用户可以使用下面的 XPath 表达式代替 XmlDocument.ChildNodes(表示整个文档)。它选定嵌套在<Request>元素(它们嵌套在<Requests>元素内)内部的<ConfirmCode>元素的所有实例：

```
dataObject.Data = doc.SelectNodes("Requests/Request/ConfirmCode")
```

当用户创建或者验证 XML 签名时，将对文档的这一部分进行计算。

3. 创建一个被封装的签名

具有一个被封装的签名之后，<Signature>元素就可以成为原始文档的一部分。其代码与前一示例的代码非常相似，但是它需要另外一个成员：XmlDsigEnvelopedSignatureTransfor 的一个实例，该实例可以直接通过 Reference 对象添加。它表示负责把 XML 数字签名映射为被封装的<Signature>元素的转换过程。

```
private void cmdEnveloped_Click(object sender, System.EventArgs e)
{
  // Load the XML data that must be signed.
  XmlDocument doc = new XmlDocument();
  doc.Load(Application.StartupPath + "\\tasks.xml");
```

```
        // Create XML signature.
        SignedXml signedXml = new SignedXml(doc);

        // Create a reference that refers to the entire document.
        Reference reference = new Reference();
        reference.Uri = "#xpointer(/)";

        // Use a transform required for enveloped signatures.
        reference.AddTransform(new XmlDsigEnvelopedSignatureTransform());

        signedXml.AddReference(reference);

        // Use the RSA algorithm for encryption.
        RSA crypt = RSA.Create();

        // Add the key details to the signature.
        signedXml.SigningKey = crypt;
        KeyInfo keyInfo = new KeyInfo();
        keyInfo.AddClause(new RSAKeyValue(crypt));
        signedXml.KeyInfo = keyInfo;

        // Calculate the signature.
        signedXml.ComputeSignature();

        // Get the XML representation of the signature.
        XmlElement xmlSignature = signedXml.GetXml();

        // Insert the XML signature into the document.
        XmlNode node = doc.ImportNode(xmlSignature, true);
        XmlNode root = doc.DocumentElement;
        root.InsertAfter(node, root.FirstChild);

        // Save the XML document with the enveloping signature.
        doc.Save(Application.StartupPath + "\\tasks_sign.xml");

        // Display the full document, with signature.
        txtSig.Text = doc.OuterXml;
    }
```

生成的签名文档如下所示。文档的第一部分和原来未签署的 XML 文档相同。后面直接跟的就是<Signature>元素。注意该文档中添加了一个前一示例中没有的<Transform>元素。

```
<Requests>
  <Request>
    <TaskCode>130</TaskCode>
```

```
      <User>23</User>
      <ConfirmCode>A</ConfirmCode>
    </Request>
  </Requests>

  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"
      />
      <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"
      />
      <Reference URI="#xpointer(/)">
        <Transforms>
          <Transform
            Algorithm=
              "http://www.w3.org/2000/09/xmldsig#enveloped-signature"
          />
        </Transforms>
        <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"
        />
        <DigestValue>wdwTi3o2GhqnO6HFR7tnTnSnH58=</DigestValue>

      </Reference>
    </SignedInfo>

    <SignatureValue>FZPhN40aE49pfayPFz...</SignatureValue>

    <KeyInfo>
      <KeyValue xmlns="http://www.w3.org/2000/09/xmldsig#">
        <RSAKeyValue>
          <Modulus>021jHOZcGm0+E3J3...</Modulus>
          <Exponent>AQAB</Exponent>
        </RSAKeyValue>
      </KeyValue>
    </KeyInfo>
  </Signature>
```

4. 验证签名

有关.NET 的 XML 签名的最好消息之一是不管所使用的签名属于哪种类型，它们的验证代码都相同。.NET 利用<Signature>元素中指定的详细信息来查询已签署的数据，并利用所定义的公钥对其进行验证。但是，如果使用分离的签名，而且没有发现引用 XML 文档，就会抛出一

个异常。这样就使签名更加容易受到损害，尤其是在用户需要把它发送到其他计算机或者平台上时。

要想验证一个 XML 签名，只需把已签名的文档(或者分离签名)加载到一个普通的 XmlDocument 类中，然后再创建一个新的、用来包装它的 SignedXml 实例。这样用户可以通过名称从源文档中获取<Signature>元素，并将其提交给 SignedXml.LoadXml()方法。最后调用 CheckSignature()方法验证签名是否有效。

```
private void cmdVerify_Click(object sender, System,EventArgs e)
{
  // Load the XML signature (which may contain the data).
  XmlDocument xmlDocument = new XmlDocument();
  xmlDocument.Load(Application.StartupPath + "\\tasks_sign.xml");

  // Create a SignedXml object for verification
  SignedXml signedXml = new SignedXml(xmlDocument);

  // Find the first signature.
  XmlNodeList nodeList =
   xmlDocument.GetElementsByTagName("Signature",
   "http://www.w3.org/2000/09/xmldsig#");
  signedXml.LoadXml((XmlElement)nodeList[0]);

  // Verify the signature.
  if (signedXml.CheckSignature())
  {
    MessageBox.Show("Signature authenticated.");
  }
  else
  {
    MessageBox.Show("Invalid signature.");
  }
}
```

如果文档中有多个签名，用户可以逐个遍历它们，并依次对其验证。这在转换过程中是很常见的，其中单个文档可能被多个不同的人修改、验证或者接收。

```
XmlNodeList nodeList =
  xmlDocument.GetElementsByTagName("Signature",
  "http://www.w3.org/2000/09/xmldsig#");

foreach (XmlNode node in XmlNodeList)
{
  // (Load signature from node and verify it here.)
}
```

注意:

如果 XML 文档中没有发现密钥信息，就需要把一个包含公钥信息的 AsymmetricAlgorithm 作为附加的参数提交给 CheckSignature()方法。

说明:

如果在文档中使用加密技术和签名，最简单的方法是先加密数据，然后再签署。这就允许应用程序在不需具备数据解密能力的情况下对签名进行验证。但是，这种排序不受 XML 签名标准的控制——它只是应用程序逻辑的附加部分。将来，随着 XML 安全标准的广泛应用，将出现创建被加密和签名的 XML 数据的标准库。

3.5　小结

本章的核心内容是加密技术在验证数据和保证数据完整性的操作中所起的作用。其中详细介绍了散列函数、密钥散列算法、数字签名以及把这些技术和其他密码术服务(例如加密)结合使用的方法。

在本章的最后，我们又详细介绍了如何创建 XML 签名。XML 签名规范允许对 XML 部分文档而不是整个文件进行签名。随着 B2B 事务处理逐渐成为标准，XML 签名规范也变得越来越重要，并且 XML 也可以用于业务信息(从多个信息源处汇集或者发送到多个信息源)的传递。

像 BizTalk 服务器这样的产品，都把 XML 作为传递消息的专用格式。就.NET Framework 未来的版本而言，Microsoft 公司一定会不断增加对其他 XML 开发标准的支持。实际上，它在新的框架版本发布之前，就可以提供技术预览或增件。如果您打算利用 XML 进行大量的工作，一定要密切关注 Microsoft 的门户网站 http://msdn.microsoft.com/xml/，其中包含了有关 XML 和.NET 集成的最新信息。

第4章　保护长期保存的数据

前一章详细介绍了数据加密和签名的有关内容。本章我们将以这些基本技术为基础，说明在保护长期保存的数据(即长期保存在某种半永久性存储器中的数据)时要实际考虑的问题。在使用该方法的过程中，您会看到一些把加密数据保存在文件、XML 文档和数据库中的新示例。我们还将学习对操作系统功能(例如加密文件系统(EFS))的支持，以及这些技术在一个安全的应用程序中所起的作用。

保护长期保存的数据引发了一些实际问题。在制定数据安全策略时至少需要考虑 5 个问题。这些问题都是在确定应用程序的身份验证和授权逻辑(应用程序如何确认用户的身份并授予其特权)时经常考虑的，但在考虑后台数据加密时一定要牢记。

- 谁有权访问数据而谁又无权访问数据？这个问题可以帮助确定如何加密数据以及使用什么密钥的规则。例如，电子商务系统需要保护信用卡号码，从而只有服务器端应用程序才有权访问它们。另外一种类型的文档存储应用程序需要保证除了文件创建者本人之外，其他任何人不得访问该文件。
- 安全数据的价值是多少？换句话说，您希望花费多少代价(来增强硬件设施或者性能交换)来保护数据。所有加密操作都会有开销，但花费 1000 美元来保护价值 10 美元的东西是毫无意义的。通常情况下，人们很难确定数据的货币价值，但它仍然很值得估计。
- 谁希望得到数据？安全就是减轻威胁，因此每种安全设计首先都应该确定可能遭受到的威胁以及如何降低这些威胁。向国家安全局(NSA)隐瞒数据所使用的技术和向一些无聊的青少年隐藏数据所用的技术是完全不同的，而与向爱管闲事的同事隐瞒数据也不同。
- 其他什么因素能够危及这种安全？如果把密钥信息存储在一个不安全的数据库中，或者使用许多用户都知道并且讨论过的密码，那么不管您进行了怎样的加密或者使用了很长的密钥，结果都一样。整个信任链都会被破坏。
- 散列或者数字签名可以相互替换吗？如果需要保证数据不被篡改，但不需要保密，散列码和数字签名都能提供较好的解决方案。

最好的建议是深层次思考问题。攻击者应该被迫去做 10 件“不可能实现”的事情，因此即使他们设法完成了一件“不可能实现”的事情，用户的安全也不会完全受到损害。

保护服务器环境是最重要的。但是，还应该包括安全的附加层，它可以限制安全破坏的程度。这些包括将信息(例如信用卡数据和用户密码等)都以加密的形式保存在数据库中。这样，有权访问服务器的攻击者就不能轻易访问到重要的秘密信息。同样，还应该对服务器和发行点(签署代码并部署给服务器的计算机)采取不同的安全措施。那样，如果攻击者窃取了用于签名代码的密钥对，我们仍然有防范措施禁止他们在服务器上运行恶意代码。本章将详细介绍这些需要考虑的问题。

4.1　把数据存储到磁盘中

第 2 章中引入了几个如何加密数据并把结果保存在文件中的示例。基本原则是：

- 在加密大量信息时使用对称加密技术，因为非对称加密技术的速度非常非常慢。
- 对称加密技术可以使用基于流的模型，允许用户把读/写操作包装到其他流(例如 FileStream)中。非对称加密技术需要调用每次只处理一块数据的加密方法，所以需要大量的工作。
- 使用对称加密技术，不仅需要一个共享的密钥，而且还需要一个已知的初始化向量(IV)。如果没有其他秘密信息可以使用，用户可以利用 PasswordDeriveBytes 用用户名创建 IV，随机生成一个 IV 并将其写入加密文件的开头，或者使用一个 0 值 IV。当然，所有这些方法都会降低防御字典攻击的安全力度，尤其是当恶意用户有权访问客户机代码并能发现所使用的 IV 值时。
- 加密技术是以二进制形式处理数据。要想把文本转换为二进制输入数据，需要使用一个类(例如 StreamWriter)，或者一个来自 System.Text 命名空间的编码类(例如 System.Text.ASCIIEncoding 或 System.Text.UTF8Encoding)。如果二进制输出数据在给定的存储格式中不受支持，可以在加密之后使用附加的步骤(例如 Base64 转换)。

说明：

如果用户必须使用非对称加密，可以保存一个利用非对称加密随机生成的对称密钥，并利用该随机对称密钥来加密大量数据。这样就支持更快的加密速度，而且它也是 Windows 加密文件系统一直使用的方法。

下面的代码可以说明如何利用对称加密和 CryptoStream 来轻松加密和解密文件。StreamWriter 和 StreamReader 类可以执行文本和二进制数据之间的转换。

```
Rijndael crypt = Rijndael.Create();

// Create the encryption transform for this algorithm.
ICryptoTransform transform = crypt.CreateEncryptor();

// Open a file for writing to.
FileStream fs = new FileStream("c:\\testfile.bin",
  FileMode.Create);

// Create a cryptographic stream.
CryptoStream cs = new CryptoStream(fs, transform,
  CryptoStreamMode.Write);

// Create a text writer.
StreamWriter w = new StreamWriter(cs);
w.Write("Secret data!");
```

```
w.Flush();
cs.FlushFinalBlock();

w.Close();

// Create the decryption transform for this algorithm.
transform = crypt.CreateDecryptor();

// Open a file for reading from.
fs = new FileStream("c:\\testfile.bin", FileMode.Open);

// Create a cryptographic stream.
cs = new CryptoStream(fs, transform, CryptoStreamMode.Read);

// Create a text reader.
StreamReader r = new StreamReader(cs);
string text = r.ReadToEnd();
r.Close();

MessageBox.Show("Retrieved: " + text);
```

在真正的应用程序中，只有这种简单的方法是不够的。例如，通常只需要加密数据的一部分。这就允许用户隐藏敏感数据并对性能进行优化。另外一种常见的要求是业务对象自动执行所需的加密操作，然后将加密技术集成到应用程序中。

4.1.1 选择性加密 XML

执行选择性加密时，有两种选择。用户可以利用应用程序逻辑来确定什么时候加密。例如，代码可以利用信用卡号码总是被加密的这个事实进行硬连接，并且在它从文件中检索数据时，可以自动解密数据。一个更加灵活的方法是把元数据存储到表示给定值是否被加密(并且可以任意添加所使用密钥的有关信息)的输出文件中。这种方法非常灵活，利用像 XML 这样的标准标记格式尤其易于实现。

例如，考虑下面的示例文档。它可以在单个节点中加密数据。Encrypted 特性可以表示该节点中包含加密文本。

```
<DocNode>
  <Node>Text</Node>
  <Node>Text</Node>
  <Node encrypted="true">zCHHw4Td76SLCWsHO0KnVQ==</Node>
  <Node>Text</Node>
</DocNode>
```

要想处理这种格式，可以创建一个派生自 XmlDocument 的特殊子类(我们称之为 XmlEncryptedDocument)，而且在保存和加载文档时，您可以根据哪个节点的 encrypted 特性被设为 True 来自动执行加密和解密操作。

```
public class XmlEncryptedDocument : XmlDocument
{ ... }
```

首先来考虑加载文档所需的步骤。XmlDocument 提供了 Load()方法的 4 种重载形式，它们分别可以接收文件名、Stream、XmlReader 和 TextReader。这些方法同样可以在 XmlEncryptedDocument 中工作，但是它们可以检索未经处理的加密数据，如上所示。除此之外，XmlEncryptedDocument 为 Load()方法添加了 4 种新的重载形式，它们可以接收自动执行所需的任何解密操作的 SymmetricAlgorithm 对象。

```
public void Load(string fileName, SymmetricAlgorithm key)
{
  this.Load(fileName);
  this.ProcessLoad(key, this.DocumentElement);
}

public void Load(Stream stream, SymmetricAlgorithm key)
{
  this.Load(stream);
  this.ProcessLoad(key, this.DocumentElement);
}

public void Load(TextReader reader, SymmetricAlgorithm key)
{
  this.Load(reader);
  this.ProcessLoad(key, this.DocumentElement);
}

public void Load(XmlReader reader, SymmetricAlgorithm key)
{
  this.Load(reader);
  this.ProcessLoad(key, this.DocumentElement);
}
```

新的 Load()方法重载可以调用 XmlDocument.Load()方法来读取文档，然后利用一个私有的子程序 ProcessLoad()来执行解密操作。注意，SymmetricAlgorithm 没有存储在 XmlEncryptedDocument 类成员中。这里没有必要获取内存中的信息，而且不这样做的话还可以增强安全性，尤其是在用户使用类连接另外一个第三方组件时。

ProcessLoad()过程只遍历节点集合，搜索 encrypted 特性。当发现该特性时，节点内的文本就可以利用用户提供的密钥解密。其他值就保留下来。

```
private void ProcessLoad(SymmetricAlgorithm key, XmlNode node)
{
  foreach (XmlNode child in node.ChildNodes)
  {
    for (int i = 0; i < node.Attributes.Count; i++)
```

```
        {
          if (child.Attributes[i].Name == "encrypted" &&
            child.Attributes[i].Value == "true" &&
            child.InnerText != "")
          {
            child.InnerText = this.Decrypt(child.InnerText, key);
          }
        }
        // Use a recursive call to get all inner nodes.
        if (child.HasChildNodes)
        {
          ProcessLoad(key, child);
        }
      }
    }
```

真正的解密操作是通过一个名为 Decrypt()的私有函数执行的。

将解密逻辑从节点遍历逻辑中分离出来有着重要意义。例如，您可以修改 XmlEncryptedDocument 类，用它代替 SymmetricAlgorithm 接收一个自身能够执行节点文本解密操作的类。该类必须实现一个通用接口(假定其名称为 IXmlCryptDecrypt)。ProcessLoad()过程将调用该接口中的方法(假定其名称为 DecryptNode)，每次在它发现加密节点时就执行解密操作。这样就可以运用非对称加密或者附加的转换来使用 XmlEncryptedDocument。另外，还可以把加密算法和密钥恢复的元数据嵌入到加密节点的特性中。

Decrypt()函数可以创建一个内存流来执行解密。它应用两种 CryptoStream 转换：一种是 Base64 编码转换形式(必须首先执行)，另一种是解密。

```
private string Decrypt(string text, SymmetricAlgorithm key)
{
  // Create an in-memory stream.
  MemoryStream ms = new MemoryStream();

  // Convert the string data to binary.
  System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
  byte[] rawData = enc.GetBytes(text);

  // Create a cryptographic stream for decryption.
  CryptoStream csDecrypt = new CryptoStream(ms,
  key.CreateDecryptor(), CryptoStreamMode.Write);

  // Create a cryptographic stream for a Base64 transform.
  ICryptoTransform transformDecode = new FromBase64Transform();
  CryptoStream csDecode = new CryptoStream(csDecrypt,
    transformDecode, CryptoStreamMode.Write);
```

```
    // Write the data to the memory stream
    // (which will then be decoded and decrypted).
    csDecode.Write(rawData, 0, rawData.Length);
    csDecode.FlushFinalBlock();

    // Now move the information out of the stream,
    // and into an array of bytes.
    byte[] bytes = new Byte[ms.Length];
    ms.Position = 0;
    ms.Read(bytes, 0, (int)ms.Length);

    return (enc.GetString(bytes));
  }
```

这些转换以写操作的模式应用于内存流中，因此它们可以像数据那样被复制到内存中。在从内存流中读取数据时仍然可以使用这种方法，但是这样就很难计算存储解密数据和解码数据所需字节的数量。用户不能像该示例一样只利用 MemoryStream.Length 属性，因为其长度与经过加密的、Base64 编码数据的长度是对应的。

接下来应该考虑保存文档所需的步骤。XmlDocument 提供了 Save()方法的 4 种重载形式，它们分别可以接收文件名、Stream、XmlWriter 和 TextWriter。XmlEncryptedDocument 为 Save()方法添加了 4 种新的重载形式，它们可以接收一个能够自动执行所需的任何加密操作的类。

```
public void Save(string fileName, SymmetricAlgorithm key)
{
  this.ProcessSave(key, this.DocumentElement);
  this.Save(fileName);
}

public void Save(Stream stream, SymmetricAlgorithm key)
{
  this.ProcessSave(key, this.DocumentElement);

  this.Save(stream);
}

public void Save(TextWriter writer, SymmetricAlgorithm key)
{
  this.ProcessSave(key, this.DocumentElement);
  this.Save(writer);
}

public void Save(XmlWriter writer, SymmetricAlgorithm key)
{
  this.ProcessSave(key, this.DocumentElement);
  this.Save(writer);
```

```
}
```

ProcessSave()子程序可以在保存文档之前搜索需要加密的数据。

```
private void ProcessSave(SymmetricAlgorithm key, XmlNode node)
{
  foreach (XmlNode child in node.ChildNodes)
  {
    for (int i = 0; i < child.Attributes.Count; i++)
    {
      if (child.Attributes[i].Name == "encrypted" &&
        child.Attributes[i].Value == "true" &&
        child.InnerText != "")
      {
        child.InnerText = this.Encrypt(child.InnerText, key);
      }
    }

    // Use a recursive call to get all inner nodes.
    if (child.HasChildNodes)
    {
      ProcessSave(key, child);
    }
  }
}
```

最后，Encrypt()方法执行和 Decrypt()方法恰好相反的操作。

```
private string Encrypt(string text, SymmetricAlgorithm key)
{
  // Create an in-memory stream.
  MemoryStream ms = new MemoryStream();

  // Create a cryptographic stream for a Base64 transform.
  ICryptoTransform transformEncode = new ToBase64Transform();

  CryptoStream csEncode = new CryptoStream(ms, transformEncode,
    CryptoStreamMode.Write);

  // Create a cryptographic stream for encryption.
  CryptoStream csEncrypt = new CryptoStream(csEncode,
    key.CreateEncryptor(), CryptoStreamMode.Write);

  // Convert the string data to binary.
  System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
  byte[] rawData = enc.GetBytes(text);
```

```
    // Write the data to the stream (which will then be encrypted
    // and encoded).
    csEncrypt.Write(rawData, 0, rawData.Length);

    // Make sure the CryptoStream has everything.
    csEncrypt.FlushFinalBlock();

    // Now move the information out of the stream,
    // and into an array of bytes.
    byte[] encryptedData = new Byte[ms.Length];
    ms.Position = 0;
    ms.Read(encryptedData, 0, (int)ms.Length);

    return (enc.GetString(encryptedData));
}
```

要想验证 XmlEncryptedDocument，可以创建一个简单的测试程序，该程序允许用户加载并编辑一个加密的 XML 文档，如图 4-1 所示。

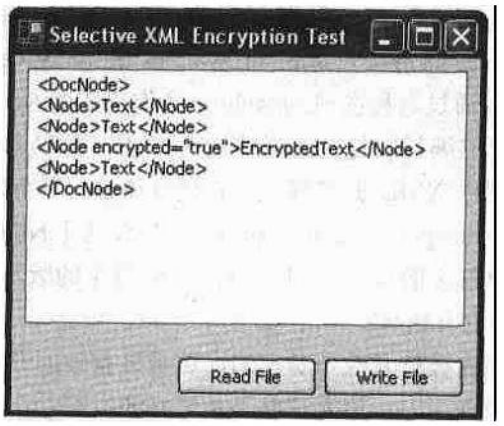

图　4-1

Read File 按钮可以创建一个 XmlEncryptedDocument 实例，用数据加载该实例(这时所有数据都将解密)并显示结果。

```
private void cmdReadFile_Click(object sender, System.EventArgs e)
{
  XmlEncryptedDocument doc = new XmlEncryptedDocument();
  doc.Load(Application.StartupPath + "\\test.xml", crypt);
  txtXml.Text = doc.InnerXml;
  MessageBox.Show("Successfully read file.");
}
```

Write File 按钮可以创建一个 XmlEncryptedDocument 实例，用文本框中的数据填充该实例，

并将其写入到磁盘中(这时具有 encrypted 特性的所有值都将被加密)。

```
private void cmdWriteFile_Click(object sender, System.EventArgs e)
{
  XmlEncryptedDocument doc = new XmlEncryptedDocument();
  doc.InnerXml = txtXml.Text;
  doc.Save(Application.StartupPath + "\\test.xml", crypt);
  MessageBox.Show("Successfully wrote file.");
}
```

当前 XmlEncryptedDocument 假定只有一个密钥可以加密整个文档。如果需要利用不同的算法或者密钥加密不同的数据，用户可以添加附加的特性信息。如下所示(用指定的密钥)：

```
<DocNode>
  <Node>Text</Node>
  <Node>Text</Node>
  <Node encrypted="true" keyName="CorpKey02">zCH476SLCWs0KnVQ==</Node>
  <Node>Text</Node>
</DocNode>
```

据推测，在某些永久性存储器(例如数据库)中保存的密钥可以通过名称识别出来。可以利用 SymmetricAlgorithm 实例的 Hashtable 将它们传递给 Save() 和 Load() 方法。然后 XmlEncryptedDocument 可以通过名称在 Hashtable 中查询所需的 SymmetricAlgorithm。

如果用户希望在跨平台的编程项目中选择性加密 XML，可以考虑一种更标准的方法。一种可能实现该操作的方法是利用 XML 加密规范，它建议通过一种标准方法标识加密元素和相关的密钥信息(见 http://www.w3.org/TR/xmlenc-core)。在首次设计.NET 时，XML 加密只是一个候选的推荐标准，但几乎可以肯定的是，.NET Framework 将来的版本将包括一些 XML 加密支持(尽管在 1.0 和 1.1 版本中都没有包含)。

该示例还假定整个对象在被打开时是解密的，这是最有效的一种方法。但是，在数据必须跨越信任界限的分布式应用程序中，该方法并不是最好的方法。在这种情况下，真正包含信息的文件可以利用不同的密钥加密。应用程序中给定的组件只需访问其中的一些信息，而且无论需要什么数据它都会小心解密。这样，应用程序的每个部分在访问敏感数据之前都被迫提供一个密钥。要想实现这种设计，用户可以不使用前面介绍的帮助类。相反，可以创建一个特殊的“信息包裹”对象，如下一小节要讲述的对象一样。

4.1.2　加密对象

在应用程序中，开发人员经常面临一个保存对象状态的任务。可能您已经非常熟悉.NET 对串行化的支持，它提供了一种标准的方法可以把对象转换为能保存在磁盘中或者在分布式系统中以消息形式发送的 XML 或二进制流。可能您认为定制一个对象的串行化过程(通过实现 ISerializable)就可以确保该串行化过程使用加密技术。遗憾的是，实际情况并不这么简单。

主要问题是一个可串行化的对象应该在不需要任何附加信息的情况下依靠自身重新构建。这就意味着，如果用户希望为串行化过程添加加密操作，就需要采取附加的措施以确保密钥信

息和对象存储在一起。这样就否定了加密的附加安全性——实际上，现在对象已变得很危险。

这就留给用户两种选择。第一种也是最明显的一种，通过包装在串行化过程中所使用的流，在把对象串行化到磁盘中时手工执行加密操作。如下所示：

```
// Wrap a CyptoStream around the FileStream.
FileStream fs = new FileStream("serialized_object.bin",
  FileMode.Create);
CryptoStream cs = new CryptoStream(fs, key.CreateEncryptor(),
  CryptoStreamMode.Write);

// Save the object using the SoapFormatter.
SoapFormatter f = new SoapFormatter();
f.Serialize(cs, objectToWrap);
cs.FlushFinalBlock();
fs.Close();
```

说明：

.NET Framework 中提供了两种串行化格式程序：BinaryFormatter 和 SoapFormatter。BinaryFormatter 比较简单，而且可以应用于许多种情况。但是，在该示例中我们将利用 SoapFormatter，以便能够简单地比较输出结果。要想使用 SoapFormatter，必须添加一个对 System.Runtime.Serialization.Formatters.Soap.dll 程序集的引用，并导入 System.Runtime.Serialization.Formatters.Soap 命名空间。

另外一个更有意义的选项是创建一个特殊的、能够包装任何对象的加密“包裹”。该对象可以根据需要对它的有效负载进行加密和解密——只要提供正确的密钥即可。该包裹最好的性能是可以包装任何对象，而且在自动串行化发生之前可以使用它，并允许用户在把数据保存到磁盘之前，或者在以消息形式传输给一个分布式组件之前加密数据。

事实证明创建这种包裹对象是非常容易的。只需要利用 MemoryStream 把被包装的对象存储在内部即可。然后利用.NET 串行化把对象复制到 MemoryStream 中，同时对其加密。

```
[Serializable]
public class EncryptedPackage
{
  private MemoryStream serializedObject;

  public void Encrypt(object objectToWrap, SymmetricAlgorithm key)
  {
    // Serialize an encrypted copy of objectToWrap in memory.
    // Encryption takes place at the same time.
    serializedObject = new MemoryStream();
    CryptoStream cs = new CryptoStream(serializedObject,
      key.CreateEncryptor(), CryptoStreamMode.Write);
    BinaryFormatter f = new BinaryFormatter();
```

```
        f.Serialize(cs, objectToWrap);
        cs.FlushFinalBlock();
    }

    public object Decrypt(SymmetricAlgorithm key)
    {
        // Decrypt the memory stream into another memory stream.
        MemoryStream ms = new MemoryStream();
        CryptoStream cs = new CryptoStream(ms,
            key.CreateDecryptor(), CryptoStreamMode.Write);
```

该操作每次可以传输(并解密)1K 字节；该操作不是必需的，但是如果需要的话，它可以很好地执行并且能够报告操作进展。

```
        byte[] buffer = new byte[1024];
        int bytesRead;
        serializedObject.Position = 0;
        do
        {
            bytesRead = serializedObject.Read(buffer, 0,
                buffer.Length);
            cs.Write(buffer, 0, bytesRead);
        } while (bytesRead > 0);
        cs.FlushFinalBlock();

        // Now deserialize the decrypted memory stream.
        BinaryFormatter f = new BinaryFormatter();
        ms.Position = 0;
        return f.Deserialize(ms);
    }
}
```

EncryptedPackage 类也可以被串行化。这就意味着用户只需把对象包装在 EncryptedPackage 中，并串行化 EncryptedPackage 对象，就可以串行化任何对象的加密副本。注意 EncryptedPackage 类使用 BinaryFormatter 来确保内部数据以一种相对简洁的形式存储。那样不能阻止把该二进制数据串行化到一个 SOAP XML 文档中，因为 SoapFormatter 的智能化程度非常高，足以自动执行 Base64 编码。

要想 EncryptedPackage 使用起来更加简单，可以添加一个能接收将要包装的对象并自动调用 Encrypt 的构造函数。注意还需要添加一个默认的(无参数的)构造函数，否则.NET 不能反串行化 EncryptedPackage。

```
public EncryptedPackage()
{
    // Default constructor required for deserialization.
}
```

```
public EncryptedPackage(object objectToWrap, SymmetricAlgorithm key)
{
  this.Encrypt(objectToWrap, key);
}
```

现在我们来使用类。要想进行一个简单的测试，可以创建一个简单的可串行化类型，例如下面的 Account 类：

```
[Serializable]
public class Account
{
  private string name;
  private decimal balance;

  public string Name
  {
    get
    { return name; }
    set
    { name = value; }
  }

  public decimal Balance
  {
    get
    { return balance; }
    set
    { balance = value; }
  }

  public Account()
  {
    // Default constructor required for deserialization.
  }

  public Account(string name, decimal balance)
  {
    this.Name = name;
    this.Balance = balance;
  }
}
```

下面的代码可以说明如何直接串行化和恢复 Account 对象：

```
Account account = new Account("Test", 1000);

FileStream fs = new FileStream(Application.StartupPath +
```

```
    "serialized.bin", FileMode.Create);

SoapFormatter f = new SoapFormatter();
f.Serialize(fs, account);
fs.Close();

// Restore the object.
fs = new FileStream(Application.StartupPath + "\\serialized.bin",
  FileMode.Open);
account = (Account)f.Deserialize(fs);
fs.Close();
MessageBox.Show("Retrieved account: " + account.Name);
```

在磁盘中，被串行化的文档如下所示(以简写形式表示)：

```
<SOAP-ENV:Envelope>
<SOAP-ENV:Body>
  <a1:Account id="ref-1">
  <name id="ref-3">Test</name>
  <balance>1000</balance>
  </a1:Account>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

可以看到，Account 类中的所有信息都可以清晰地显示出来。但是，用户可以简单地修改代码来利用加密的包装器。

```
SymmetricAlgorithm crypt = SymmetricAlgorithm.Create();

Account account = new Account("Test", 1000);
EncryptedPackage encryptedAccount = new EncryptedPackage(account,
 crypt);

FileStream fs = new FileStream(Application.StartupPath +
  "serialized.bin", FileMode.Create);

SoapFormatter f = new SoapFormatter();
f.Serialize(fs, encryptedAccount);
fs.Close();
```

下面的代码可以恢复对象：

```
fs = new FileStream(Application.StartupPath + "serialized.bin",
  FileMode.Open);
encryptedAccount = (EncryptedPackage)f.Deserialize(fs);
account = (Account)encryptedAccount.Decrypt(crypt);
fs.Close();
MessageBox.Show("Retrieved account: " + account.Name);
```

新被串行化的文档实际上很难解密。它包括被串行化的 EncryptedPackage 对象，引用的 MemoryStream 对象，以及保存在 MemoryStream 中的内存缓冲。缓冲中包括以加密的 Base64 传输格式保存的 Account 对象数据。

```
<SOAP-ENV:Envelope>
<SOAP-ENV:Body>
  <a1:EncryptedPackage id="ref-1">
    <serializedObject href="#ref-3"/>
  </a1:EncryptedPackage>
  <a2:MemoryStream id="ref-3">
    <_buffer href="#ref-4"/>

  </a2:MemoryStream>
  <SOAP-ENC:Array id="ref-4" xsi:type="SOAP-ENC:Base64">guDuGbAQ5mC4sA
2j7BXSBYyeJlkwOhGFW3AfmlNYObFpJmmav8hMBMlAcA4VveeGzxn7xxUjJrFQKyZHftFdnHFx7Ou
0aECCavU7LMQ4TGlvWFTgduM9n1hngV3FfxRsg11toSAgjn7R92qqTyUGO2xQ2gQ+uz8xQ5KKF+1TE/Vy
zNB1jGPoPx90u3V/M6+b3FTJxdB2TEMp1PzKdKvxuZpa79nnEbo8Ma0HlblRe0AAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA==</SOAP-ENC:Array>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

EncryptedPackage 包装器最好的功能是用户在分布式系统中发送消息时同样可以很好地利用它，我们将在下一章中对其进行讨论。用户还可以使用一种类似的方法来数字签署对象(可能通过创建 SignedPackage 对象)。类不再加密内存流，它只需利用提供的密钥签署它即可。然后还可以灵活地把两个对象结合在一起执行签名－加密或者加密－签名操作。

有趣的是，用户希望开发出一种不同的方法来加密数据库。办法是从 DataRow、DataTable 和 DataSet 中派生出强类型的数据类。用户可以添加一种方法(或者创建一个帮助类)，在把数据保存到 DataRow 中之前利用它加密数据。有关强类型 DataSet 对象的具体内容可以参考 Wrox 公司出版的 Professional ADO.NET 一书。

4.1.3　加密文件系统(EFS)

我们已经看到，手工进行的应用程序级加密操作需要进行大量的操作，而且会出现预想不到的相关问题。但是，如果在应用程序中有一种方法不需要把加密逻辑放在业务代码旁边就可以自动加密信息会怎么样呢？实际上，Windows 2000、Windows XP 和 Windows .NET Server 操作系统都支持对存储在 NTFS 区域中的数据自动进行用户级加密的操作。这种支持就是我们所说的加密文件系统(EFS)，它对应用程序级的加密操作起补充作用。它允许用户保护系统，使攻击者不能轻易从被破坏的计算机(或者被盗窃的硬件设备)中获取敏感数据。

对于当前的数据系统来说，默认的数据保护设置很少。用户密码可以阻止恶意用户启动一台被窃取的计算机上的操作系统，但是其他一些工具可以简单地扫描硬件设备，识别 NTFS 文件结构，并允许检索信息。其他可以保护计算机的设置，例如 BIOS 密码，如果用户物理访问机器的话(通过对主板电池放电，或者只需把硬件设备转移到一台新的计算机上)，很容易就可

以攻破。

惟一一种真正可以阻止攻击者访问数据的方法是对数据加密。利用加密文件系统，指定目录下的数据可以在磁盘上加密。每个文件都利用一个惟一的对称密钥进行加密(利用 DES 算法)，密钥可以随机生成，然后依次利用当前用户的公钥进行编码。因此，检索对文件解密所需的对称密钥的惟一方法是获得用户私钥的访问权。有权访问该密钥的用户可以访问文件，而其他人则被拒绝。

EFS 在实际情况中很值得注意，它对实现和安全都是透明的(它总是生成强密钥，而且从来不把它们保存在可以交换到磁盘交换并被发现的分页内存中)。实际上，EFS 基于 System.Security.Cryptography 命名空间中许多类型之下的同一个 CryptoAPI 库。在一个具有 Windows 2000 或 Windows .NET Server 域控制器的网络环境中，还可以配置一个恢复策略，允许域管理员恢复那些随机生成的、用来加密各个文件的对称密钥(但不是用户的私有非对称密钥)。具体信息可以参考 http://www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp 中的内容。

有了 EFS，指定目录中的数据可以在磁盘上加密。每个文件都可以利用惟一的对称密钥(使用 DES 算法)加密，该密钥是随机生成的，然后再利用当前用户的公钥进行编码。那么，检索文件解密所需的对称密钥的惟一方法就是获得用户私钥的访问权。有权访问该密钥的用户可以访问文件，而其他用户则被拒绝。在网络环境中，根密钥还可以通过网络上的一些证书颁发机构签署，因此不能直接从硬件设备中恢复。

EFS 的用法非常简单。用户可以利用 Windows Explorer 选择一个文件夹来自动加密。只需右击文件夹并选择属性，然后单击 Advanced 按钮。在高级属性窗口，选择 Encrypt contents to secure data，并单击 OK。如图 3-2 所示。

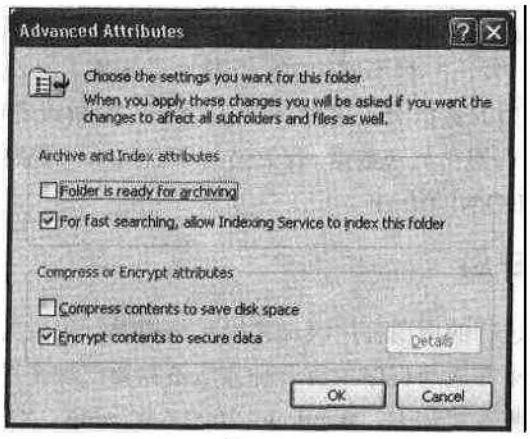

图　3-2

目录中现有的任何文件都可以自动加密。如果有子目录的话，将被提示是否希望对它们加密。如图 3-3 所示。

所有被加密的文件夹和文件都使用特殊的绿色文字在 Windows Explorer 中显示。EFS 加密遵守以下规则：

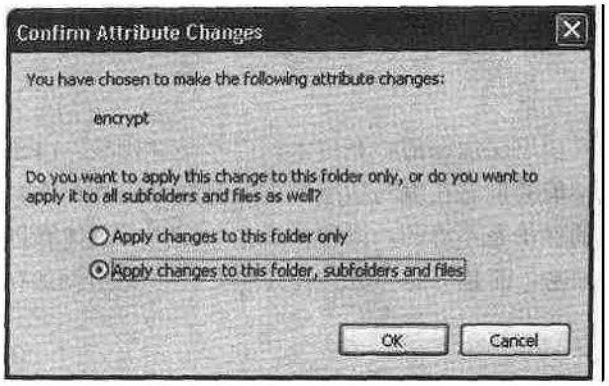

图　3-3

- 移动到加密目录下的新文件可以自动加密。
- 从加密目录下移出的文件仍然是加密的。
- 把文件从加密目录下移到不支持加密操作的位置(例如非 NTFS 区域或者可移动介质)上将生成一则警告消息，提醒用户将撤销加密操作。如果单击 Ignore ，就可以忽略警告并创建文件的未加密副本。

用户还可以选择加密各个文件，但我们并不推荐使用该方法。关键问题是由于应用程序不知道原始文件是加密的，所以它可能把数据复制到一个未加密的文件中(例如同一目录下的临时文件)。如果选择加密一个单独的文件，那么首先就会接到一则警告，如图 3-4 所示。

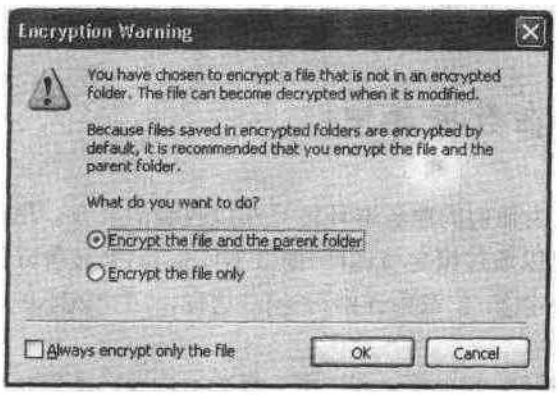

图　3-4

最后，还可以利用一个名为 cipher.exe 的工具从命令行中访问 EFS 密码服务，该程序可以在系统目录下找到(例如 C:\Windows\System32)。使用 cipher.exe，用户可以利用下面的命令行加密目录：

```
> cipher /e MyDirectory
```

并利用下面的命令解密：

```
> cipher /d MyDirectory
```

这样就可以使加密成为应用程序安装脚本的一部分。要获得支持的密码参数的完整列表，

输入下面的命令：

```
> cipher /?
```

牢记，EFS 是一种对应用层加密的补充技术，而不是替代品。EFS 的一个缺点是用户不能从代码内部实施它，也不能验证它在编写敏感数据之前是否完全有效。根据应用程序和部署它的环境类型，这种等级的保护是不够的。同样，如果数据离开了计算机或者在消息中发送给其他计算机，它就不能被加密，而且可能会遭受到其他类型的攻击(例如窃听)。

使用 Triple DES

在默认情况下，EFS 利用 DESX 加密技术，虽然它比明文安全得多，但是现代的标准并不认为 DES 是非常安全的。在那些利用高加密包的 Windows XP(专业版)、.NET Server 和 Windows 2000 操作系统上，如果用户感到需要特别的安全防范措施，他们就可以利用较为安全的 Triple DES 技术。这样就可以利用组策略——这也可以为 IP 地址的安全启用 Triple DES。用户可以为利用 Group Policy Object Editor MMC 管理单元的机器编辑组策略。

在 MMC 窗口中选择 Computer Configuration | Windows Settings | Security Settings | Local Policies | Security Options 菜单命令。

- 双击 System cryptography: Use FIPS compliant algorithms for encryption, hashing and signing
- 选择 Enabled，然后单击 OK。

4.2 把数据保存在数据库中

许多组织机构都误认为他们的服务器环境是难以攻破的要塞。尽管有权进入服务器的攻击者很难禁止，但是我们可以通过锁定资源(例如数据库信息)来显著降低由于暂时中断而造成的损害。(还有一些好经验将在第 8 章中详细讨论)。这样，用户就不能轻易窃取用来攻击的一系列信用卡号码或者密码。

把加密数据保存到数据库中非常简单。用户只需要创建一个二进制字段。但是，把加密数据保存在数据库中会迫使用户牺牲一些灵活性。例如：

- 不能轻易操作存储过程或者 SELECT 语句中的字段。通常情况下，这不会成为问题。
- 丢失与未加密数据类型相关的任何信息。例如，不清楚要解密的二进制数据应该转换为小数还是整数。同样，也不明白使用什么编码方法来处理字符串。我们可以通过认真的文件管理操作来减少这些问题的发生，而且可以把所有具体的内容都封装在一个专用的数据库组件中，但是这样仍然会出现麻烦。

用户还将丢失一些好的东西，例如固定长度的字符串。数据库中二进制字段的长度必须由用来加密的密钥长度确定。如果使用 RSA 非对称加密技术，在默认情况下数据块的长度是 1024 位(或者 128 字节)。利用 Rijndael 算法的非对称加密技术，默认的数据块长度是 256 位(或者 32 字节)。图 3-5 定义了一个用来存储密码的二进制字段，该字段很大，它足以保存一个单独的、

利用 Rijndael 算法加密的数据块。

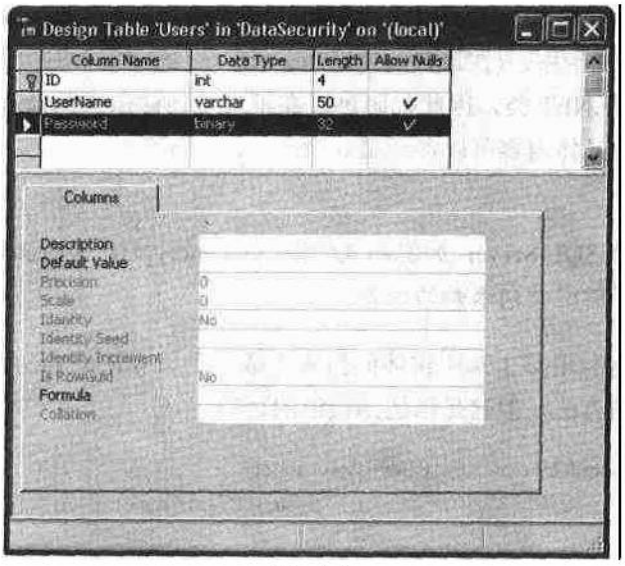

图　3-5

您可以参考第 1 章中的表，获得更多有关算法块长度的信息。

4.2.1　保存加密的数据

处理数据块中二进制数据最简单的方法是利用存储过程调用或者带参数的命令。这些方法都是利用参数来代替动态插入的 SQL 子句，它们可以阻止大部分的 SQL 注入攻击。利用参数可以轻松地保存数据，这是因为提供者可以进行必要的转义处理。这就意味着用户不必浪费时间来构思一个允许把字节数组转换为字符串表达式的子程序。

我们将创建一个简单的测试程序(数据库存储器)来验证该原则。应用程序可以利用 Generate Key 按钮生成一个基于密码的密钥，通过 Save 按钮把加密数据保存到数据库中，并利用 Verify 按钮根据数据库来验证用户名-密码组合。如图 3-6 所示。

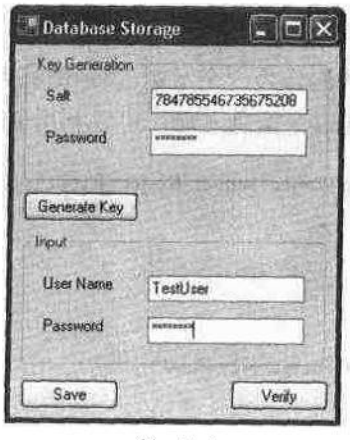

图　3-6

要想使用这种方法，那么在创建密码记录时使用的密钥必须和验证它时使用的密钥完全相同。用户还需要寻找一个极为安全的地方来存储该密钥信息——理论上将其保存在一个单独的 ROM 硬件设备例如智能卡中。(在最好的情况下，用户将具有一个可以扩充密码系统并包装密钥信息的专用 CSP 或者.NET 类，因此代码可以在不直接访问信息的前提下加密和解密数据)。有关存储和管理密钥的具体内容可以参考第 6 章。

说明：

决不能使用无证的 SQL Server 加密函数(例如 pwdencrypt)。这些函数只能内部使用，它们很容易被修改，而且可能遭受到未知的攻击。

为了测试，我们将利用第 1 章中讲述的技术生成一个口令派生的密钥。很明显，这在实际操作中并不是最安全的方法，但它足以达到论证的目的。

```
private void cmdGenerateKey_Click(object sender,
                                  System.EventArgs e)
{
  // Here we have hard-coded the salt value and password value
  ulong hexSalt = ulong.Parse(txtSalt.Text);
  byte[] saltValue = BitConverter.GetBytes(hexSalt);
  crypt = Rijndael.Create();
  // Generate Password-Based Key
  RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
  PasswordDeriveBytes pdb = new
    PasswordDeriveBytes(txtKeyGenPassword.Text, saltValue);
  byte[] key = pdb.GetBytes(32);
  crypt.Key = key;
  crypt.IV = new byte[16];
  //
  // TODO: Add any constructor code after InitializeComponent call
  //
  MessageBox.Show("Key Generated");
}
```

我们可以把 Save 按钮的后台代码作为一个示例，该示例利用一个带参数的命令和 SQL Server 提供者插入一个新的用户。在添加口令信息之前，可以利用 Rijndael 算法对它进行加密。

```
private void cmdSave_Click(object sender, System.EventArgs e)
{

  if (crypt == null)
  {
    MessageBox.Show("Generate Key First!");
  }
  else
  {
```

```
// Create a cryptographic stream for encryption.
MemoryStream ms = new MemoryStream();
CryptoStream cs = new CryptoStream(ms,
  crypt.CreateEncryptor(), CryptoStreamMode.Write);

// Write the password text to an encrypted memory stream.
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
byte[] password = enc.GetBytes(txtPassword.Text);
cs.Write(password, 0, password.Length);
cs.FlushFinalBlock();

// Now move the information out of the stream
// and into an array of bytes.
password = new Byte[ms.Length];
ms.Position = 0;
ms.Read(password, 0, (int)ms.Length);

// Define the database you want to connect to
string connectionString = "Data Source=localhost;" +
  "Initial Catalog=DataSecurity;Integrated Security=SSPI";

// Create a parameterized command with placeholders
string SQL = "INSERT Users (UserName, Password) " +
  "VALUES (@UserName, @Password)";

SqlConnection con = new SqlConnection(connectionString);
SqlCommand cmd = new SqlCommand(SQL, con);

// Add a @UserName parameter of "TestUser"
SqlParameter param;
param = cmd.Parameters.Add("@UserName", SqlDbType.VarChar,
                           50);
param.Value = txtUserName.Text;

// Add the encrypted data to a @Password parameter
param = cmd.Parameters.Add("@Password", SqlDbType.Binary, 32);
param.Value = password;

try
{
  // Insert the record
  con.Open();
  int rows = cmd.ExecuteNonQuery();
  MessageBox.Show(rows.ToString() + " row inserted.");
```

```
        }
        catch (Exception err)
        {
          MessageBox.Show(err.ToString());
        }
        finally
        {
          con.Close();
        }
      }
    }
```

说明：

如果利用 OLE DB 提供者，就不能在带参数的命令中使用指定的占位符(例如@UserName)。相反，必须使用一个简单的问号(?)。在这种情况下，一定要保证将向 Command.Parameters 集合中添加参数的顺序和它们在命名文本中列出的顺序相同。

该示例可以直接处理一个被转换为字节数组的字符串，但是还可以将该代码和前面一些示例的代码合并在一起，把一个被串行化并加密的对象或者 XML 文档写入数据库字段中。图 3-7 显示了数据在数据库中的形式(就像在查询分析器中显示的那样)。

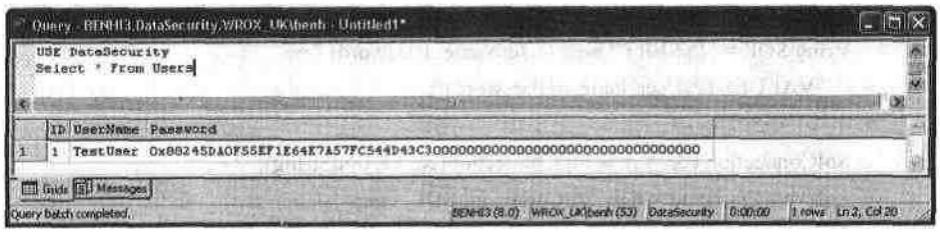

图　3-7

该代码可以利用另一种带参数的命令来选择用户名和密码都相符的记录。在匹配密码之前，用户提供的值必须被转换为字节数组，并利用相同的密钥和算法加密。

```
private void cmdVerify_Click(object sender, System.EventArgs e)
{
  if (crypt == null)
  {
    MessageBox.Show("Generate Key First!");
  }
  else
  {
    // Create a cryptographic stream for encryption.
    MemoryStream ms = new MemoryStream();

    CryptoStream cs = new CryptoStream(ms,
      crypt.CreateEncryptor(), CryptoStreamMode.Write);
```

```
// Write the password text to an encrypted memory stream.
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
byte[] password = enc.GetBytes(txtPassword.Text);
cs.Write(password, 0, password.Length);
cs.FlushFinalBlock();

// Now move the information out of the stream,
// and into an array of bytes.
password = new Byte[ms.Length];
ms.Position = 0;
ms.Read(password, 0, (int)ms.Length);

// Define the database you want to connect to.
string connectionString = "Data Source=localhost;" +
  "Initial Catalog=DataSecurity;Integrated Security=SSPI";

// Create a parameterized SELECT command with placeholders.
string SQL = "SELECT * FROM Users " +
  "WHERE UserName=@UserName and Password=@Password";
SqlConnection con = new SqlConnection(connectionString);
SqlCommand cmd = new SqlCommand(SQL, con);

// Add parameters
SqlParameter param;
param = cmd.Parameters.Add("@UserName", SqlDbType.VarChar,
                           50);
param.Value = txtUserName.Text;
param = cmd.Parameters.Add("@Password", SqlDbType.Binary, 32);
param.Value = password;

try
{
  con.Open();
  SqlDataReader r = cmd.ExecuteReader();
  if (r.Read())
  {
    // A row was returned
    MessageBox.Show("Authentication succeeded.");
  }
  else
  {
    // A row was not returned.
    // You may want to log this error to a security log
    MessageBox.Show("Authentication failed.");
```

```
            }
            r.Close();
        }
        catch (Exception err)
        {
            MessageBox.Show(err.ToString());
        }
        finally
        {
            con.Close();
        }
    }
}
```

在利用加密数据和数据库时，所有文本的比较都区分大小写。在默认情况下，SQL 中的字符串比较是不区分大小写的，因此“OpenSesame”和“OPENsesame”被认为是相同的。如果希望忽略大小写，只需在密码保存到数据库中之前，把它们都转换为规范格式(例如全部小写)即可，用户提供的任何值也按照同样的方法处理。口令加密的主要弱点是加密值可能都非常短，这样就很容易遭受蛮力攻击。有一种解决方案是把像 PasswordDeriveBytes 这样的类当作中介使用，生成口令的二进制信息，并对该数据加密。

4.2.2 利用口令散列进行身份验证

如果把加密的口令保存在数据库中，口令将永远存在，而且如果需要的话可以重新找到它并对其解密。但是，如果只需验证用户，而不需要对现存的密码解密，那么可以使用一种单向散列函数，并把口令散列保存在数据库中。这种方法的好处是：

- 不要求生成散列的保密值。
- 不经过代价昂贵的蛮力攻击就无法确定密码，因为这里没有发现任何秘密密钥。
- 代码非常简单，因为可以调用 ComputeHash() 方法来代替使用流。

用户还不能放松安全，虽然从理论上看，发现另外一个可以生成相同散列的值和对口令解密一样困难。下面的代码片断展示了把散列码存储在数据库中所需的一些代码修改。记住，应该把字段的长度设置为散列的长度以避免浪费空间。散列的位数可以当作 HashAlgorithm.HashSize 属性(每 8 位数组划分为一个字节)使用。

```
// Generate the hash.
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
byte[] password = enc.GetBytes("OpenSesame");
password = HashAlgorithm.Create().ComputeHash(password);
```

4.2.3 利用加 salt 值的口令散列进行身份验证

一种更安全的方法是使用加 salt 值的散列。在存储口令时，许多操作系统都采用这种方法，而且它的好处是加大字典攻击的难度。基本原则是利用一个随机字节序列(即“salt”)生成散列

值。在口令表中，每个口令都具有不同的 salt 值。这就意味着如果攻击者检索到一个完整的口令列表并希望利用蛮力攻击找到相符的值，那么攻击者就需要分别攻击每个散列。这样就会极大地增加破解所有口令所需的时间。

下面将介绍一种既执行散列操作又执行加 salt 值操作的方法。从本质上看，它的工作原理如图 3-8 所示。在创建口令散列时需要执行以下步骤：

(1) 把口令转换为字节数组。

(2) 按一般的方式散列口令。

(3) 为口令添加随机的 salt 值，并再次散列它。

(4) 把随机的 salt 值添加到散列数据之后。然后把结果保存在数据库中。

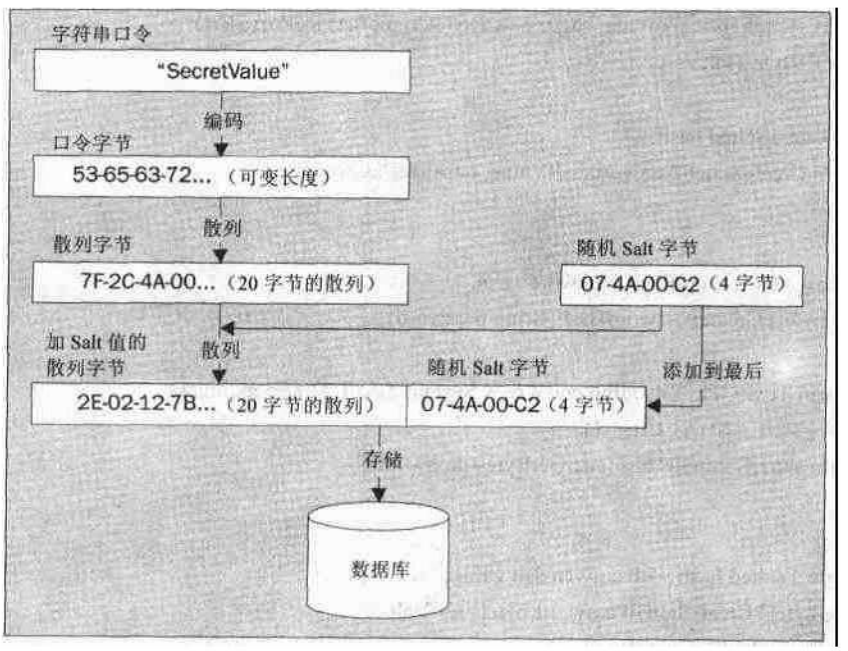

图　3-8

用户可以创建一个实用程序类来封装这些步骤：

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

public sealed class HashHelper
{
 private const int saltLength = 4;

  // (Code omitted.)
}
```

该类提供了一种 CreateDBPassword()方法，它可以通过把口令转换为一个散列(利用 CreatePasswordHash()方法)，然后对一个随机的 salt 值进行散列来创建一个加 salt 值的口令(利用 CreateSaltedPassword()方法)。用户只能在首次创建存储到数据库中的用户记录时使用该方

法。(如果对同一个口令多次使用该方法，每次都会接收到不同的散列，这是因为所选的随机 salt 值不同)。

```
// Creates a salted password to save in the database.
public byte [] CreateDBPassword(string password)
{
  // Create the unsalted password hash.
  byte[] unsaltedPassword = CreatePasswordHash(password);

  // Generate a random salt value.
  byte[] saltValue = new byte[saltLength];
  RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
  rng.GetBytes(saltValue);

  // Create a salted hash.
  return CreateSaltedPassword(saltValue, unsaltedPassword);
}

// Creates a basic (unsalted) password hash.
public byte[] CreatePasswordHash(string password)
{
  System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
  SHA1 sha1 = SHA1.Create();
  return sha1.ComputeHash(enc.GetBytes(password));
}

// Create a salted hash with a given salt value.
private byte[] CreateSaltedPassword(byte[] saltValue,
  byte[] unsaltedPassword)
{
  // Add the salt to the hash.
  byte[] rawSalted   = new byte[unsaltedPassword.Length +
    saltValue.Length];

  unsaltedPassword.CopyTo(rawSalted, 0);
  saltValue.CopyTo(rawSalted, unsaltedPassword.Length);

  // Create the salted hash.
  SHA1 sha1 = SHA1.Create();
  byte[] saltedPassword = sha1.ComputeHash(rawSalted);

  // Add the salt value to the salted hash.
  byte[] dbPassword   = new byte[saltedPassword.Length +
    saltValue.Length];
  saltedPassword.CopyTo(dbPassword, 0);
```

```
        saltValue.CopyTo(dbPassword, saltedPassword.Length);

        return dbPassword;
    }
```

希望通过口令进行身份验证的客户机只需调用 CreateSaltedPassword()方法就可以生成一个口令散列。然后把口令散列通过网络发送出去(但是仍然要对其加密以防止被偷听)。

然后服务器获取来自数据库的加 salt 值的散列，并调用 HashHelper.CompareHash()方法对加 salt 值的散列和用户提供的散列进行比较。该方法的工作原理如下：

(1) 从加 salt 值的散列的末端提取最初的 salt 值。

(2) 利用用户提供的散列(带有 salt 值)来计算 salted 散列。

(3) 然后把这两个加 salt 值的散列逐个字节进行比较。

```
    // Compare the hashed password against the stored password
    public bool ComparePasswords(byte[] storedPassword,
        byte[] unsaltedPassword)
    {
        if (storedPassword == null || unsaltedPassword == null ||
            unsaltedPassword.Length != storedPassword.Length - saltLength)
            return false;

        // Retrieve the salt value.
        byte[] saltValue = new byte[saltLength];
        int saltOffset = storedPassword.Length - saltLength;
        for (int i = 0; i < saltLength; i++)
        saltValue[i] = storedPassword[saltOffset + i];

        // Convert the hashed password to a salted password.
        byte[] saltedPassword = CreateSaltedPassword(saltValue,
            unsaltedPassword);

        // Compare the two salted hashes (they should be the same).
        return CompareByteArray(storedPassword, saltedPassword);
    }

    // Compare the contents of two byte arrays.
    private bool CompareByteArray(byte[] array1, byte[] array2)
    {
        if (array1.Length != array2.Length)
            return false;

        for (int i = 0; i < array1.Length; i++)
        {
            if (array1[i] != array2[i])
                return false;
```

```
        }
        return true;
    }
```

说明：

要想看到该方法在分布式应用程序中的实际运用，可以参考第 8 章，其中一个完全端对端的示例就使用了这种方法。

4.3 创建防止篡改的文件

在第 3 章中，我们了解了如何创建散列、密钥散列和数据签名来验证数据的内容是否被修改。散列经常用在分布式通信环境中，但是在保存长期存在的数据时同样非常有用。

需要牢记的是单独一个简单的散列码不足以验证数据，因为恶意用户可以生成新的数据和与之匹配的新散列码。这样，用户就必须使用一些加密形式(例如，密钥散列或者数据签名)，或者必须把散列保存在一个安全的地方。有一种办法是把一列散列码保存在数据库中。在使用文件中的数据之前，可以快速计算它的散列码并与最后一个被记录的值进行比较。

散列并不仅仅局限于数据文件。用户还可以利用散列保护可执行文件。在默认情况下，.NET 使用非对称加密技术允许发布者为他们的程序集签名，并允许用户根据被允许的发布者设置安全策略。如果存在签名的话，在执行程序集之前需要验证签名。但是数字签名系统没有提供保护密钥不被窃取的任何方法。获得密钥的恶意用户可以发布看起来和原来代码相同的恶意代码。用户可以在强名称签名之后应用 Authenticode 发布者签名来避免这些问题。但是在任务比较重要的环境下，基于发布者的信任模型是不够的。例如，它不能通过同一个发布者来防止对现有应用程序有错误存在的更新操作。

有一种解决方案可以不基于签署代码的人而基于代码所包含的内容向代码授权——换句话说，就是利用一个散列算法采集代码程序集的数字指纹。Windows 2000 就引入了这种功能和 Windows 文件保护(WFP)。WFP 利用一种散列机制将散列码和在数字签名的目录文件中支持的值进行比较。它可以保护操作系统文件，和已经被 Microsoft 鉴定为 Windows 兼容的第三方代码。利用这种散列方法极大地增加了修改这些受保护文件的难度。用户不能只“模仿”正确的文件报头或者版本信息；相反，必须用一个包含了操作系统文件恶意替代物的散列的目录取代散列的数字签名目录。要想了解 WFP 和如何实现它的详细信息，可以参考 http://www.microsoft.com/hwdev/driver/digitsign.asp 和 MSDN 知识库。

利用类似的方法可以锁定一个生产服务器环境吗？它取决于所创建的应用程序的类型和愿意承担的附加工作量。我们将在下面两个小节中考虑两种方法。

4.3.1 在代码访问安全系统中使用散列

.NET 提供了一种重要的代码访问安全系统，允许用户根据各种因素(统称为证据)授权。在默认情况下，所有的本地代码(在本地硬件设备上执行的代码)都被授予完全的权限。但是，它可以加强系统安全，而且只能根据程序集的散列码授权。

当然，本书并不是专门介绍代码访问安全的(有关该主题的内容可以参考清华大学出版社引进并出版的 Eric Lippert 的《Visual Basic .NET 代码安全手册一书)但是，我们将引用一个示例说明如何利用 caspool.exe 命令行实用程序配置.NET 的安全系统，使其安全地使用散列码。

首先，可以利用下面的命令确定所使用的安全等级：

```
> caspol -lg
```

标准的摘要如下所示：

```
Security is ON
Execution checking is ON
Policy change prompt is ON

Level = Machine

Code Groups:

1.  All code: Nothing
   1.1.  Zone - MyComputer: FullTrust
      1.1.1.  StrongName - Microsoft : FullTrust

      1.1.2.  StrongName - ECMA : FullTrust
   1.2.  Zone - Intranet: LocalIntranet
      1.2.1.  All code: Same site Web.
      1.2.2.  All code: Same directory FileIO - 'Read, PathDiscovery'.
   1.3.  Zone - Internet: Internet
      1.3.1.  All code: Same site Web.
   1.4.  Zone - Untrusted: Nothing
   1.5.  Zone - Trusted: Internet
      1.5.1.  All code: Same site Web.
```

这个默认策略最重要的部分是末尾的代码组树。.NET 设法使代码与这些代码组中的一个相匹配以便授予权限。在默认情况下，1.1 区域用于本地硬件设备上的所有代码，它可以授予 FullTrust 权限集。

要想根据散列码对程序集授权，首先要把这个 FullTrust 权限集从 1.1 区域中删除(虽然这是正确的——例如，可以利用散列码来验证从 Internet 或者内部网上下载的代码，在这种情况下，应该把它当作子组添加到 1.3 或者 1.2 区域中)。在该示例中，本地代码被授予 LocalIntranet 权限集，因此它受到很多限制(它包括执行，但是其他几乎所有的操作都要受到限制，包括对文件或者系统的访问)。

```
> caspol -cg 1.1 LocalIntranet
```

下一步根据它们的散列码，显式添加包含程序集的组。下面的代码可以说明如何授予完全权限来确保程序集 MyLibrary.dll：

```
> caspol -ag 1.1 -hash MD5 -file MyLibrary.dll FullTrust
```

现在如果再次显示策略，将看到一个新的策略树：

```
1.  All code: Nothing
  1.1.  Zone - MyComputer: LocalIntranet
    1.1.1.  StrongName - Microsoft: FullTrust
    1.1.2.  StrongName - ECMA : FullTrust
    1.1.3.  Hash - System.Security.Cryptography.MD5CryptoServiceProvider, =
D41D8CD98F00B204E9800998ECF8427E: FullTrust
  1.2.  Zone - Intranet: LocalIntranet
    1.2.1.  All code: Same site Web.
    1.2.2.  All code: Same directory FileIO - 'Read, PathDiscovery'.
  1.3.  Zone - Internet: Internet
    1.3.1.  All code: Same site Web.
  1.4.  Zone - Untrusted: Nothing
  1.5.  Zone - Trusted: Internet
    1.5.1.  All code: Same site Web
```

目前在计算机上运行的所有代码都将授予 LocalIntranet 权限集合，除非它来自 Microsoft 和 ECMA，或者它具有和 MyLibrary.dll 相同的 MD5 散列。注意，用户可以利用.NET Framework 安全工具添加并配置代码组。但是，利用 caspool.exe 是立刻看到完整的代码组树的简便方法。

该方法的功能非常强大，并且在根据条件范围为程序集授予权限时提供大量的灵活操作。当然，它需要一些消耗时间的策略对其进行调整和测试。

4.3.2　通过编程方式检验散列

由 Jason Coombs(2002 年 9 月在 MSDN 杂志上刊登的 Cryptographic Hash Algorithms Let You Detect Malicious Code in ASP.NET)首先提出的另一种方法是利用自定义的 HTTP 模块来保护 Web 服务和 Web 应用程序。该模块可以对 AuthorizeRequest 事件做出反应，并在继续下一步操作之前自动验证那些要求为当前请求服务的文件的散列码。其原型代码如下所示。在该示例中，假定用户正在使用一个普通的散列码；但是如果害怕散列清单被轻易篡改的话，可以替换为密钥散列或者数字签名。

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

public class HashVerificationModule : System.Web.IHttpModule
{
    public void Init(HttpApplication context)
    {
        context.AuthorizeRequest += new
        EventHandler(this.HashAuthorization);
    }
```

```
    public void Dispose() {}

    public void HashAuthorization(object sender,EventArgs e)
    {
        HttpApplication app = (HttpApplication)sender;

        FileStream f = File.Open(app.Request.PhysicalPath,
          FileMode.Open,FileAccess.Read, FileShare.ReadWrite);
        HashAlgorithm hash = HashAlgorithm.Create();
        byte[] hashValue = hash.ComputeHash(f);
        f.Close();

        // Compare the computed hash code with a value stored
        // in a database, and throw an error if it does not agree.
    }
}
```

用户可以通过向 machine.config 配置文件的<httpModules>部分添加一个新的<add>元素，为该模块注册自动处理请求的功能。首先，需要赋予程序集一个强名并将其添加到 GAC 中。然后在 add 元素中指定完整的强名信息，如下所示：

```
<httpModules>
  <add name="HashVerification"
  type="HashVerification.HashVerificationModule, HashVerification,
  Version=1.0.3300.0, Culture=neutral,
  PublicKeyToken=b77a5c561934e089" allowLocation="false""/>
</httpModules>
```

用户还可以创建一个严格限制的实用应用程序，它可以利用新的散列信息更新数据库。

遗憾的是，该系统具有两个明显的不足：

- 目前只有被验证的文件(即.aspx 文件)才是请求目标。后台代码程序集文件和页可以利用的任何组件都不能被验证。因此，只有在用户使用没有组件的内联编码方法时系统才有效(如所写的那样)，这肯定是一个很差的设计。
- IHttpModule 扫描的是被预编译的页，而不是 ASP.NET 自动生成的机器码。如果攻击者找到一种在不修改.aspx 页的情况下代替这些文件的方法，验证就可以顺利进行，但是也将执行非法的代码。

这些缺陷会损害到代码的安全性，但是我们可以利用不同的方法为散列提供类似的安全。例如，我们可以创建一个组件来定时扫描服务器上应用程序文件的目录并对其进行验证。或者，可以扩充 IHttpModule 使其读取数据库上的相关信息目录。最后，在一个 ASP.NET 应用程序或者会话开始运行时，利用 global.asax 事件扫描并验证 bin 目录下的应用程序。该代码可以异步实现，并确保用户不用等待，以及性能开销也会尽可能地小。

4.4 小结

本章介绍了在计划如何以一种安全的形式存储数据时所面临的各种问题。其中包括例如安全保存对象和使用选择性加密这样的应用程序问题。我们还介绍了一种在数据库中处理加密数据的好方法，以及如何利用散列码和.NET 保护代码程序集和数据，从而防止未被识别或者修改过的代码运行。当然，应该注意分层设计。要想取得最大的成功，就必须把本章介绍的技术和其他方法(例如第 8 章中的好经验)结合使用，以便保证代码中不留下任何可攻击的缺陷。

在下一章中，我们将进一步拓宽视野，来了解密码术在一个较大的、基于 Remoting 或者 Web 服务的分布式系统中所起的作用。

第5章　保护通信数据

本章将重点介绍如何使用加密技术和身份验证技术来保护分布式系统中各个组件之间的通信。不管通信是在局域网上进行还是在 Internet 网络上进行，这些操作步骤都可以确保被截获的消息不被读取或者修改。它们同样还可以很好地处理任何消息形式和传输协议，这就意味着用户可以利用这些技术处理原套接字、.NET Web 服务或远程通信。

站在应用程序编程人员的立场来看，利用加密技术保护数据时可以通过以下两种方法：

- 利用一种内置在传输格式中的自动(或者透明的)加密技术。例如，可以使用 SSL 或 IPSec。在这种情况下，整个消息都被自动加密，而且用户几乎不能控制过程。但是，也可从通过测试、行业性强的加密方法中受益。
- 在数据以消息形式发送出去之前，利用.NET 的密码类可以有选择性地加密部分数据。这时用户必须谨慎编码以确保没有引入安全漏洞，但灵活性不受任何限制。

通常情况下，第一种方法比较易于实现，它的优点是不需要在应用程序逻辑旁边插入加密代码。但是，它的可用性也很容易受到影响：根据应用程序的不同，有些用户就不可能使用自动加密服务。而且，还会丧失只对信息交换的某部分进行编码的灵活性。本章将讨论对 SSL 使用自动加密服务。

第二种方法涉及到对.NET Framework 的了解以及它对加密操作的支持。该方法允许通过减少加密信息的数量来优化加密逻辑。为了正确地实现该逻辑，用户需要经常修改系统的接口。例如，以字符串形式传递的数据必须以字节数字的形式发送。这些操作都冒险把应用程序和指定的加密策略紧密耦合在一起。本章我们将考虑如何在一个分布式应用程序中使用选择性加密，和一些可以最小化耦合问题的策略。

当然，还存在一个问题——究竟哪种方法更好些呢？这取决于所处的环境、数据的价值、攻击者的类型以及出现各种威胁的可能性。SSL 在执行关键任务的应用程序中是非常理想的一种工具，因为它可以对任何事物加密，所以能够防止攻击者检测到任何允许他们利用您的系统的用法模式。如果使用选择性加密，稍不留意就会导致某个数据未被加密，虽然数据本身没有什么价值，但它可以用来确定应用程序的弱点。然而自动加密也需要支持它(例如 IIS)的环境，而且通常要求更多的 CPU 处理。这样就会降低每次处理数百个 SSL 请求的服务器的运行速度，尽管可以利用硬件 SSL 加速器(而且经常用在 Web 服务器中)。

在考虑传输层安全时，最重要的是理解可能面临的攻击。这些都将在第 7 章中详细介绍，下面简单概括了最常见的几种类型：

- 偷听，允许攻击者在客户机和服务器之间遍历时检索敏感信息。一般情况下都通过对消息加密的形式来防止偷听。
- 篡改，是当攻击者设法修改消息时，可能会导致应用程序中出现错误。要想防止篡改，可以使用某种签名或者密钥散列码来验证数据是否已经被修改。

- 中间人攻击，是在攻击者扮演客户机或者服务器时，为了获得特权数据，或者只是导致应用程序出现错误而发起的进攻。要想抵御中间人攻击，需要利用上面介绍的全部技术和一些形式的身份验证，例如数字证书。
- 重放攻击，是在攻击者截获一则消息并设法在后面重新使用它时发生的。例如，即使攻击者不能理解或者篡改登录消息，攻击者可以设法用它获得访问权。重放攻击一般利用某种序列号或者时间戳系统来检测过期的或者次序混乱的消息。

说明：

下面许多示例都假定用户已经知道了如何创建 Web 服务和如何通过与组件主机之间的远程通信发布组件。如果您对 Web 服务或者远程通信还很陌生，建议您首先阅读一些与该主题相关的专业书籍，然后阅读本章的内容，以学习如何把远程组件和.NET 加密结合在一起。您可以参考清华大学出版社引进并出版的《C# Web 服务高级编程》一书。

5.1 SSL

安全套接字层(Secure Sockets Lager，SSL)技术是通过 HTTP 协议来加密通信。通常用来加密在客户机和 Web 站点之间交换的数据。要想使服务器支持 SSL 连接，就必须安装一个 X.509 证书。证书是一种用来建立身份的数字文档。证书可以从可信的第三方机构(例如 VeriSign)处购买，我们将在第 6 章对其做详细的介绍。

SSL 在分布式系统中可以用于下面两种情况：

- 用户使用的是 Web 服务。因为 Web 服务是通过 IIS(Internet Information Server)驻留的，所以它们都支持 SSL 加密。
- 用户使用的是远程通信，而且远程组件驻留在 IIS 中。

这两种情况都可以通过 IIS 来利用 SSL。除此之外，还有一些其他的低级选项。例如，可以直接和 Microsoft 公司非托管的安全支持提供者接口(Security Support Provider Interface，SSPI)API 连接，以便在 IIS 外部使用 SSL。该项任务已经超出了本书的范围，而且在一般情况下是不鼓励这样做的，因为它要求具备较高水平的专业技术才能正确实现(没有被忽略的安全漏洞)。用户可以在 MSDN 上的 http://www.microsoft.com/technet/prodtechnol/windows2000serv/maintain/security/sspi2k.asp 中获得更多信息。

要想一起使用 IIS 和 SSL，首先需要购买或者生成一个证书并安装它。用户可以在 IIS 中配置一个或者多个虚拟目录来请求安全通信。连接 Web 页、Web 服务或者通过这些虚拟目录提供远程对象的客户机都需要使用开头为 https://(而不是 http://)的 URL。我们将在下一小节中通过一个简单的 Web 服务来介绍这些步骤。

配置一个远程组件使其驻留在 IIS 中是相当容易的，但是这里我们不再介绍具体的操作步骤。要想了解有关远程通信的更多信息，可以参考清华大学出版社引进并出版的《C# Web 服务高级编程》一书。注意 IIS 支持 SSL3.0 版本。

5.1.1　证书简介

在许多分布式应用程序中，客户机必须在发送敏感数据之前确定是否信任一个 Web 站点。我们在第三方的帮助下建立一种可信的关系，使证书为该目的服务。从概念上看，证书与司机的执照类似。司机的执照是机动车辆管理部门对其身份(和驾驶能力)的一种担保。同样，证书是证书颁发机构(CA)对用户身份的担保。就像信用卡公司可以根据有效的驾驶执照接受申请一样，客户机可以根据它是否具有可信的第三方签署的证书，从而选择把敏感数据提交给 Web 服务器。

要想使用 SSL，组织就必须从一家著名的证书颁发机构购买证书，并将其安装在 Web 服务器上。客户机信任证书颁发机构，因此它也愿意隐式信任 CA 验证过的证书信息。该模型的工作效率很高，因为恶意用户不可能购买并安装一个伪造身份的证书。即使这样做了，CA 也会保留与每个注册用户相关的信息，并可以取消证书。但是不管以何种方式，证书都不能确保服务器的可信性、应用程序的安全性和业务的合法性。从根本上看，证书被限定在一个范围内——不仅恶意的应用程序可以合法注册，而且在某些情况下，信任链也允许客户机信任一个服务器，否则就不能(而且也不应该这样)。

证书本身包括某些标识信息。利用证书颁发机构的私钥，可以保证该信息是可信的而且没有被修改过。业界标准的证书类型即众所周知的 x.509v3，包括下面的基本信息：

- 持有者的名称、组织和地址
- 持有者的公钥
- 证书的验证日期
- 证书的序列号

除此之外，证书还包含具体业务的信息，例如证书持有者的行业、入行时间等。第 6 章中介绍了有关证书和它们如何建立信任链的更多信息。还解释了如何撤销证书和计算机如何根据安装的根证书做出信任的决定。

5.1.2　SSL 简介

可以看到，每个证书都包含一个公钥。这就可以进行非对称加密操作。客户机利用公钥进行编码的任何信息都只能通过服务器解密。这是建立和 SSL 会话的基础。客户机可以建立和服务器之间的信任关系并安全地交换用来加密剩余通信信息的对称加密密钥。具体来看，该过程包括：

(1) 客户机发送一个请求来连接服务器。

(2) 服务器为它的证书签名并将其发送给客户机。这种签名操作(利用服务器的私人密钥执行)可以确保消息不被篡改。由此可以推断出信息交换的部分。

(3) 客户机检验证书是否由它所信任的 CA 颁布。如果是，可以继续下一步操作。在 Web 浏览器中，如果客户机识别出 CA，就可以利用一则带有预兆性的消息警告用户，并让它们决定是否继续执行。注意，客户机使用 CA 的公钥来验证证书签名。

(4) 客户机把证书中的信息和从站点上接收到的信息进行比较(包括它的域名和公钥)。客户机还可以验证服务器端的证书是否有效，是否还没有撤销，以及是否由可信的 CA 颁布。这样

客户机就可以接受或者拒绝连接。

(5) 客户机告诉服务器在进行通信时它支持什么类型的加密密钥。

(6) 服务器选择它和客户机都可以使用的最强的密钥长度，并告知客户机。

(7) 客户机随机生成具有指定长度的对称加密密钥。该密钥被称作会话密钥。在服务器和客户机之间的事务处理中可以使用该密钥。它可以确保更好的性能，因为对称加密比非对称加密的速度快得多。

(8) 客户机利用服务器的公钥(来自证书)加密对话密钥，然后把加密的会话密钥发送给服务器。

(9) 服务器接受加密的会话密钥并利用私钥解密。现在客户机和服务器都具有共享的秘密，它们可以对会话过程中的所有通信加密。

如果打算实现选择性加密，那么理解这些步骤是非常重要的，因为可能会遇到相同的模式。惟一的差别是加密和身份验证逻辑将在应用程序代码内部发生。用户应该注意下面几点：

- 非对称加密只能用在对称密钥的交换中。这样可以更好地保证其性能。
- 对称密钥是随机生成的，而且只能用在会话的持续时间内。这样就可以限制安全风险。首先，它很难利用密码术破译加密的消息，因为这里不能使用其他会话中的消息。第二，即使密钥是由恶意用户确定的，在会话过程中它仍然可以保持有效。
- 客户机必须生成对称密钥。这是因为客户机具有服务器的公钥，利用该公钥可以对只有服务器才能读取的消息加密。服务器没有与客户机相对应的信息，因此也不能加密消息。

最后一点非常有趣。如果客户机提供了一个弱密钥，整个交互作用过程就会受到损害。例如，较老版本的 Netscape 浏览器是利用较弱的随机数字生成器来创建对称密钥。这样恶意用户很容易就可以利用蛮力攻击猜出密钥。

双向证书验证

在任何使用 SSL 的情况下，加密操作总是双向进行的，它既可以把消息发送给服务器，又可以把它发送给客户机。但是，上面描述的身份验证是单向的：服务器提供一个可以证实其身份的证书，客户机同意开始会话。

这样并不能解决验证客户机的问题。有以下几种可能性：

- 利用某种应用程序指定的自定义身份验证方法，把客户机提交的信息与数据库中的信息进行比较。第 8 章演示了该技术的一个基本示例。
- 利用 IIS 提供的身份验证模式，例如集成的 Windows 身份验证(这是目前最安全的一种方法)。如果使用 IIS 身份验证，当客户机请求指定虚拟目录下的资源时，IIS 将设法使客户机自动和 Windows 域服务器进行比较。如果客户机不能被身份验证，就会拒绝请求。
- 利用客户机端的证书。在这种情况下，如果客户机不能产生被可信的 CA 验证过的证书，就拒绝连接。

最后一个选项特别有意义，是因为它是 SSL 标准的一部分。但它很少用在 Web 服务环境下，因为如果把证书分发给所有可能的客户机，所耗费的开销会非常大。也就是说，双向身份验证在某些 B2B(企业对企业)环境中是非常有用的。

利用双向身份验证，服务器可以为客户机提供一系列它所信任的证书颁发机构。如果客户机处理一个由其中某一证书颁发机构颁发的证书，那么它就可以把该证书的副本发送给服务器进行验证。如果证书是有效的，IIS 就可以对映射指定证书的用户进行身份验证。

所有的 IIS 验证都是在虚拟目录的基础上实现的。要想实现双向证书身份验证，就要在 IIS 中找到相关的虚拟目录，右击证书并选择属性。打开 Directory Security 选项卡，选定 Secure Communications 并单击 Edit。这时就可以把证书指定为账户映射。要想了解更多有关映射规则的信息，可以参考 http://www.microsoft.com/windows2000/en/server/iis/htm/core/iimapsc.htm 中的 IIS 文档规范。

5.1.3　使用证书

与如何获取(或者生成)与本章中讲述的应用程序一起使用的证书有关的信息，可以在第 6 章的“密钥和证书管理”一节中找到。

.NET 包含一个 System.Security.Cryptography.X509Certificates 命名空间和一个允许读取证书信息的类。X509Certificate 类提供了两个静态成员：CreateFromFile()和 CreateFromSignedFile()，它们都允许用户实例化一个 X509Certificate 对象并用证书文件中的信息填充。X509Certificate 类不提供任何属性，但是提供了许多允许检索证书相关信息的实例方法，例如 GetIssuerName()和 GetKeyAlgorithm()。

X509Certificate 类受到的限制很多，它只能用来提供信息。公钥数据被作为一个字节数组(或者 Base64 编码字符串)导入，用户不能直接使用它构建 AsymmetricAlgorithm 对象。Microsoft 公司的 WSE(Microsoft .NET 的 Web Services Enhancements 工具)为该类提供了一个功能更强大的版本，它可以从 http://msdn.microsoft.com/webservices/building/wse 处下载。使用 WSE 可以从证书中检索密钥，用它在.NET 中执行程序化的加密操作。还可以从计算机的密钥存储单元(而不仅仅是文件)中检索证书。第 2 章最后对该技术以及来自 Microsoft.Web.Services.Security.X509 命名空间的 X509CertificateStore 类和 X509Certificate 类作了介绍。它们共同组成了未来版本的.NET Framework 的核心。

说明：

如果正在利用 ASP.NET 开发一个 Web 应用程序、驻留在 IIS 中的 Web 服务或者远程组件，用户也可以利用 HttpRequest.ClientCertificate 属性检索发送给客户机对其进行身份验证的证书，但是这些证书使用的都是 System.Web.HttpClientCertificate 类，而不是 X509Certificate 类。

5.1.4　在 IIS 中安装证书

在开发应用程序时，用户可能想从一些真正的证书颁发机构(例如 VeriSign，参见 http://www.verisign.com)处购买证书。IIS 管理器允许自动创建一个证书请求。首先打开 Internet 服务管理器(也就是 IIS 管理器)。展开 Web Site 组，右击 Web 站点项(通常命名为 Default Web Site)并选择 Properties。打开 Directory Security 选项卡，将发现一个 Server Certificate 按钮。单击该按钮可以打开 IIS Certificate Wizard，如图 5-1 所示。该向导请求一些基本的组织信息，并生成一个请求文件。用户还需要为密钥提供一个位数长度——位长度数字越高，密钥就越强。这样

非对称密钥的位长度就会分配给证书(而不是动态生成的会话密钥)。

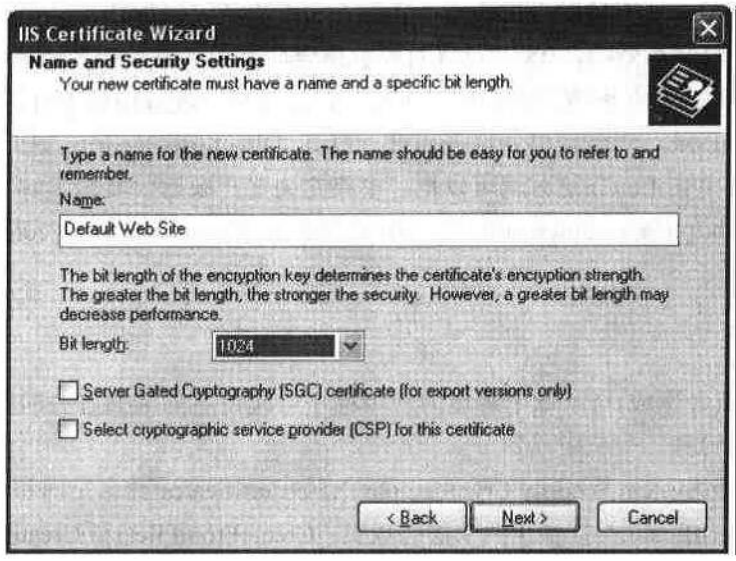

图　5-1

生成的文件保存为文本文件，但它最后必须通过电子邮件发送给证书颁发机构。证书请数据被自动加密。一个简单的(有些内容被缩减)请求文件示例如下：

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIB1DCCAT0CAQAwgZMxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhOZXcgWW9yazEQ
MA4GA1UEBxMHQnVmZmFsbzEeMBwGA1UEChMVVW5pdmVyc2l0eSBhdCBCdWZmYWxv
MRwwGgYDVQQLExNSZXNlYXJjaCBGb3VuZGF0aW9uMSEwHwYDVQQDExh3d3cucmVz
ZWFyY2guYnVmZmFsby5iZHUwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBALJO
hbsCagHN4KMbl7uz0GwvcjJeWH8JqIUFVFi352tnoA15PZfCxW18KNtFeBtrb0pf
-----END NEW CERTIFICATE REQUEST-----
```

证书颁发机构将返回一个证书，用户可以根据说明进行安装。按照惯例，用户应该通过端口号为 443 的端口运行所有的 SSL 通信，并通过端口号为 80 的端口运行普通的 Web 通信服务。

在默认情况下，用户不能把 IIS 和利用 makecert 生成的证书结合在一起使用。相反，建议用户从著名的证书颁发机构中购买证书，或者使用 Windows 2000 Server 或者 Windows .NET Server 的证书服务器。Microsoft 公司提供了逐步安装其知识库(http://support.microsoft.com/default.aspx?scid=kb;EN-US;q299525)中证书的具体步骤。

5.1.5　利用 SSL 对信息编码

只要安装了证书，就很容易使用 SSL 通信了。例如，看一下下面这个非常基本的 Web 服务。它提供了一个名为 TestSSL()的方法，该方法通过检验 HttpRequest.IsSecureConnection 属性可以验证 SSL 是否正在被当前连接使用。

```
public class SecureService : WebService
{
```

```
[WebMethod()]
public bool TestSSL()
{
  if (!Context.Request.IsSecureConnection)
  {
    // In this case, we throw a security error (because this is a
    // web service). In the case of a web page, you might just want
    // to redirect the user to a non-secure page.
    throw new SecurityException(
     "This method requires SSL.");
  }

  // (In a real web service, you would place your application
  // code here.)
  return true;
 }
}
```

另外一个步骤就是修改客户机的请求，使用 https://开头的 URL 代替 http://开头的 URL。在远程通信中，用户可以修改配置文件中的 URL。在 Web 服务中，可以修改所生成的代理类(该类是从 System.Web.Services.Protocols 命名空间的 WebClientProtocol 类中派生的)的 url 属性。

```
SecureTest.SecureService proxy;
proxy = new SecureTest.SecureService();
proxy.Url = "https://WebServer/SecureTest/SecureService.asmx";

// You can now send an SSL-encrypted message.
// This method will return true.
Console.WriteLine(proxy.TestSSL());

proxy.Url = "http://WebServer/SecureTest/SecureService.asmx";

// You can now send normal unencrypted requests.
// This method will generate an error.
Console.WriteLine(proxy.TestSSL());
```

该技术在处理使用身份验证的一些服务时非常有用，但不是对所有的方法都有用。在该示例中，用户只有在调用 Login()方法时才使用 SSL。要想在不对 URL 硬编码的情况下执行所需的 URL 操作，可以使用 System.Uri 类：

```
SSLTest.SSLService proxy = new SSLTest.SSLService();
Uri uri = new Uri(proxy.Url);

// Use SSL.
Proxy.Url = "https://" + uri.Host + uri.AbsolutePath;
```

```
// Use ordinary HTTP.
proxy.Url = "http://" + uri.Host + uri.AbsoultePath;
```

说明：

许多人经常会犯同一个错误，那就是利用本地主机或者其他别名作为 SSL 连接中服务器主机的名称。出现这样的错误后系统就不能工作，因为客户机在 SSL 交换的信号交换过程中会验证服务器证书的主题名称部分的公用名称(CN)是否与 HTTP 请求中发现的主机名称相符。因此，即便在进行测试的时候，也要使用服务器名称。

因为所有的加密和解密操作都恰好发生在传送消息之前(或者在消息被检索到之后)，应用程序不需要担心手工解密数据、处理字节数组、使用正确的字符编码方法等。

在服务器端，还可以实施 SSL 连接，因此在不对通信加密的情况下和 Web 服务发生交互作用是不可能的。只需在 IIS 管理器上右击 Web 站点，并选择 Directory Security 选项卡即可。在 Secure Communications 中，单击 Edit 按钮(只有在证书安装之后才可以使用)。然后选择 Require Secure Channe(SSL)，如图 5-2 所示。

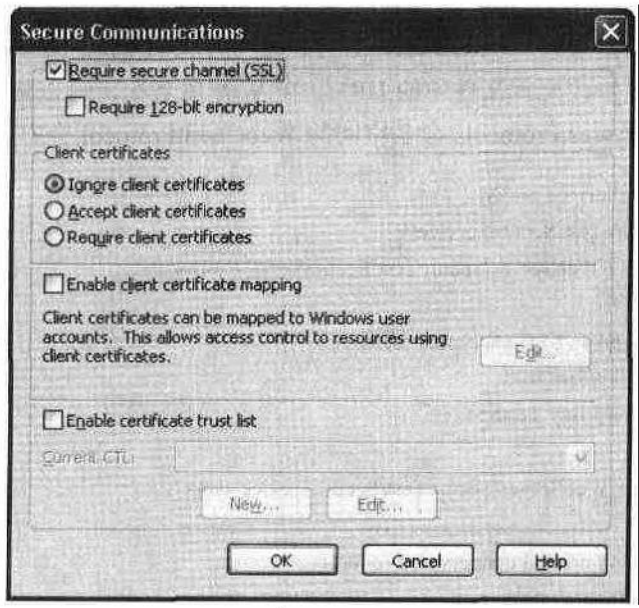

图　5-2

记住，我们有很好的理由不去实施 SSL 连接。例如，用户可能希望保护 Web 服务中的一些方法调用，而不仅仅是保护那些没有返回敏感信息的方法。这就允许用户提高性能并减少服务器执行的工作。

利用 SSL，所有传输数据都可以被加密，而不仅仅是对敏感数据加密。由于这个原因，许多 Web 服务器都利用一个硬件加速器改进利用 SSL 的加密性能。

说明：

有关其他形式的传输层安全的信息可以参考附录 A(主要讨论 IPSec)的有关内容。

5.2　应用层加密

使用 SSL 后，代码只需要进行少量的可选的修改，就允许用户检验当前通信是否加密。如果需要执行自己的选择性的加密操作，就需要编写大量附加的代码。但是用户可以按照这里讲述的设计模式，把加密逻辑移入一个专用的、可重复使用的程序集中来简化编码和维护操作。

在实现应用层加密之前，证明这样做是否有正确的原因是非常重要的。下面列出了两条不需要使用应用层加密的理由：

- 替换一个行业性强的身份验证或者加密协议(像 Kerberos 或 SSL)。
- 作为保护关键任务数据的最佳方法。

另一方面，用户希望利用应用程序加密来防止各部分信息成为大量不受保护的数据，或者在需要一个灵活性较高的解决方案时，发起攻击的威胁(和攻击成功后造成的后果)较小。这在攻击者利用特权访问网络的中间人攻击中尤为突出，保护代码不受中间人攻击是非常困难的。

说明：

我们并不是暗示基于应用程序的加密代码是无用的——其实它非常有用——但是对于加密客户机-服务器通信这项任务而言，使用 SSL 总是正确的。对于加密客户机上不用于服务器编程或者其他应用层加密任务的信息，需要利用.NET 的加密装置。

应用层编码只需利用.NET Framework，因此它可以处理几乎所有类型的分布式应用程序。从理论上看，应用层编码适合 ASP.NET Web 服务，因为它们包括其他使其易于实现的服务，例如缓存和对话状态。

首先我们要考虑如何利用非对称加密来保护发送给服务器的信息，然后讨论如何利用和对称加密的对话来提高其性能。在本小节的最后，我们将分析自定义方法所带来的风险，和一些可以使用的攻击类型。然后学习如何利用会得到增强的性能来修补这些安全漏洞，并期待在第 8 章中看到端对端的解决方案。

5.2.1　简单的非对称加密

下面示例说明的是一项简单的 Web 服务，该服务允许客户机用一种只有它才能解密数据的方法来加密数据。

```
public class SecuredService : WebService
{ ... }
```

Web 服务通过实例化 RSACryptoServiceProvider 对象，利用一种私有方法创建出一个新的公共-私有密钥对。该对象保存在 ASP.NET 应用程序状态下，它可以确保即使在 Web 服务不运行的情况下该对象仍然保留在内存中，并且只有在 Web 服务器重新启动或者循环利用应用程序域(可能为了适应最新编译成的 Web 服务程序集)时才删除它。用户可以利用从安全位置读取到的信息(例如第 6 章中讨论的本地证书存储单元或者智能卡)随意构建 RSACryptoServiceProvider 对象。

```
private RSACryptoServiceProvider GetKeyFromState()
{
  RSACryptoServiceProvider crypt = null;

  // Check if the key has been created yet.
  // This ensures that the key is only created once.
  if (Application["Key"] == null)
  {
    // Create a key for RSA encryption.
    CspParameters param = new CspParameters();
    param.Flags = CspProviderFlags.UseMachineKeyStore;
    crypt = new RSACryptoServiceProvider(param);

    // Store the key in the server memory.
    Application["Key"] = crypt;
  }
  else
  {
    crypt = (RSACryptoServiceProvider)Application["Key"];
  }
  return crypt;
}
```

注意：

我们还需要一些额外的步骤来访问密钥容器，因为 ASP.NET 进程被指定为一个非交互作用的用户。要想了解有关它的更多信息可以参考第 2 章的内容。

下一步，添加一个允许客户机检索密钥公用部分的方法：

```
[WebMethod()]
public string GetPublicKey()
{
  // Retrieve the key object.
  RSACryptoServiceProvider crypt = GetKeyFromState();

  // Return the private portion of the key only.
  return crypt.ToXmlString(false);
}
```

最后，可以创建一个需要加密信息的 Web 方法。最简单的方法是接受所有加密参数的原始二进制数据。这样，客户机就不能错误地提交原来未加密的值。通过表明方法中包含加密参数和表明适当的基本数据类型的 WebMethod 属性，可以添加一项描述，这也是一种很好的方法。目前还没有一种标准的方法可以标记这些参数。

```
[WebMethod(Description = "Parameters to this method " +
  "must be encrypted with the web service public key. " +
```

```
"encryptedAccountNumber is an encrypted String; " +
"encryptedAmount is an encrypted String (containing a decimal).")]
public bool DepositFunds(byte[] encryptedAccountNumber,
                         byte[] encryptedAmount)
{ ... }
```

Web 服务本身也可以包含下面的文档规范：

```
[WebService(Description = "All encrypted strings use " +
 "UTF-8 Unicode encoding")]
public class SecureService : WebService
{ ... }
```

最终结果如图 5-3 所示。

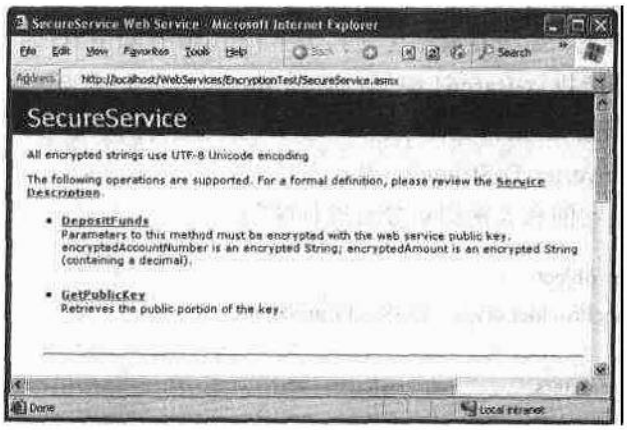

图　5-3

注意：

即便字符串数据类型是小数值，它也可以用于 encryptedAmount。这是因为它很难用一种标准化格式把小数编码为字节。最快捷的方式是把它们转换为字符串，然后再把字符串转换为字节。另外，许多相关的方法都使用 Decimal.GetBits()方法，它将返回一个包括 4 个整数的数组，分别表示小数值的不同信息。然后利用 System.BitConverter 类中重载的 GetBytes()方法把这些整数转换为字节。或者也可以使用支持小数类型的 BinaryWriter 和 BinaryReader 类。

现在客户机可以检索密钥，并在调用 DepositFunds() Web 方法之前利用它加密方法参数。

```
EncryptionTest.SecureService proxy =
  new EncryptionTest.SecureService();

// Instantiate a cryptographic object using the web service key.
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider();
crypt.FromXmlString(proxy.GetPublicKey());

// Create some sample parameters for DepositFunds().
```

```
string accountNumber = "000-M334";
string amount = "1000.42";

// Encrypt the parameters.
byte[] encryptedAccountNumber, encryptedAmount;
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
encryptedAccountNumber = crypt.Encrypt(enc.GetBytes(accountNumber),
  false);
encryptedAmount = crypt.Encrypt(enc.GetBytes(amount), false);

// Call the method with the encrypted parameters.
proxy.DepositFunds(encryptedAccountNumber, encryptedAmount);
```

因为信息是以字节数组形式发送的，所以.NET 将自动把它转换为所需的 Base64 编码形式插入 SOAP 消息中。但是，如果将其手工转换为字符串(没有利用 Base64 编码方法编码)，并发送该字符串，那么就不能执行 Base64 编码，而且 SOAP 消息中可能包含 XML 中不合法的特殊字符。因此，很少需要把加密数据当作字符串发送，建议在转换字节数组时用 Convert.ToBase64String()方法代替 BitConverter.ToString()方法。

最后，服务器使用它的私人密钥对参数进行解码：

```
// Retrieve the key object.
RSACryptoServiceProvider crypt = GetKeyFromState();

// Decrypt the parameters.
string decryptedAccountNumber;
decimal decryptedAmount;
System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();
decryptedAccountNumber = enc.GetString(
  crypt.Decrypt(encryptedAccountNumber, false));
decryptedAmount = Decimal.Parse(enc.GetString(
  crypt.Decrypt(encryptedAmount, false)));

// (Use decryptedAccountNumber and decryptedAmount.)
```

到目前为止，我们已经考虑了 Web 服务，但该技术很容易就能应用于通过.NET 远程控制展示的组件。惟一的差别在于需要使用一个单独的或者客户机激活的远程对象。这样在客户机访问过该对象之后，将继续存在于一个固定的生存期策略下。这样可以确保与组件相关的公钥不会改变。另外一种方法是从每个方法调用(也可能是硬件设备)开头的安全位置处读取公钥信息。这样就可以取代当前用来访问应用程序状态的代码，它只能通过 ASP.NET 实现。

这时可以通过扩充该方法来支持双向加密。客户机只需在调用方法时提交它的公钥。然后利用该公钥加密任何返回值。

```
public bool DepositFunds(byte[] encryptedAccountNumber,
  byte[] encryptedAmount, string clientKeyXml)
```

但是，非对称加密本来就非常慢——比对称加密要慢几百倍。扩充该模型的更好方法是利用对话和动态生成的对称密钥，它与 SSL 协议的操作方法非常相似。我们将在下一小节内容中考虑这种方法。

5.2.2　使用会话

SSL 引入了一个会话概念，在客户机通过 SSL 访问页时就会开始一次会话。在会话过程中，所有通信都受同一个随机生成的对称密钥的控制。使用选择性加密时，必须遵循同样的设计。这样可以减少开销，因为它能够确保在首次处理共享的密钥时，只需使用非对称加密操作一次。之后所有的交互作用将使用对称加密。

要想使这种设计模式起作用，服务器必须把一些基本信息存储在内存中：当前客户会话列表和它们正在使用的对称密钥。在 ASP.NET Web 服务中，可以把该信息保存在会话状态下或者缓存中。ASP.NET 进程仍然会保留信息，并允许 Web 服务类以无状态的形式执行。在远程通信组件中，可能需要采取额外的步骤来保存会话信息，例如将其保存在一个后台数据库中，或者使用一个客户机启动的对象，以使每个远程对象都只为一个客户机服务。这些方法都不是最好的，数据库存储将把附加的开销施加在每个方法调用中，而客户机启动的对象也不适合所有的环境，并且不像无状态实现那样健壮。

说明：

基于会话的加密是说明如何实现层中安全的最好示例。因为对称密钥只能在短期内使用，在恶意用户使用蛮力攻击它的时候就不再有效。同样，如果恶意用户试图捕获并稍后再次发送同一则消息时(即“重放”攻击)，攻击将失败，这是因为密钥已经改变。

在基于会话的系统中，经常会利用客户机所调用的特定方法来启动会话。在一般情况下，该方法指的是验证客户机所提供的证书(例如用户名和密码)的 Login()方法。该方法也会返回一个票据(ticket)，用来惟一地标别会话。该票据通常是一个由 Web 服务利用 Guid.NewGuid()方法创建的 GUID。对于其他的会话，客户机将在每次方法调用时重新提交票据。Web 服务将寻找相应的对称密钥信息，并利用它来解密所提交的全部参数。因此票据可以用来为会话验证客户机，以及跟踪服务器上的密钥信息。

服务器端组件用来保存票据和密钥信息的方法完全由您做决定。但是，下面这三种方法是最常用的：

- ASP.NET 的应用程序状态。这是最简便的一种方法。但是需要采取额外的步骤确保在会话结束时信息已经被删除。如果没有，就会浪费大量的服务器内存。
- ASP.NET 的内置会话装置。在这种情况下，信息只能由指定的用户利用自动生成的 cookie 进行识别。但是如果 Web 服务客户机被配置为不接受 cookie 并在后面的请求中重新提交它，信息就会丢失。在会话状态下，会话的超时取决于 web.config 的设置，通常将会话的有效时间限定为 20 分钟。
- 一个能够随意和缓存结合的数据库。这样就提供了能够长期保存数据的存储器(如果需要的话，还有可以保存大量信息的存储器)。通常情况下，要把密钥插入到 ASP.NET 的数据缓存中。在查找密钥时，可以先利用缓存，只有在找不到所需的信息时才查询数据

库。在没有把信息存储在其他更长久性的存储器(例如数据库)中时，不应该使用缓存。因为在服务器内存不足时，信息将被删除，即便只超出了 1 秒。

本章我们将利用第一种方法，但是也可以通过简单地修改代码来使用其他方法。假如您允许会话持续较长的时间(例如几天)并仍然具有高度的可伸缩性，就可以使用第三种方法。牢记在这种情况下，必须注意数据库的创建日期或者期满日期，并通过编程方式确定会话在什么时候到期。

5.2.3 使用会话和 Web 服务

下面这个 Login()方法示例把这些概念都集中在一起。它可以接受一个已经利用公用 Web 服务密钥加密的对称密钥。对该密钥解密，将其保存在应用程序状态下，并利用随机生成的 GUID 值建立索引。

```
[WebMethod(Description = "The encryptedClientKey should be a " +
  "secret value for Rijndael encryption. It should be encrypted " +
  "using the web service public key.")]
public string Login(byte[] encryptedClientKey)
{
  // Retrieve the server key.
  RSACryptoServiceProvider serverKey = GetKeyFromState();
  SymmetricAlgorithm clientKey = Rijndael.Create();
  clientKey.Key = serverKey.Decrypt(encryptedClientKey, false);

  // The initialization vector is not used.
  clientKey.IV = new byte[clientKey.IV.Length];

  // Create a new ticket.
  string ticket = Guid.NewGuid().ToString();

  // Store this key in application state.
  Application[ticket] = clientKey;

  return ticket;
}
```

为了简化操作，这个 Login()方法并不要求任何附加的用户信息，在创建会话之前也不执行用户身份验证。这样就将重点放在数据的安全性上，而不是用户的身份验证服务。但是，在实际操作中，有可能同时执行这两种任务，那么登录方法可以采取下面的形式：

```
[WebMethod(Description = "The encryptedClientKey should be a " +
  "secret value for Rijndael encryption. It should be encrypted " +
  "using the web service public key.")]
public string Login(byte[] encryptedUserName,
  byte[] encryptedPassword,  byte[] encryptedClientKey)
{
```

```
    // (Decrypt the user name and password.)

    // Verify that the user name and password are in the database
    // using a private member function.
    if (ValidateUser(decryptedUserName, decryptedPassword))
    {
      // (Create and store the ticket here.)
    }
    else
    {
      throw new SecurityException("Unrecognized user. " +
        "Cannot start session.);
      return null;
    }
  }
```

除此之外，您可能希望把聚集了多条客户机信息(例如，对称会话密钥和用户名)的自定义对象保存在应用程序状态下。

应用程序状态信息只有在应用程序重新启动时才发布。因此为了删除服务器端的信息，在会话结束时应该添加一个对应的 Logout()方法供客户机调用。

```
[WebMethod()]
public void Logout(string ticket)
{
  Application[ticket] = null;
}
```

当然，如果网络出现问题，禁止客户机调用 Web 服务并终止会话，或者当客户机疏忽了它的责任时，就不能调用 Logout()方法。这是利用应用程序状态存储这种类型的信息时所出现的一种问题。解决该问题的方法是，根据 Web 服务的通信量，通过设置 machine.config 选项，再循环利用应用程序域，或者切换到另外一种方式把密钥信息存储在缓存和后端数据库中，而不是应用程序状态。

下一步，修改 Web 服务中的其他方法来接收票据参数。注意，票据参数从来都是不加密的。

```
public bool DepositFunds(byte[] encryptedAccountNumber,
  byte[] encryptedAmount, string ticket)
{
    // Validate ticket.
    SymmetricAlgorithm clientKey = (clientKey)Application[ticket];

    if (clientKey == null)
    {
        throw new SecurityException("Invalid ticket.");
    }
```

```
        // (Application specific logic goes here.)
    }
```

说明：

可以利用 SOAP 报头或者在远程通信情况下利用一个调用上下文装置来简化设计。这时每个方法调用都会自动提交票据，而且不需要把它当作一个独立的方法参数添加。要想了解有关这些内容的更多信息，可以参考有关 Web 服务或者远程通信方面的专业书籍。

下面的代码示例可以说明客户机如何与服务器发生交互作用：

```
EncryptionTest.SecureService proxy =
  new EncryptionTest.SecureService();

// Instantiate a cryptographic object using the web service key.
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider();
crypt.FromXmlString(proxy.GetPublicKey());

// Create a random symmetric key.
SymmetricAlgorithm clientKey = Rijndael.Create();

// The initialization vector is not used.
clientKey.IV = new byte[clientKey.IV.Length];

// Start a new session by sending an encrypted copy of the random
// symmetric key to the server.
string ticket = proxy.Login(crypt.Encrypt(clientKey.Key, false));

// Create some sample parameters for DepositFunds().
string accountNumber = "000-M334";
string amount = "1000.42";

// Encrypt the parameters with the symmetric key.
byte[] encryptedAccountNumber, encryptedAmount
encryptedAccountNumber = this.EncryptString(accountNumber, clientKey);
encryptedAmount = this.EncryptString(amount, clientKey);

// Call the method with the encrypted parameters.
proxy.DepositFunds(encryptedAccountNumber, encryptedAmount, ticket);
```

使用对称加密时，必须对 CryptoStream 进行写操作。使用 Web 服务或者远程通信时，不能直接访问请求和响应流。相反，必须在发送数据之前利用内存中的流对其加密。为了简化该代码，可以创建并使用名为 EncryptString()的帮助函数，其代码如下所示：

```
private static byte[] EncryptString(string text,
 SymmetricAlgorithm crypt)
```

```
{
    UTF8Encoding enc = new UTF8Encoding();

    // Create a memory stream.
    MemoryStream ms = new MemoryStream();

    // Encrypt information into the memory stream.
    CryptoStream cs = new CryptoStream(ms,
      crypt.CreateEncryptor(), CryptoStreamMode.Write);
    StreamWriter w = new StreamWriter(cs);
    w.WriteLine(text);
    cs.FlushFinalBlock();

    byte[] encryptedBytes = new byte[ms.Length];
    ms.Position = 0;
    ms.Read(encryptedBytes, 0, ms.Length);
    ms.Close();

    // Return the final byte array of encrypted data.
    return encryptedBytes;
}
```

服务器可以执行 DepositFunds()中相同的步骤来解密参数，不过这次是利用 DecryptString()方法：

```
// Retrieve the key object.
SymmetricAlgorithm clientKey =
  (SymmetricAlgorithm)Application[ticket];

// Decrypt the parameters.
string decryptedAccountNumber;
decimal decryptedAmount;

decryptedAccountNumber = DecryptString(encryptedAccountNumber, clientKey);
decryptedAmount = DecryptString(encryptedAmount, key);

// (Use decryptedAccountNumber and decryptedAmount.)
```

DecryptString()方法的代码如下：

```
private static string DecryptString(byte[] encryptedText,
 SymmetricAlgorithm crypt)
{
  System.Text.UTF8Encoding enc = new System.Text.UTF8Encoding();

  // Create a memory stream.
```

```
    MemoryStream ms = new MemoryStream();
    CryptoStream cs = new CryptoStream(ms, crypt.CreateDecryptor(),
                                       CryptoStreamMode.Write);

    // Place the array of bytes into a memory stream.
    cs.Write(encryptedText, 0, encryptedText.Length);
    cs.FlushFinalBlock();

    // Decrypt and display the data.
    StreamReader r = new StreamReader(ms);
    ms.Position = 0;
    return r.ReadToEnd();
}
```

其效果和前面引用的非对称示例的效果相同，但是代码的执行速度更快，可以减轻服务器上的负担。我们将在第 8 章中重新利用该方式来构建一个更加完整的示例。但是，首先要探讨它所带来的一些安全风险。

5.2.4 自定义方法中的安全风险

到目前为止，我们自定义的加密解决方案的行为与 SSL 有些相似，但重要的是要记住，它并不是 SSL。尤其是当实现遭受一些严重的安全漏洞破坏时。有些问题很容易解决，而其他一些却非常复杂(最好通过 SSL 实现)。在下面的内容中，我们将详细分析这些问题，并且将在第 8 章中利用大量附加的代码解决它们。您会发现把我们基本的身份验证方法和更健壮的行业标准(例如 Kerberos)进行比较是非常有意义的。可以参考 http://web.mit.edu/kerberos/www/dialogue.html 中对 Kerberos 以及它所解决的安全问题的讲解，这些内容都是以一种独特的对话形式编写的。

1. 口令和散列

现在用户的整个口令被加密并通过连接发送出去。口令以明文形式存储在数据库中，这样做不仅很容易输入和更新，而且很容易成为任何有权访问服务器的攻击者的目标。一种较好的办法是把口令的加 salt 值的散列值存储在数据库中。我们首次在第 4 章中引入了该技术，在第 8 章的端对端示例中将其和自定义的 Web 服务方法结合在一起。

2. 可猜测的 GUID 票据

目前，我们把 GUID 当作票据使用。问题是创建 GUID 的算法还没有得到加密保护。换句话说，Guid 类允许生成惟一的 GUID，但是它不能保证攻击者不会根据 GUID 的生成算法及时预测出在特定时间点上生成的 GUID。有关该主题的内容将在附录 B 中详细介绍。

要想解决该问题，可以用任何从 RNGCryptoServiceProvider 类中(以加密的随机格式)派生的随机字节构成的字符串来代替 GUID。利用 16 字节(和 GUID 相同的数字)来防止发生冲突是一个好办法。可以把随机字节以 Base 字符串的形式(利用 Convert.ToBase64()方法)或者 16 进制字符串的形式(利用 BitConverter.ToString()方法)返回给客户机。

```
// Create a new ticket.
byte[] ticketBytes = new byte[16];
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
rng.GetBytes(ticketBytes);
string ticket = BitConverter.ToString(ticketBytes);
```

另外，仍然可以使用 GUID，但是要基于随机数据来创建。这就是在第 8 章中使用的技术。

```
// Create a new ticket.
byte[] ticketBytes = new byte[16];
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
rng.GetBytes(ticketBytes);
Guid ticketGuid = new Guid(ticketBytes);
string ticket = ticketGuid.ToString();
```

3. 消息身份验证

加密使攻击者很难解密消息，但它不能阻止破坏消息。目前，Web 服务缺少能够防止这种类型的消息被篡改的方法(在被篡改的数据不能成功解密并转换为所需的数据类型时就会出现运行库错误)。有一种更健壮的方法是使用消息身份验证代码来为已加密的数据签名。在这种情况下，需要把所有数据合并在一个加密的包中，或者分别对数据的每一个加密部分签名。

HMAC 密码散列算法是消息身份验证散列码的一个好选择。可以利用该算法连接会话密钥。第 3 章对该方法做了详细的介绍，我们将在第 8 章中引用一个完整的 Web 服务示例对其进行说明。

4. 重放攻击

如果攻击者能够截获登录消息并将其保存到磁盘上，那么这种专业类型的攻击就有可能支持自定义的 Web 服务。然后攻击者重新使用该消息在稍后的时间点上进入服务。可以避免这种风险的一种方法是对与创建包的日期和时间相关的数据以及包本身加密。然后服务器可以验证该数据是不是所期望的数据。将在第 7 章中详细介绍该方法，而且在第 8 章中会引用一个完整的示例进行说明。

5. 身份和中间人

Web 服务示例中最难以解决的问题之一是对身份验证和身份的询问。例如，如果攻击者可以重新连接网络并通过假装成服务器而引入中间人攻击，会出现什么情况呢？在客户机调用例如 GetPublicKey()这样的方法时，就无法知道它是不是和真正的服务器进行通信。数据仍然被加密，但利用的是攻击者的公钥！

正如我们在本章开头讨论的那样，解决这类身份问题的惟一方法是依靠第三方，例如证书颁发机构。遗憾的是，即便在首次创建对话时就把已签名的证书数据传递给客户机，客户机也没有简便的方法可以验证该信息，因为.NET 没有提供用来验证证书数据的 API。这种缺少真正的身份验证操作是自定义加密方法的主要问题之一，这也是它和 SSL 协议之间的主要区别之一。当然，可以在风险估计中确定像这样的攻击相对来说是不可能的——在实际操作中，偷听、假

冒、故障利用或者拒绝服务等攻击都不常见。

解决该问题的一个方法是在不需要证书颁发机构帮助的前提下支持客户机执行它自己的密钥验证。例如，每个客户机都可以存储一系列可信的服务器密钥。如果它检索到一个不属于该列表的密钥，可以决定不开始会话。服务器还可以利用相同的方法验证客户机。这也为消息完整性验证提供了一个好的解决办法，因为每则消息都可以利用适当的非对称密钥数字签名。

说明：

自定义加密的目标不是从头开始重做什么。自定义加密代码对于特定的应用程序非常灵活而且可定制，但它不能复制 SSL。

6. 对象和串行化

目前，该项 Web 服务只支持可以直接转换为字节的简单数据类型。更具扩展性的好方法是利用对象的串行化并加密结果生成的二进制数据，如第 4 章所述。我们将在第 8 章中利用该方法加密完整的对象。自定义对象还允许用代码存储更多的客户机身份验证信息。

5.2.5　利用会话进行远程通信

正如前面所说明的那样，远程通信中不包括任何用来保存共享内存的内置装置。这就意味着不能使用在前一小节中远程使用无状态(SingleCall)组件的方法。但是，可以轻松地对它使用客户机激活的对象。在这种情况下，Login()方法就可以把对称密钥保存在一个类成员变量中。集合不是必需的，因为只有一个用户才可以和一个客户机激活的对象的任何给定实例发生交互作用。但是需要特别关注生存期租用的配置。一旦服务器端的对象被撤销，客户机就需要重新创建它，并调用 Login()方法开始一次新的会话。

如果需要的话，还可以使用一种基于会话的方法来处理单独的对象。在这种情况下，每个客户机都使用同一个实例，而且需要把客户机信息保存在一个散列集合中，该集合将作为远程对象的成员变量包含在内。但是，该方法是最复杂的一种方法。需要添加锁定码以确保不同线程上的多个用户不能同时访问集合。

在使用远程通信时，不需要依靠字节数组传递被编码的数据。实际上，可以利用一个能够存储数据并具有加密或解密功能的自定义对象。可以利用前一章引用的示例来传递安全信息：EncryptedPackage 对象。通过把解密逻辑集中在一个类中，就可以避免出现编码错误和其他麻烦。出现问题的惟一原因是如果客户机和服务器使用的是同一对象的不兼容版本，并且禁止严格的版本检验。但是，该方法不能在 Web 服务模式下使用，因为 Web 服务只支持几种常见的数据类型。可以创建一种自定义的类型在 Web 服务中使用，但是客户机只能检索到带有公共数据成员的结构。所有的方法、构造函数和属性过程代码都将被忽略。

5.2.6　密钥交换类

在进行分布式通信时使用密码术最常见的错误之一是不能正确地执行密钥交换。如果不能适当地生成一个随机值，或者不能正确地编码保密值信息，就会损害到整个会话。为了解决这些问题，.NET Framework 中包含一些特殊的专门用来确保安全密钥交换的密钥交换类。

使用 RSA 公钥时，有两种方法可以交换密钥，它们都派生自 AsymmetricKeyExchangeFormatter：

- RSAOAEPKeyExchangeFormatter 利用 OAEP(Optimal Asymmetric Encryption Padding)来创建加密的密钥交换数据。RSAOAEPKeyExchangeDeformatter 可以解密该密钥交换数据。
- RSAPKCS1KeyExchangeFormatter 利用 PKCS#1 填充法创建加密的密钥交换数据。RSAPKCS1KeyExchangeDeformatter 可以解密该密钥交换数据。

这些密钥交换格式程序都可以提供可以随机生成会话密钥的 CreateKeyExchange()方法，然后利用非对称加密技术对其加密。惟一所需的信息是用来加密随机对称密钥的公钥。在生成密钥交换数据之前可以调用 SetKey()方法来指定密钥。

下面是一些修改过的客户程序代码，它可以利用 Login()方法提交会话密钥。现在，密钥数据不能手工创建和加密，但是可以直接从 RSAPKCS1KeyExchangeFormatter 中导出。

```
EncryptionTest.SecureService proxy =
  new EncryptionTest.SecureService();

// Instantiate a cryptographic object using the web service key.
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider();
crypt.FromXmlString(proxy.GetPublicKey());

// Instantiate the key exchange formatter.
RSAPKCS1KeyExchangeFormatter exchange =
  new RSAPKCS1KeyExchangeFormatter();

// Specify the public key to use for encryption.
exchange.SetKey(crypt);

// Create a random symmetric key.
SymmetricAlgorithm clientKey = Rijndael.Create());

// Create the key exchange data.
byte[] exchangeData = exchange.CreateKeyExchange(clientKey.Key);

// Start a new session by sending the key exchange data.
proxy.Login(exchangeData);
```

在服务器端，Login()方法必须使用对应的反格式程序检索密钥交换数据中的秘密值。

```
[WebMethod()]
public string Login(byte[] encryptedClientKey)
{
  // Instantiate the key exchange deformatter.
  RSAPKCS1KeyExchangeDeformatter exchange =
    new RSAPKCS1KeyExchangeDeformatter();
```

```
        // Specify the server key.
        exchange.SetKey(GetKeyFromState());

        // Decrypt the symmetric key.
        SymmetricAlgorithm clientKey =
          (SymmetricAlgorithm)(new RijndaelManaged());
        clientKey.Key = exchange.DecryptKeyExchange(encryptedClientKey);

        // Create a new ticket.
        string ticket = Guid.NewGuid().ToString();

        // Store this key in application state.
        Application[ticket] = clientKey;

        // Return the ticket.
        return ticket;
    }
```

注意:
该方法仍然有前面所指出的安全弱点。

5.3 高级选项

本章讨论了两个基本层的加密支持。在传输层中，可以使用 SSL 在应用程序外部对数据加密和解密，而不需要考虑密码操作的具体情况。如图 5-4 所示。

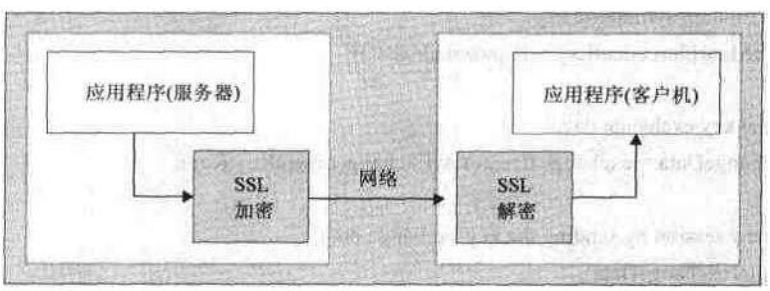

图　5-4

本章介绍的其他方法是应用层加密，其中所有的操作都是在代码的监督下执行的，如图 5-5 所示。

在这两层之间还有一些附加的选项。这些都是手工执行加密操作的普通.NET 代码模块，但它们是为大多数与应用程序没有任何交互作用的部分工作的。两个经常引用的示例是:

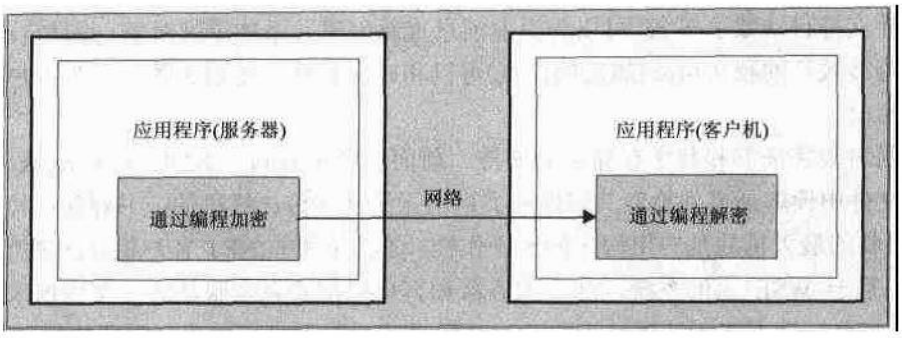

图　5-5

- 自定义的 SOAP 扩展，它只能在 SOAP 消息发送之前自动加密或者在收到消息之后自动对其解密。由于已经发布了 WSE，所以不经常使用该方法。
- 自定义的远程信道接收器，它只能在消息发送之前对其加密或者在收到消息之后对其解密。该技术在 C# Web Services: Building Web Services with .NET Remoting and ASP.NET, ISBN 1-86100-439-7 一书中做了详细的介绍。

这些示例都指定了所需的技术，而且必须在客户机和服务器端运行。它们都具有重要的专业用途(例如，不使用 IIS 就把透明的加密技术添加到远程通信中，或者自动运行自定义的算法，例如组合的加密、压缩和验证)。这些方法还需要许多操作，并且要把您的解决方案和指定的技术紧密结合。通常情况下，它们需要对整个信息加密，这就意味着它们的执行速度可能要比选择性加密慢。最大的危险是不能安全地实现这些安全防卫措施。多数攻击都不使用蛮力或者指定加密算法中的弱点，但是它依赖于程序上的问题(例如，不安全的数据库中一系列未加密的密码)，或者实现加密操作的软件中存在的漏洞。如果您设计一种自定义的 SOAP 加密方法或者远程加密信道，就可能意外地将一个密钥值暴露在内存中，或者允许未经处理的错误来展示敏感的未加密信息。

由于这些原因，作者强烈推荐不要试着独立开发这部分基础措施，而把它留给公司(例如 Microsoft)的专家来完成。安全问题是非常复杂的，在独立开发一部分安全防卫措施时，很容易留下许多漏洞，尤其是在有时间约束和资源有限的条件下编程。

WSE 和 SOAP 安全的未来

用于把请求和响应消息发送给.NET Web 服务的 SOAP 标准发展得非常迅速。许多被提议的扩展仍然还没有合并在.NET Framework 中，包括 WS-Security。随着这些标准越来越具体，Microsoft 公司正在设法提供一种允许开发人员简单且安全地使用它们的对象模型。其中一种工具就是针对 Microsoft .NET 安全性的 Web Services Enhancements(WSE)，该工具可以从 http://msdn.microsoft.com/webservices/building/wse 处下载。WSE 了解利用.NET 进行的安全 Web 服务编程的前景，而且将来可能被并入核心框架中。

当前，WSE 被结构化为一组对 SOAP 请求和响应消息进行转换的扩展。它们可以生成 WSE 需要安装的自定义 HttpModule。在安全区域中，这些扩展包括许多有意义的功能：

- 支持自动加密。可以通过 WS- Security 报头设置加密参数，而且 SOAP 消息的正文会被自动加密。

- 同样支持自动数字签名。可以为当前消息重新创建并添加令牌对象，并且自动进行签名。
- 支持被散列的秘密值(例如密码)，它可以和时间戳结合使用来防止恶意用户截取和重用秘密值。
- 增强对数字证书和数字存储器的支持。前面已经了解到，.NET framework 提供了可以从文件中读取证书并检索其属性的类，但无法从当前计算机的证书存储中检索它们。

我们面临的最大挑战是应用程序的互操作性，尤其是和非.NET 客户机之间的互操作性。目前，如果要基于 WSE 上的系统，那么服务器和客户机都要完全服从同一个协议来成功地处理加密或验证操作。现在还无法通过读取 SOAP 消息的元数据来自动确定需要执行的步骤。

5.4 小结

本章深入探讨了在分布式环境下进行安全通信时需要使用的技术。您已经看到如何利用 SSL 或者.NET 加密类来保护通过连接发送的敏感数据。虽然本章大部分内容都集中在远程通信和 Web 服务上，但通过任何协议都可以应用相同的技术。

要实现安全的分布式编程，最重要的是您首先要计划好项目的初期阶段。引入加密技术可以改变服务器端组件的接口，它可以请求不同的部署策略，而且在某些情况下，可能需要开发安全防卫措施的某一特定部分，例如自定义的加密信道。这些具体内容可以(而且必须)利用共享的组件和负责的组织控制，否则就会搞乱应用程序的指定业务部分。将来，随着 Web 服务的发展，Microsoft 公司将把新的功能集成在.NET Framework 中，这些安全任务可能变得更加透明，而且所需的自定义代码也越来越少。但是现在它们仍然需要开发人员大量的时间投入和努力。

第 6 章将继续深入探讨有关证书和信任链(trust chain)的内容，而第 8 章将引用一个端对端的 Web 服务示例来说明如何处理自定义代码中加密技术的一些复杂操作。我们还将讨论如何进一步防御在第 7 章中讨论到的攻击。

第6章　密钥和证书管理

密钥管理是密码术的一个最为重要的方面，尽管它通常不易于理解。如果加密信息的密钥不是安全的，那么该信息最好是公开的。签名数据也是如此，因为如果没有将私人密钥放在安全的地方，那么它的所有者很容易就否认其责任，而不管数字签名。

有一个词可以表示本章的核心，那就是 trust(信任)。它有着重要意义，如果出现在不同的上下文中，那么很难作解释，它是本章最重要的一个词汇。很显然您信任使用的技术(如果不想发明您自己的)和(假定)信任本书的内容。这并不是本章要介绍的；本章通过实现加密技术和发布实践声明，讨论如何运用加密技术使密钥安全，如何可以信任其他人以及如何让其他人信任您。

说明：

加密技术并不能建立双方的信任；进行声明以及人之间的交流也是必要的。

本章将向读者介绍如何使用.NET 环境将密钥存储在 Windows 中和如何访问它们。还将讨论下列内容：

- 数字证书的概念
- 什么是证书
- 证书的作用
- 如何管理证书

6.1　数字证书

数字证书的作用就是为所有数字事务处理构建一个统一平台，其中信任是每个事务处理的一部分。证书包括它所表示的实体的信息，这不一定是人；信息就包括所有者是谁(什么)、实体所属的组织、从何处签发证书、有效日期、公共密钥、密钥长度、密钥用法等。尽管有多个证书规范，但只有一个是全球认可的，即 X.509 规范。X.509 标准已发展到第 3 版了，这是目前所遵循的。从http://www.ietf.org/rfc/rfc2459.txt的 RFC 2459 中可获取更详细的信息。

证书的一个常见用法就是用作服务器证书；Web 服务器使用这些确保浏览器和 Web 服务器之间的信道安全。如果访问一个具备服务器证书的站点，例如https://www.microsoft.com，使用 Internet Explorer 将看到一个小的黄色锁图标出现在状态栏中，如图 6-1 所示。

图 6-1

如果双击该锁，将得到图 6-2 所示的结果，显示了有关证书的信息。

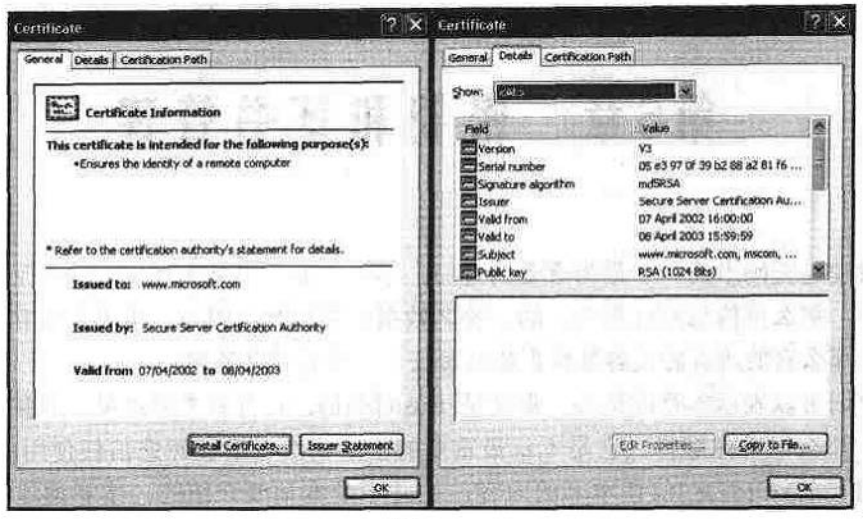

图　6-2

这就是确保用户和服务器之间通信安全的实际证书。由于所有的 X.509 证书都基于公共密钥加密原则，证书本身只是一个公共对象，从中可找到所有者身份、发行者和公共密钥。每个证书都有一个相应的私人密钥(像公共密钥加密一节中介绍的)，它不和证书一起分发；它由所有者保管。如图 6-3 所示。

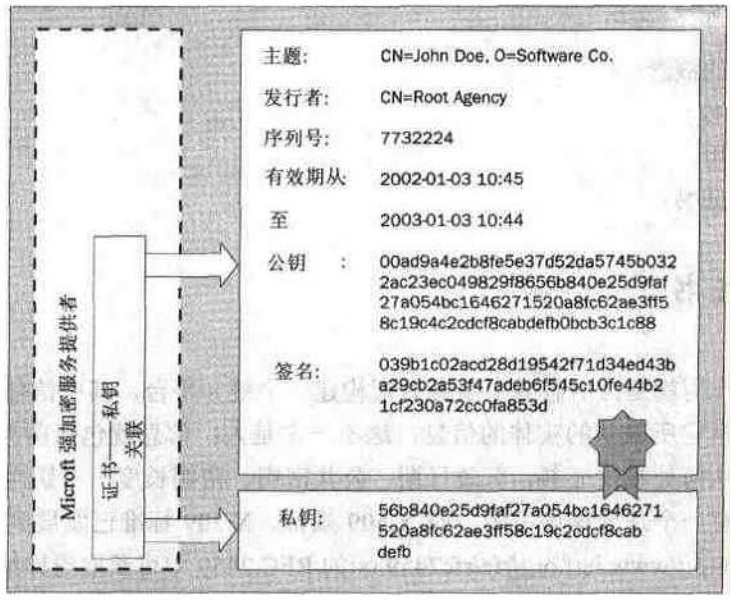

图　6-3

尽管公共密钥和私人密钥之间存在着数学关系，证书和私人密钥之间的关联是在客户机上显式指定的，例如通过 Microsoft Cryptographic Service Provider (CSP)接口。用户一般是不知道这种关联的。

第 3 版的 X.509(X.509v3)规范包括了许多标准字段，但它也允许您使用扩展格式化证书的

其余部分。凡事有好的一面，也有不好的一面。好的方面就是可以在特定上下文中研究 PKI 的全部潜能，例如包括所有者图像。不好的就是其他任何人都不能解释您的特定格式。现在已出于致力于指定这些字段的明确重要性的国际性的努力(像 RFC 2459，http://www.ietf.org/rfc/rfc2459.txt)，至少如果您准备接受其他根证书时，那么在您的组织中可能就必须采用这些规范。

6.1.1　一般用途

在创建证书时，也同时定义了其用途。并不是所有的证书都创建为服务器证书，还有其他用途，例如使邮件安全(S/MIME)，客户机身份验证，签名(不可否认的)，标记名称的代码(但只有少量的)。

1. 安全的电子邮件证书

S/MIME 标准定义了证书如何保护通过 Internet 发送的电子邮件(RFC 2632, 2633 和 2634)，许多电子邮件客户软件厂商已采用了它，例如 Microsoft Outlook 或 Netscape Messenger。

那就需要一个电子邮件证书，可从 Thawte (http://www.thawte.com)获取一个免费的证书。在接收和安装证书后，必须将电子邮件客户软件配置为使用它。

我们可以用新的证书为所有发出的邮件签名，但这并不包括秘密。如果将消息签给 Alice，她就可以加密她给您的消息，因为她已接收到您的私人密钥。为了能够加密发给 Alice 的消息，她必须获取证书，标记发给您的消息，接着您可以有她的公共密钥，电子邮件客户软件将可以加密给她的所有消息。

2. 客户机身份验证

这一般由希望通过使用双向 SSL(也就是服务器和客户机有一个使它们之间信道安全的证书)提高安全性的 Web 站点使用。例如，Internet 银行可以将证书发给它的在线客户，配置其 Web 站点，这样客户必须提供他们的证书来访问该站点。

可在服务端从客户机证书中获取身份，这使得服务器可以相信客户机为一个特定的客户。然而，如果发行的证书存储在软件中，没有对私人密钥的口令保护(这是很经常的事，因为这是最不昂贵的实现)，很有可能(例如)站点将使用 ID 和 PIN 实现基于证书的客户机身份验证。

3. 不可否认的签名

这些证书可用于标记的法律关系的事物，如协议或合同，它们需要强(strong)信任。强信任意味着证书根是众所周知的，完全可信任的，发行者的策略和实践包括保护证书主题(证书的接收者)的标识符。

不过，还必须信任终端用户证书存储，这一般意味着这些证书现在必须存储在硬件令牌上。例如，存储在软件中的证书在一个开放的公司里并没有提供充分的保护，其中许多计算机每天都会有几分钟的时间无人看管，而这点时间对于滥用其他人签名的人来说已经足够了。

4. 代码标记

证书的使用日益流行，因为它阻止用户安装从 Internet 下载的有恶意的软件。通过使用代码签名证书，软件厂商可以简化它们的软件发布，因为下载他们软件的终端用户可以放心地安

装它。

软件厂商只是用可信的证书来对代码进行标记，接着用户可以在安装它前检验该签名。

Microsoft 已在其浏览器和软件开发工具中实现了这一技术，即 Authenticode。这使得程序集在创建后可以自动进行标记，Internet Explorer 将自动检验所有下载的软件的签名。如果下载了没有签名的软件，那么它将警告您它可能含有有恶意的软件。

6.1.2 实际的 PKI 或身份管理

现在的 PKI 工作组是野心勃勃的，他们计划解决使用 Internet 而引起的所有身份问题，比如确切地知道给您发送邮件的人，或者是来自 Joe 软件商场的 Joe Smith 从您的网上比萨递送商店订购了 200 美元的比萨。遗憾的是，这些努力都是无足轻重的，这意味着很难让那些不在 PKI 核心集团中的人理解规范的真正含义。因此，这里尝试讲述 PKI 的最为重要的方面：

说明：
简写的 PKI 越来越不流行；现在的专用术语是身份管理。

1. 信任

身份管理是指不同上下文中的信任。例如，这一技术使得您可以信任协议中的内容，或让 Internet 银行知道您帐户上的所有事务处理是由您而不是其他人做的，可以知道证书背后的身份是它所声明的现实生活中的人，可以信任数字签名，因为您知道协议或委托是由现实生活中的人制定的。提供这种信任的技术如果不管上下文，那么就是一样的；可信的第三方的实践和策略决定了信任的强度(strength)。

说明：
技术不决定信任的强度；决定强度的是可信的第三方的实践和策略。

2. 技术

它的技术原理是什么？从技术上讲，信任构建于根证书(或证书根)基础之上。这一证书可以发行其他证书，像终端用户证书或授权证书(也叫作 CA 证书)。CA 证书可以依次发行证书，这样就可以创建一个信任链。终端用户证书安全地链接到根证书，只要根证书是可信的，链上的所有证书都可信。这就是为什么全球性品牌的证书根经常放在高度安全的地方。

该链是由签发子证书的与父证书相关的私人密钥建立的，因此只要根据父证书的公共密钥来检验包括在子证书中的签名，就可以根据父证书对子证书进行检验。那么是谁签发根证书，根证书有没有父证书？答案就是根证书用它自己的私人密钥签名，一切也就那么简单。

在图 6-4 中，名称 CA01 和 CA11 并没有表示任何特别的东西，它们只是中间的证书授权机构的示例名称。在实际生活中，终端用户证书可以作为您的一个雇员；CA11 可以是 CA 服务器部门，CA01 是法人级别的证书服务器，证书根是众所周知的 CA，像 Verisign 或 Thawte。

根据签发的证书中的公共密钥对每个证书中的签名进行验证，可以很容易地验证这信任链的完整性。最后的签名是根据它自己的证书进行验证，因为它是个根证书，因此是自签名的。通常，验证过程也包括验证有效日期，有时也包括密钥用法。

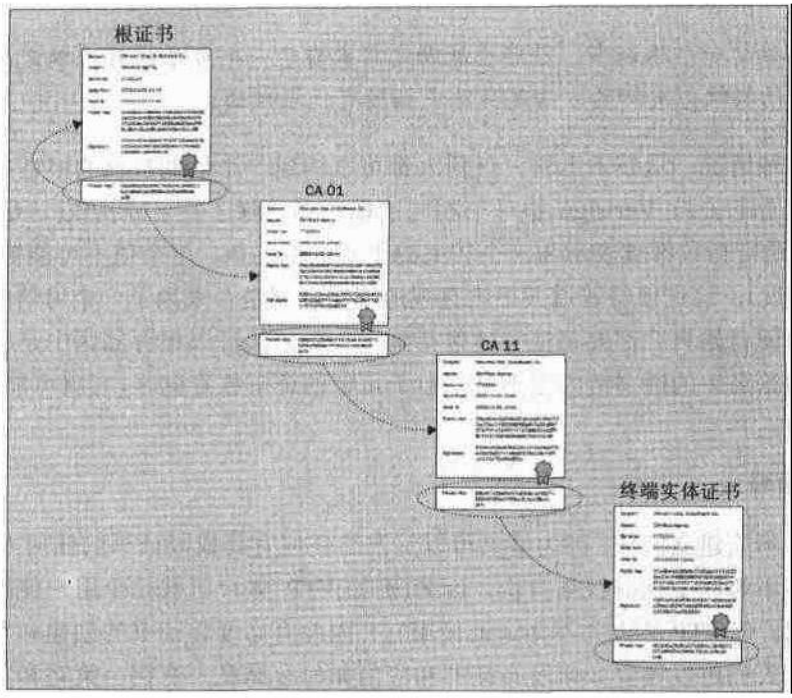

图　6-4

3. 根证书

了解根证书很重要，因为它们提供了信任的技术基础。因此，有一个用于可信的根证书的独立存储器，它不像其他存储器那样容易被修改，这一点是很重要的。否则可能就会得到一个看上去还不错的根证书，实际却来自于一些有恶意的源。

Windows 已经提供了一个相当长的可信的根证书列表，使用命令行实用程序 certmgr 可以看到。选择 Trusted Root Certificate Authorities 选项卡，就可以看到像 Verisign、Thawte、Entrust 和 Valicert 这样的名称。

将一个新的可信根证书添加到可信的根证书存储区中，Windows 将提示是否真的想添加其来预防，即如图 6-5 所示的对话框。

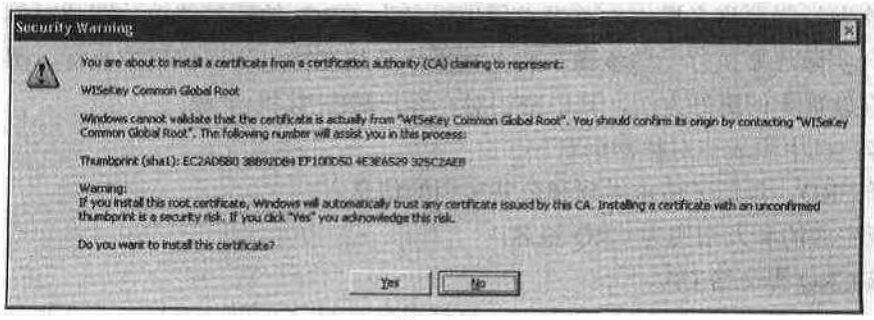

图　6-5

说明：

决不要轻易接受一个根证书，除非您能断定它来自于一个可信的源。确定的惟一方法就是手工地检验该源的指纹，更好的方法是从源中调用有代表性的。

考虑下面这种情况：从技术上讲，任何人都可以构建一个他们自己的根证书，这就使得任何人都可以创建看上去与 Verisign 根证书相同的根证书，除了签名或指纹(这是不可以伪造的)不同外。不过，通常指纹都被表示为一个较大的十六进制的值，这个值不可能被任何人记住(也可能有例外的情况)，这也使得普通人很难直观地确定证书是否来源于一个可信的根证书。

因此所有验证自始自终都要将证书链包括到一个可信的证书根存储区中是很重要的，因为如果您所验证的签名中的证书链的根不是来自于可信的证书根存储区，它就可能来源于任何人，很可能是无用的。

4. 实践和策略

技术本身并不能建立信任，因为证书可以签发给任何有其权限证书的任何人，供他们使用。拥有权限证书并不等同于拥有权限身份，因为(例如)FTP 帐户可被几个用户使用，这意味着它们都有权限证书，但 FTP 站点并不知道谁使用它！因此有必要将证书的创建和发布封装到策略和实践中，因此人们可以设置如何发布证书和应当如何存储私人密钥。策略和实践的强度也暗示着证书的信任强度。听起来很悬，是吗？下面对此作了讨论：

任何人都可以创建根证书，终端实体(终端用户、服务器)证书可以派生自它。通过这样一个根证书，可以很容易地用 Bill Gates 的名字创建一个证书；试想一下如果用该证书为价值高的事务签名；我将为世人注目。

因此，严格的证书授权机构建立了大量策略和实践声明，以便保证证书所有者的身份。通常，他们也定义应当如何保存私人密钥。还有一个“限定的”证书配置文件(RFC 3039)，这意味着它是根据有关于身份所有者的非常严格的规则发布的。这些配置文件也声明私人密钥必须安全，例如放在智能卡上或其他硬件令牌上，因为基于软件的加密存储区通常都是不够安全的。这些证书一般都被认为是所有者的真正数字身份，它们可以安全地用于任何需要它的在线事务处理。

6.1.3 发行

证书首次发行需要许多操作。首先必须生成名为 CRQ 的证书请求。即使对创建过程作了很好的定义，实际还要执行一些步骤，而一些操作必须在特定物理位置中执行。图 6-6 说明了将证书存储在软件中的更好方法，其中密钥对在客户端(计算机)生成。

(1) 在客户机中生成非对称密钥对。

(2) 编译 CRQ，包括了请求主题和公共密钥的信息。

(3) 用新生成的私人密钥为 CRQ 签名。

(4) 将该 CRQ 提交给 CA。

(5) CA 验证代理验证该 CRQ 的内容，包括主题信息。如果证书被确定为标识该主题，代理或其他机构代表必须手工地验证主题的身份。

(6) CA 为 CRQ 签名，生成一个证书。

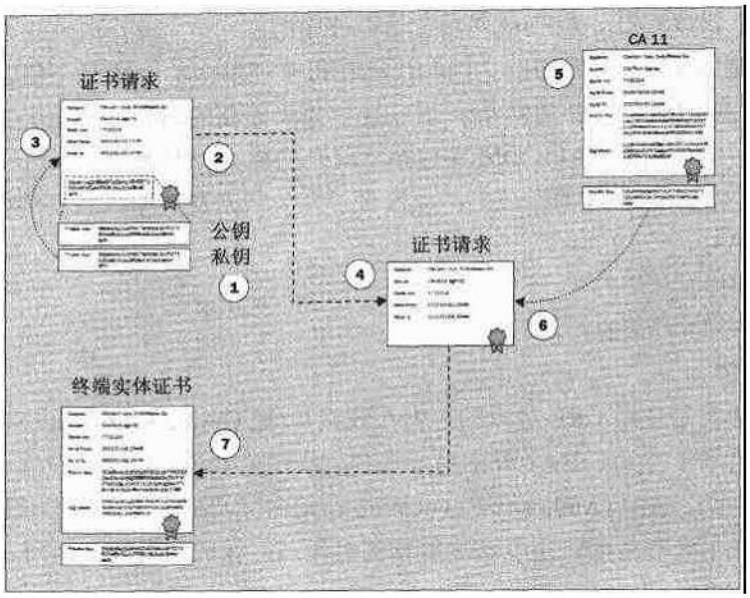

图　6-6

(7) 在客户机上安装该证书，与私人密钥相关联。

这是个简单的示例，其中没有执行所有者的身份验证。如果证书的目的是标识某人，那么必须在向客户计算机上安装证书之前采取一些额外的动作。例如，这类动作可以使用以前的 PIN 代码执行。将 PIN 提供给手工标识的一方，接着他们可以安装该证书，在安装了证书之后，它将是无用的。

6.1.4　检验

证书检验是对证书情况作一个简单检查的过程，例如确保它来自于一个可信的根证书，和有效日期签出。下面说明了检验证书的不同步骤：

(1) 在检验日期前后，证书有一个有限的验证日期。

(2) 根据父证书验证签名(如果是根证书就用其本身验证)。

(3) 用父证书重复这样的验证，直到处理根证书。

显然，执行验证的计算机中的日期和时间对于结果来说很重要。更好的方法是在进行签名前验证证书来节省时间，但仍必须在对该签名感兴趣的应用程序中进行检验和验证，以便支持可接受的安全级别。

6.1.5　验证-撤销

如果私人密钥被泄露给其他人而不是所有者，那么会发生什么情况？很自然，所有者并不希望其他人未经他们的授权来使用其密钥。CA 制定了在您知道其他人未经授权而使用您的私人密钥时采取的措施的有关规则(在发行证书前)。如果是内部发行了证书，那么您可能不必做任何事，但全球性的品牌一般都要求您尽可能地通知他们可疑的密钥(私人密钥被公开或被所有者之外的其他人使用)，这样它们可以撤销该证书(使之不可信)。在撤销证书时，它被添加到证书撤销列表中，即 CAL 中。由于这一动作并不能阻止使用私人密钥，所有想知道证书撤销情况

的应用程序必须下载该 CRL，检查证书看它是否被撤销。

CRL 通常是通过 Internet 发布的，因此包括在每个证书中的一般是个特定的字段，即 CRL Distribution Point 或 CDP。CDP 通常是一个 URL，如果需要的话，可从中手工下载 CRL，但正如您将看到的，后面提到的 CAPICOM 库包括了这一功能，因此您不必亲自实现它。

CRL 并不是只列出“具体的”证书，也包含了有关它的发布的信息，例如：

- 发行者(CA)
- 这次更新时间
- 下次更新时间(可选的)
- 发行者签名

现在，验证通常是使用 CRL 进行的，但在线接口(像 Online Certificate Status Protocol)也日益广泛使用，主要是作为除 CRL 之外的另一选择。从理论上讲，也可以泄露 CA(从而泄露由该 CA 发行的所有证书)，ARL(Authority Revocation List)由父 CA 发布。也必须验证 CA 证书，以便确保终端用户证书是有效的。重复这一过程到根证书，保证它没有撤销列表。

6.2 数字身份

本节将看一下数字身份的复杂性。这里将介绍您需要彻底理解的一些问题，以便实现高度安全的系统或进程。

6.2.1 数字身份简介

数字身份可以指所有事物，也可以指所有事物中的一个，但有一个常用的定义，即证书。正如我们已经讨论过的，证书包含了构成人(或其他什么如服务器)的数字表示的不同值。数字表示的完整性和强度是由技术本身和可信的第三方一起保护的。TTP 开发了策略和实践声明，试图使得任何人必须持有权限证书才可以使用私人密钥。

如果证书是由您公司的 IT 管理员内部发行的，那么情况又是怎样？是否会因为没有任何公司策略声明发行证书的规则而取消它的数字身份？答案是否定的，在这种情况下，该证书仍表示您，它肯定可以用于内部身份标识。难题就是明确定义证书在其中提供和未提供强身份标识的上下文。尽管您的公司证书可以充分标识您是在这个公司，但它对于其他公司来说就不意味着什么。

6.2.2 数字签名

数字签名和数字身份是同样的。尽管身份标识和签名从技术上讲是类似的，还是有一些小的不同点，比如用途。常见的一个例子就是通过 Internet 发布软件。下载软件的用户不能确定它是否被病毒篡改过，除非安装文件进行了签名。签过名的文件使得用户感觉安装它是安全的，只要为文件签的证书是来源于可信的根证书(参考根证书一节)，但它并不证明个人身份。

说明：

理解数字签名的所有内涵很重要，而不仅仅是技术方面的。

用现在的工具可以很容易地生成数字签名，但要确保在使用它们之前，知道每个数字签名在每种情况和每个上下文中的真正意思。本书不可能讲述签名在特定上下文中的意思；必须逐个例子地进行分析。接下来就下面几点进行讨论：

- 定义上下文
- 评估可用的 CA
- 评估签名工具和格式
- 将数据归档

1. 定义上下文

这可能是数字签名最为重要的方面，如果没有明确地定义上下文，那么您的数字签名可能就无意义了。

- 签名的目的是什么；例如它是否可以证明可下载软件的完整性或是证明有意识的动作，如签订合同？
- 签名的重要性；例如是否有与签名相关的时间或金钱费用或是否有法律色彩？
- 签名属于个人、角色还是服务？

还应当考虑告诉签名者签名含义的必要性。如果上下文定义一个签名等同于手工签名，被签的文档就有很高的责任性，在签名者签文档之前告诉他那些含义，应该是个不错的主意，在签文档时提到它也很好。在对数据签名时提到数字签名的意义比较好，特别是如果签名与手工签名一样都证明动作时更是如此。例如，如果希望纯文本上另一个人的数字签名表示他人接受该协议，那么就要声明签名也表示接受协议文本，这对其他人来说就能够一目了然。

说明：

如果不能表明事先已告诉过签名者，那么就可以否认数字签名，而不管其强度。

2. 评估可用的 CA

如果您正使用一个外部的 CA，对 CA 作一下评估很重要。应该安装您自己的 CA 吗？是否寻求全球性的品牌？选择权当然在您手里；只要确保身份强度符合您的需要。记住身份强度也来自于硬件令牌！

选择 CA 的另一重要方面是与您通信的一方也必须信任您所选择的 CA。外部方未必会接受您所拥有的根证书，因此，即使您在您自己的公司里畅通无阻，您也一定要有一个来自于可信的外部根的证书，例如 Verisign。

3. 评估签名工具和格式

评估工具是表示为您提供任何要签的内容的应用程序，这意味着它每次以完全一样的方式向您显示要签的内容是很重要的。

由于有像隐写术这样的技术，因此评估用于签名的工具很重要。允许签的数据格式越复杂，那么证明签名者实际看到和理解他们所签内容的问题就越多。即使出于安全的原因，只允许签纯文本，也会使得签名工具舍去文本的重要部分。

说明：

纯文本是用于签名的很好格式，并不一定需要 Microsoft Word 文档，是因为 VBA(Visual Basic for Applications)可用于处理可供给查看的信息。

要获取更多的原文信息，如果想确保被签的实际内容，那么不应接受任何脚本格式。

签名工具最好是只接受只读纯文本或如果可能的话，只使它“只读”。如果被签名的数据不是按位保护或存储的，从而可以恢复成和签名时同样的位数据，那么检验签名就有困难了。

4. 将数据归档

如果您有将所有已签名的数据归档的习惯，那么您已为必须验证签名的事件作好了准备。最好将签名和证书链与签名的时间戳捆绑起来，在归档之前标记这一捆绑。如果保存这些内容，就有了必要的证据来证明某人签了特定的数据。即使最初的签名者的私人密钥后来被泄露了，时间戳也可以使得它成为有效的证据。因此，在装载它之前进行捆绑和签名：

- 原始数据
- 签名
- 证书链
- 签名的时间戳(从签名数据的时间开始)

5. 示例：公司内部的文档签名

这个示例中 Alice 和 Bob 在同一个公司工作，Alice 在采购部，而 Bob 是服务部的经理。这个公司有它自己的证书根，所有雇员都有通过 CA 证书部门根据这个根证书签发的个人证书。在 Bob 办公室工作的 Carol 写了一个购买新的电脑的请求，她用她的证书对其签名，接着将请求发给 Bob 以获得批准。Bob 知道她需要一个新电脑，他就签名批准了这一请求。Bob 将请求发送给 Alice，她检验 Bob 的签名，验证 Bob 作这类请求的权限。Alice 购买了电脑，同时对所有相关的文档进行签名，这些都是在归档以前由管理员系统捆绑、盖上时间戳和进行签名。

由于 Bob 有正确的权限，Carol 最终将获得她的新电脑，而财务部会因为这一请求的有效管理而欣喜若狂。之所以一切工作这么顺利，是因为他们在设计该系统时注意了下列几点：

- 定义上下文。签名表示单个请求和它们各自的授权机构。每个已经对文档签名的人很难否认他们的签名和他们的行为。这是个公司内部的交换。
- 评估 CA。这是个非常简单的例子，因为这是个公司，可以很容易地在证书所有者的身份中建立强信任，因为该证书总是在公司内部签发的，可以被管理线验证。
- 评估签名工具和格式。这同样很简单，因为所有人都可以使用相同的管理系统，因此他们将看到相同的内容。系统也可以很容易地决定易于正确显示的简单格式。
- 将数据归档。由于所有的数据项都被放入系统中，因此可以很容易地在归档之前对它们进行捆绑、盖上时间戳和签名。

6. 示例：公司外部的文档签名

本示例通过讨论实际购买的管理扩展了前一个示例。Alice 所在的公司要购买电脑，而 Bob 所在的公司将电脑卖给 Alice 的公司。

Alice 给 Bob 写了一个订购单，签名并发送。由于 Bob 和 Alice 不在同一个公司上班，因此他不知道 Alice 公司的证书根，他就不能接受她的公司所签的证书。因此在将订单签给 Bob 时，Alice 必须使用另一个证书，该证书必须来源于 Bob 接受的证书根。在 Alice 获得 Bob 可接受的证书后，她也必须与他在签名工具和格式上达成一致。由于两人都使用 Outlook 2000，因此他们都认为只要所有消息都是纯文本，使用 Outlook 2000 中的 S/MIME 给所有消息签名和加密就足够了。

Alice 需要另外做的惟一一件事就是将她与 Bob 的通信信件归档在公司管理系统中。这整个过程之所以运行得这么好，是因为他们注意了下列几点：

- 定义上下文——签名批准了 Alice 的订单，验证了 Bob 的确认信息，这就涉及到了公司间的交换。
- 评估 CA——Bob 并不知道 Alice 公司的证书根，他们必须从外部的可信的根证书(例如 Verisign)中获取证书。只要他们都接受其他根证书，他们就不必具备来自同一个根证书的证书。
- 评估签名工具和格式——使用同样的工具生成和检验签名是个好办法；如果工具来自于可被信任的第三方更是这样。他们也选择纯文本，这样就显示所有内容，这和 HTML 不同，它可以在不显示的内容中提供注释。
- 将数据归档——Outlook 有其自己的归档功能，但为邮件盖上时间戳、用各自的证书对它们签名，将它们记录在安全档案中仍不失为一个好方法。

7. 示例：代码签名

Alice 所在的软件公司通过 Internet 发布它的所有软件，选择对它进行签名，向用户证明它的来源和完整性。Bob 是 Alice 软件的一个专门用户，经常下载它。他知道可以安全地安装该软件，因为 Internet Explorer 检验了代码的签名，在 Bob 安装它之前提供了代码签名证书。

Bob 会信任该证书，因为它来自于可信的根(根证书在他信任的根证书存储区中)，他没有安装其他根证书。这个例子还强调了不在信任的证书根存储区中安装根证书的重要性，除非您能 100%确定根证书来自于可信的源。这样才能确保安全。

这个示例也说明以前提到的指导原则只是指导性的，因为本例中并没有提及归档数据的知识点。不过，下面几点内容还需要注意：

- 定义上下文——签名表示了软件的源和自它生产出来后未被更改过的证明。代码是由公司外部的终端用户下载的。软件只针对 Microsoft Windows 系统。
- 评估 CA——Alice 必须使用已在 Bob 信任的证书根存储区中的证书，因为 Bob 对其他证书根一无所知。
- 评估签名工具和格式——由于针对的是 MS Windows 系统，因此显而易见是选择 Authenticode，下载时由 Internet Explorer 自动检验。

8. 示例：归档

在这个例子中，Alice 是代理人，她使用来自可信的证书根的签名证书。她用这一证书对由她处理的所有法律文档进行签名，接着把它们发送到公司归档系统。这一系统首先检验所有签名，然后将所有文档和签名一起压缩到一个文件档案中，添加一个包含时间戳的文件(当然使用

来自可信的时间源的时间)，接着用同样来自可信的证书根的证书为压缩文件档案签名。接下来将压缩的文件档案和系统签名放入系统档案中。

由于 Alice 的证书和系统证书都来自于可信的证书根，因此任何人都可以检验签名。由于时间戳是使用可信的时间源添加的，公司可以很好地保护该文档。

6.3 Windows 中的证书

这是密码术的一个重要方面，本节讨论如何在 Windows 中处理密钥。其中会碰到 3 个容易混淆的概念：存储单元(store)、系统存储区(system store location)和存储提供者(store provider)这三个概念。

存储区只是一组几个相关的存储单元。有与 Current User、Local Machine 和其他通用的 Windows 角色相关的系统存储区。每个系统存储区都至少有 4 个预定义的存储单元，划分为 4 个不同的上下文：MY (个人密钥和证书)、Root (可信的根证书), Trust (可信的第三方的证书)和 CA (中间 CA 证书)。存储单元(自 Windows 2000 起)是逻辑的，而非物理的，因此每个存储单元与一个或多个存储提供者相关。存储提供者表示存储单元的实际实现，像临时内存存储、系统注册表的存储或基于硬件的存储。不必担心密钥实际存储的位置，因为这取决于操作系统版本和配置。

存储单元、存储区和存储提供者之间的关系很复杂，很难讲透彻。Web Services Enhancements for .NET (WSE)和 CAPICOM 库(后面将介绍)提供的接口可以简化对 Microsoft 最通用的存储单元的访问，可以用更易于理解的方式组合这三方面，

6.3.1 系统存储区

存储区描述了可从中进行证书存储的上下文，例如，一般的存储区在当前用户或本地机器上下文中。下面所示为在 Crypto API 中可用的典型存储区例子：

- Current Service
- Current User
- Local Machine
- Services

所有这些可用编程的方式通过 CryptoAPI 获得(使用权限证书)，但也可以使用 MMC.exe (Microsoft 管理控制台)或 CertMgr.exe 手工管理。

1. .NET 中的系统存储区

少数存储区可通过已有的库从.NET 中获得。CryptoAPI 对访问这些存储区提供了广泛支持，实现了添加自己的存储区的可能。CAPICOM 和 WSE 支持可通过 CryptoAPI 获得的部分存储区。

WSE 库支持表 6-1 所示的 4 个存储区。

表 6-1

存　储　区	说　　明
Current User	这是当前用户的存储区，这种类型的存储区可能是读/写存储区。如果是的话，会保存对存储内容的更改
Local Machine	如果用户有读/写权限，那么本地机器存储区可以是读/写存储区。如果用户有读/定权限，如果该存储区以读/写模式打开，那么会保存对存储内容的更改
Services	这是像 Event Log 这样的应用程序所使用的指定的本地服务帐户的存储区
Unknown	这个存储区是未知的

2.0.0.1 版本的 CAPICOM 库支持表 6-2 所示的存储区。

表 6-2

存　储　区	说　　明
Memory	这个存储区是临时内存存储区；不会保存对存储内容的更改
Current User	这是当前用户的存储区；这一类型的存储可以是读/写存储。如果是这样，保存对存储内容的更改
Local Machine	如果用户有读/写权限，那么本地机器存储区可以是读/写存储区。如果用户有读/写权限，如果存储区是以读/写模式打开的，那么会保存对存储内容的更改
Active Directory	这个存储区是活动目录存储区。如果活动目录以读/写模式打开，那么不会产生任何错误，但将不会保存对存储内容的更改
Smart Card User	该存储区是个智能卡组。CAPICOM 2.0 中引入了它，该存储区很安全，因为密钥放在一个可移动的硬件上。使用口令限制对卡的密钥的访问可以进一步提高安全性

如果在框架类库、WSE 或 CAPICOM 中没有看到这些存储区，那么先看一下 CryptoAPI。

6.3.2　证书存储单元

Microsoft 加密库使用的存储单元是面向它们所在的上下文，例如 My 用于您自己的个人密钥(和证书)，而 Root 用于根公共密钥(主要与证书相关)。

标准的系统存储单元有：My、CA、Trust 和 Root。

- My——包含您的个人证书和对他们各自的私人密钥的引用(不管他们在任何物理位置上)。
- CA——也叫作中间 CA 存储单元。在这里保存了所有根证书和终端实体证书之间的证书。
- Trust——这包含了与可信的第三方相关的密钥。
- Root——这里存放所有可信的根证书。必须使该存储单元安全，因为它提供了信任链中的一个重要部分。有恶意的根证书可能会伪装成其他根证书，因为可以很容易地复制所

有可见字段。

.NET 中的存储单元

由于 CAPICOM、.NET Framework 和 WSE 加密库构建于其之上，在其中实现了运算的 CryptoAPI 库提供了 Microsoft Windows 中的基本加密技术。这些库只提供了对 CrytoAPI 的部分功能的访问。

对称密钥不可以保存在可直接从.NET 中访问的存储器中，这很容易理解，因为它们通常都由被封装的数据(后面作讨论)中的非对称密钥来保护，它们是易失的会话密钥。也可以保存对称的密钥，但如果这样做，必须研究低级别的 CryptoAPI。

表 6-3 所示为 WSE 和 CAPICOM 支持的各个库。

表 6-3

存 储 单 元	WSE	CAPICOM
My	√	√
CA	√	√
Root	√	√
其他		√

6.3.3 手工的系统存储管理

查看当前用户可用的证书的最简单方法就是运行 CertMgr 命令行实用程序(Start | Run: certmgr)，或是使用 Internet Explorer 中的 Tools 菜单(Tools | Internet Options... | Content | Certificates...)。

另一种显示存储区的所有安装证书的方法就是使用管理控制台的证书管理单元(W2K 和 XP 中可用)。打开 MMC 应用程序；一种方法就是通过 Start | Run: mmc.exe，接下来的步骤如下所示：

(1) 从 Console 菜单中选择 Add/Remove Snap-in...。

(2) 单击 Add...项。

(3) 选择 Certificate 管理单元。

(4) 选择想管理的存储区。

(5) 如果想管理多个存储区，重复第 2 步。

(6) 单击 OK 按钮。

(7) 浏览存储区，凭个人喜好添加和删除证书。当然进行这些操作时要小心；如果您删除了个人证书，很可能也删除了私人密钥，接着就不能解密所有用该公共密钥加密的信息。另外也不要随意篡改可信的根证书，因为它们是所有信任的基础所在。

图 6-7 显示了 Windows XP 计算机上与 EventLog 服务帐户相关的证书。

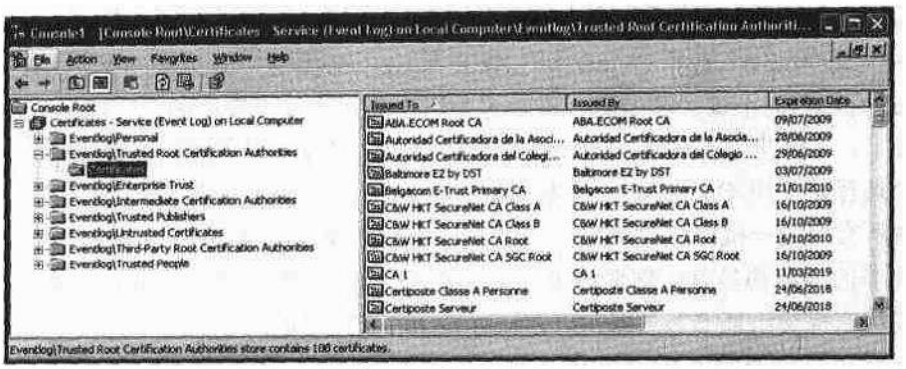

图　6-7

6.4　获取证书

在创建证书时还有一些选择；从何处获得证书主要取决于希望谁信任证书以及它的作用。例如，将用于创建您的身份的证书签给未知的第三方就没好处。

6.4.1　从公认的 CA 中获取证书

发现公认的 CA 的方法就是使用表 6-4 中所列出的或是使用 certmgr 并查看 Trusted Root Certification Authorities 选项卡，列出可信的证书根。默认情况下，它会提供大部分 Windows 用户信任的 CA 证书列表。在其他条件都相同的情况下，人们更有可能去信任信誉良好的证书授权机构。

表 6-4

名　称	URL	服　务
Verisign	http://www.verisign.com	个人和服务器证书
Thawte	http://www.thawte.com	个人和服务器证书
Entrust	http://www.entrust.com	服务器证书

由于他们有其自己的签发证书的方法，因此必须访问他们的 Web 站点，知道如何从中获取证书。例如 Thawte 发行免费的电子邮件证书，但它们并不包括您的身份。如果您希望获取包括身份的证书，必须提供证明您自己的证据，可能还得付费。

如果您从上面列出的 CA 中获取了个人证书，那么您的数字签名很有可能会在世界大部分地区被接受，因此这是很值得的。

6.4.2　从内部证书服务器中获取证书

您可以创建证书根，为任何人提供证书。不过，如果您建立了自己的证书根，派生自它的证书未必可以在您所服务的区域外被接受。为内部公司身份设置自己的根服务器可能是个不错的方法。为所有雇员提供个人证书可以创造许多新的可能，这是使用口令身份验证所不能提供

的，例如公司内部的所有动作和委托的不可否认记录。

Windows 2000 Server 和 Windows .NET Server 都提供了证书服务器(只有使用 Active Directory 才可获得)。不过，证书服务器确实为用户提供了一个简单的 Web 接口，在其中它们可以申请证书，如果您已经使用了 Active Directory，这可能是您更好的选择。

另一个选择是使用 OpenSSL(一个免费的开放源代码的 SSL 工具包)创建 CA。缺点就是必须亲自实现所有的用户接口。本章最后讨论了 OpenSSL 的使用。

所有证书的创建都是首先创建一个证书请求即 CRQ。通常在生成非对称密钥时本地生成 CRQ，接着将它提交给证书服务器，以便进行验证和获得批准。

Windows 中有一个名为 xenroll 的 ActiveX 的组件(本地 VBScripts 中可用)，它提供了创建 CRQ(也叫作 PKCS10 对象，PKCS#10 是 CRQ 格式的规范)的简单接口。

下面是生成 CRQ 的 VBScript(在浏览器中本地脚本生成)的简单示例：

```
<HTML>
<HEAD>
<TITLE>VBScript Certificate Enrollment Control Sample
</TITLE>
<OBJECT classid="clsid:43F8F289-7A20-11D0-8F06-00C04FC295E1"
        codebase="xenroll.dll"
        id=Enroll >
</OBJECT>
<OBJECT classid="clsid:98AFF3F0-5524-11D0-8812-00A0C903B83C"
        codebase="certcli.dll"
        id=Request >
</OBJECT>
<BR>
Certificate Enrollment Control Request Sample
<BR>
<BR>

<SCRIPT language="VBScript">
' Declare the distinguished name variable
Dim strDN

' Declare the request variable.
Dim strReq

' Declare a local variable for request disposition
Dim nDisp

' Enable error handling.
On Error Resume Next

' Declare consts used by CertRequest object
```

```
const CR_IN_BASE64 = &H1
const CR_IN_PKCS10 = &H100

' Build the DN.
strDN =   "CN=Erik Johansson" _
        & ",OU=Research & Development" _
        & ",O=DynaZon AB" _
        & ",L=Västerås" _
        & ",S=Västerås" _
        & ",C=SE"
' Attempt to use the control, in this case, to create a PKCS #10
MsgBox("Creating PKCS #10 " & strDN)
strReq = Enroll.createPKCS10( strDN, "1.3.6.1.5.5.7.3.2")
' If above line failed, Err.Number will not be 0.
if ( Err.Number <> 0 ) then
    MsgBox("Error in call to createPKCS10 " & Err.Number & _
           ", strReq=" & strReq)
    err.clear
else
    MsgBox("Submitting request " & strReq)
    nDisp = Request.Submit( CR_IN_BASE64 OR CR_IN_PKCS10, _
                            strReq, _
                            "", _
                            "Machine\CertAuth")
    ' If the preceding line failed, Err.Number will not be 0.
    if ( Err.Number <> 0 ) then
        MsgBox("Error in Request Submit " & Err.Number)
        err.clear
    else
        MsgBox("Submitted certificate; disposition = " & nDisp)
    end if

end if
</SCRIPT>
<BR>
</HEAD>
</HTML>
```

6.4.3　生成测试证书

如果您只需要用于测试的证书，不想花钱购买证书或是创建证书服务器，那么可以使用.NET Framework 中的 makecert 命令行实用程序生成您自己的证书。makecert 实用程序包含了大量选项，包括设置期满日期、公司名称和加密算法选项的参数。不过，要生成默认的证书，只需要指定文件名：

```
> makecert c:\myCertFile.cer
```

该证书将包括一些有关于“Joe's Software Emporium”的基本信息。这将不能通过证书授权机构验证。您可以双击 Windows Explorer 上的证书文件来查看证书的详细信息。如图 6-8 所示。

图　6-8

说明：

您也可以使用其他 makecert 选项或是 certmgr 实用程序，创建新的证书并把它安装到当前计算机中的密钥存储器中。不用任何参数运行 certmgr 就可以启动 GUI。

遗憾的是，WSE 类要求私人密钥是可输出的，随.NET Framework SDK 装载的 makecert 版本并没有将它们标记为可输出的。如果使用随 Visual Studio .NET 2002 装载的 makecert 实用程序(5.131.2157.1 版本)创建测试证书，将不能在后面的 WSE 示例中使用它们。不过，2002 年 8 月推出的 SDK 平台中的 makecert 版本(5.131.3639.0 版本或更高级的)却可以！您可从 http://www.microsoft.com/msdownload/platformsdk/sdkupdate/下载 SDK 平台的最新版本。

生成根证书

为了达到测试的目的，您可能希望模拟整个信任链。正如我们已经讨论过的，根证书与一般的有些不同。实际上，您可以将自己作为证书机构。创建一个根证书和其相关的私人密钥文件。

```
> makecert -a sha1 -sv richard.pvk -n "CN=Richard" -d Richard -r richard.cer
```

接着创建另一个证书，导入上面自签名的证书作为它的根证书机构。

```
> makecert -$ commercial -d "Example website" -n "CN=www.example.org" -iv richard.pvk -ic richard.cer
example.org.cer -pe
```

这将创建一条有效的信任链。技术文档中详细介绍了其中的开关。使用应用程序级协议可以按原样显示给客户机。

说明：

-pe 选项将私人密钥标记为可输出的，这在随 Visual Studio .NET 2002 装载的 makecert 版本中是不可用的。它是随 2002 年 8 月推出的 Microsoft Platform SDK 装载的。

6.5　证书和 WSE

尽管标准的 Framework Class Library 中有 X509Certificates 命名空间，即 System.Security.Cryptography.X509Certificates，但它并不包括任何证书存储类。下面的示例使用 Microsoft .NET 包的 Web Services Enhancements 1.0(这是 Web Services Development Kit Technology Preview 的支持版本)的命名空间：Microsoft.Web.Services.Security.X509。

6.5.1　Web Services Enhancements for .NET

找到下面的链接，获取如何下载和安装 Web Services Enhancements for .NET (WSE)的说明。这一发布是与技术文档一起提供的，可从 http://msdn.microsoft.com/webservices/building/wse/获取。System.Security.Cryptography 命名空间用于其他所有加密算法。

为了将 WSE 包括在 C#项目中，从 Add Reference 对话框的.NET 选项卡中选择 Microsoft.Web.Services.dll(当然在安装 WSE 之后)。

说明：

.NET 安全命名空间(包括 WSE)易于使用，但并不总是支持必要的运算来满足高级安全的需要，如保护智能卡的密钥，至少目前还没有。可考虑用 CAPICOM(后面作介绍)补充这些运算或是一开始就使用 CAPICOM 编写所有实现。

6.5.2　列出自己的存储单元

在首次测试 WSE 中的存储类时，我们查看了您的私人证书存储单元。您不可能有一个带有相应私人密钥的证书，除非您使用下面的示例找到了它。它们通常都放在默认(CurrentUser)系统存储区的“My”存储单元中。

记住，所有的 X509 类都来自于 WSE 库，而所有其他加密类来自于 FCL。

```
// ListCertsWSE.cs
using System;
using System.Security.Cryptography;
using Microsoft.Web.Services.Security.X509;
```

创建一个表示当前用户的个人证书的证书存储对象：

```
class ListCertsWSE{
```

```
static void Main()
{
  X509CertificateStore store =
    X509CertificateStore.CurrentUserStore(
                                        X509CertificateStore.MyStore);
```

以只读模式打开它，这样就不必担心会破坏它：

```
store.OpenRead();
```

出存储单元中的所有证书，用人们易读的方式输出主题。记住，X509Certificate 类来自于 WSE，而非 FCL！

```
foreach ( X509Certificate cert in store.Certificates )
        Console.WriteLine(cert.GetName());

store.Close();
  }
}
```

说明：

所有的 X509 类来自于 WSE 命名空间 Microsoft.Web.Services.Security.X509。

这个存储列表如下所示：

```
CN=Arne Anka, OU=Development, O=Software & Co, L=Västerås, C=SE
C=SE, OU=AddTrust2Mail, O=AddTrust, CN=Arne Anka, E=arne.anka@acme.com
```

由于 OSI Directory 也叫作 X.500，因此 CN、OU、O 等标记符的名称很特别。现在还沿用一些标准的名称，可查看 RFC 1779 (http://www.ietf.org/rfc/rfc1779.txt)获取详细信息。如果您对 LDAP (Lightweight Directory Access Protocol)熟悉的话，那么可以从那儿知道这些名称的用途。

6.5.3　检查自己的证书

.NET Framework 提供了易于访问一些证书字段的方法，这些字段相对比较难于提取。不过，这些字段的意义或多或少有些片面，因为很难归纳一个特定的含义。这并不意味着他们不包含重要的信息；您只需要知道在不同上下文中要找的内容。例如，在瑞典的每个公民都有一个惟一的编号(实际是在 1984 年前发明的)，这使得在许多不同上下文中标识人变得简单，并不只是数字的。不过，已经开发了一个名为 eID 的 X.509 配置文件，它包括了该编号，使用遵循于该文件的证书可以很容易地标识瑞典人。配置文件策略在这些证书发布中还定义了严格的身份标识方法，这使得它们可以安全地在 Internet 上使用。

当然，问题是知道要寻找的内容；例如，我们如何知道证书是否遵循于一个特定的证书配置文件，从而知道是否可以获取一个惟一的编号？不过，目前没有简单的方法。我们可以做的就是在检查证书内容时进一步看一下可能性。

```
// ExamineCertsWSE.cs
using System;
using System.Security.Cryptography;
using Microsoft.Web.Services.Security.X509;

class ExamineCertsWSE{

  static void Main()
  {

    // Set up the personal certificate store and list the
    // certificates in it
    X509CertificateStore store =
      X509CertificateStore.CurrentUserStore(
                                          X509CertificateStore.MyStore);
    store.OpenRead();

    foreach ( X509Certificate cert in store.Certificates )
    {
      // The issuer specifies the distinguished names of the CA
      // that produced this certificate (signed the CRQ)
      Console.WriteLine("issuer     : " + cert.GetIssuerName());

      // The subject is your own distinguished names
      Console.WriteLine("subject    : " + cert.GetName());

      // The serial number is unique within the issuer (CA)
      Console.WriteLine("serial     : " + cert.GetSerialNumberString());

      // A certificate isn't valid unless the current date is between
      // the notBefore and notAfter dates
      Console.WriteLine("notBefore: " +
        cert.GetEffectiveDateString());
      Console.WriteLine("notAfter : " +
                          cert.GetExpirationDateString());
```

下面的字段注重于更高级的上下文。其中有 KeyUsage 和 ExtendedKeyUsage 字段，它们定义了证书的用途，像身份识别、不可否认性和密钥加密(还有很多)。通过该框架可获得两个用途标记，尽管这并不意味着没有其他的用途。

```
      Console.WriteLine("Data encryption   : " +
                          cert.SupportsDataEncryption);
      Console.WriteLine("Digital signature: " +
    cert.SupportsDigitalSignature);
```

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

        store.Close();

        Console.ReadLine();
    }
}
```

该示例的输出如下所示：

```
issuer    : CN=CA01, OU=Examples, O=Test CA, L=Västerås, C=SE
subject   : CN=Arne Anka, OU=MyDep, O=MyCompany, L=MyTown, C=US
serial    : A949CD0AEC00
notBefore: 2002-02-14 09:18:27
notAfter : 2003-02-14 09:18:27
Data encryption   : True
Digital signature: True
-----------------------------------

issuer    : C=SE, L=Stockholm, O=Another CA, OU=Test, CN=CA02
subject   : C=SE, L=Västerås, OU=1443 MyRoad, CN=Arne Anka
serial    : CA3912000100408B692A
notBefore: 2001-11-06 01:27:12
notAfter : 2002-11-06 01:37:12
Data encryption   : True
Digital signature: True
-----------------------------------
```

6.5.4　为简单的纯文本签名

接下来看一下如何通过证书存储单元中的证书生成数字签名。这一示例显示了如何对简单的文本字符串签名，以及签名在转换为 base64 编码(一种常见的将二进制数据转换为文本的方法，在 MIME 中大量使用)后的外观。

说明：

对于这个例子来说，证书的私人密钥必须是可输出的，如果不是，那么将抛出 System.Security.Cryptography.CryptographicException 异常。参考“获取证书”一节，可以创建允许输出私人密钥的测试证书。

注意，我们通过更改 Certificates[0]索引来选择证书。您可以获得与前一个示例中列出的同样多的证书。

```
// SimpleSignWSE.cs
using System;
```

```
using System.Security.Cryptography;
using Microsoft.Web.Services.Security.X509;

class SimpleSignWSE.cs {

  static void Main()
  {
    // Create a certificate store object representing the
    // current users personal certificates
    X509CertificateStore store =
      X509CertificateStore.CurrentUserStore(
                                          X509CertificateStore.MyStore);

    store.OpenRead();

    // Pick out the first certificate we find in the store
    X509Certificate cert = (X509Certificate)store.Certificates[0];
    store.Close();

    // Get the private key from the certificate, specified by 'true'
    // as argument to ExportParameters
    RSAParameters privateKey = cert.Key.ExportParameters(true);

    // Set up a RSA provider and set the private key handle
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
    rsa.ImportParameters(privateKey);

    // Create a hash provider used for the signature
    SHA1CryptoServiceProvider sha1 = new SHA1CryptoServiceProvider();
```

安装一个实用程序编码器，以便获得签名的特定格式。如果被签名的数据用完全相同的格式保存(例如使用相同的字符编码)，或是如果在检验时知道原来和特定的格式，那么检验签名就不难。考虑将所有纯文本当作二进制数据，可以避免进行编码转换。

```
    System.Text.UTF8Encoding utf8Encoder = new
      System.Text.UTF8Encoding();

    byte[] plaintext = utf8Encoder.GetBytes("Sign this");
    byte[] signature = rsa.SignData(plaintext, sha1);

    Console.WriteLine(Convert.ToBase64String(signature));
  }
}
```

这个代码片段的一般输出是 base64 编码形式的签名：

XX2pUsZa2s4yKws1MQIaZMwCFQI89gLfc/qwBrshr1rfW+Yl0NQnTHj9juL5XJwcWI+CdqbqfIzGoLDf
outd0rXgB4YSps5xRyZiKYztGxvawHGAVX4CCr1nUKlcVY4jSaBoq5KadVy6D+7AnTQ9BBHWSF7uQKK
EB6X5r0aFzuyTLwGqZbsYDYirof1f3oQFj/8DRfo0ACZEvubBOcBtIBOM/OIYMelgeR4/lnDzbdFwWinEk+l
BstC6dRiROzaTy4xm0Om7JUx7ypRmat5sfOoYe6HlLm2fHbLAYIA8nZw77o/7OY2phGdr8+Pt2NzPhrAVJ4
h0qm93coAOFo3UFA==

6.5.5　检验签名

现在根据证书的公共密钥对签名进行检验。VerifyData()方法只根据对其签名的证书来检验数据：它并不检验证书链。对证书链进行检验以便确保生成签名的证书来自于一个可信的证书根，这是个不错的习惯，但由于 FCL v1.0 不提供这一支持，需要考虑使用 CAPICOM，因为它包括了全部的检验功能。

这一示例使用前面的代码以便节省篇幅；首先从和前一个例子中一样的证书中选出公共密钥的句柄，用它设置 RSA 提供者。

```
RSAParameters publicKey = cert.Key.ExportParameters(false);
rsa.ImportParameters(publicKey);
```

接着根据前一示例中生成的纯文本检验该签名，也使用相同的散列提供者：

```
bool result = rsa.VerifyData(plaintext, sha1, signature);

Console.WriteLine("signature verified: " + result);
```

如果签名或纯文本没有更改(觉得好玩的话，后面可以试一下)，那么检查签名将不会出现问题。

说明：

这种签名检验非常简单，如果证书源可疑，则不要使用 VerifyData()方法。

在任何时候都考虑使用 CAPICOM，因为它支持整个证书链检验。可参考本章后面的 CAPICOM 一节。

本示例的输出如下所示：

```
signature verified: True
```

6.5.6　保护另一个密钥

本示例说明了如何使用对称(秘密)密钥加密简单的文本字符串，接着如何用证书公共密钥加密加密该秘密密钥(从而保护它)。加密的密钥是用与该证书相关的私人密钥解密的。这是个如何用安全的方式保护个人信息的典型示例。也是用于共享被保护的信息的最为基本的模式，因为其他人的公共密钥也可以加密秘密密钥。该示例中的术语发送者和接受者是为了说明那种可能性。

```
// ProtectKeyWSE.cs
using System;
```

```
using System.Security.Cryptography;
using Microsoft.Web.Services.Security.X509;

class ProtectKeyWSE{

  static void Main()
  {
    System.Text.UTF8Encoding utf8 = new System.Text.UTF8Encoding();
```

为发送者创建对称密码。这也是用于加密明文的秘密密钥。

```
    SymmetricAlgorithm sCipher = SymmetricAlgorithm.Create();
    ICryptoTransform enc = sCipher.CreateEncryptor();

    // Set up a simple plaintext example and print it to the console
    byte[] sPlaintext = utf8.GetBytes(
      "The red fox jumps over the brown dog");
    Console.WriteLine("Sender's plaintext    : " +
      utf8.GetString(sPlaintext));
```

将明文加密成密文，以 UTF-8 编码字符串的形式将密文输出到控制台。这没有任何意义；只是为了达到说明的目的：

```
    byte[] ciphertext = enc.TransformFinalBlock(sPlaintext, 0,
      sPlaintext.GetLength(0));
    Console.WriteLine("Ciphertext              : " +
      utf8.GetString(ciphertext));
```

现在使用和以往(在前面的例子中)同样的方式，从个人存储区中获取证书。选择公共密钥，创建 RSA 密码对象。

记住，Certificates 集合包含了存储区中的所有证书，索引值 0 表示第一个枚举的证书。

```
    X509CertificateStore sStore =
      X509CertificateStore.CurrentUserStore(
        X509CertificateStore.MyStore);

    sStore.OpenRead();
    X509Certificate sCert = (X509Certificate)sStore.Certificates[0];
    RSAParameters publicKey = sCert.Key.ExportParameters(false);
    sStore.Close();

    RSACryptoServiceProvider sRSA = new RSACryptoServiceProvider();
    sRSA.ImportParameters(publicKey);
```

在这里，我们实际用接受者的公共密钥保护秘密密钥。如果不是使用 ECB 模式，那么也必须保存 IV。

```
byte[] eKey = sRSA.Encrypt(sCipher.Key, false);
byte[] iv = sCipher.IV;
```

这里将被保护的秘密密钥、密钥参数(像 IV)和密文发送给接受者。下面一段使用上面一段中的符号 eKey、iv 和 ciphertext。这次用相反的顺序再次做所有事情。

```
X509CertificateStore rStore =
  X509CertificateStore.CurrentUserStore(
    X509CertificateStore.MyStore);

rStore.OpenRead();
X509Certificate rCert = (X509Certificate)rStore.Certificates[0];
RSAParameters privateKey = rCert.Key.ExportParameters(true);
rStore.Close();

RSACryptoServiceProvider rRSA = new RSACryptoServiceProvider();
rRSA.ImportParameters(privateKey);
```

再次用从发送者(IV)处获取的参数加密秘密密钥，创建默认的密码。

```
byte[] sKey = rRSA.Decrypt(eKey, false);

SymmetricAlgorithm rCipher = SymmetricAlgorithm.Create();
rCipher.Key = sKey;
rCipher.IV = iv;

ICryptoTransform dec = rCipher.CreateDecryptor();
byte[] rPlaintext = dec.TransformFinalBlock(ciphertext, 0,
  ciphertext.GetLength(0));

Console.Write("Recipient's plaintext: " +
                utf8.GetString(rPlaintext));

Console.ReadLine();
  }
}
```

一般输出如下所示(注意混乱的密文输出)：

```
sender's plaintext    : The red fox jumps over the brown dog
ciphertext              : P$?lZLs)?ztl)H?ZW?|f?(_
recipient's plaintext: The red fox jumps over the brown dog
```

这个特定示例只是演示了密钥交换背后的基本原理，但必须理解其他许多问题。这里所有的对称密码参数都是“默认的”，几乎可以说只有.NET Framework 可以定义默认的密码参数。这使得实现简单的加密任务变得容易些，但它也隐藏了重要的内容，像算法 ID、模式等，如果是在系统间交换密文，那么知道这些很有必要。

说明：

要清楚在.NET 中使用默认的密码配置会冒互操作的危险！考虑使用特定的密码和模式，在两台有着不同默认设置的计算机中交换信息。

6.6　证书和 CAPICOM

CAPICOM 是 Microsoft 加密 API 的 COM 包装器，也叫作 CryptoAPI 或 CAPI，因此有了 CAPICOM 这个名称。CAPICOM 接口与 System.Security.Cryptography 命名空间在抽象级别上有些差异。CAPICOM 接口几乎只实现处理证书、证书存储以及任何与证书相关的事物的接口。例如，签名和验证运算稍微特别点，处理标准格式例如被封装的数据 (PKCS#7，RFC 2315：http://www.ietf.org/rfc/rfc2315.txt)，而框架提供了更基本的运算。CAPICOM 也用 PKCS#7 格式提供 EnvelopedData 对象，它处理秘密密钥的对称加密和非对称加密，而不是强制显式地去做。这使它在某些方面更具吸引力，但也降低了灵活性。

CAPICOM 2.0 可从 http://www.microsoft.com/downloads/search.asp?中下载，选择 Keyword Search，使用 CAPICOM 作为关键字。编写本书时最新的版本为 2.0.0.0。在下载了这之后，选取 CAPICOM.dll，将它放入系统的 Windows/Stystem32 目录中，接着将它注册为 COM 服务器：

```
> regsvr32 CAPICOM.dll
```

如果使用 Visual Studio .NET，将 CAPICOM 包括在 C#项目中所需要做的就是从 Add Reference 对话框的 COM 选项卡中选择 CAPICOM v2.0 Library(当然在安装 CAPICOM 后)。

如果您只使用 SDK 框架进行开发，那么将需要生成一个运行时可调用的包装器：

```
> tlbimp CAPICOM.dll /out:CAPICOM_RCW.dll /namescape:CAPICOM
```

6.6.1　列出自己的存储单元

在 CAPICOM 中列出个人存储单元与使用 WSE 几乎一样简单。CAPICOM 中有更多的参数名称，有时这很让人烦，但它确实提供了更为高级的选项，例如可以选择智能卡存储或是完整的签名检验。

```
// ListCertCAPICOM.cs

using System;
using CAPICOM;
class ListCertCAPICOM
{
  static void Main()
  {
    Store store = new Store();
    store.Open(CAPICOM_STORE_LOCATION.CAPICOM_CURRENT_USER_STORE,
               "My",
```

```
                CAPICOM_STORE_OPEN_MODE.CAPICOM_STORE_OPEN_READ_ONLY);
        foreach ( Certificate cert in store.Certificates )
          Console.WriteLine(cert.SubjectName);
        Console.ReadLine();
      }
    }
```

上面这个示例的输出如下所示：

```
C=SE, L=Västerås, O=Software & Ca, OU=Development, CN=Arne Anka E=arne.anka@acme.com,
CN=Arne Anka, O=AddTrust, OU=AddTrust2Mail, C=SE
```

我们猜想这一输出应与 WSE 示例一样，但事实并不是如此。这里没有描述如何显示具体名称的规则，全部是解析和转换证书字段的运算。

6.6.2 检查自己的证书

正如在 WSE 例子中说明的一样，.NET Framework 只直接支持证书中少量字段的分析，在一些需要严格安全的应用程序中会限制其有效性。如果没有在这一领域扩展 FCL，那么将强制直接使用 CAPICOM 或是 CAPI。

```
// ExamineCertCAPICOM.cs

using System;
using CAPICOM;

class ExamineCertCAPICOM
{
  static void Main()
  {
    Store store = new Store();
    store.Open(CAPICOM_STORE_LOCATION.CAPICOM_CURRENT_USER_STORE,
    "My",
    CAPICOM_STORE_OPEN_MODE.CAPICOM_STORE_OPEN_READ_ONLY);

    foreach ( Certificate cert in store.Certificates )
    {
    // The issuer specifies the unique names of the CA
    // that produced this certificate (signed the CRQ)
    Console.WriteLine("issuer    : " + cert.IssuerName);

    // The subject is your own distinguished names
    Console.WriteLine("subject   : " + cert.SubjectName);

    // The serial number is unique within the issuer (CA)
    Console.WriteLine("serial    : " + cert.SerialNumber.ToString());
```

```
// A certificate isn't valid unless the current date is between
// the notBefore and notAfter dates
Console.WriteLine("notBefore: " + cert.ValidFromDate.ToString());
Console.WriteLine("notAfter : " + cert.ValidToDate.ToString());
```

下列字段注重于更为高级的上下文。其中有 KeyUsage 和 ExtendedKeyUsage 字段，它们定义了证书的用途，像身份识别、不可否认性、代码签名和名称的密钥加密(但只有少量的)。标准规范 RFC 2459 和 3280 中提到了标准字段，可从 http://www.ietf.com/rfc.html 中找到。如果想知道 CAPICOM 中的更多可用字段，需要阅读 CAPICOM 参考手册：http://msdn.microsoft.com/library/en-us/security/security/capicom_oid.asp。

```
KeyUsage ku = cert.KeyUsage();
Console.WriteLine("\nKey usage ++++++++++++++++++++++++++++++++");
Console.WriteLine("Critical                  : " + ku.IsCritical);
Console.WriteLine("May sign CRL:s            : " +
                  ku.IsCRLSignEnabled);
Console.WriteLine("May sign certificates: " +
                  ku.IsKeyCertSignEnabled);
Console.WriteLine("Digital signature      : " +
                  ku.IsDigitalSignatureEnabled);
Console.WriteLine("Data encryption        : " +
                  ku.IsDataEnciphermentEnabled);
Console.WriteLine("Encryption only        : " +
                  ku.IsEncipherOnlyEnabled);
Console.WriteLine("Decryption only        : " +
                  ku.IsDecipherOnlyEnabled);
Console.WriteLine("Key agreement          : " +
                  ku.IsKeyAgreementEnabled);
Console.WriteLine("Key encryption         : " +
                  ku.IsKeyEnciphermentEnabled);
Console.WriteLine("Non-repudiation        : " +
                  ku.IsNonRepudiationEnabled);

ExtendedKeyUsage ekus = cert.ExtendedKeyUsage();
Console.WriteLine("\nExtended Key usage  ++++++++++++++++++++");
foreach ( EKU eku in ekus.EKUs )
{
  Console.WriteLine(eku.Name + " : " + eku.OID);
}
Console.WriteLine("\nCertificate extensions ++++++++++++++++");
foreach (Extension ex in cert.Extensions())
{
  Console.WriteLine(ex.OID.Name + " : " + ex.OID.Value);
```

```
            }
            Console.WriteLine("\nExtended properties ++++++++++++++++++++");
            foreach (ExtendedProperty ep in cert.ExtendedProperties())
            {
              Console.WriteLine(ep.PropID + " : " +
                ep.get_Value(CAPICOM_ENCODING_TYPE.CAPICOM_ENCODE_ANY));
            }
            Console.WriteLine("----------------------------------------\n");
          }
          Console.ReadLine();
        }
      }
```

该示例的输出如下所示：

```
issuer    : C=SE, L=Västerås, O=Test CA, OU=Examples, CN=CA01
subject   : C=SE, L=MyTown, O=MyCompany, OU=MyDep, CN=Arne Anka
serial    : 00EC0ACD49A9
notBefore: 2002-02-14 19:18:27
notAfter : 2003-02-14 19:18:27

Key usage ++++++++++++++++++++++++++++++++
Critical                : True
May sign CRL:s          : False
May sign certificates: False
Digital signature       : True
Data encryption         : True
Encryption only         : False
Decryption only         : True
Key agreement           : False
Key encryption          : True
Non-repudiation         : False

Extended Key usage ++++++++++++++++++++

Certificate extensions +++++++++++++++++
CAPICOM_OID_SUBJECT_KEY_IDENTIFIER_EXTENSION : 2.5.29.14
CAPICOM_OID_KEY_USAGE_EXTENSION : 2.5.29.15

Extended properties ++++++++++++++++++++
CAPICOM_PROPID_KEY_PROV_INFO :
pDEXAL4xFwABAAAAAAAAAAAAAAAAAAAAAQAAAEgAbQBoAHgAYwBrAFoAUgBaADkAU
gBzAAAATQBpAGMAcgBvAHMAbwBmAHQAIABFAG4AaABhAG4AYwBlAGQAIABDAHIAeQBwAH
QAbwBnAHIAYQBwAGgAaQBjACAAUABvAG8AdgBpAGQAZQByACAAdgAxAC4AMAAAAA==

CAPICOM_PROPID_HASH_PROP : 1DRuiYoYHeMR4Svq9rQOWe9DiX0=
```

```
CAPICOM_PROPID_MD5_HASH : xWAoDekixcTkVIpCOq6GJg==

CAPICOM_PROPID_KEY_IDENTIFIER : j7e2j5BGwTt87Q4dVHnJy0rsPUc=

-----------------------------------
```

从这些简单的示例中可以知道，处理证书不是件容易的工作，因为每个库都有其自己用可读取的格式显示内容的方法。如果比较证书间的字段，建议您先学习更多的实际内容格式。

正如在示例输出中看到的，在证书中有许多不同的字段，那就很难获取与您相关的字段。常见的字段是通过用户友好的界面发布的，但也有其他字段值得研究。

表 6-5 列出了部分扩展字段。

表 6-5

字　段	说　明
CAPICOM_OID_CRL_DIST_POINTS_EXTENSION	指定发布 CA CRL 的位置(通常是 URL)。证书中必须有这一字段，以便检查撤销情况
CAPICOM_OID_CLIENT_AUTH_EKU	不可以通过用户友好的 FCL 界面获取的 EKU(Extended Key Usage)。这个字段说明证书的其中一个用途就是充当客户端 SSL 证书(客户机身份验证)
CAPICOM_OID_SERVER_AUTH_EKU	用于 SSL 的所有服务器证书中必须有的另一个 EKU
*_EKU	值得研究的其他一些 EKU 值。访问 MSDN 站点可获取有关 CAPICOM_OID 的更多内容

如果想知道更多可用的扩展，可以访问下列 MSDN Web 站点：http://msdn.microsoft.com/library/en-us/security/security/capicom_oid.asp。

6.6.3　为简单的纯文本签名

在 WSE 示例中，我们生成了一个简单的签名作为单个对象。我们必须提供证书和纯文本，以便检验它。不过，CAPICOM 在 Sign()运算中生成了一个 PKCS#7 格式的对象，包括纯文本(可选的)、签名和证书链。这一对象不是很灵活，因为您不能直接提取签名实体，但它支持像联合签名和完全验证这样的运算。

```
// SimpleSignCAPICOM.cs
using System;
using CAPICOM;

class SimpleSignCAPICOM
{
```

```
static void Main()
{
  // Open the personal store as usual
  Store store = new Store();
  store.Open(CAPICOM_STORE_LOCATION.CAPICOM_CURRENT_USER_STORE,
             "My",
             CAPICOM_STORE_OPEN_MODE.CAPICOM_STORE_OPEN_READ_ONLY);

  // Pick out the first certificate (one based array)
  ICertificate cert = (Certificate)store.Certificates[1];

  // Create a Signer class as holder of the certificate
  Signer signer = new SignerClass();
  signer.Certificate = cert;

  // Create a SignedData class including the original message
  SignedData sd = new SignedDataClass();
  sd.Content = "The red fox jumped over the brown dog";
```

使用保存证书的 Signer 对象为数据签名。Base64 是可处理的简洁格式，因此我们请求那种编码格式的签名。结果是 PKCS#7 格式的。

```
  string signedMessage = sd.Sign(signer, false,
    CAPICOM_ENCODING_TYPE.CAPICOM_ENCODE_BASE64);

  Console.WriteLine(signedMessage);
  Console.ReadLine();
 }
}
```

这一输出比 WSE 示例中的签名大得多，主要是因为包括了证书链(不可选)，也就是说包括了终端用户证书以及所有祖先证书。如果选择包括此内容，使用 VerifyData()运算，接着检查 Content 属性，就可以从这块数据中提取它。下列所示是上面这个程序的部分输出(删减过)：

```
MIIJPQYJKoZIhvcNAQcCoIIJLjCCCSoCAQExCzAJBgUrDgMCGgUAMFkGCSqGSIb3DQEHAaBMB
EpUAGgAZQAgAHIAZQBkACAAZgBvAHgAIABqAHUAbQBwAGUAZAAv6cZjo+zyewuLTcW2
...
8AhLa5S0e5lzxuP7F3zULLfUTbxjl2ew3Yeh8nXgA7WjJvm3FbfRBGzTE/4y1HHUofjvZwIONj772N4E5
QTZe7p/lkKXV8LqpvI5+vrfKMgf7+82Xc=
```

6.6.4　检验签名

为了减少篇幅，假定已经用上面列出的数据设置了 signedMessage 对象。检验签名消息很简单，包括证书的检验。VerifyData()运算并不返回任何值，但如果因为某种原因检验失败，那么就会抛出异常。

创建一个新的 SignedData 对象，检验 signedMessage 内容。在成功检验后，Content 属性包含了被签名的原始消息(除非纯文本不在 Sign()方法中)。

```
SignedData verifyThis = new SignedDataClass();
verifyThis.Verify(signedMessage, false,
  CAPICOM_SIGNED_DATA_VERIFY_FLAG.CAPICOM_VERIFY_SIGNATURE
    _AND_CERTIFICATE);

Console.WriteLine("content = " + verifyThis.Content);

Console.ReadLine();
```

这个示例的输出是原始签名的纯文本：

```
content = The red fox jumped over the brown dog
```

6.6.5　验证证书

正如上一节提到的，证书验证通常不只包括证书链的检验，例如证书撤销检查。这种检查执行起来比较复杂，但 CAPICOM 接口包括了处理这种检查的运算，方法就是下载可用的 CRL，本地缓存它们。表 6-6 列出了可能的验证标记。

表 6-6

标　记	说　明
CAPICOM_CHECK_NONE	没进行验证
CAPICOM_CHECK_TRUSTED_ROOT	确保根证书是可信的
CAPICOM_CHECK_SIGNATURE_VALIDITY	检查证书链中所有证书的签名
CAPICOM_CHECK_ONLINE_REVOCATION_STATUS	根据 CA CRL 检查证书。这要求证书中有 CRL Distribution Point (CDP)字段，其 URL 必须可用。检查证书中的 CDP 扩展
CAPICOM_CHECK_OFFLINE_REVOCATION_STATUS	首先检查本地 CRL 缓存，如果在脱机缓存中没有发现，那么就试着在线获取它
CAPICOM_CHECK_COMPLETE_CHAIN	检查完整的证书链
CAPICOM_CHECK_NAME_CONSTRAINTS	检查名称限制，可查看 RFC 2459 获取详细信息
CAPICOM_CHECK_BASIC_CONSTRAINTS	检查基本限制，例如路径长度，可查看 RFC 2459 获取详细信息
CAPICOM_CHECK_NESTED_VALIDITY_PERIOD	检查证书链的任何证书中嵌套的有效性日期
CAPICOM_CHECK_ONLINE_ALL	检查证书链中所有证书的撤销情况，在线的根证书除外
CAPICOM_CHECK_OFFLINE_ALL	检查证书链中所有证书的撤销情况，脱机的根证书除外

该示例列出了您的个人存储单元中的证书，运用 CDP(CRL Distribution Point)扩展检查所有证书的在线撤销情况；另外执行了简单的验证。

```
// ValidateCertCAPICOM.cs

using System;
using CAPICOM;

class ValidateCertCAPICOM
{
  static void Main()
  {
    Store store = new StoreClass();

    store.Open(CAPICOM_STORE_LOCATION.CAPICOM_CURRENT_USER_STORE,
               "My",
               CAPICOM_STORE_OPEN_MODE.CAPICOM_STORE_OPEN_READ_ONLY);

    foreach (Certificate cert in store.Certificates)
    {
      CertificateStatus cs = (CertificateStatus)cert.IsValid();
```

上面的代码用只读的模式打开了个人存储单元，接着循环遍历其中的所有证书，根据存在 CDP 项与否来检查有效性。

这一部分有些倒退，但 IsValid()方法创建了 CertificateStatus 对象，它用于再次检查撤销情况(除了需要默认的标记)。在每次更新 CheckFlag 属性后更新 Result 值。

这里检查 CRL Distribution Point 字段是否在证书中。

```
bool cdpPresent = false;
foreach ( Extension e in cert.Extensions() )
{
  if ( e.OID.Name ==
    CAPICOM_OID.CAPICOM_OID_CRL_DIST_POINTS_EXTENSION )
  {
    cdpPresent = true;
    break;
  }
}

if ( cdpPresent )
{
```

如果存在 CDP 字段，我们请求证书链中所有证书(包括 CA 证书)的在线撤销状态。

```
            cs.CheckFlag = CAPICOM_CHECK_FLAG.CAPICOM_CHECK_ONLINE_ALL;

            if ( cs.Result == false )
                Console.WriteLine("online check failed: " +
                                  cert.SubjectName);
            }
            else
            {
```

如果不存在 CDP，我们只检查每个证书中可用的字段和证书链的完整性。

```
            cs.CheckFlag =
              CAPICOM_CHECK_FLAG.CAPICOM_CHECK_COMPLETE_CHAIN;

            if ( cs.Result == false )
              Console.WriteLine("complete check failed: " +
                                cert.SubjectName);
          }
        }
        Console.WriteLine("Done!");
        Console.ReadLine();
    }
  }
```

6.6.6　处理证书链

举个例子，如果您想知道有关 CA 和根证书的更多信息，就有 Chain 类帮助对证书链进行检查和处理。Chain 类提供了一个访问链中所有证书的简单接口。如果通过调用 IsValid()初始化了 CertificateStatus 对象，Chain.Build()方法将执行在该状态对象中指定的所有验证。

```
// CertChainCAPICOM.cs

using System;
using CAPICOM;

class CertChainCAPICOM
{

  static void Main(string[] args)
  {
    Store store = new StoreClass();
    store.Open(CAPICOM_STORE_LOCATION.CAPICOM_CURRENT_USER_STORE,
    "My",
    CAPICOM_STORE_OPEN_MODE.CAPICOM_STORE_OPEN_READ_ONLY);
```

遍历存储单元中的所有证书，为它们构建 Chain 对象。Build()方法根据终端证书的

CertificateStatus 验证链中的所有证书。

```
foreach ( Certificate cert in store.Certificates )
{
  Chain chain = new ChainClass();
  bool result = chain.Build(cert);
  if ( result == false )
    Console.WriteLine("\nValidation failed for certificate: " +
      cert.SubjectName);

  Console.WriteLine("\nListing certificates in chain:");
```

由于 chain.Certificates 集合是有序的，我们就获得了一个表示链中正确顺序的列表，从终端证书开始。

```
  int c = 1;
  foreach ( Certificate parent in chain.Certificates )
    Console.WriteLine("chain[" + c++ + "]: " +
                      parent.SubjectName);
}
Console.ReadLine();
}
}
```

如果需要其他验证标记，必须在构建证书链之前，使用 cert.IsValid() 获取一个 CertificateStatus 对象，使用 CheckFlag 属性设置新的标记。在此过程中，所有证书成功检验，设置了默认的验证标记。

```
Listing certificates in chain:
chain[1]: C=SE, L=Västerås, O=MyCompany, OU=MyDep, CN=Arne Anka
chain[2]: C=SE, L=Västerås, O=SomeCA, OU=Class 1 Domain, CN=CA01
chain[3]: C=SE, L=Västerås, O=SomeRoot, OU=Class 1 Domain, CN=Root CA

Listing certificates in chain:
chain[1]: E=arne.anka@acme.com, CN=Arne Anka, O=AddTrust, OU=AddTrust2Mail, C=SE
chain[2]: CN=AddTrust2Mail CA, OU=AddTrust TTP Network, O=AddTrust AB, C=SE
chain[3]: E=info@valicert.com, CN=http://www.valicert.com/, OU=ValiCert Class 1
Policy Validation Authority, O="ValiCert, Inc.", L=ValiCert Validation Network
```

6.6.7　被封装的数据

PKCS#7 格式也包括被封装的数据，表示数据是用秘密密钥加密的，而秘密密钥又是由接受者的公共密钥保护的。正如前面提到的，这是保护数据的常用格式(例如用于 S/MIME 中，它是安全电子邮件标准)，CAPICOM 中有一个处理这些运算的对象。

说明：

在与别人交换消息时考虑使用这个类而不是自己实现密钥保护模式。

这个示例类似于前一个示例，但都在一个对象中。

```
// EnvelopedCAPICOM.cs

using System;
using CAPICOM;

class EnvelopedCAPICOM
{
  static void Main(string[] args)
  {
    StoreClass store = new StoreClass();
    store.Open(CAPICOM_STORE_LOCATION.CAPICOM_CURRENT_USER_STORE,
    "My",
    CAPICOM_STORE_OPEN_MODE.CAPICOM_STORE_OPEN_READ_ONLY);

    // Create an object for enveloped data and fill in the content
    EnvelopedDataClass ed = new EnvelopedDataClass();
    ed.Content = "This is the actual content";

    // Set secret key parameters
    ed.Algorithm.Name =
      CAPICOM_ENCRYPTION_ALGORITHM.CAPICOM_ENCRYPTION_ALGORITHM_3DES;
    ed.Algorithm.KeyLength =
      CAPICOM_ENCRYPTION_KEY_LENGTH.CAPICOM_ENCRYPTION_KEY
        _LENGTH_MAXIMUM;
```

添加第一个证书作为加密消息的接受者。公共密钥用于加密秘密密钥。首先，用秘密密钥加密内容，接着用每个接受者(可以有多个接受者)的公共密钥加密秘密密钥。

记住，Certificates 集合包含了存储单元中的所有证书，索引值 0 表示第一个被枚举的证书。

```
    ed.Recipients.Add( (Certificate)store.Certificates[1]);

    string eContent = ed.Encrypt(
      CAPICOM_ENCODING_TYPE.CAPICOM_ENCODE_BASE64);

    // Print out the encrypted result and secret key information
    Console.WriteLine("Key algorithm = " + ed.Algorithm.Name);
    Console.WriteLine("Key length      = " + ed.Algorithm.KeyLength);
    Console.WriteLine(eContent + "\n");
```

创建“接受者”站点，其中只有加密的内容继承自上面的代码：

```
EnvelopedDataClass rEd = new EnvelopedDataClass();
```

加密被封装的数据。这一运算试图在您存储的证书和封装中的证书之间找到匹配的。存储的证书必须有一个相应的私人密钥。

```
rEd.Decrypt(eContent);

// Print out the original plaintext and secret key information
Console.WriteLine("Key algorithm = " + rEd.Algorithm.Name);
Console.WriteLine("Key length      = " + rEd.Algorithm.KeyLength);
Console.WriteLine(rEd.Content);

Console.ReadLine();

    }
}
```

一般输出如下所示：

```
Key algorithm = CAPICOM_ENCRYPTION_ALGORITHM_3DES
Key length      = CAPICOM_ENCRYPTION_KEY_LENGTH_MAXIMUM
MIICEAYJKoZIhvcNAQcDoIICATCCAf0CAQAxggGZMIIBlQIBADB9MHMxETAPBgNVBAMTCER
UUyBDQTAxMRwwGgYDVQQLExNEb21haW4gVHJ1c3QgU2VydmVyMSAwHgYDVQQKExdJbmZyYXR
ydXN0IFRlY2hub2xvZ2llczERMA8GA1UEBxMIVmFzdGVyYXMxCzAJBgNVBAYTAlNFAgYA7ArNSakw
DQYJKoZIhvcNAQEBBQAEggEALJTUTRa0wpNCelfTyY6TLbWz40ucw9HcmGhnxy+LRFrm+wn78BdTi8t
acBqOPRfsS0qR21bpyzRowP9dulN3S7ML3oYwcLIXV9Q3+VM22cDK/CzVuZ3wWhCTY66EVElsXVdho4
WJpaX0y91UwdJq3amCW7tdcqWOjDDZAzlRB9WAfqDMEQqOJ7CToX05Cbi+mbAKOBCV0D1gMNHt9a
wi5P9E7We7ESWHgeOsg8LEfXq2GsyQ8lX90j9JTpj06r3u6yv8uQeWQuQE/kduwEJgw183ZGwFJOXaGI4U
KxMqKP8eemSKbYzeM2U3yQLkSPLPwK1/M/nLijasdG2zrEo0jBbBgkqhkiG9w0BBwEwFAYIKoZIhvcNA
wcECBMLp2+MdvNsgDidBPzChHblEHyZW7Xt6vxneNCOoo5rVITAASeo43oLPb32lZE42mDf/p3zcHfCPS2
EDh5SRgUnvg==
```

```
Key algorithm = CAPICOM_ENCRYPTION_ALGORITHM_3DES
Key length      = CAPICOM_ENCRYPTION_KEY_LENGTH_MAXIMUM
This is the actual content
```

6.7　互操作性

我们已经知道了许多有关于如何在明确的环境下使数据安全以及相关方面的内容。如果配置文件中有其他系统或是想通过 Internet 交换信息和信任，必须考虑其他一些问题。这样的加密情况已相当成熟，但加密库还没有跟上步伐。还有与标准格式相关的互操作性问题，因此在注意了.NET Framework 安全后不要假定没有任何问题。您可以访问 http://www.microsoft.com/windows2000/techinfo/howitworks/security/pkiintro.asp，看一下 Microsoft 在其自己的 W2K PKI 黄页上对 PKI 的评价。

Microsoft 外部也很抽象地描述了 PKI 或 Identification Management(身份管理)，以及如何在全局上下文中实现它；从 http://www.ietf.org/internet-drafts/draft-ietf-pkix-roadmap-09.txt 中可获取更多信息。

这节中的示例使用 OpenSSL 应用程序(包括在 OpenSSL 发布中)来帮助说明特定事物。OpenSSL 是个可从下列链接中获取的免费 C 库：http://www.openssl.org。该站点只提供 C 源代码，因此需要自己编译它。另外，还可通过 Web 站点的链接获取 OpenSSL 的二进制 Windows 发布，例如 http://www.stunnel.org/download/binaries.html。

6.7.1　ASN.1、DER 和 PEM

ASN.1 (Abstract Syntax Notation 1)中描述了许多公共密钥加密格式，它类似于其他抽象的语法定义符号或像 EBNF(Extended Backus-Naur Form)这样的语言。由于 ASN.1 并没有指定如何将符号转换为位和字节，有像 DER(Distinguished Encoding Rules)这样的编码规范来描述。

大部分 X.509 证书、签名和 PKCS 对象是 DER 编码的。由于 DER 内容是二进制的，在有些情况下很难处理，因此经常用 BASE64 进行编码，从而获得对象的原文表示(不易读)。

PEM (Privacy Enhanced Mail)格式为 BASE64 编码格式的 DER 实体添加报头和报尾，以便标识它(在其他内容中)。一般的报头如下所示：

```
-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----

-----BEGIN PRIVATE KEY-----
-----END PRIVATE KEY-----

-----BEGIN PKCS7-----
-----END PKCS7-----
```

这样，如果获得了一个 PEM 格式的文件，删除报头和报尾，对 BASE64 内容进行解码，将得到一个 DER 编码的对象。

由于 X.509 证书是用 ASN.1 描述的，可以在任何计算机系统间交换 X.509 证书。由于二进制内容不能很好地输出，这个示例中的所有对象都是 PEM 格式的，例如下列这个证书：

```
-----BEGIN CERTIFICATE-----
MIIDPDCCAqWgAwIBAgIBAjANBgkqhkiG9w0BAQQFADBYMQswCQYDVQQGEwJTRTER
MA8GA1UEBxMIVmFzdGVyYXMxDTALBgNVBAoTBENBMDExEDAOBgNVBAsTB0V4YW1w
bGUxFTATBgNVBAMTDEV4YW1wbGUgQ0EwMTAeFw0wMjEwMjIxNzM5MTRaFw0wMzEw
MjIxNzM5MTRaMHsxCzAJBgNVBAYTAlNFMREwDwYDVQQHEwhWYXN0ZXJhczESMBAG
A1UEChMJTXlDb21wYW55MQ4wDAYDVQQLEwVNeURlcDESMBAGA1UEAxMJQXJuZSBB
bmthMSEwHwYJKoZIhvcNAQkBFhJhcm5lLmFua2FAYWNtZS5jb20wgZ8wDQYJKoZI
hvcNAQEBBQADgY0AMIGJAoGBAK7ak4gZAn7CkMF+Ug6ngxunSKQJpYhWELI/ERwx
sJtRN7rqdhRlQzIv7AR8IeowsJZJU/AccaKpU02rSQI/m8VD7nZqLuRE+4PuzMuB
vpSuzkVhrnSL63+epuW08DHxNXK0iM1BuzodQ0roAumxQdN/mKROVHoZozHEB1pD
X5KVAgMBAAGjgfIwge8wCQYDVR0TBAIwADALBgNVHQ8EBAMCBeAwMwYJYIZIAYb4
QgENBCYWJEMjIERhdGEgU2VjdXJpdHkgRXhhbXBsZSBDZXJ0aWZpY2F0ZTAdBgNV
```

```
HQ4EFgQULXVjsdz2pW3VPbi/roqr6VBc4RcwgYAGA1UdIwR5MHeAFHul/pbZJ7T7
GvLHCmYOjdyV6FEnoVykWjBYMQswCQYDVQQGEwJTRTERMA8GA1UEBxMIVmFzdGVy
YXMxDTALBgNVBAoTBFJvb3QxEDAOBgNVBAsTB0V4YW1wbGUxFTATBgNVBAMTDEV4
YW1wbGUgUm9vdIIBATANBgkqhkiG9w0BAQQFAAOBgQBZMi+uzlqleb7jbydfcvn2
8PVksdFMMNHf/ZEkr+KIEEruztAd3sibKBW5P8rAf3u13TSIGLDgInvw5vcG3SPE
EhQzYLQDIV7gv1YfGDK1FCfrburpEWo1vTxX+96KNlNe8wUTsJSJ0gPrik0Qagx5
EwzuZ/Qk1PTnOLUdmR8sRA==
-----END CERTIFICATE-----
```

OpenSSL 发布详细说明了如何创建您自己的测试证书，包括根证书和中间 CA 证书。这些示例中的证书是在 OpenSSL 环境中创建的。其中一些示例假定文件系统中有下列文件：

- arneanka.pem ——一个简单的 PEM 格式的终端用户证书，没有私人密钥。
- arneanka.cer ——相同的证书，但却是 DER 二进制格式的；这类文件可以用 certmgr.exe 应用程序打开。
- arneanka.pfx ——仍是相同的证书，但是作为证书存储单元中的输出，包括证书链上的所有证书和私人密钥(PKCS#12 格式的)。这个示例中使用的口令为 hemligt。

如果您的证书存储单元中有一个包括私人密钥的证书，那么可以使用 certmgr.exe 应用程序创建这些文件。使用 certmgr 或 mmc 应用程序列出您的证书存储单元。

6.7.2 显示证书内容

在 OpenSSL 中输出证书信息的最简单的方法就是使用 openssl x509 命令。x509 参数并不为.pfx 文件工作，因为它的格式不同(PKCS#12)。

使用 OpenSSL 发布的命令行实用程序 openssl，下列两个语句可以得到同样的输出：

```
openssl> x509 < arneanka.pem –text –noout

openssl> x509 < arneanka.der –inform DER –text -noout
```

用尽可能易读的格式输出证书：

```
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 2 (0x2)
    Signature Algorithm: md5WithRSAEncryption
    Issuer: C=SE, L=Vasteras, O=CA01, OU=Example, CN=Example CA01
    Validity
      Not Before: Oct 22 17:39:14 2002 GMT
      Not After : Oct 22 17:39:14 2003 GMT
    Subject: C=SE, L=Vasteras, O=MyCompany, OU=MyDep, CN=Arne
     Anka/Email=arne.anka@acme.com
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public Key: (1024 bit)
```

```
        Modulus (1024 bit):
          00:ae:da:93:88:19:02:7e:c2:90:c1:7e:52:0e:a7:
          83:1b:a7:48:a4:09:a5:88:56:10:b2:3f:11:1c:31:
          b0:9b:51:37:ba:ea:76:14:48:43:32:2f:ec:04:7c:
          21:ea:30:b0:96:49:53:f0:1c:71:a2:a9:53:4d:ab:
          49:02:3f:9b:c5:43:ee:76:6a:2e:e4:44:fb:83:ee:
          cc:cb:81:be:94:ae:ce:45:61:ae:74:8b:eb:7f:9e:
          a6:e5:b4:f0:31:f1:35:72:b4:88:cd:41:bb:3a:1d:
          43:4a:e8:02:e9:b1:41:d3:7f:98:a4:4e:54:7a:19:
          a3:31:c4:07:5a:43:5f:92:95
        Exponent: 65537 (0x10001)
    X509v3 extensions:
      X509v3 Basic Constraints:
        CA:FALSE
      X509v3 Key Usage:
        Digital Signature, Non Repudiation, Key Encipherment
      Netscape Comment:
        C# Data Security Example Certificate
      X509v3 Subject Key Identifier:
        2D:75:63:B1:DC:F6:A5:6D:D5:3D:B8:BF:AE:8A:AB:E9:50:5C:E1:17
      X509v3 Authority Key Identifier:
        keyid:7B
          :A5:FE:96:D9:23:B4:FB:1A:F2:C7:0A:66:0E:8D:DC:95:E8:51:27
        DirName:/C=SE/L=Vasteras/O=Root/OU=Example/CN=Example Root
        serial:01

  Signature Algorithm: md5WithRSAEncryption
    59:32:2f:ae:ce:5a:a5:79:be:e3:6f:27:5f:72:f9:f6:f0:f5:
    64:b1:d1:4c:30:d1:df:fd:91:24:af:e2:a5:10:4a:ee:ce:d0:
    1d:de:c8:9b:28:15:b9:3f:ca:c0:7f:7b:b5:dd:34:88:18:b0:
    e0:22:7b:f0:e6:f7:86:dd:23:c4:12:14:33:60:b4:03:95:5e:
    e0:bf:56:1f:18:32:b5:14:27:eb:6e:ea:e9:11:6a:35:bd:3c:
    57:fb:de:8a:36:53:5e:f3:05:13:b0:94:89:d2:03:eb:8a:4d:
    10:6a:0c:79:13:0c:ee:67:f4:24:d4:f4:e7:38:b5:1d:99:1f:
    2c:44
```

要显示.pfx 文件 (PKCS#12 格式的文件)，该文件经常包括私人密钥，必须知道用于保护.pfx 文件的口令。该示例中的文件是用口令 hemligt 保护的。

```
openssl> pkcs12 < arneanka.pfx -clcerts
Enter Import Password:
MAC verified OK
Bag Attributes
  localKeyID: 01 00 00 00
  friendlyName: {AA941D8C-EC07-4891-B71B-C8360CD320E5}
  1.3.6.1.4.1.311.17.1: Microsoft Enhanced Cryptographic Provider v1.0
```

```
Key Attributes
    X509v3 Key Usage: 10
Enter PEM pass phrase:
Verifying password - Enter PEM pass phrase:
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,749161BB4C214F29

Yg1OHMFz0YYWTuouaWGhFG48sG/fiwptKSq3HrYHC3vGxqpSXei7xx0eLFoKBmul
pubFIZJE/2pfhglWQBTWw+LqhA5nZYZmestTJ0LcY4f2eG965vj0H550sxPcjH9G
cqe3IwODcX4mXvaApMPhBU0NaO+ph885hjo1EpacgpMDmjWXIaTbu0rIg1Ke75No
z65ujoFFWmfC0+lc1sY1hUMwq8b/G2xd+TjGZ5P6J+6r3+0QcwjGFEYOZDdGHnIc
g6bdI6NdEbmGOIpFKZ3MSPFvsLTMB8LOD1ncavoH7GVE1WT6XCLb2Sz4qNcO1qFM
G6JbEspTkivLEziEEfRzbKLFPoGN795C01QgDmhPtdb2eIgNvwLTAzHs/VgulXN6
KswR0tUDZp1r2SQilRStT25QhmEPy6g4hPyOugh7wek59tVsFca/5MOOYPkyBomT
Kxs1vYcUZ9CdpPEMsUQUmbE/KrG1EZ29RMKM3PNE2UiGGY6D4p2LdZUnBuHdK401
ab4NeC+GR2fQHsgcwqcifGSYjV2SNw0JXICIq7NPTRjBzQp1b6wBOPx8hdRj4Bb6
xQ7OWTa9gd+5QJvbkPYAJkqujFNLtFQvBq1vdz7imzizbYPTuQUEDKtZkKDB/l1q
ocK7QQ3cVemI0vizSJxOicCp/W1ChaUjMnH0CB9B7fl5e79uw+9aYFPaYVHG45OH
0ZwgFjZsGoZ09mT7GwWVlQV+XNpGcyGULDt3H954B/Jx5GPAZOS6HP8S2hB95f8y
iObxaGl9cVOas6alBRUd1gBrVEqxyklkCdlfVOv4/ywfgrqsjR6++A==
-----END RSA PRIVATE KEY-----
Bag Attributes
    localKeyID: 01 00 00 00
    friendlyName: My Certificate
subject=/C=SE/L=Vasteras/O=MyCompany/OU=MyDep/CN=Arne Anka/Email=arne.anka@acme.com
issuer= /C=SE/L=Vasteras/O=CA01/OU=Example/CN=Example CA01

-----BEGIN CERTIFICATE-----
MIIDPDCCAqWgAwIBAgIBAjANBgkqhkiG9w0BAQQFADBYMQswCQYDVQQGEwJTRTERMA8G
A1UEBxMIVmFzdGVyYXMxDTALBgNVBAoTBENBMDExEDAOBgNVBAsTB0V4YW1wbGUxFTATBg
NVBAMTDEV4YW1wbGUgQ0EwMTAeFw0wMjEwMjIxNzM5MTRaFw0wMzEwMjIxNzM5MTRaMHsxCz
AJBgNVBAYTAlNFMREwDwYDVQQHEwhWYXN0ZXJhczESMBAGA1UEChMJTXlDb21wYW55MQ4w
DAYDVQQLEwVNeURlcDESMBAGA1UEAxMJQXJuZSBBbmthMSEwHwYJKoZIhvcNAQkBFhJhcm5lLm
Fua2FAYWNtZS5jb20wgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAK7ak4gZAn7CkMF+Ug6ngxu
nSKQJpYhWELl/ERwxsJtRN7rqdhRIQzIv7AR8IeowsJZJU/AccaKpU02rSQI/m8VD7nZqLuRE+4PuzMuBvp
SuzkVhrnSL63+epuW08DHxNXK0iM1BuzodQ0roAumxQdN/mKROVHoZozHEB1pDX5KVAgMBAAGjgfl
wge8wCQYDVR0TBAIwADALBgNVHQ8EBAMCBeAwMwYJYIZIAYb4QgENBCYWJEMjIERhdGEgU2V
jdXJpdHkgRXhhbXBsZSBDZXJ0aWZpY2F0ZTAdBgNVHQ4EFgQULXVjsdz2pW3VPbi/roqr6VBc4RcwgY
AGA1UdIwR5MHeAFHul/pbZI7T7GvLHCmYOjdyV6FEnoVykWjBYMQswCQYDVQQGEwJTRTERMA8G
A1UEBxMIVmFzdGVyYXMxDTALBgNVBAoTBFJvb3QxEDAOBgNVBAsTB0V4YW1wbGUxFTATBgNV
BAMTDEV4YW1wbGUgUm9vdIIBATANBgkqhkiG9w0BAQQFAAOBgQBZMi+uzlqleb7jbydfcvn28PVksdF
MMNHf/ZEkr+KlEEruztAd3sibKBW5P8rAf3u13TSIGLDgInvw5veG3SPEEhQzYLQDlV7gv1YfGDK1FCfrb
urpEWo1vTxX+96KNlNe8wUTsJSJ0gPrik0Qagx5EwzuZ/Qk1PTnOLUdmR8sRA==
-----END CERTIFICATE-----
```

6.7.3　在 PEM 和 DER 之间进行转换

由于编写一个能为 X.509 证书创建 DER 和 REM 之间的转换器的应用不是很困难，所以如果已经安装了 openssl，可以使用它。

下列语句是从 PEM 转换为 DER：

```
openssl> x509 < arneanka.pem > arneanka.der –outform DER
```

而下列语句是从 DER 转换为 PEM：

```
openssl> x509 < arneanka.der > arneanka.pem –outform PEM
```

6.8　小结

本章重点讨论了证书和它们如何帮助保护数据和密钥。公共密钥加密模式对此很适合，与证书结合起来，它提供了一个可以帮助通过 Internet 用电子形式与别人建立信任的工具。

公共密钥模式易于使用，还提供了一些可以利用的很好工具。您必须注意的不仅是您对接受者所有的私有密钥保管措施有多信任，还有对私人密钥的所有者的信任度；那么您对证书内容有多大的信任呢？

第7章　密码术——最佳和最坏做法

编写安全代码很不容易。它的编程思想与普通编程不同，它通常把重点放在是否经济、有效性、重用性和性能等方面。要创建安全代码，开发人员不仅需要理解加密原理及本书中介绍的概念，而且还必须能预见工作代码中的弱点，并消除使得心怀恶意的用户可进行攻击的漏洞。

安全编程需要慎密周全并且要关注细节问题，这点要求甚至对经验丰富的程序员都是挑战。不断追求新的安全主动性的 Microsoft，现在也会让新雇佣的开发人员参加公司内部组织的特殊安全班进行学习，甚至还会点名记录出席情况！

造成编写安全代码困难的一些原因如下：

- 安全性涉及到代码的方方面面。它们包括对用户进行身份验证和授权用户请求、屏蔽敏感数据和检验数据真实性的逻辑。另外，可以使用代码访问安全和机器级设置防止病毒和引诱性攻击。
- 安全问题涉及到编程过程中从开发到部署的各个阶段。它们包括在代码中隐藏秘密信息(这样做并不好)这样的编码问题、系统体系结构(它确定需要通过网络进行交换的信息类型)这样的设计问题和部署问题(如许可和锁定服务器环境)。
- 安全风险有许多不同的类型。攻击者试图窃取敏感信息、偷听通信信息，因此他们可以了解足够信息供以后攻击时使用、破坏其他用户的会话或只是暂时控制您的 Web 服务器。
- 要评估给定代码的安全风险比较困难。许多不安全的应用程序在受到攻击前看起来都是坚不可破的。

本章将列举最常见的攻击类型，以及应该如何制定计划防止这些攻击。我们还会介绍一些最佳做法和编写应用程序时会犯的最糟糕的错误，并列举一些比较规范的示例。

7.1　攻击类型

设计安全代码就必须知道攻击者可能使用的工具和技术的功能。通常能很好保护代码免受一种类型的攻击却让人担心会受到另一种攻击。甚至大型组织会犯的常见错误就是假定攻击者只对窃取信息有兴趣。实际上，攻击者很可能通过提交被破坏的信息来破坏应用程序，或满足于使用多余请求使服务器超载。

攻击方法有许多种。代码的易受攻击性与受到指定类型攻击的可能性都取决于应用程序类型(如桌面与分布式应用程序)、它所提供的服务(如金融与娱乐)以及攻击者的动机。某些进攻是被动的，也就是说信息是受到监控的。其他攻击是主动的，也就是说，它们以危害应用程序或其环境为目的而生成或修改信息。

从本质上看可以将攻击分为下面 4 类：

- 未经授权就使用资源，攻击者在没有权限的情况下使用攻击目标的资源。
- 数据入侵，攻击者窃取攻击目标的数据或通过非法监听消息获取信息。
- 破坏数据，攻击者通过欺骗使之接受和存储非法信息(并可能重写正确信息)危及系统安全。
- 拒绝服务(DoS)，攻击者努力使攻击目标的运行速度慢下来、阻止或防止攻击目标进行操作(通常其攻击目标是驻留应用程序的服务器)。

下面将对一些具体的攻击类型进行解释说明，它们经常用于攻击企业应用程序。一般而言，基于网络的利用如嗅探和重放攻击(sniffing and the replay attack)对黑客来说要更为困难一些，也不常用，可能需要访问公司网络的专用部分。不过，如果攻击成功，它们通常都会具有破坏性，使用无线网络技术后，这些攻击方式就越来越常见。利用用户、网络管理人员或程序员的失误的攻击方式(如社会工程攻击、第三方攻击或蛮力攻击破解比较脆弱的口令)则要常见得多。值得注意的是，许多这些攻击至少要有可信的雇员(或以前的雇员)的特有信息或帮助才行。

1. 嗅探或窥探

这就是攻击者偷听应用程序的组件间的网络通信的过程。它可以是客户机和服务器端组件(如 Web 服务)之间的通信或两个后端组件(通过消息队列将消息路由到任务调度程序的 Web 服务)之间的通信。通常，嗅探攻击用于窃取可以供以后发动另一次攻击时使用的信息(如用户口令)。不过，嗅探攻击可只用于研究网络通信量，提供攻击者所需的最终导致网络崩溃或被破坏的信息。

为了防止嗅探攻击，就必须加密。在某些受控制的环境中，可以使用传输级技术，如附录中将介绍的 IPSec(IP 安全)。不过，多数情况下，需要 SSL 或使用.NET 类进行应用程序级加密，前面几章已对此进行探讨。

攻击者对各种类型的信息都有兴趣，记住这点很重要。换言之，只是对口令和其他敏感信息进行加密处理是不够的。攻击者仍然可以看到足以确定谁在使用应用程序的其他信息，并能推断他们正在进行什么操作。在这种情况下，也可以在性能和安全之间作一个权衡，因为对所有信息进行加密需要耗费时间。最后，即使加密所有数据，攻击者仍然可以确定与数据有关的计算机、通信频率并且可能推断出计划下次攻击时要使用的其他详细信息。例如在一个著名的示例中，使用通信分析来确定总统出现之前特务机关的情况。在这一例子中，如果呼机数据突然变得频繁，这一点就很有用，就不需要破解消息所使用的加密密码。

2. 假冒或哄骗

攻击者常常假扮成合法用户进行这种攻击。这个攻击分为两个阶段。首先，攻击者会窃取必要的身份验证信息(如口令和用户名组合)。一旦获得这些信息，攻击者就会使用该身份访问系统，或者利用平台软件中的安全故障(例如，IIS 或 Windows 操作系统)以获取更高的特权。

为了检索身份验证信息，攻击者就会使用上述嗅探器程序或其他程序。另外可能还会使用恶意的客户端应用程序，这样的程序可以透明地监控击键情况，并使用口令 Trojan 假装成客户应用程序的前端，或允许攻击者出现在客户和服务器之间的重新连接的网络中。最后一种方法即中间人攻击特别隐蔽，由于客户应用程序相信它正在与服务器应用程序进行通信，但是实际上它正在与心怀恶意的应用程序交互，这样的应用程序会记录下由用户发送过来的敏感信息。

要防范中间人攻击相当困难，现在的许多应用程序都受到它们的困扰。

绝大多数安全代码都试图通过阻止攻击的第一个阶段来防止假冒式攻击。例如，对通信进行加密，这样在明文中就看不到口令、以加密或散列格式存储数据库口令、非管理员级用户不能访问用户表。不过，这些步骤都不能解决一些最为常见的风险，其中包括用户存储口令信息(如，在纸上或个人电脑的明文文件中)或具有所需的访问权限的内部用户泄露用户信息。

不过，目前还无法确定口令是否遭到破坏或提交口令的用户是否就是正确的用户。不过，可以使用下面的技术降低风险：

- 强制用户使用从密码技术的角度看不容易破解的口令，劝告他们不要将信息存储在内存以外的任何地方。强口令应该长一些，不要与任何语言中的词一致，并且要大小写、数字及其他字符混合使用。
- 定期使口令期满并强制用户换用新值。当然这可能有点过份。如果口令期满过于频繁，用户就会使用容易破解且容易记住(因此也容易猜)的值。
- 使用日志跟踪用户的操作情况，并定期评审是否有可能危及安全的可疑行为模式(例如，在工作时间以外访问系统的用户)。
- 将某些基本安全检查引入到应用程序中。例如，如果正设计以会话为基础的 Web 服务，就可以使用 Request.UserHostAddress 属性检查用户的 IP 地址。如果同一个用户试图在一个会话中从两个不同的 IP 地址访问系统，这种情况下就可以假定可能会发生假冒式攻击，就拒绝其请求，使会话期满并将该事件记录到安全日志中。
- 考虑使用依靠硬件(如智能卡)或安装证书进行用户身份验证的设备。这样用户就不会参与到这个过程中，也不能危及口令安全。不过，也暴露出新的风险——如果攻击者可以通过物理方式访问安装有硬件或认证的计算机，这就会危及系统安全。因此，维护这些系统的工作会变得更为复杂，费用也更昂贵。

最后，如果服务器安全曾经遭到破坏，则认为所有口令都不再安全，并要强制用户创建新口令。

说明：

上面的示例主要说明应用程序级的假冒攻击行为，不过攻击者也可以在会话期间使用假冒的 IP 地址来假冒您的计算机。使用 IPSec 这样的协议或进行应用程序级加密(防止用户知道被截获的信息)来防止出现这种攻击。

3. 重放攻击

更为隐蔽的假冒形式就是通过重放攻击来实现。在重放攻击中，黑客使用某种协议分析器来监控并复制在网络中传输的数据包。不过，与嗅探攻击不同的是，采用这种攻击方法黑客不用查找和解密用户口令。黑客只是保存数据包的副本。以后黑客将数据包重新发送到服务器就可通过身份验证。

重放攻击之所以有效，是因为攻击者不需要对消息进行解码以重用消息。假定所有用户会话都以完全相同的登录顺序开始，那么攻击者就可以通过发送与经过身份验证的用户相同的数据包假冒用户。

防止重放攻击，有几个低级别的选择。Windows 2000 网络中可以使用 IPSec，IPSec 使用(并验证其有效性)序列号来防止这种攻击。也可以使用 Windows 身份验证，这种身份验证使用询问/响应身份验证。由于询问总是不同，所以不能重用以前的响应。也可以使用 SSL，SSL 使用动态生成的会话密钥对数据包进行加密。

应用程序级可以采用下列几项技术来确保身份验证的数据包总是不同：

- 允许客户定期新建会话密钥。严格地讲，这并不能防止重放攻击，不过它会限定攻击者的攻击能力，因为密钥可使用的时间更短。
- 使用询问/响应系统。 例如，在交互开始时，服务器先向客户发送一些随机字节。然后，客户把这些字节添加到口令中，并计算结果的散列。这种响应不易受到重放攻击，因为每次询问(随机字节)都不相同。
- 使用安全审查。例如，客户可以生成惟一的静态 GUID，并将它与口令散列一起加密，然后发送到服务器。然后，服务器就可以解密消息并将 GUID 存储到数据库中。如果 GUID 在后面的登录请求中出现，服务器就知道遭到了重放攻击。

例如，客户可以使用下面的模式，在加密前将用户名和口令组合成一个日期字符串，这样就能很好地防止重放攻击。

```
// Instantiate a cryptographic object using the web service key.
EncryptionTest.SecureService proxy =
  new EncryptionTest.SecureService();
RSACryptoServiceProvider crypt = new RSACryptoServiceProvider();
crypt.FromXmlString(proxy.GetPublicKey());

// Create the login authentication information.
string userName = "testuser";
string password = "opensesame";

// Note that the following method uses UTC (universal time) to ensure
// that the code is independent of the client's time zone.
string now = DateTime.UtcNow.ToString("yyyy-dd-MM-HH-mm-ss-tt");

// Convert the authentication information to bytes.
UTF8Encoding enc = new UTF8Encoding();
byte[] userNameBytes, passwordBytes;
userNameBytes = enc.GetBytes(userName + now);
passwordBytes = enc.GetBytes(password + now);

// Encrypt the authentication information.
userNameBytes = crypt.Encrypt(userNameBytes, true);
passwordBytes = crypt.Encrypt(passwordBytes, true);

// Log in.
proxy.Login(userNameBytes, passwordBytes);
```

Web 服务会解密这种信息，并检验处理处理请求前的时间是否是合理的(可能是从当前时间算起 5 分钟内)。请注意，要规范的数据格式应达成一致意见以便它可以正常运行，这点极为重要。否则，Web 服务就会去掉字符串中错误的部分，并且不能验证正确消息的有效性。为了执行该操作，Web 服务代码使用 DateTime.ParseExact()方法而不是 DateTime.Parse()。

```
[WebMethod()]
public string Login(byte[] encryptedUserName,
                    byte[] encryptedPassword)
{
  // Retrieve the server key (using a private function).
  RSACryptoServiceProvider serverKey = GetKeyFromState();

  // Decrypt the user name and password.
  byte[] userNameBytes, passwordBytes;
  userNameBytes = serverKey.Decrypt(encryptedUserName, false);
  passwordBytes = serverKey.Decrypt(encryptedPassword, false);

  // Convert the authentication information into strings
  string userName, password;
  UTF8Encoding enc = new UTF8Encoding();
  userName = enc.GetString(userNameBytes);
  password = enc.GetString(passwordBytes);

  // Verify that both dates match.
  string dateString = userName.Substring(userName.Length, 22);
  if (dateString != password.Substring(password.Length, 22))
  {
    throw new SecurityException("Cannot start session.");
  }

  // Verify that the date is not in the future,
  // and is not more than five minutes in the past.
  DateTime date = DateTime.ParseExact(dateString,
    "yyyy-dd-MM-HH-mm-ss-tt", null);
  if (date.AddMinutes(5) < DateTime.UtcNow ||
    date > DateTime.UtcNow)
  {
    throw new SecurityException("Cannot start session.");
  }

  // Verify that the user name and password are in the database
  // using a private function.
  if (!ValidateUser(decryptedUserName, decryptedPassword))
  {
    throw new SecurityException("Cannot start session.");
```

```
    }
    // (Only an authenticated user will reach this point.)
    // (Register the user and issue a ticket here.)
}
```

上述方法惟一的弱点就是它为攻击者提供了 5 分钟的时间间隙。通过向每则消息指定惟一的随机标识符就可以防止受到攻击。然后，Web 服务就会将这些标识符缓存 6 分钟(比消息到期日期长一分钟)，每次收到请求都会检查缓存。拒绝重复的请求。我们把这种标识符称为 nonce。

基于 SOAP 的 Web 服务的 WS-安全标准定义了挫败重放攻击的标准方法，在以后的.NET Framework 版本中很可能加入这些方法。它使用口令、nonce 和日期创建生成口令散列，并将它添加到消息中作为 SOAP 题头。

示例如下所示：

```
<wsse:UsernameToken
      xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"
      xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
  <wsse:Username>NNK</wsse:Username>
  <wsse:Password Type="wsse:PasswordDigest ">FEdR...</wsse:Password>
  <wsse:Nonce>FKJh...</wsse:Nonce>
  <wsu:Created>2001-10-13T09:00:00Z </wsu:Created>
</wsse:UsernameToken>
```

Microsoft 推出了 WS-Security 的实现 Web Service Enhancements for Microsoft .NET，可以从 http://msdn.microsoft.com/webservices/building/wse/处获得。请记住，WS-Security 标准还只是一个草案，在它完全被吸收到编程架构如.NET 之前还可能修改它。

4. 会话重放或会话劫持

会话重放是一种特殊的重放攻击方式，它主要攻击采用基于票据(ticket)的身份验证的 Web 服务。黑客使用它就可能通过窃取用户的会话票据来劫持会话。我们通常很容易找到这种票据，因为每条消息都会将它添加为 SAOP 题头。如果消息被截获，黑客就可以窃取这种票据并将其应用于其他消息。

会话重放所造成的损害与身份验证重放攻击所造成的损害相比要有限得多，因为会话的持续时间有限。一旦票据过期，它也就没有用了。要防止受到会话重放攻击，则使用加密安全的票据。GUID 值比简单序列号或时间戳好得多，因为攻击者不需要截获数据包也可以很容易地猜出这些值。不过，GUID 值并不能保证从密码技术的角度看是安全的，.NET Framework 如何创建它们并不清楚(以及是否可以利用算法预测计算机可能生成的随机 GUID 值)。因此，最不容易破解的票据使用由 RNGCryptoServiceProvider 派生而来的随机字节序列，或使用 Web 服务的私钥在 GUID 票据上签名。第 8 章将剖析这项技术。

最后，应该采用严格的过期策略来限制票据的有效性，并且要给用户提供可以根据自己的意愿终止会话的退出功能。您还应该记录下客户的 IP 地址和会话信息，并在后续的会话中对它的有效性进行验证，确保不能使用与其他计算机相同的会话(除非攻击者将 IP 哄骗攻击法配合窃取票据使用)。

5. 蛮力攻击

在蛮力攻击中，攻击者不用特殊信息或技术。攻击者只是试用操作的每种可能组合形式，直到达到其攻击目的。密码技术中的蛮力攻击通常指通过猜测秘密密钥解密通信或用户的登录信息。使用绝大多数加密算法，蛮力攻击将花费很多时间而变得不可行。在蛮力攻击成功之前，数据就可能不再有价值。不过，随着分布式计算处理能力的不断进步，允许由计算机组成的大型网络联合作业，计算机处理能力在不断提高(采用这种方式，要破解由 DES 加密的数据，保守估计也只需要大约 3 个小时)。通过增加加密算法的位长度就可以提高安全性。

说明：

请记住，加密技术的目标不是要使系统完全不受攻击的影响，而是使攻击者的成本足够高昂(按时间或必需的资源计算)，这样对攻击者来说就没有实用价值。某些数据的生命周期相对短，因此就不需要采取相同的保护措施。

成功的蛮力攻击几乎总是根据对用户信息或其他信息的假设来缩小攻击的可能性范围才得以成功。例如，使用密钥的每种可能的字节序列来破解加密消息的蛮力攻击总是需要花费很长时间才能成功。不过，使用字典(称之为字典攻击法)中已知的每个单词来破解加密消息则要快得多。切记，加密强度取决于秘密值的强度。

通常可以很轻易猜出用户口令，这就是蛮力攻击成为最常用的攻击方法的原因。要使加密信息不容易破解，则只有使用口令进行用户身份验证。如果要加密长期保存的数据可交换的消息，就使用随机生成的密钥。在将密钥与加密信息存储在一起时，可以使用服务器端密钥或用户口令加密密钥。

您还需要特别注意用户口令，这是任何大型系统最常见的薄弱环节之一。强制用户使用从密码学角度看不容易破解的口令。例如，拒绝使用字典中的单词，建议不要使用容易猜测的值(如名字和日期)，使用最小的长度或强制要求包括一个或多个数字或大小写混合使用。这样就大大增加了破解密钥需要尝试的可能性。例如，一本优秀的口令破解字典可能有上百万条条目，而 10 个字符长度的非字典单词口令的可能数量就多达数万亿。另外，您也应该指定预先生成的随机口令(尽管口令越复杂，用户就采用不安全方式记录的可能性就越高，如记在小纸片上)。

您应该也使用好的审查做法来跟踪注册失败的情况。这样就可以发现可疑的操作模式(如同一个用户试图注册几十次均未成功)。

6. 拒绝服务

拒绝服务(DoS)攻击法是可以在 Web 环境中用于攻击服务器的最常见的方法，很大程度上是由于它们很容易实施。拒绝服务攻击法进行的操作很少，也不需要特别访问网络或专用信息。进行拒绝服务攻击时，攻击者就是努力使服务器超负荷运行(通常使用伪造的请求)，最后强制脱机。有时，使用术语“分布式拒绝服务”攻击(DDoS)表示由多个外部计算机对同一台服务器发动的拒绝服务攻击。从概念上看，拒绝服务攻击就是在向攻击目标吼叫直到攻击目标失去判断力。

许多拒绝服务攻击不是发生在应用层。其中一些甚至不是攻击计算机，而是以路由器、打印机或网络基础结构的其他部分作为攻击对象。下面就是一些常见的示例(不过还有许多类

似示例)：

- Ping of Death 攻击法使用请求服务器状态的 ping 请求充满整个服务器。这样的 Ping 请求将使服务器不能执行任何其他任务(关闭路由器级的 ICMP 13 和 18 就可以阻止绝大多数 ping 攻击)。
- Smurf 攻击与 ping of death 攻击法相似，不过它使用中间网络和广播消息。
- SYN Flooding 攻击法哄骗服务器打开不能完成的 TCP 连接，因为请求系统不能完成 TCP 握手。可使用路由控制软件限制由单一来源产生的这种影响。
- Teardrop 攻击将 IP 数据分割成极小的数据包，这种数据包可以绕过许多路由器和入侵监测系统。如果重新集中数据包，它们就会致使缓冲器溢出。
- FTP Bounce Back 使用 FTP 上传耗费大量时间的脚本，然后请求将脚本传输到 Web 服务器，并在 Web 服务器上进行破坏性操作。

攻击者通常使用某种 IP 哄骗技术结合这些攻击方法，以便 IP 地址不能阻止这些数据包。为了防止这种攻击，就需要知道您软件的弱点，并安装必要的补丁和安全软件。这属于网络管理员的工作范畴。

许多程序员不太明白的就是应用层有许多途径可以进行拒绝服务攻击。例如，如果 Web 服务的登录方法要耗大量时间根据数据库检验用户，那么攻击者就可以连续调用该方法数千次，牢牢控制住数据库。要防止这种攻击并不总是这么容易。其中一种方法是使用某种散列检验技术。

例如，在 Microsoft 的 Favorite Service(MSDN 上描述的一种平台)中，使用 GUID 创建所有票据，使用 GUID 的散列创建其他部分。使用服务器的私钥在散列上签名，这样散列就不能被篡改。在查询数据库检查票据是否合法之前，该服务就会确认散列代码是否正确。这样，就可以更迅速地拒绝不能与经过身份验证的用户相对应的伪造请求，从而降低了拒绝服务攻击的风险。简而言之，目标就是使未经身份验证的用户可以使用的资源量最少。

说明：

对许多组织而言，拒绝服务攻击没有入侵或信息失窃那么严重。它惟一的影响就是会使 Web 服务器临时脱机。不过，在要执行重要任务的情况下，长期停机造成的损失会相当大。

7. 社会工程攻击

社会工程攻击就是骗用户执行某操作或泄露信息。这可能是最为古老的攻击形式，它有数千种变化形式。其中一个示例就是攻击者假装成管理员向用户发送电子邮件，通知用户更改或泄露他们的口令。另一个示例就是电子邮件病毒，这种病毒含有伪装成幽默故事或有趣的应用程序的恶意代码。

我们不可能采用代码来防止社会工程攻击，用户需要不断地吸取教训。不过，对于防止社会工程攻击，开发人员仍然要承担部分责任，设计应用程序时必须要考虑受到社会工程攻击的可能性。例如，对于设计欠佳的客户接口都会受到许多社会工程攻击。默认设置授予用户太多特权(在电子邮件程序中，授予用户的特权就足以运行可执行的邮件附件)，或只是强制用户决定它们是否合格(如下载到 Web 页中的 ActiveX 控件)。要求用户作出信任的决定一般都是个馊

主意，因为用户缺少作出决定所需的信息，他们一般只了解需要完成的任务，而不是很了解所面临的风险。用户只可以执行工作的惟一方法就是接受所有代码，习惯性地就会在常规安全对话框中单击 Yes 或 OK 按钮。

8. 第三方攻击

第三方攻击由在同一台计算机上运行应用程序或其他应用程序的软件的弱点决定。其中包括操作系统缺陷、或程序如 IIS 中的安全漏洞。这些安全漏洞相当常见，因为人们都知道流行应用程序或操作系统中的安全隐患并讨论这些问题。例如专用 Web 服务。尽管 Web 服务比它在其上运行的 ISS 软件的安全缺陷还要多，但是知道 ISS 局限性的人更多，而发现应用程序的缺陷则要花一定时间。

防止第三方攻击的简单方法就是保持消息灵通，不断应用合适的补丁。如果 Microsoft 的安全公告声明存在的潜在问题，整个黑客界现在都知道这些问题，以及如何利用这安全隐患。Internet 上有许多关于安全信息的优秀资料，其中包括 CERT 协调中心(http://www.cert.org)和 NTBugTrack (http://www.ntbugtraq.com)。另外，就算万一发现您所用算法从数学的角度看存在漏洞，这也不会防害对最新加密技术的了解。

9. 应用层攻击

应用层攻击可以利用软件中存在的安全漏洞或局限。应用层攻击之所以常见是因为绝大多数软件都存在可以利用的缺陷。示例如下：

- 编码错误允许用户执行超出用户被允许范围的操作。这些错误通常都属于格式错误和用户输入无效(即标准错误)。
- 不安全的编码做法，攻击者通过引发不同类型的错误就可以利用这些编码发现用于攻击有关您的服务的足够信息。
- 代码中的路径不安全(后门)，不能执行适当的身份验证步骤。

解决应用层问题没有通用的方法，更正每种错误的方法都不相同。本章的“最佳和最坏做法”一节中将列举这些常见错误(以及如何修复它们)。

10. 病毒和引诱攻击

病毒是最为普遍的客户端安全问题之一——从本质上讲，它就是安装在计算机中进行破坏操作的微型程序。病毒之所以不易防范，是因为操作系统没有办法区分代码的好坏。如果将病毒安装在系统上，通常它就能完全控制计算机并执行它要进行的任何操作。

防范病毒需要某种沙盒，它可以根据某些条件来限制代码的能力。例如，如果将 Java 小程序载入浏览器中，就假定它不能执行破坏性操作，如修改系统注册表。不过，您仍然经常会面临引诱攻击，有恶意的代码会哄骗受信任的实用程序代码执行一些破坏性操作。实用程序代码允许进行这样的操作，是因为它不能识别出可能存在的安全问题，有恶意的代码就可以执行在其他情况下不能执行的操作。要防范这样的问题尤其困难，也是.NET Framework 中创建代码访问安全的主要目的。

7.2　最佳和最坏做法

本节将介绍创建安全代码时开发人员要采纳的一些最佳做法和最坏做法。这些示例都围绕密码技术展开，而不是其他问题如代码访问安全或授权和身份验证。不过，许多这些做法都涉及到其他一些领域，因为安全与代码开发的方方面面都有关系。

当然这只是说还有一些安全的重要领域没有涉及，即代码访问安全和信任模型。严格地讲，这些问题都与密码技术不相关。不过，知道有恶意的代码可以有许多方法与您的应用程序代码交互是很重要的。例如，如果您在使用当前很流行的“可修改应用程序”设计方案中的一种方法，如从 Internet 下载组件的 Windows Form 应用程序或使用反射动态实例化类型，那么您就为攻击者开辟了许多存在潜在安全漏洞的新领域。不过，用户不会意识到基本技术的变化，并希望您能确保任何插件、可插组件、可下载更新等安全性。

注意：

有关代码访问的更多信息，请参见清华大学出版社引用并出版的《Visual Basic .NET 代码安全手册一书。有关编写安全代码的指导，请参见 Microsoft 的内部安全手册 Writing Secure Code (Microsoft Press)，目前正在更新该手册，第二版中将包含.NET 特有的更多信息。

1. 最坏做法：用模糊处理实现安全

客户应用程序中假定攻击者知道应用程序的一切工作原理。分布式的应用程序中则假定攻击者完全了解您所使用的加密算法和密钥长度以及用于生成随机数字、票据和随机对称密钥的技术。

说明：

密码技术的强度取决于算法的强度和密钥的保密性而不是算法的保密性。好的加密技术可以确保即使在保攻击者知道加密消息的方法也不能解密消息。

我们有足够理由让您假定自己不能保守这些秘密。首先，可以从一个组织(通过物理或电子方式)窃取系统体系结构的高级别信息，或在公众场合中在您没有意识到的情况下，就可以从纸张或其他材料上获取相关信息。要么，攻击者能从应用程序中可以访问的另一部分(如客户的前端)推断这种信息，要么可以通过理论知识进行推导。如果您的应用程序安全视保密操作而定，如果秘密信息被人知道，就需要用新版本替换您的整个应用程序。这不仅不实用，反而在攻击者获取这种信息时也不可能会知道。

在客户应用程序中容易获取关于代码的信息。.NET 代码可以视为 MSIl，MSII 是比常规 Windows 可执行程序中使用的本地代码要高级得多的语言。即使不熟悉 IL 的程序员也可以很容易地从 IL 代码中识别这种"hello world"应用程序。下面的代码中已突出显示 WriteLine()和 ReadLine()方法调用以及要显示的字符串。

```
.namespace HelloWorld
{
.class private auto ansi beforefieldinit Class1
```

```
        extends [mscorlib]System.Object
{   .method private hidebysig static void
        Main(string[] args) cil managed
{
    .entrypoint
    .custom instance void
    [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )

    .maxstack 1
    IL_0000: ldstr      "Hello World"
    IL_0005: call       void
                  [mscorlib]System.Console::WriteLine(string)
    IL_000a: call       string [mscorlib]System.Console::ReadLine()
    IL_000f: pop
    IL_0010: ret
  } // end of method Class1::Main
 } // end of class Class1
} // end of namespace RandomTest
```

生成这段代码需要.NET Framework 的 ILDasm (IL Disassembler)实用程序。可以加载 GUI，使用 GUI 就可以按类型查看程序集类型，也可以在命令行输出如下的调试信息：

```
> ildasm HelloWorld.exe /TEXT
```

可以清楚地看到用于.NET Framework 类的方法和属性，正如硬编码信息(如字符串)。更有有趣的就是用户可以对更为复杂的反汇编程序进行处理。这种说明概念的示例之一就是 Exemplar，可以从 http://www.saurik.com/net/exemplar/免费下载。如果将 Exemplar 用于上述控制台应用程序，通过更改某些不重要的格式和使用不同的语义来检索所有代码(例如，所有类成员名称都是完全限定的)。

```
namespace HelloWorld {
  class Class1 {

    private static void Main(string[] args) {
      System.Console.WriteLine("Hello World");
      System.Console.ReadLine();
    }

    public Class1() : base() {
    }
  }
}
```

当然，采用模糊性未必就不好。您没有必要泄露任何有关应用程序的其他信息，以使得用户可以设计攻击策略。不过，不要依赖于这种模糊性，因为这在生产环境中是远远不够的。事

实上，应该总是假定您的代码是可以反汇编的。在.NET 中这相当容易，不过在任何语言中都是有可能的。

2. 最佳做法：不要将保密信息存储在代码中

所有这些都说明一个事实：绝不要依赖于保密操作或将保密信息嵌入客户应用程序。最糟糕的错误就是对对称密钥、口令或数据库连接字符串这样的秘密值进行硬编码。采用二进制格式来保存这样的信息或使用其他可能的十几种方法将保密信息掩藏在代码中都是没有用的，因为可以访问程序集文件的攻击者可以看到与数据和操作相关的信息。

我们还可以使用某些第三方的模糊处理方法，这些方法通过重命名方法和变量名来使其更难理解，也可能重新排列某些逻辑(其中包括与 NET Framework1.1 版本捆绑在一起的方法)。这些模糊处理方法对于专业黑客几乎不管用，如果模糊处理的缺陷引起代码不兼容的重新排列，则可能引发一些问题。

事实上，软件程序中没有十分简单的方法可以使使用软件程序的用户看不到保密信息。对于任何代码都是如此，不过在.NET 中，反汇编过程很简单。存储保密信息可以选择使用专门的硬件，如智能卡，不过这些硬件一般要昂贵得多(且仍然不能完全防止遭到攻击)。

3. 最坏做法：将用户保密信息存储为参数

在可以存储信息的地方中代码并不是惟一不安全的地方。许多应用程序都使用专业的加密信息来保护数据库中的和传输过程中的口令，添加使用方便的功能，安全就容易受到威胁。常见的示例就是自动登录功能，或对话框上“保存该口令”复选框。通常使用这些功能时，都会将保密信息存储在不安全的地方，如文件、Windows 注册表或 cookies 中。某些运转良好的应用程序如 Windows Messenger 会很好地确保这种信息的安全，这样在不知道用户口令的情况下就不能获取这种信息，可是，托管的类都没有包装这种功能。

我们不能设计这样的功能，程序员必须拒绝实现危及安全的功能。并不是只有在这种情况下，安全与其他功能之间才出现冲突。

4. 最坏做法：使用特殊的加密算法

许多编程书为了介绍概念会介绍一些没有经过论证的加密技术。其中最常用的就是 XOR 加密，它对字符串中的字符和私钥中的值进行 XOR 位运算。这些算法示例供学习使用，很有趣，不过它们不适合用于专业环境。例如，WordPerfect 使用 XOR 加密来加密用口令保护的文件。至少会有一个第三方提供可以通过蛮力攻击破解 WordPerfect 文件的实用程序，如果用于加密文档的密钥不多于 10 个字符，这种实用程序就可以迅速破解这种文件。

除非您是经过严格教育的密码专家，否则您的加密系统都很可能受到攻击如频繁分析攻击。您还需要高度复杂的数学问题，如容易破解的密钥、链接模式等。事实上，对加密算法的绝大多数蛮力攻击都利用实现特定算法时的缺陷，而不是算法本身的弱点(算法很可能经过数以百计的密码学专家推敲过)。也不是要认为编码(如将字符转化成字节或将几段数据组成一个更大的数据结构)就可以使攻击者看不出您的数据。这些方法都不管用。

请记住.NET 提供了对产品级加密算法的强大支持。请使用它们。

最佳做法：使用 CBC 链接模式

默认情况下，对称密钥算法采用 CBC 链接模式，不过在此有必要强调一下。使用 CBC 时，在加密数据块前，通过位异或运算将数据块与前一个数据块的密文合并到一起。它确保即使前一个数据块包含许多相似的数据，加密后的数据也不会相同。虽然这不需要处理器进行其他处理，但是它会大大降低某些分析攻击的有效性。例如，使用所有可能的密钥加密通用的数据片段(可能是全为 0 的数据块)，其中至少有一次攻击可以成功。然后，攻击者扫描通信信息，查找与其中一个生成的数据块相匹配的数据块。如果存在相匹配的数据块，攻击者就可能使用相同的密钥来解密其他通信信息。使用 CBC，这种攻击法要想成功就要难得多。

使用 CBC 链接模式，则将 SymmetricAlgorithm.CipherMode 属性设置为 CipherMode.CBC。

最坏做法：使用 System.Random

Random 类是一种伪随机数生成器，它使用数学算法从有限的数字中选择“随机”数。从统计的角度看，这些数字对于许多应用程序都是随机的(例如，生成示例数据)，但是从密码学的角度看却并不安全。它们存在如下问题：

- 如果攻击者知道创建随机数的算法，就可以预见您要生成的随机数。采用 IL 代码可以方便窃取这种信息。
- 如果攻击者所监听到的您生成的随机数足够多，他们甚至不需要知道算法是什么。最终，他们就可以将随机数与广泛使用的种子值关联起来(如当前时间的毫秒值)。
- 随着时间的推移，某些随机数序列就会再次出现。如果识别出这种顺序，就可以预测后面的“随机”数，即使这些数字在统计上看似随机的也可预测。
- Random 类的默认构造函数使用时间的毫秒值作为种子值。不过，如果您试图同时使用这个类的副本实例生成多个随机数，这些随机数都相同。这个缺陷很容易重复出现(附录 B 将演示说明)。

幸好，.NET 提供了在 System.Security.Cryptography 命名空间的 RNGCryptoServiceProvider 类中生成的密码安全的随机数。有关这个类的更多信息，请参见附录 B。

5. 最坏做法：对较大数据使用非对称加密

我们可以使用非对称加密对较大数据进行安全编码，如较大消息或数据文件。

不过，性能很不好(比使用对称加密要慢 100~1000 倍)，加密后的数据则要比必要的尺寸大得多。不过，没有简单的解决方法可以使用。如果要处理较大数据或只能使用非对称密钥，请按下列操作步骤进行操作：

(1) 随机地生成新的对称密钥。通过实例化新的 SymmetricAlgorithm 对象就可以隐式地进行这样的操作。

(2) 将该密钥与消息或文件保存在一起，并使用非对称公钥对其加密。只有拥有密钥对中的相应私钥的用户才能解密对称密钥。

(3) 使用对称密钥来加密数据的余下部分。

第 4 章(保护长期保存数据)和第 5 章(分布式系统中的密钥交换)列举了该技术的示例。

6. 最佳做法：分层设计

安全代码并不意味着使用破坏安全性更困难，它也只是限制破坏安全所引起的损害。其中一种最佳做法就是分层设计安全基础结构，这样就可以使用几层不同的安全机制来保护最为重要的信息。分层安全设计的几个优秀示例如下：

- 加密存在在数据库中的任何敏感信息，这样就算用户可以访问数据库，也不能访问敏感数据。对于某些数据类型，则只需要存储数据散列。
- 在内部网络或 SLL 中使用网络技术如 IPSec 就可以使网络监听和窃听更为困难，同时仍然要加密应用程序层的有效负载。
- 使用随机生成的密钥确保消息和较大数据的安全。采用这种方法获得对称私钥的用户却不能解密多个文档。
- 使用 EFS(Windows 加密文件系统)就可以使除管理员用户以外的其他用户都看不到服务器计算机上的数据文件。添加到应用程序上的这个加密层是透明的，不过如果心怀恶意的用户获得访问内部网络或物理计算机的权限，添加的这个额外保护层就可以防止攻击者检索数据。

总之一句话，绝不要依赖一种防范措施。

7. 最坏做法：返回敏感信息

discovery 就是黑客在检查系统是否存在可利用应用程序级缺陷时使用的工具之一。黑客使用“发现”这个工具引发不同的错误，就可以得到系统的大量信息。只要有足够时间，就可以收集到大量信息，这种信息可以用于后面发动攻击。

下面的示例就是编写 Login()方法的最坏做法。

```
public bool Login(string userName, string password)
{
  SqlConnection con = new SqlConnection(connectionString);

  // Define command for checking the user.
  string SQL = "SELECT Password From Users " +
    "WHERE UserName='" + userName;

  SqlCommand GetUser = new SqlCommand(SQL, con);

  try
  {
    con.Open();
    SqlDataReader r = GetUser.ExecuteReader();

    if (!r.Read())
    {
      throw new ApplicationException("User does not exist.");
    }
    else if (password != r["Password"].ToString())
```

```
        {
          throw new ApplicationException("Invalid password.");
        }
      }
      finally
      {
        con.Close();
      }

      // (Register logged in user here.)
      return true;
    }
```

现在只需稍作努力并使用蛮力攻击猜测密钥，黑客就可以确定系统上的用户列表。(生成可能的名字列表，使用自动化工具逐一测试并检查所返回的错误消息类型，就是其中一种可能的方法。)这种信息是实施针对性更强的攻击并努力猜出特定用户口令的良好基础。通过泄漏专用信息，攻击者使用这段代码就可以更容易地猜出口令。不幸的是，在产品级代码中这种做法并不少见。

较好的做法如下所示：

```
    public void Login(string userName, string password)
    {
      SqlConnection con = new SqlConnection(connectionString);

      // Define command for checking the user.
      string SQL = "SELECT Password From Users " +
        "WHERE UserName='" + userName +
        "' AND Password='" + password + "'";

      SqlCommand getUser = new SqlCommand(SQL, con);

      bool isAuthenticated = false;
      try
      {
        con.Open();
        SqlDataReader r = getUser.ExecuteReader();

        if (r.Read())
        {
          if (password == r["Password"].ToString())
          {
            isAuthenticated = true;
          }
        }
      }
```

```
        finally
        {
          con.Close();
        }
        if (isAuthenticated == true)
        {
          // (Register logged in user here.)
        }
        else
        {
          throw new SecurityException("User could not be logged in.");
        }
      }
```

通过将数据库代码封装到独立组件中仍然可以大大改进这段代码。另外，它使用动态构建的 SQL 字符串，这就使得它容易遭受 SQL 拒绝攻击。存储过程或带参数的命令是更好的方法，下节将介绍。

8. 最佳做法：用户输入标准化

标准错误是一种特定的应用程序错误类型，如果代码假定用户提供的值总是采用标准化格式，就会出现这种错误。标准错误是低技术性错误，不过后果却十分严重，这种错误的结果通常就是：用户可以执行应该受到限制的操作。

其中一种大家都很厌恶的标准错误就是 SQL 注入式攻击(SQL injection)，用户提交格式不正确的值，哄骗应用程序执行经过修改的 SQL 命令。例如，下面的示例使用代码验证用户的效性。可以使用如下的查询：

```
string SQL = "SELECT Password From Users " +
  "WHERE UserName='" + userName +
  "' AND Password='" + password + "'";
```

攻击者通过提交"InvalidPassword OR '1'='1"格式的口令就可以利用这种查询。动态查询如下所示：

```
SELECT Password From Users WHERE UserName='InvalidUser'
  AND Password='InvalidPassword' Or '1'='1';
```

该查询的最后部分('1"='1')计算所得的值总是为 True，这样查询就会返回一个包含所有用户口令的表。如果身份验证代码只是检验返回的 resultset 中是否至少只有一个记录(例如，通过调用 r.Read()并检验它是否返回 true 值)，那么身份验证就会成功执行。

这里使用 SQL 注入式攻击利用用户身份验证代码，但是可以采用相同的方法操纵查询，以便它可以返回敏感信息。在某些情况下，用户能够添加分号来创建批查询，并在查询的末尾附加一个任意的 SQL 语句。假设 SQL Server 包含可以运行命令行程序的 xp_cmdshell 存储过程，那么这种情况就极其严重了。

为了防止 SQL 注入式攻击，可以通过检查省略号等字符并删除，就可以清除输入信息中夹

杂的非法字符。也可以使用带参数的命令，这些命令自动执行必要的转义操作。下面就是与 SQL Server 等价的带参数的命令：

```
// Define the SQL string with named parameters.
string SQL = "SELECT Password From Users WHERE UserName=@UserName " +
  "AND Password=@Password";
SqlCommand getUser = new SqlCommand(SQL, con);
// Add the parameter values to the command.
SqlParameter param;
param = getUser.Parameters.Add("@UserName", SqlDbType.NChar, 25);
param.Value = userName;
param = getUser.Parameters.Add("@Password", SqlDbType.NChar, 25);
param.Value = password;
```

如果心怀恶意的用户提交相同的非法值，就会对查询进行如下转义操作：

```
SQL = "SELECT * FROM Users WHERE UserID='InvalidUser'
   AND Password='InvalidPassword" OR "1"="1'
```

它不返回记录，因为没有口令与所提供的字符串(InvalidPassword" OR "1"="1) 相匹配。

文件路径和 URL 也可能出现其他形式的标准问题。例如，下面的 Web 方法就从固定的文档目录中返回文件数据。

```
[WebMethod()]
public byte[] DownloadFile(string filename)
{
  FileInfo f;
  f = new FileInfo(Server.MapPath("Documents\\" + filename));
  FileStream fs = f.OpenRead();

  // Create the byte array.
  byte[] bytes = new byte[fs.Length];

  // Read the file into the byte array.
  fs.Read(bytes, 0, (int)fs.Length);
  fs.Close();

  return bytes;
}
```

这段代码看起来很简单。它将用户提供的文件名与 Documents 路径连接到一起，允许用户检索这个目录中任何文件的数据。

问题就是可以使用多种格式表示文件名称。攻击者不是提交合法的文件名，而是提交限定性文件名如“"..\filename"”。连接路径“WebApp\Documents\..\filename”实际上检索 Documents 目录的父目录(WebApp)。采用相似的方法，用户就可以指定 Web 应用程序驱动器上的任何文

件名。由于只是根据 ASP.NET 工作过程的限定条件来限制 Web 服务，所以用户可以下载服务器端的敏感文件。

修复这段代码相当容易。可以使用 Path 类(在 System.IO 目录中)删除字符串中的文件名最后部分。

```
[WebMethod()]
public byte[] DownloadFile(string filename)
{
  filename = Path.GetFileName(filename);

  FileInfo f;
  f = new FileInfo(Server.MapPath("Documents\\" + filename));
  FileStream fs = f.OpenRead();

  // Create the byte array.
  byte[] bytes = new byte[fs.Length];

  // Read the file into the byte array.
  fs.Read(bytes, 0, (int)fs.Length);
  fs.Close();

  return bytes;
}
```

这样就确保将用户强制在正确的目录中。如果正处理 URL，使用 System.Uri 类型就可以产生相似的效果。例如，下面就是如何从 URI 中删除查询字符串虚参，并确定它就是指指定的服务器和虚拟目录。

```
string uriString = "http://www.wrongsite.com/page.aspx?cmd=run";

Uri uri = new Uri(uriString);
string page = System.IO.Path.GetFileName(uri.AbsolutePath);
// page is now just "page.aspx"

Uri baseUri = new Uri("http://www.rightsite.com");
uri = new Uri(baseUri, page);
// uri now stores the path "http://www.rightsite.com/page.aspx"
```

9. 最佳做法：尽早失败

路径发现经常使用不能掩饰其异常的服务器端组件。例如，心怀恶意的用户会使用非法参数和没有得到授权执行操作的不充分证书调用服务器端方法。不过，如果方法代码编写正确，它就会在验证用户身份前返回异常，显示数据库或文件系统的其他信息。该信息不仅对客户没有用，而且还有助于心怀恶意的用户策划对系统的攻击。

例如，下面的代码以字符串形式返回文件内容：

```
public string GetFile(string filename, TicketInfo ticket)
{
  if File.Exists(filename)
  {
    throw new ApplicationException("File does not exist");
  }
  else
  {
    // Validate the user using the ticket.
    if ! ValidateTicker(ticket)
    {
      throw new SecurityException("Invalid user");
    }
    // Open the file and return the contents.
  }
}
```

这个问题允许文件系统的路径发现。攻击者使用不同路径就可以枚举整个目录的结构，并了解可以协助以后攻击的信息(如敏感文件的位置、安装的应用程序版本等)。这个示例中通过重新排列代码就可以解决这个问题：

```
public string GetFile(string filename, TicketInfo ticket)
{
  // Validate the user using the ticket.
  if (!ValidateTicker(ticket))
  {
    throw new SecurityException("Invalid user.");
  }

  if (File.Exists(filename))
  {
    throw new ApplicationException("File does not exist.");
  }
  else
  {
    // Open the file and return the contents.
  }
}
```

不过，在许多情况下问题都没有这么明显，敏感信息通常是从出乎您意料的异常中泄漏的。例如下面的模式：

```
public string GetFile(string filename, TicketInfo ticket)
{
```

```
    // 1. Acquire file.
    // 2. Authenticate user.
    // 3. Open file.
}
```

该示例的第 1 步中使用的代码就会导致 FileNotFoundException 异常。这样就会在不经意间将文件系统的信息返给攻击者。

要解决这些问题，就得确保不要总是先进行用户身份验证。而是使用异常处理代码来限制任何错误，然后抛出更为一般的异常。即使用户通过身份验证，也没有理由向他们提供有关非法数据库操作的明确信息，除非用户有某种方法可以解决问题。下面的示例表明如何处理数据库访问问题：

```
try
{
  con.Open();

  // (Perform database tasks.)
}
catch
{
  throw new ApplicationException("Data access error.");
}
finally
{
  con.Close();
}
```

较好的方法就是记录下服务器上发生的异常，这样就可以在需要时检索相关信息，ApplicationException 错误字符串中包含联系信息，这样用户就可以选择合法的方式来处理问题。在任何可能发生这种错误的地方添加异常处理——也换言之，无论何时访问文件、数据库、注册表或硬件等外部资源时都要添加异常处理。没有必要给出会暴露代码内部工作原理的任何低级信息。

说明：

对开发测试合适的未必就适合于生产环境。进行开发测试时，返回原始异常很重要。不过，在进行版本测试时，就必须修改这种异常处理代码。可以使用有条件的编译来确保某些代码不会随发行的版本一起提供给用户。

还有一个要尽早失败的原因。它有助于减少攻击者进行拒绝服务攻击的机会，进行拒绝服务攻击时，心怀恶意的用户就会使您的服务器不停地处理伪造的请求。如果可以在执行耗费大量时间的任务(如连接到数据库)之前就引发故障，攻击者要控制服务器就要难得多。

10. 最佳做法：记录异常

典型的蛮力攻击的工作量很大。例如，黑客在查找到用户名和口令组合之前，会向 Web 服

务器发送数以百万计的请求。如果有早期警示系统识别这种可疑行为，就可以在黑客的攻击造成损害之前发现攻击的迹象。

最佳的做法之一就是记录安全异常。该信息可以存储在中心数据库中，或计算机特有的事件日志中。由于事件日志很简单，所以通常使用它。尽管每台计算机都有自己的事件日志，但是您也可以使用.NET 代码来检查远程计算机的事件日志。下面的代码表明如何记录注册失败的访问：

```
// Create the log if needed.
if (!EventLog.Exists("MyAppLog"))
{
    EventLog.CreateEventSource("MyWebService", "MyAppLog");
}

EventLog log = new EventLog("MyAppLog");
log.Source = "MyWebService";
```

请注意我们没有存储试用过的口令。这样有助于跟踪问题的原因，不过如果另一个用户读取日志也可以知道存在的安全隐患。

```
log.WriteEntry("Attempt to login for user " + userName + " failed.",
  EventLogEntryType.FailureAudit);
```

一些公司使用自动化工具扫描事件日志，查看是否存在可疑行为方式，不过，管理员迅速审查该信息就可以轻易发现潜在的问题，这样更方便。日志和审核日志时所面临的挑战就是，某些微不足道的事件也可能表明如果问题重复出现后果就更严重(如注册失败)。正确获得这种通知逻辑很困难。如果存在疑问，就将其存储到可以持续使用的地方，以供以后评审之用。

说明：

不要使用将向管理员发送有关问题的电子邮件的异常日志代码，除非问题很少见且很严重。否则，用于通知的邮件帐户就可能被小错误塞满，最终重要的消息也会被忽略。

如果出现安全漏洞，就必须正确修复以便限制它造成的损害。也就是说，您必须制定事件响应计划和编写相关摘要，其中包括网络工程师、Web 管理员和 ISP。您手里应该准备一些基本的网络工具如数据包嗅探器。有关详细信息都超出本书讲述的范围。不过，您可以参见网络安全方面的专业书籍(请参见附录 C)。

11. 最佳做法：使用散列避免使用数据库

要防止拒绝服务攻击，在对用户进行验证之前，不应该执行任何耗费时间的任务。如果可能，您应该避免使用数据库，因为数据库连接属于有限资源并且需要一些基本的系统开销才能建立这样的连接。

不过，使用数据库是不可能避免的。例如，如果先验证用户的身份，可能就需要将证书与用户表相比较。不过，在基于票据的身份验证中，可以使用散列代码在后续请求中避免使用数据库。

不使用散列代码的情况下基于票据的身份验证的操作步骤如下所示：

(1) 用户注册并收到票据。票据就是动态生成的 GUID。

(2) 票据信息存储在数据库和高速缓存中并返回给用户。

(3) 用户在后续请求中出示票据。服务器端代码在高速缓存中查找该票据。

(4) 如果高速缓存中没有票据，服务器端代码就会检查数据库。

这里的潜在问题就是第 4 步，也是消耗时间最多的一步。如果提交非法票据，服务器仍然会浪费时间检查数据库，这样就为拒绝服务攻击提供了机会。为了降低受到拒绝服务攻击的可能性，服务器需要采用一种方法来识别它是否生成所提供的票据。散列代码就是一种简单的解决方案。

使用散列代码的情况下基于票据的身份验证的操作步骤如下所示：

(1) 用户注册并收到票据。票据就是动态生成的 GUID，它末尾附有一串简短的字节。这些字节是使用服务器私钥签名的 GUID 散列生成的数字签名。

(2) 票据信息存储在数据库和高速缓存中并返回给用户。

(3) 用户在后续请求中出示票据。服务器端代码通过验证签名的有效性并检查过期日期来验证票据。

(4) 假定票据有效，服务器端代码就会在高速缓存中查找该票据。

(5) 如果高速缓存中没有票据，服务器端代码就会检查数据库。

说明：

只有消息身份验证代码的普通散列是不够的，因为心怀恶意的用户可以很容易地生成新消息和新散列。否则，就必须使用密钥散列或签名的散列。

12. 最坏做法：试图压缩加密数据

加密后再压缩不会节约大量存储数据的空间。如果加密算法很优秀，它就会从一系列随机数字生成从统计上不能区分的输出。因为通过识别模式压缩软件就可以压缩数据，没有哪种压缩算法可以处理表面上看起来随机的数据(反之，如果发现可以将大块加密数据压缩许多，那么加密算法就存在缺陷)。

如果没有将数据压缩到总大小小于一个数据块的大小，就可以在加密前进行压缩。压缩后的数据越小，加密这些数据所占用 CPU 时间就越少。而且压缩后的数据都更紧凑，要使用某些攻击来破解加密数据就更困难。

13. 最坏做法：不能保护所有消息发送协议

记住在保护传输中的数据时，需要考虑系统各部分之间进行通信所使用的所有方法。例如，这些都太简单，以至于不用开发加密系统来保护客户和 Web 服务之间的通信，却忘记保护通过消息队列发送到后端的信息。Microsoft Message Queuing 这样的系统会提供加密服务，却要由您决定是否使用它们。

万一要通过多个组件来传递数据，则需要确定这些组件都没有存储或发布敏感信息。作为经验法则，组件不应该解密数据，除非它需要使用数据。多层应用程序使用的最安全的模式就是通过几个组件传递加密后的对象，并且只在最后才解密这些数据。

14. 最佳做法：存储口令散列而不是口令

如果创建定义身份验证系统，就不需要将没有加密的口令存储在数据库中。如果将没有加密的口令存储在数据库中，就正好为攻击者创建一个很有价值的攻击目标。可以存储口令散列。在攻击者可以毫无限制地访问数据库的最糟糕的情况下，对付字典攻击法的最佳方法就是使用加 salt 值的散列。第 4 章将详细讲述这项技术，第 8 章将在端对端的示例中应用它。

15. 最坏做法：没有加密的配置文件

加密处理会有系统开销。在配置应用程序时，必须考虑到这点以便可以确定使用哪些必要的硬件和潜在的吞吐量。如果将加密技术视为瓶颈，通过减少对 SSL(所有交互都不需要)的使用或安装专门的加密硬件就可以减轻系统负担。例如，许多 Web 服务器就使用 SSL 加速卡。Microsoft 的 MSDN 网站(http://msdn.microsoft.com/en-us/dnbda/html/bdadotnetarch15.asp)对不同的加密算法的性能作了有趣的比较。

16. 最佳做法：标准化编码和外部的详细信息

由于引入了应用程序级加密，所以代码变得更为复杂。要处理字符串编码、缓冲复制和流这样的详细信息就显得有些笨拙。不幸的是，代码越复杂，在适当条件下就越可能出现危及安全的错误。

为了降低这种可能性，就应该标准化这些详细信息，将这些标准形成文档，在可能的情况地方将加密信息封装到客户和服务器都可以使用的专用组件中。第 5 章介绍了该设计方案的示例。

17. 最坏做法：先加密有价值的数据，然后放宽密钥条件

当然，首先要考虑防止有恶意的用户访问密钥。不过，要是由于驱动器发生毁灭性的故障或硬件设备失灵破坏了密钥，那该怎么办呢？出现这样的问题不能做太多的工作，不过您可能会考虑使用信任的第三方存储密钥，从而防止出现这样的问题。除此之外，某些情况下不要太信赖单个密钥。例如，在第 8 章的虚拟 Web 驱动器中，使用每个用户个人的密钥来加密数据。只使用服务器密钥建立安全会话。

18. 最佳做法：保留数字签名文档的日志

如第 3 章使用 XML 签名所进行的讨论一样，不使用数字签名来防止恶意用户冒充经过身份验证的用户。他们也建立不可否认性(non-repudiation)。换言之，如果您有一个进行数字签名的股票报价定单，提出请求的客户就不能否认所进行的交易。数字签名文档的日志在解决争端时非常有用。

19. 最佳做法：减少内存中的敏感信息

如果黑客可以通过物理方式访问运行您的软件的计算机，黑客就可以有许多攻击方法。其中一种做法就是引发没有绑定的异常，然后使用调试器捕获当前内存中的内容。稍后黑客就可以搜索这些信息，查找存储的秘密信息和其他有用的信息。

一般极少会使用这种攻击方法，因为它假定心怀恶意的用户对系统最重要的部分也有访问特权。不过，使用下面的最佳做法仍然可以限制可通过内存转储攻击法恢复的信息量。

- 使用加密对象后，立即调用 Dispose()方法。在加密类中重载这种方法，这样不仅可以释

放内存资源，还可以重写包含密钥数据的那部分内存。

- 限制使用包含密钥数据或口令的字符串或字节数组。即使事后覆盖该信息，.NET 也会保留它。如果检索该信息，就将其直接放入加密对象的适当属性中。
- 如果访问数据库，将连接字符串直接检索到 Connection 对象中而不是中间字符串变量。在.NET Framework1.1 及更新的版本中，会在内存中对该信息稍做加密处理。
- 不要试图直接触发垃圾收集器。垃圾收集器只是将内存标记为可用，实际上它并没有更改这些内存中的内容。也是就是该方法对于删除敏感信息并没有什么帮助。
- 请尽可能使用 Dispose()方法。不要浪费时间使用垃圾值覆盖数据—— 这通常不能达到预定的效果，因为.NET 可以动态重新分配内存(在没有原始信息后，也会将数据副本保存较长时间)，字符串这样的引用类型不变，也就是说修改它们就可以释放对象，这样就可以废除它们。

在 ASP.NET 应用程序中，要特别小心存储在高速缓存中的信息。同一个 Web 应用程序中任何其他页面或 Web 服务都可以获得该信息。访问 Web 服务器的攻击者可以方便地创建恶意页面，这样的页面会枚举高速缓存中的内容并访问您的全部敏感数据。

```
string itemList = "";

foreach(DictionaryEntry item in Context.Cache)
{
  Response.Write(item.Key.ToString());
  Response.Write(Context.Cache[item.Key].ToString());
}
```

它也可以攻击应用程序中的信息。不过，没有方法可以直接限制对 Cache 或 Application 集合的访问。您也许需要在提高性能和 Web 服务器的安全受到威胁的可能性之间进行权衡。另一方面，访问会话状态就更难一些，因为使用随机生成的标识客户的 GUID 票据对其进行索引。

7.3　小结

本章介绍了代码易受到的攻击类型和可疑的错误类型。尽管绝大多数示例看起来都很简单，但是要在实际的代码中发现它们却相当困难，发现它们需要花费较长时间并且情况要复杂得多。多数安全缺陷都与业务代码和容易伪装的外来详细信息混合在一起。但是，安全缺陷的破坏性只有在黑客发现缺陷后才能确定有多大。

请记住，编写代码容易，而要创建真正安全的代码总是很困难，并且需要花费一定时间。安全性不能出现问题后再实现。相反，项目从设计到部署(先设计威胁模型)每个阶段都要重视安全问题。确保按正确的过程执行是惟一可以确保创建安全代码的途径。也就是说，应该安排公司内部的安全审查并制定详细的应急响应计划。甚至公布所有安全代码，这样让尽可能多的人帮助您发现代码中存在的缺陷。

最后，记住不要让用户获取所有敏感信息。如果担心通过网络发送某种数据的安全问题，就要考虑如何将某部分排除在外(不包含其中某个部分)，并减少传输的次数。

第8章　设计安全的应用程序

在前 6 章中，您已经学习了密钥数据安全的有关概念和如何利用.NET 提供的工具来实现它们的方法。但是就一个完整的应用程序而言，还需要做许多工作。将面临的挑战之一就是管理大量的详细资料，例如密钥长度、算法和字节编码等。您还需要把多种不同的加密技术合并在同一个应用程序中。例如，可能需要把加密操作和验证操作结合在一起，或者使用不同形式的加密技术来保护那些需要长期保存的数据和通过网络发送的数据。最后，您还要把密码术和数据安全技术以及其他类型的应用层安全(例如用户的身份验证和授权)结合在一起。

本章将引用一个端对端的示例把前几章中介绍的各种技术结合在一起使用。该示例演示了一个“虚拟硬盘”Web 服务，它允许用户在远程服务器上安全地保存和检索任何类型的数据。它展示的一些安全概念包括：

- 利用“不可猜测”的票据，对数据库进行基于票据的用户身份验证
- 数据(通过网络传递的数据和长期保存的数据)的加密技术
- 利用加 salt 值的口令散列代替加密的口令进行身份验证
- 防止重放攻击的时间戳
- 防止篡改消息的消息身份验证代码
- 确保长期保存的数据没有被篡改的数据验证
- 用户提供的，用来防止安全漏洞的标准值

为了展示所需的全部代码，该示例不再使用 Windows 加密文件系统(EFS)、SSL 或者其他任何类型的“自动化”加密技术和身份验证服务，尽管这些方法在使用时都是完全有效的。实际上，本章将利用自定义的加密代码开发一个完整的示例。这就需要我们探讨.NET 加密系统的完整性能，了解在构建安全的端对端应用程序时所面临的挑战，和作好评估第三方加密技术和现用组件的准备。如果您需要在预先构建的安全服务不可用的环境下，或者在创建一个高度专用化的解决方案时设计应用程序，这些技术也是非常有用的。在本章的最后，我们将回顾应用程序可能遭受到的攻击，并考虑它在哪种环境下是安全的——以及在哪些地方仍然需要改进。

说明：

在大多数情况下，最好的安全选择是利用预先构建的安全服务，例如身份验证使用的 Kerberos 协议，数据传输使用的 SSL 和 IP/Sec，以及长期保存数据时使用的 Windows 加密文件系统(EFS)。如果在一个产品应用程序中编写了自己的加密代码，必须通过密码术专家的认真评审——非专业人员会遗漏许多可能出现的弱点。

8.1 VirtualWebDrive 服务概述

该示例引入了一项允许文档被上传和检索的 Web 服务(称之为 VirtualWebDrive)。由于数据在服务器上已经加密，因此我们不能读取它，而且数据库会跟踪用户和用户创建的文件。

从概念上看，VirtualWebDrive 是一个虚拟的安全保管箱服务。每个客户机都可以接收一个箱(客户机程序)，它们在箱中可以锁定一些项目(数据)。然后把该箱发送给 Web 服务，Web 服务又把箱放置在一个受保护的位置上。客户机可以应要求找出它们所提交的箱的数量，并请求返回一个箱。但是，Web 服务本身不能参加对客户机数据的加密和解密(虽然它需要使用加密技术来保护报文通信)。这种情况下所面临的真正挑战是不能创建客户机数据上的锁以及安全的服务器端存储器的锁——它可以识别客户机，验证它们是否被模拟，并确保它们不能随意接收其他用户的箱。

VirtualWebDrive Web 服务总共包括 6 种方法：

- 第一种方法是 GetKey()，它可以获取 Web 服务的公钥。
- Login()和 Logout()方法都可以开始或结束一次客户机会话。
- 剩下的 3 种 Web 方法 GetFileInfo()、GetFile()和 SaveFile()都可以获取与当前存储的文件相关的信息，并上传或者下载单独的文件。

在一个较为复杂的系统中，您可能要添加其他的 Web 方法来管理被存储的文件(例如，允许用户删除一个被存储的文件)，并包含一种机制来自动创建用户记录。当前，这些方法都需要管理员利用一个单独的实用程序手工添加。

用户信息被存储在包含两个表的后端数据库中，如图 8-1 所示。第一个表 Users 包含一系列用户和他们的口令。第二个表 Files 列出了已经被上传的文件，文件在服务器上的路径，以及拥有这些文件的用户。已上传文件的真正内容没有保存在数据库中，而是保存在服务器硬盘中。这样就可以确保数据库查询和管理操作始终都可以迅速执行。

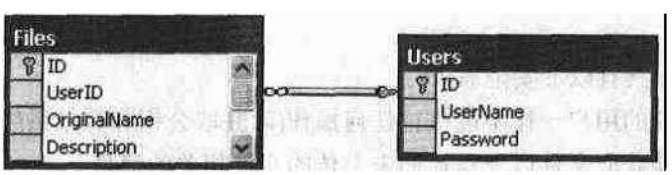

图　8-1

为了确保文件名是随机的，GUID 值可以由服务器创建并透明地使用。您可以使用一些其他的方法(例如序列号)，但是 GUID 在安全的程序设计中是非常有用的，因为在一般情况下很难准确地猜出它们。例如，给定用户会知道可以标识他们文件的 GUID。这样就不能轻松地猜出其他用户的文件使用什么 GUID 值，使一些类型的攻击更加难以进行。序列号或者其他添加了随机数字的时间戳都很容易实现逆向工程。

加密逻辑被封装在一个客户机和服务器都能使用的专用组件程序集中。该示例可以很好地说明如何在不需混合使用加密代码和业务逻辑的前提下执行应用层加密，但这并不意味着代码和.NET Framework 紧密地联系在一起，并因而更加难于使用跨平台的客户机。这就是操作的简易性和互操作性之间的关键折衷点。严格地说，客户机不需要使用自定义的组件，但没有它的

话，客户机将难以按照正确的顺序复制加密步骤。

VirtualWebDrive 示例(如图 8-2 所示)还包含一个能够处理 Web 服务的专用 Windows 客户机。它可以为上传和获取文件提供用户接口，并执行一些加密操作。Windows 客户机和 Web 服务共享同一个组件，对于所有的加密任务来说，它们都在一个较高的层次上使用该组件。

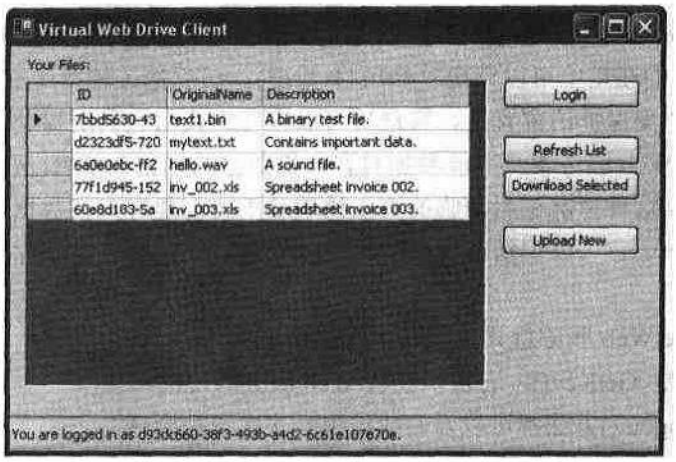

图　8-2

说明：

为了快速了解 Windows 客户机的工作原理，您可能想直接跳到本章最后一部分讲述客户机的内容中。您可以从 http://www.prosetech.com 或 http://www.wrox.com/books/1861008015.htm 下载完整的代码(和安装数据库的脚本)。

8.2　安全分析

VirtualWebDrive 具有以下安全需求：

- 未经身份验证的用户一律不能执行任何操作(除获取公钥和登录系统之外)。
- 用户一律不能获取文件以及与他们未上传的文件相关的信息。
- 客户机和 Web 服务之间的数据传递应该以密钥和其他信息都不可解密的方式加密。除此之外，使重放攻击不可能发生。
- 即使攻击者有权访问服务器，他们也看不到数据，已存储的数据应该以这种方式加密。从理论上看，服务器组件无法解密文件，因为它不需要检查所包含的数据。因此，即使服务器受到损害(甚至被盗窃)，攻击者只能获取口令散列和完整的加密数据。
- 应该实现一些安全措施来验证文件在上传过程和下载过程期间没有被篡改。在 VirtualWebDrive 服务中，这些检验都是由服务器执行的。

第一个要求利用基于票据的身份验证系统是很容易实现的。在该系统下，程序必须在进行其他任何操作之前调用 Web 服务 Login()方法接收票据。同时执行密钥交换(如第 5 章中演示的那样)也是很有意义的。客户机创建一个随机对称密钥，该密钥可以加密交互作用的剩余数据。

如图 8-3 所示：长期保存的数据被客户机预先加密，并以加密的形式保存在服务器的硬件设备中。用来加密该数据的密钥与服务器密钥和会话中使用的密钥隔离开，以确保数据不能在传输过程或服务器端读取。要想加密文件，就要生成并使用一个随机的对称密钥。对称密钥利用客户机的公钥加密，并保存在文件数据的开头。客户机的非对称密钥可以通过一些安全机制(例如客户机证书)提供。在 VirtualWebDrive 示例中，只是对其进行硬编码。

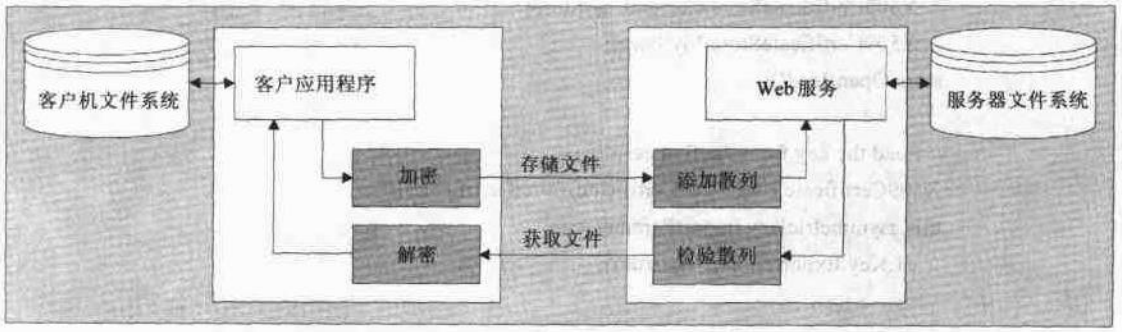

图　8-3

8.3　VirtualWebDrive 组件

我们将要介绍的第一组类属于服务和客户机使用的组件。该组件包括加密逻辑本身。我们可以把该组件中包含的功能再细分为子组件，一个包含与安全相关的逻辑，另一个包含与数据库相关的逻辑。下面我们依次对其进行介绍。

8.3.1　安全组件

安全组件是一组被编译为单个 DLL 程序集的类，它被分布给客户机和服务器。安全组件的目标是隐藏低级的加密信息并用高级的加密“任务”替换它们。从理论上看，客户机不必担心格式、密钥、流的用法等就可以调用诸如“EncryptDataForServer”这样的方法。

安全组件包括两个核心部分：在通过连接发送数据之前加密数据和在接收数据时对其解密的类，和加密文件的类(在客户端执行)。

1. 文件加密类

文件加密类是最直观的。它需要一个非对称密钥，用来加密动态生成的对称密钥。该类的框架代码如下所示：

```
public sealed class FileEncryptor
{
    private RSACryptoServiceProvider asymmetricKey;

    public FileEncryptor()
    {
        this.asymmetricKey = new RSACryptoServiceProvider ();
```

```
            // You can load the key from some source with the
            // RSACryptoServiceProvider.FromXmlString() method.
            // In the following implementation, the key is retrieved
            // from the user's certificate store.
            // This requires WSE (as explained in Chapter 2).
            X509CertificateStore store =
              X509CertificateStore.CurrentUserStore(
              X509CertificateStore.MyStore);
            store.OpenRead();

            // Read the key from the first certificate.
            X509Certificate c = (X509Certificate)store.Certificates[0];
            this.asymmetricKey.ImportParameters(
              c1.Key.ExportParameters(true));
        }

        public byte[] EncryptFile(byte[] fileContent)
        { ... }

        public byte[] DecryptFile(byte[] fileContent)
        { ... }
    }
```

注意，在安全的程序设计中，您始终都应该把类声明为密封的，因为这样恶意代码就不可能从您的类中继承和重写它的功能。

EncryptFile()方法的完整代码如下所示。在该示例中，我们使用的是 Rijndael 算法，它可以提供在.NET 平台下使用的最强的密钥长度。把初始向量设为一个空的字节数组，因为随机生成的密钥所提供的安全被认为是足够强的。但是，为了更有力地抵御蛮力攻击，您可以使用非 0 的 IV 并将它和对称密钥数据一起保存在文件中。这两部分数据都将利用客户机的非对称密钥加密。

```
public byte[] EncryptFile(byte[] fileContent)
{
    // Create the random key.
    SymmetricAlgorithm key =   Rijndael.Create ();
    key.IV = new byte[key.IV.Length];
    byte[] keyData = asymmetricKey.Encrypt(key.Key, false);

    // Write the random key data (which has been encrypted using
    // the asymmetric key) and key size information
    // to the beginning of the file.
    MemoryStream ms = new MemoryStream();
    byte[] keyDataLength = BitConverter.GetBytes(keyData.Length);
    ms.Write(keyDataLength, 0, keyDataLength.Length);
```

```
        ms.Write(keyData, 0, keyData.Length);

        // Now encrypt and write the file data with the symmetric key.
        CryptoStream cs = new CryptoStream(ms,
        key.CreateEncryptor(), CryptoStreamMode.Write);
        cs.Write(fileContent, 0, fileContent.Length);
        cs.FlushFinalBlock();

        // Return all the data.
        return ms.ToArray();
    }
```

注意，EncryptFile()方法要按顺序预先考虑文件内容前的两部分信息：用字节表示的加密密钥的长度，和加密密钥数据。注意加密类是用指定算法的形式(而不是利用基本的SymmetricAlgorithm.Create()方法)创建的。这样可以保证所使用的算法总是相同的。否则从理论上看，如果指定机器的配置已经改变，那么客户机就利用同一种方法创建出不同的加密算法。

对应的 DecryptFile()方法代码如下所示：

```
public byte[] DecryptFile(byte[] fileContent)
{
    // Retrieve the key size information.
    // This is the only unencrypted part of the file.
    byte[] keyDataLength = new byte[4];
    Buffer.BlockCopy(fileContent, 0, keyDataLength,
      0, keyDataLength.Length);
    int keyLength = BitConverter.ToInt32(keyDataLength, 0);

    // Check that the keyLength makes sense.
    // Otherwise a hacked file could conceivably trick the client
    // into allocating a huge amount of memory, and crashing.
    if (keyLength > fileContent.Length)
    {
        throw new ApplicationException("Invalid file data.");
    }

    // Retrieve the random symmetric key that was used,
    // and decrypt it.
    byte[] keyData = new byte[keyLength];
    Buffer.BlockCopy(fileContent, keyDataLength.Length,
                     keyData, 0, keyLength);
    SymmetricAlgorithm key = Rijndael.Create();
    key.IV = new byte[key.IV.Length];
    key.Key = asymmetricKey.Decrypt(keyData, false);

    // Decrypt the remaining file contents with the symmetric key.
```

```
        MemoryStream ms = new MemoryStream();
        CryptoStream cs = new CryptoStream(ms,
                key.CreateDecryptor(), CryptoStreamMode.Write);
        cs.Write(fileContent, keyDataLength.Length + keyLength,
                fileContent.Length - keyDataLength.Length - keyLength);
        cs.FlushFinalBlock();

        // Return the decrypted data.
        return ms.ToArray();
    }
```

该类最大的优点就是它可以封装用来查询动态加密文件密钥的规则。执行该项任务没有任何标准，因此不同的实现可以在文件的开头用不定数量的字节保存密钥，也可以在文件的末尾保存，其中可以包含密钥长度信息，也可以不包含。它还可以利用 XML 加密标准把数据和密钥信息分离，但是您需要通过编程方式创建一个 XML 文档(因为.NET 目前不能提供一个 XML 加密 API)。

2. 会话加密类

会话加密类可以对那些需要通过连接发送的其他任何类型的数据加密。该信息可以利用动态生成的会话密钥加密。

会话加密类比文件加密类更复杂，这是因为它们在客户机和服务器上的操作是不同的，而且需要处理不同类型的数据。为了支持这种设计，VirtualWebDrive 要使用 3 种类。核心功能包含在一个名为 SecureSession 的抽象类中。基于该类构建的其他两个类 SecureClientSession 和 SecureServerSession 可以提供高级加密任务。

SecureSession 基类

如下所示为 SecureSession 抽象类的框架代码。它定义了能够对称加密或非对称加密任何可串行化对象的核心功能。您还会注意到它包括对非对称服务器密钥(Web 服务提供的)的引用和对称客户机密钥(供会话使用，由客户机动态生成)的引用。

```
public abstract class SecureSession
{
    protected RSACryptoServiceProvider serverKey;
    protected SymmetricAlgorithm clientKey;

    protected byte[] EncryptSymmetric(object objectToEncrypt)
    { ... }

    protected object DecryptSymmetric(byte[] dataToDecrypt)
    { ... }

    protected byte[] EncryptAsymmetric(object objectToEncrypt)
    { ... }
```

```
        protected object DecryptAsymmetric(byte[] dataToDecrypt)
        { ... }

        protected bool CompareByteArray(byte[] array1, byte[] array2)
        { ... }
    }
```

说明：

利用对象串行化后不需要为每个类都编写自定义的代码就可以容易地加密很大范围内的可串行化对象。但是，如果串行化一个.NET Framework 对象，而且运行 Web 服务的框架版本与运行客户机的版本不同，就会出现串行化问题。解决办法是只对您可以确定版本和控制的自定义对象进行串行化。

SecureSession 代码有些冗长。在对称方面，要求所有的对称加密都必须是基于流的。因此，要想对称加密数据，需要首先将其转换为流。

除此之外，EncryptSymmetric()方法使用一种消息身份验证代码，该代码可以利用客户机密钥计算散列码，并将其添加到消息数据的最后。

```
protected byte[] EncryptSymmetric(object objectToEncrypt)
{
    MemoryStream ms = new MemoryStream();
    CryptoStream cs = new CryptoStream(ms,
    clientKey.CreateEncryptor(), CryptoStreamMode.Write);

    BinaryFormatter f = new BinaryFormatter();
    f.Serialize(cs, objectToEncrypt);
    cs.FlushFinalBlock();

    // Create hash code.
    KeyedHashAlgorithm hash = HMACSHA1.Create();
    hash.Key = clientKey.Key;
    ms.Position = 0;
    hash.ComputeHash(ms);

    // Write the hash code to the end of the message.
    // Because it is a keyed hash code, it does not need further
    // encryption.
    ms.Write(hash.Hash, 0, hash.Hash.Length);

    return ms.ToArray();
}
```

DecryptSymmetric() 方法首先要验证消息身份验证代码以确保数据没有被篡改。然后它再反串行化对象。

```
protected object DecryptSymmetric(byte[] dataToDecrypt)
{
    // Extract the hash from the data.
    byte[] data = new byte[dataToDecrypt.Length - hash.HashSize / 8];
    Buffer.BlockCopy(dataToDecrypt, 0, data, 0, data.Length);
    byte[] MAC = new byte[hash.HashSize / 8];
    Buffer.BlockCopy(dataToDecrypt, data.Length, MAC, 0, MAC.Length);

    // Recalculate the keyed hash code.
    KeyedHashAlgorithm hash = HMACSHA1.Create();
    hash.Key = clientKey.Key;
    hash.ComputeHash(data, 0, data.Length);

    // Verify the calculated hash with the hash on the message.
    if (!this.CompareByteArray(hash.Hash, MAC))
    {
        throw new ApplicationException("Invalid data.");
    }

    MemoryStream ms = new MemoryStream();
    CryptoStream cs = new CryptoStream(ms,
                                       clientKey.CreateDecryptor(),
                                       CryptoStreamMode.Write);

    cs.Write(data, 0, data.Length);
    cs.FlushFinalBlock();

    // Now deserialize the decrypted memory stream.
    ms.Position = 0;
    BinaryFormatter f = new BinaryFormatter();
    return f.Deserialize(ms);
}
```

非对称加密方法较为复杂。我们在这里需要考虑到非对称加密每次只能处理一个块。这对我们前面提到的每次只加密一个数据(密钥信息)块的文件加密示例是非常好的。但是较长的数据需要逐块加密，然后再拼凑到一起。下面列出的代码与第 2 章引用的非对称示例非常相似，并且该示例在第 2 章中已经描述得很详细。牢记 VirtualWebDrive 服务只能在交互作用的最开头利用非对称加密来处理密钥交换。

```
protected byte[] EncryptAsymmetric(object objectToEncrypt)
{
    // Create the memory streams.
    MemoryStream msRaw = new MemoryStream();
    MemoryStream msEncrypted = new MemoryStream();
```

```
        BinaryFormatter f = new BinaryFormatter();
        f.Serialize(msRaw, objectToEncrypt);

        byte[] bytes = msRaw.ToArray();

        // Determine the optimum block size for encryption.
        int blockSize = 0;
        if (serverKey.KeySize == 1024)
        {
            blockSize = 16;
        }
        else
        {
            blockSize = 5;
        }

        // Move through the data one block at a time.
        byte[] rawBlock, encryptedBlock;
        for (int i = 0; i < bytes.Length; i += blockSize)
        {
            if ((bytes.Length - i) > blockSize)
            {
                rawBlock = new byte[blockSize];
            }
            else
            {
                rawBlock = new byte[bytes.Length - i];
            }

            // Copy a block of data.
            Buffer.BlockCopy(bytes, i, rawBlock, 0, rawBlock.Length);

            // Encrypt the block of data.
            encryptedBlock = serverKey.Encrypt(rawBlock, false);

            // Write the block of data.
            msEncrypted.Write(encryptedBlock, 0, encryptedBlock.Length);
        }

        return msEncrypted.ToArray();
    }

    protected object DecryptAsymmetric(byte[] dataToDecrypt)
    {
        // Create the memory stream where the decrypted data
```

```
        // will be stored.
        MemoryStream msDecrypted = new MemoryStream();

        // Determine the block size for decrypting.
        int keySize = serverKey.KeySize / 8;

        // Move through the data one block at a time.
        byte[] decryptedBlock, rawBlock;
        for (int i = 0; i < dataToDecrypt.Length; i += keySize)
        {
            if ((dataToDecrypt.Length - i) > keySize)
            {
                rawBlock = new byte[keySize];
            }
            else
            {
                rawBlock = new byte[dataToDecrypt.Length - i];
            }

            // Copy a block of data.
            Buffer.BlockCopy(dataToDecrypt, i, rawBlock, 0,
                             rawBlock.Length);

            // Decrypt a block of data.
            decryptedBlock = serverKey.Decrypt(rawBlock, false);

            // Write the decrypted data to the in-memory stream.
            msDecrypted.Write(decryptedBlock, 0, decryptedBlock.Length);
        }

        msDecrypted.Position = 0;
        BinaryFormatter f = new BinaryFormatter();
        return f.Deserialize(msDecrypted);
    }
```

最后一种 SecureSession 方法是 CompareByteArray()，它在测试散列值时非常有用：

```
    protected bool CompareByteArray(byte[] array1, byte[] array2)
    {
        if (array1.Length != array2.Length)
            return false;
        for (int i = 0; i < array1.Length; i++)
        {
            if (array1[i] != array2[i])
                return false;
        }
```

```
        return true;
    }
```

该代码非常普通，您可以考虑在一个单独的实用程序类中实现它，并利用它处理多个加密组件。但是，在实际操作中它很容易偏离该逻辑，因此您不需要拆散应用程序的一些相关部分就可以修改它来处理不同的任务。注意所有这些方法都是受保护的，这就意味着除了那些派生自 SecureSession 的类之外其他任何类都不能使用。

我们从 SecureSession 中派生了两个专用的类：一个专门为服务器使用，另一个专门为客户机使用。

```
public class SecureClientSession : SecureSession
{ ... }

public class SecureServerSession : SecureSession
{ ... }
```

这些类可以在服务器和客户机上包装指定的加密任务，我们将在下一小节中详细描述。

3. 登录过程

要想理解 SecureClientSession 和 SecureServerSession 类的工作原理，了解客户机在访问 VirtualWebDrive 服务时所经历的过程对此是非常有帮助的。首先它们要设法进入系统。为了执行该登录操作，客户机必须提交一些被 LoginInfo 类封装的证书：

```
[Serializable()]
public class LoginInfo
{
    public string UserName;
    public byte[] Password;
    public DateTime CreatedTime;
    public byte[] ClientKey;

    public LoginInfo(string userName, string password,
                     byte[] clientKey, DateTime serverDate)
    {
        this.UserName = userName;
        this.Password = password;
        this.CreatedTime = serverDate;
        this.ClientKey = clientKey;
    }

    public LoginInfo()
    {
        // Required for deserialization.
    }
}
```

LoginInfo 类中包括用户名和密码，还包括一个 CreatedTime 字段。这是因为 LoginInfo 对象中的数据将被串行化并加密为一个单独的字节数组。通过所包含的日期和时间，服务器可以拒绝旧的登录包，并防止黑客重新使用原来被截断的网络通信(也就是我们常说的重放攻击)。除此之外，LoginInfo 还包括一个随机生成的对称密钥，将来所有的通信都可以使用它(假设登录成功的话)。

说明：

为了避免出现时区问题(或者时间差异)，客户机必须使用服务器的时间。Web 服务为此提供了一种 GetServerDate()方法。客户机调用该方法，并对 LoginInfo 包使用返回时间。

图 8-4 所示的是创建、提交，然后再验证登录信息的过程。该图还列出了数据库验证的步骤，我们将在下一小节的内容中对其讨论。

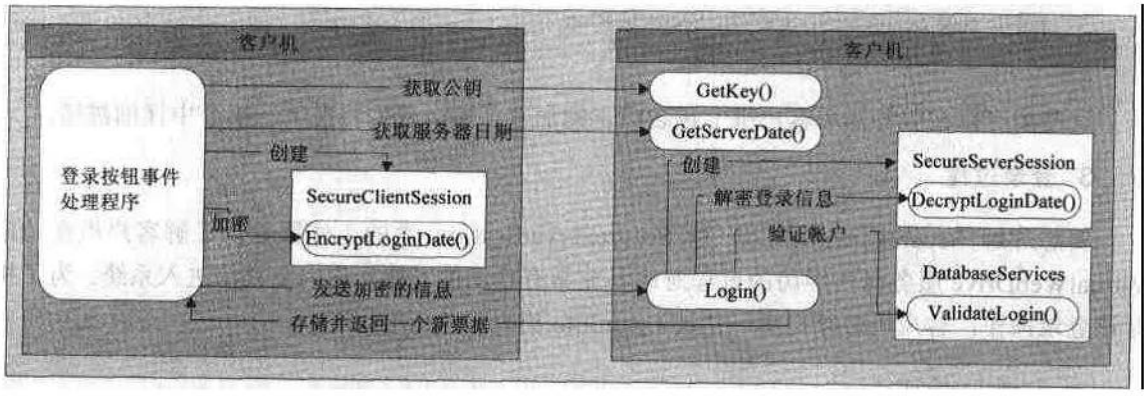

图　8-4

客户机通过创建类的一个实例开始使用 SecureClientSession。这时它需要以 XML 字符串的形式提供服务器密钥的公共部分。该信息是 SecureClientSession 构造函数必需的。

```
public SecureClientSession(string serverKeyXml)
{
    this.serverKey = new RSACryptoServiceProvider();
    this.serverKey.FromXmlString(serverKeyXml);
}
```

现在，客户机利用用户名和密码调用 SecureClientSession 的 EncryptLoginData()方法。该方法将创建一个带有客户机信息的 LoginInfo 对象和一个随机生成的密钥，并利用服务器的非对称密钥将其加密为一个字节数组。然后它将返回一个字节数组结果。

```
public byte[] EncryptLoginData(string userName, string password)
{
   this.clientKey = (SymmetricAlgorithm)(new RijndaelManaged());
   this.clientKey.IV = new byte[this.clientKey.IV.Length];

   LoginInfo loginInfo = new LoginInfo(userName, password,
                                       clientKey.Key);
```

```
        return this.EncryptAsymmetric(loginInfo);
    }
```

当我们检验 Windows 客户机时，将看到这样的操作会使客户机代码变得非常简单。

SecureServerSession 类与此非常相似，但是 Web 服务需要执行定制的服务。它的构造函数需要非对称密钥对象。

```
public SecureServerSession(RSACryptoServiceProvider serverKey)
{
    this.serverKey = serverKey;
}
```

SecureServerSession 类的 DecryptLoginData()方法可以接收原始的 LoginInfo 对象。在客户机上，EncryptObjectForClient()和 DecryptObjectFromClient()都允许 SecureServerSesion 对象处理对称加密的数据。

```
public LoginInfo DecryptLoginData(byte[] loginData)
{
    return (LoginInfo)this.DecryptAsymmetric(loginData);
}
```

对于其他所有的方法来说，客户机和服务器之间的交互作用是由对称加密控制的。SecureClientSession 类为该任务提供了其他两种方法，它们可以利用基本的 SecureSession 方法加密和解密对象。

```
public byte[] EncryptObjectForServer(object objectToEncrypt)
{
    return this.EncryptSymmetric(objectToEncrypt);
}

public object DecryptObjectFromServer(byte[] dataToDecrypt)
{
    return this.DecryptSymmetric(dataToDecrypt);
}
```

SecureServerSession 类使用下面两种补充的方法：

```
public byte[] EncryptObjectForClient(object objectToEncrypt)
{
    return this.EncryptSymmetric(objectToEncrypt);
}

public object DecryptObjectFromClient(byte[] dataToDecrypt)
{
    return this.DecryptSymmetric(dataToDecrypt);
}
```

记着，Web 服务是无状态的，而且每次可以和数百个客户机发生交互操作。在每个客户机交互作用的开始，Web 服务因此都需要获取适当的客户机密钥，并将其应用于 SecureServerSession 类。这些操作是通过该类的 SetClientKey()方法实现的。

```
public void SetClientKey(byte[] keyData)
{
    this.clientKey = Rijndael.Create();
    this.clientKey.Key = keyData;
    this.clientKey.IV = new byte[this.clientKey.IV.Length];
}
```

最后，两种附加的帮助方法允许 Web 服务创建并验证文件散列。由于使用了加密类，您可以直接创建一个指定的散列算法实现类以防止出现配置被修改的错误。利用基本的 SecureSession.CompareByteArray()方法可以测试散列是否相等。

```
public byte[] CalculateHash(byte[] data)
{
    HashAlgorithm hash =
      (HashAlgorithm)SHA1CryptoServiceProvider.Create();
    return hash.ComputeHash(data, 0, data.Length);
}

public bool VerifyHash(byte[] data, byte[] storedHash)
{
    HashAlgorithm hash = SHA1CryptoServiceProvider.Create();

    return this.CompareByteArray(hash.ComputeHash(data, 0,
     data.Length), storedHash);
}
```

这样在 VirtualWebDrive 系统中用来保护通信的加密代码就完成了。

8.3.2　数据库组件

确定您在开发周期的前期需要执行的数据库操作也非常重要。在 VirtualWebDrive 示例中，这些操作包括添加文件记录、检索文件信息和验证登录信息。从理论上看，这些任务都将通过存储过程执行。在该示例中，我们利用的是带参数的命令。它可以保护数据库不受 SQL 注入攻击并提供一种简单的存储过程的代码迁移路径。

首先要定义一个可串行化的 FileInfo 结构，它可以和描述文件记录的信息捆绑在一起。注意这并不包括真正的文件数据，真正的文件数据被单独保存。与所有的可串行化类一样，该类可以把信息发送给 Web 服务。

```
[Serializable()]
public class FileInfo
{
```

```
    public Guid ID;
    public int UserID;
    public string OriginalName;
    public string Description;
    public string ServerPath;
    public byte[] Hash;

    public FileInfo(string originalName, string description)
    {
        this.OriginalName = originalName;
        this.Description = description;
    }

    public FileInfo()
    {
        // Required for deserialization.
    }
}
```

数据库代码被封装在一个单独的名为 DatabaseServices 的类中。在创建类时，可以从配置文件(在该示例中，指的是存放 Web 服务的虚拟目录的 web.config 文件)中读取数据库的连接字符串。连接字符串应该只授予修改 Files 表和访问 Users 表所需的权限。

```
public class DatabaseServices
{
    private string connectionString;

    public DatabaseServices()
    {
        connectionString =
          ConfigurationSettings.AppSettings["DatabaseConnection"];
    }

    // (Other code omitted.)
}
```

DatabaseServices 类提供了 4 种方法，它们都使用带参数的命令并按照几乎完全相同的方式进行操作(编写安全代码的最佳方法是编写令人厌烦的代码)！AddUser()方法可以创建一个新的用户记录，而且在把密码存储在数据库之前自动散列它并为其加 salt 值。为了支持该技术，可以使用在第 4 章中引入的 HashHelper 类。

```
public void AddUser(string userName, string password)
{
    // Define the SQL string with named parameters.
    string SQL = "INSERT Users (UserName, Password) VALUES " +
```

```
        "(@UserName, @Password)";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(SQL, con);

    // Add the parameter values to the command.
    SqlParameter param;
    param = cmd.Parameters.Add("@UserName", SqlDbType.VarChar, 50);
    param.Value = userName;
    param = cmd.Parameters.Add("@Password", SqlDbType.VarBinary, 24);
    // Compute the salted password hash.
    HashHelper hashUtil = new HashHelper();
    param.Value = hashUtil.CreateDBPassword(password);

    // Add the record to the database.
    try
    {
        con.Open();
        cmd.ExecuteNonQuery();
    }
    catch (Exception err)
    {
        System.Diagnostics.Debug.WriteLine(err.ToString());
        throw new ApplicationException("Database error.");
    }
    finally
    {
        con.Close();
    }
}
```

注意 Web 服务不能直接使用 AddUser()方法。实际上，该方法由一个快速允许您为数据库添加用户记录的专业管理实用程序来使用。该实用程序和下载代码包括在一起，但这里不再列出。

ValidateLogin()方法可以搜索数据库中的用户，并返回相应的用户 ID 编号和口令散列。然后在 HashHelper 类的帮助下验证口令散列。如果没有与所提供的信息相符的用户，口令散列不匹配，或者时间戳不正确，那么 ValidateLogin()就会抛出一个异常。

```
public int ValidateLogin(LoginInfo loginInfo)
{
    if (loginInfo.CreatedTime.AddMinutes(5) < DateTime.Now ||
        loginInfo.CreatedTime > DateTime.Now)
    {
        throw new ApplicationException("Invalid login attempt.");
    }
```

```
// Define the SQL string with named parameters.
string SQL = "SELECT ID, UserName, Password From Users WHERE " +
 "UserName=@UserName";
SqlConnection con = new SqlConnection(connectionString);
SqlCommand cmd = new SqlCommand(SQL, con);

// Add the parameter values to the command.
SqlParameter param;
param = cmd.Parameters.Add("@UserName", SqlDbType.NChar, 50);
param.Value = loginInfo.UserName;

int userID = null;
byte[] passwordHash = null;
try
{
    con.Open();
    SqlDataReader r = cmd.ExecuteReader();

    if (r.Read())
    {
        userID = (int)r["ID"];
        passwordHash = (byte[])r["Password"];
    }
}
catch (Exception err)
{
    System.Diagnostics.Debug.WriteLine(err.ToString());

    throw new ApplicationException("Database error.");
}
finally
{
    if (r!=null) r.Close();
    con.Close();
}

// Verify the password hash.
HashHelper hashUtil = new HashHelper();
if (hashUtil.ComparePasswords(passwordHash,
    loginInfo.PasswordHash))
{
    return userID;
}

else
```

```
        {
            throw new ApplicationException("Invalid login attempt.");
        }
    }
```

注意如何用一个不能向意欲进行攻击的人展示任何信息的非关键性 ApplicationException 来代替出现的任何异常。在测试该代码时，您可能需要注释出该逻辑来简化故障的标记符(尽管可以依靠日志)。

GetFilesForUser()方法利用一个用户 ID，并返回一系列相关的文件(就像具有一个 DataTable 的 DataSet 一样)。另外，信息还可以以 FileInfo 对象数组的形式返回，但是 DataSet 方法可以简化客户端数据绑定的操作。

```
public DataSet GetFilesForUser(int userID)
{
    // Define the SQL string with named parameters.
    string SQL = "SELECT ID, OriginalName, Description FROM " +
      "Files WHERE UserID=@UserID ORDER BY CreateTime";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(SQL, con);

    // Add the parameter values to the command.
    SqlParameter param;
    param = cmd.Parameters.Add("@UserID", SqlDbType.Int);
    param.Value = userID;

    SqlDataAdapter adapter = new SqlDataAdapter(cmd);
    DataSet ds = new DataSet("Virtual Web Drive");
    try
    {
        con.Open();
        adapter.Fill(ds);
    }
    catch (Exception err)
    {
        // Log error here.

        System.Diagnostics.Debug.WriteLine(err.ToString());
        throw new ApplicationException("Database error.");
    }
    finally
    {
        con.Close();
    }
    return ds;
}
```

注意 GetFilesForUser()方法只能返回 3 部分对用户很有意义的信息。有关正确的服务器路径、散列码等信息不能返回。另一方面，GetFileInfo()方法可以返回那些 Web 服务在把文件返回给用户之前查找并验证文件时所需要的信息。您可以创建一些更普通的方法，利用它们来返回所有信息，而且以后还可以手工删除那些不重要的详细资料，但是这种方法在该示例中更有效。

```
public FileInfo GetFileInfo(Guid fileID)
{
    // Define the SQL string with named parameters.
    string SQL = "SELECT ServerPath, Hash, UserID FROM " +
      "Files WHERE ID=@ID";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(SQL, con);

    // Add the parameter values to the command.
    SqlParameter param;
    param = cmd.Parameters.Add("@ID", SqlDbType.UniqueIdentifier);
    param.Value = fileID;

    FileInfo file = null;
    try
    {
        con.Open();
        SqlDataReader r = cmd.ExecuteReader();

        if (r.Read())
        {
            file = new FileInfo();
            file.UserID = (int)r["UserID"];
            file.ServerPath = r["ServerPath"].ToString();
            file.Hash = (byte[])r["Hash"];
        }

        r.Close();
    }
    catch (Exception err)
    {
        // Log error here.
        System.Diagnostics.Debug.WriteLine(err.ToString());

        throw new ApplicationException("Database error.");
    }
    finally
    {
        con.Close();
    }
```

```
    return file;
}
```

最后，AddFile()方法基于信息向 FileInfo 对象中插入一个新的文件记录。该方法不能生成相同的 GUID，它们将由 Web 服务提供(并用于文件名中)。

```
public void AddFile(FileInfo fileInfo)
{
    // Define the SQL string with named parameters.
    string SQL = "INSERT INTO Files (ID, UserID, OriginalName, " +
      "ServerPath, Hash, Description) VALUES (@ID, @UserID, " +
      "@OriginalName, @ServerPath, @Hash, @Description)";
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand(SQL, con);

    // Add the parameter values to the command.
    SqlParameter param;
    param = cmd.Parameters.Add("@ID", SqlDbType.UniqueIdentifier);
    param.Value = fileInfo.ID;
    param = cmd.Parameters.Add("@UserID", SqlDbType.Int);
    param.Value = fileInfo.UserID;
    param = cmd.Parameters.Add("@OriginalName", SqlDbType.VarChar,
                               50);
    param.Value = fileInfo.OriginalName;
    param = cmd.Parameters.Add("@ServerPath", SqlDbType.VarChar, 100);
    param.Value = fileInfo.ServerPath;
    param = cmd.Parameters.Add("@Hash", SqlDbType.Binary, 20);
    param.Value = fileInfo.Hash;
    param = cmd.Parameters.Add("@Description", SqlDbType.VarChar,
                               200);
    param.Value = fileInfo.Description;

    try
    {
        con.Open();
        cmd.ExecuteNonQuery();
    }
    catch (Exception err)
    {
        // Log error here.

        System.Diagnostics.Debug.WriteLine(err.ToString());
        throw new ApplicationException("Database error.");
    }
    finally
    {
```

```
            con.Close();
        }
    }
```

8.4　VirtualWebDrive 服务

既然安全和数据库的基础结构已经安排就绪，那么构建客户机和 Web 服务就相当简单了。第一步要创建一个存储文件路径和数据库连接字符串的 web.config 文件。如下所示(但是您需要根据所使用的测试系统调整文件路径和数据库连接字符串)：

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

  <appSettings>
    <add key="DatabaseConnection"
     value="Initial Catalog=Wrox;Data Source=localhost;
     Integrated Security=SSPI" />
    <add key="FilePath" value="f:\VirtualWeb\UserFiles\" />
  </appSettings>

  <system.web>
    <!-- Other web application settings go here. -->
  </system.web>
</configuration>
```

VirtualWebDriveService 类定义了 3 种私有成员变量。一个用来存储非对称加密密钥，其他两个分别引用 SecureServerSession 和 DatabaseServices 类的实例。

```
public class VirtualWebDriveService : System.Web.Services.WebService
{
    private RSACryptoServiceProvider crypt;
    private VirtualWebDriveComponent.SecureServerSession session;
    private VirtualWebDriveComponent.DatabaseServices DB;

    // (Code omitted.)
}
```

Web 服务的构造函数可以初始化这些值：

```
public VirtualWebDriveService()
{
    InitializeComponent();
    this.crypt = GetKeyFromState();
    this.session = new
```

```
            VirtualWebDriveComponent.SecureServerSession(crypt);
        this.DB = new VirtualWebDriveComponent.DatabaseServices();
    }
```

记着，Web 服务是完全无状态的。因此该构造函数要在每个客户机请求的开头调用。因为每个客户机都可以接收一个新的 VirtualWebDriveService 实例，所以成员变量之间就不可能发生同步冲突。

私有的帮助方法可以从应用程序状态那里接收非对称 Web 服务密钥对。这就可以确保 Web 服务的所有实例都可以访问同一个密钥。

```
private RSACryptoServiceProvider GetKeyFromState()
{
    RSACryptoServiceProvider crypt = null;

    // Check if the key has been created yet.
    // This ensures that the key is only created once,
    // the first time this web service is accessed.
    // Alternatively, this key could be retrieved from a secure
    // storage location.
    if (Application["Key"] == null)
    {
        // Create a key for RSA encryption.
        CspParameters param = new CspParameters();
        param.Flags = CspProviderFlags.UseMachineKeyStore;
        crypt = new RSACryptoServiceProvider(param);

        // Store the key in the server memory.
        Application["Key"] = crypt;
    }
    else
    {
        crypt = (RSACryptoServiceProvider)Application["Key"];
    }
    return crypt;
}
```

GetKey()Web 方法允许客户机获取该密钥的公共部分：

```
// Return the public portion of the key only.
[WebMethod()]
public string GetKey()
{
    return crypt.ToXmlString(false);
}
```

8.4.1　基于票据的身份验证

基于票据的身份验证是服务的关键部分。它要求那些非常耗时的数据库验证操作只在会话的开头进行一次。在成功的登录时，客户机发布一个能够为剩余会话快速验证客户机的票据。您可以自定义票据的失效期以及它是如何失效的。我们在第 5 章中介绍过基于票据的身份验证。

在利用基于票据的身份验证时首先要定义一个可以封装所有票据详细资料的类。在该示例中，票据可以存储用户 ID、用户的 IP 地址、票据创建的时间以及客户密钥：

```
public class TicketInfo
{
    public int UserID;
    public DateTime LastUsedDate;
    public string IPAddress;
    public byte[] ClientKey;

    public TicketInfo(int userID, string IPAddress, byte[] clientKey)
    {
        this.UserID = userID;
        this.LastUsedDate = DateTime.Now;
        this.IPAddress = IPAddress;
        this.ClientKey = clientKey;
    }
}
```

跟踪 IP 地址并不能提高安全性，因为在许多情况下，整个组织都将作为一个代理服务器或者使用网络地址转换把多台计算机映射到一个 IP 地址上。但是，跟踪 IP 地址也不会降低安全性，而且如果出现安全问题的话，它还可以发出一些可用日志代码记录的有意义信息。

下面我们回过头来看 VirtualWebDriveService 类。由于我们已经创建了加密组件和数据库组件，所以 Login()方法非常直观。SecureServerSession 类可以解密登录数据，然后将其传递给 DatabaseServices 组件进行验证。如果用户不能被验证，就会抛出一个普通的异常。否则，就利用新的 GUID 和 LoginInfo 对象中的密钥信息生成一个新的票据。

```
[WebMethod()]
public string Login(byte[] loginData)
{
    LoginInfo loginInfo = session.DecryptLoginData(loginData);

    // An unhandled, generic exception will halt the process here if
    // the user is not found.
    int userID = DB.ValidateLogin(loginInfo);

    // Create a cryptographically random ticket.
    byte[] ticketBytes = new byte[16];
    RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
```

```
        rng.GetBytes(ticketBytes);
        Guid ticketGuid = new Guid(ticketBytes);
        string ticket = ticketGuid.ToString();

        // Store this ticket in application state.
        Application[ticket] = new TicketInfo(userID,
          Context.Request.UserHostAddress, loginInfo.ClientKey);

        // Return the ticket.
        return ticket;
    }
```

Logout()方法可以简单地删除票据：

```
    [WebMethod()]
    public void Logout(string ticket)
    {
        Application.Remove(ticket);
    }
```

基于票据的身份验证系统需要一个更详细的资料：验证被提交的票据是否有效的私有函数。该函数可以验证 GUID 是否与有效的票据相对应，票据是否已经过期(它可能表示一次重放攻击)，以及用户的 IP 地址是否被修改(它可能表示一次会话截取)。如果这些标准都不能满足，就会抛出一个 “Invalid ticket”异常。

如果满足这些标准，就把客户机的会话密钥提交给 SecureServerSession 对象，它在该对象中可以解密或者加密任何被交换的数据。

```
    private TicketInfo ValidateTicket(string ticket)
    {
        TicketInfo ticketInfo = (TicketInfo)Application[ticket];
        if (ticketInfo == null)
        {
            throw new ApplicationException("Invalid ticket.");
        }

        // Check IP address.
        if (ticketInfo.IPAddress != this.Context.Request.UserHostAddress)
        {
            throw new ApplicationException("Invalid ticket.");
        }

        // Apply expiration policy allowing up to twenty minutes
        // between requests.
        // This information could be read from the configuration file.
        if (ticketInfo.CreatedDate.AddMinutes(20) < DateTime.Now)
```

```
        {
            throw new ApplicationException("Invalid ticket.");
        }
        else
        {
            ticketInfo.LastUsedDate = DateTime.Now;
        }

        session.SetClientKey(ticketInfo.ClientKey);
        return ticketInfo;
    }
```

注意，您可以在 ValidateTicket()方法中定义自己的异常策略。在 VirtualWebDrive 中，票据在禁用 20 分钟后就会过期。每次可以接收一个请求来检验它的有效期，如果票据有效，那么创建日期就更新为当前日期。

8.4.2　管理文件

VirtualWebDriveService 类所剩下的就是 3 种管理文件和文件信息的 Web 方法：GetFileInfo()、GetFile()和 SaveFile()。每种方法首先都要验证票据，还要把对应的客户机密钥放置到 ServerSecureSession 中，因此可以用来加密和解密数据。

GetFileInfo()方法只能利用 GetFilesForUser()方法返回文件信息，并利用 EncryptObjectForClient()方法来加密。

```
[WebMethod()]
public byte[] GetFileInfo(string ticket)
{
    TicketInfo ticketInfo = this.ValidateTicket(ticket);

    return session.EncryptObjectForClient(
            DB.GetFilesForUser(ticketInfo.UserID));
}
```

GetFile()方法可以检索一个指定文件的信息，从硬盘中读取文件，并以字节数组的形式返回内容。该数据在传输之前不能加密，因为它已经被客户机加密过。

GetFile()方法可以在验证票据之后执行两种附加的安全检查：

- 检验所需的文件是否由发出请求的用户创建。
- 检验存储在数据库中的散列是否与文件的散列相符。这样可以确保文件在上传之后还没有修改，假设数据库中的信息也没有被篡改。

如果这些检验都失败，就返回一个普通的异常。客户机无法确定操作失败是由于安全异常、数据库问题还是文件 I/O 错误造成的。如果需要的话，您可以定制或者添加日志代码，以便在操作失败之后，能够在服务器上检索信息。

```
[WebMethod()]
public byte[] GetFile(string ticket, string fileID)
{
    TicketInfo ticketInfo = this.ValidateTicket(ticket);
    FileInfo fileInfo = DB.GetFileInfo(new Guid(fileID));

    if (fileInfo.UserID != ticketInfo.UserID)
    {
        throw new ApplicationException("Cannot return file.");
    }

    byte[] fileData;
    try
    {
        FileStream fs = File.OpenRead(fileInfo.ServerPath);
        fileData = new byte [fs.Length];
        fs.Read(fileData, 0, (int)fs.Length);
        fs.Close();
    }
    catch
    {
        throw new ApplicationException("Cannot return file.");
    }

    if (!session.VerifyHash(fileData, fileInfo.Hash))
    {
        throw new ApplicationException("Cannot return file.");
    }

    // No encryption needed. Data is still encrypted.
    return fileData;
}
```

最后，SaveFile()方法把一个新的文件保存在服务器的硬盘上并创建相应的数据库记录。把文件信息当作一个加密的 FileInfo 对象和加密的文件内容一起传递。FileInfo 数据和当前的会话密钥已经传递出去，而且它可以通过服务器解密。但是加密的文件内容不能在服务器端解密，而是以加密的形式写入磁盘中。SaveFile()方法也可以计算文件的散列值，并把它保存在数据库记录中。

```
[WebMethod()]
public string SaveFile(string ticket, byte[] fileInfoData,
  byte[] fileContent)
{
    TicketInfo ticketInfo = this.ValidateTicket(ticket);
```

```
        FileInfo fileInfo =
          (FileInfo)session.DecryptObjectFromClient(fileInfoData);

        fileInfo.UserID = ticketInfo.UserID;

        // Add the hash information.
        fileInfo.Hash = session.CalculateHash(fileContent);

        // Generate the ID.
        fileInfo.ID = Guid.NewGuid();

        try
        {
            // Save file.
            // No encryption needed. Data is already encrypted.
            string path = ConfigurationSettings.AppSettings["FilePath"] +
              fileInfo.ID.ToString();
            fileInfo.ServerPath = path;
            FileStream fs = new FileStream(path, FileMode.CreateNew);
            fs.Write(fileContent, 0, fileContent.Length);
            fs.Close();

            DB.AddFile(fileInfo);
        }
        catch (Exception err)
        {
            // Log error.

            System.Diagnostics.Debug.WriteLine(err.ToString());
            throw new ApplicationException("Cannot save file.");
        }
        return fileInfo.ID.ToString();
    }
```

SaveFile()方法还有一个缺陷。如果发生数据库异常，它可以不创建相关的数据库记录就把一个“孤立”的文件写入磁盘中。为了解决这个问题，如果数据库异常发生，您可以删除文件，或者使用一个 COM+事物处理并颠倒操作的顺序。这样在出现文件 I/O 错误之后，您可以重新回滚数据库更改。

8.5　Windows 客户程序

该系统的最后一部分是和 VirtualWebDrive 服务发生交互作用的 Windows 客户程序。该客户程序如图 8-5 所示。

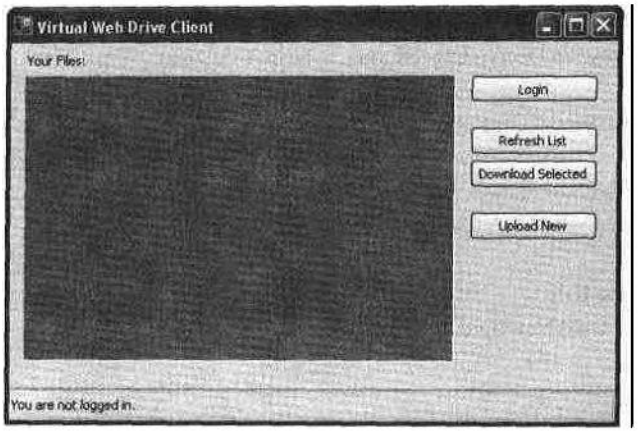

图　8-5

客户机程序使用多个成员变量来引用 Web 服务代理，当前会话和 FileEncryption 类的一个实例。它还可以跟踪当前票据和表示客户机是否登录的标志。

```
public class MainForm : System.Windows.Forms.Form
{
    private localhost.VirtualWebDriveService proxy =
     new localhost.VirtualWebDriveService();
    private VirtualWebDriveComponent.SecureClientSession session;
    private VirtualWebDriveComponent.FileEncryption FileUtil =
     new VirtualWebDriveComponent.FileEncryption();
    private string ticket;
    private bool loggedIn = false;

    // (Designer code and event handlers omitted.)
}
```

当单击 Login 按钮时，Login()的单击事件处理程序可以创建一个新的 LoginForm(如图 8-6 所示)。然后检索用户 ID 和密码，并以加密的形式提交给 Web 服务。注意客户机不需要知道数据是如何加密的，是否使用了时间戳，甚至是否为了进行数据交换而创建一个新的对称密钥。窗体将在登录之后利用票据信息更新。

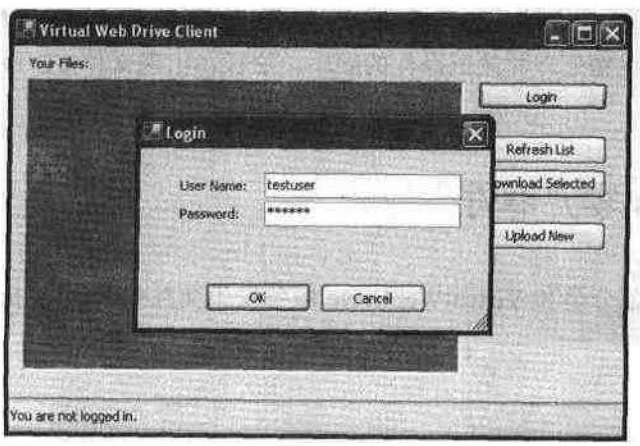

图　8-6

```
private void Login(object sender, System.EventArgs e)
{
    LoginForm login = new LoginForm();
    if (login.ShowDialog() == DialogResult.OK)
    {
        try
        {
            session = new SecureClientSession(proxy.GetKey());
            ticket = proxy.Login(session.EncryptLoginData(
              login.UserName, login.Password));
            status.Panels[0].Text = "You are logged in as " +
              ticket + ".";

            loggedIn = true;
            RefreshList();
        }
        catch (Exception err)
        {
            MessageBox.Show(err.ToString());
            loggedIn = false;
            status.Panels[0].Text = "You are not logged in.";
        }
    }
    else

    {
        ticket = null;
        gridFiles.DataSource = null;
    }
}
```

如图 8-7 所示，MainForm 的 RefreshList()方法可以自动下载最新的一系列文件并在表格中显示出来。注意：返回的信息可以利用 SecureClientSession 类无缝解密。

```
private void RefreshList()
{
    DataSet ds = (DataSet)session.DecryptObjectFromServer(
        proxy.GetFileInfo(ticket));
    gridFiles.DataSource = ds.Tables[0];
}
```

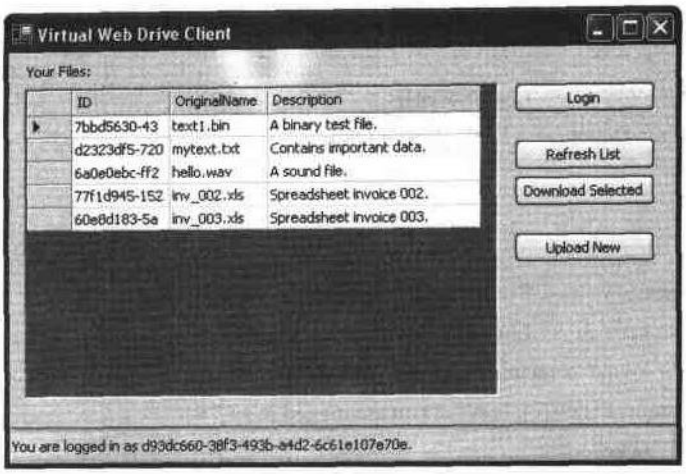

图　8-7

当关闭窗体时，如果需要的话，退出该客户程序。

```
private void MainForm_Closed(object sender, System.EventArgs e)
{
    if (loggedIn)
    {
        proxy.Logout(ticket);
    }
}
```

下面只剩下实现上传文件和下载文件的功能了。再次使用加密组件来简化操作。客户程序只需利用 OpenFileDialog 或者 SaveFileDialog 组件给用户提示源文件或者目的文件即可。文件可以利用 FileUtility 类加密或者解密。

```
private void cmdUpload_Click(object sender, System.EventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        try
        {
            Stream fs = dlg.OpenFile();
            byte[] fileContent = new byte[fs.Length];
            fs.Read(fileContent, 0, fileContent.Length);
            fs.Close();

            // Encrypt file.
            fileContent = FileUtil.EncryptFile(fileContent);

            FileInfo fileInfo = new FileInfo(
              Path.GetFileName(dlg.FileName), dlg.FileName);
```

```
                proxy.SaveFile(ticket,
                  session.EncryptObjectForServer(fileInfo), fileContent);

                // Refresh list.
                RefreshList();
            }
            catch (Exception err)
            {
                MessageBox.Show(err.ToString());
            }
        }
    }
```

在下载文件时，可以从 DataGrid 中检索所选的文件，然后进行相同的操作过程，不过这次顺序正好相反。

```
    private void cmdDownload_Click(object sender, System.EventArgs e)
    {
        DataTable dt = (DataTable)gridFiles.DataSource;

        SaveFileDialog dlg = new SaveFileDialog();
        dlg.FileName =
          dt.Rows[gridFiles.CurrentRowIndex]["OriginalName"].ToString();
        if (dlg.ShowDialog() == DialogResult.OK)
        {
            try

            {
                byte[] fileContent = proxy.GetFile(ticket,
                  dt.Rows[gridFiles.CurrentRowIndex]["ID"].ToString());
                // Decrypt file.
                fileContent = FileUtil.DecryptFile(fileContent);

                Stream fs = new FileStream(dlg.FileName, FileMode.Create);
                fs.Write(fileContent, 0, fileContent.Length);
                fs.Close();
            }
            catch (Exception err)
            {
                MessageBox.Show(err.ToString());
            }
        }
    }
```

可以创建一个简单的文本文件来测试该逻辑，如图 8-8 所示。

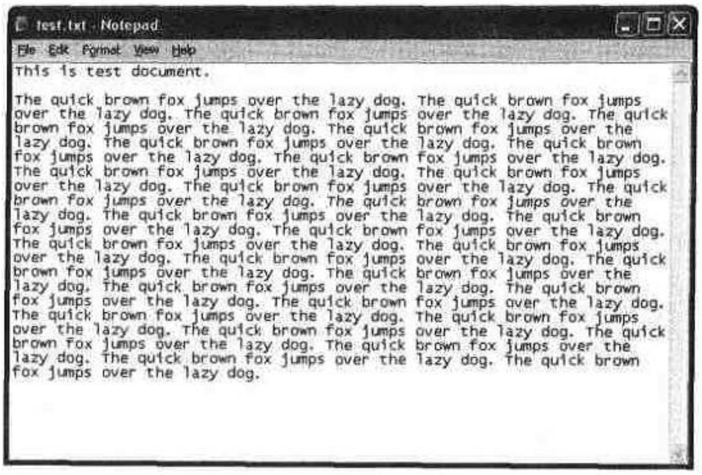

图　8-8

在把文本文件上传给服务器时，会给定一个惟一的文件名(在该示例中指的是 dafd059e-27eb-4ae1-b959-ca6326e91c8a)。如果您设法在记事本中打开该文件，就会看到如图 8-9 所示的内容。

图　8-9

最后，当把文件下载到客户机硬盘上时，它可以解密回原来的格式。

8.6　可能的改进措施

本章开发的 VirtualWebDrive 服务利用大量加密代码来保护通信数据和数据库中的数据。在该示例中，我们没有利用任何安全服务(例如 Windows 身份验证或 SSL)，而且它引入了我们需要添加的代码层。但即便有了这些具体的措施，解决方案中仍然存在许多安全风险，其中有一部分比较容易解决。

- 没有充足的保护措施来防御那些经过身份验证的恶意用户。尤其是经过身份验证的用户

可以上传大量文件来最终填满服务器硬盘。从本质上看，这就是拒绝服务式攻击。要想解决该问题，您必须实现某些策略规则，而且最好使用单独的存储器。单独的存储器允许利用.NET 运行库提供的虚拟文件系统安全地存储数据。它还限定了数据尺寸。

- 一些数据以明文形式保存在数据库中，例如原始的文件名、描述等等。这种风险非常小，因为攻击者必须在他们访问该信息之前(这时他们可能检索到更有价值的数据)进入服务器计算机。不过，可以通过加密该信息来进行保护。您可以利用客户机的密钥，在这种情况下，服务器从来没有检验过该信息，或者您也可以使用服务器的公钥，这时要求该密钥一定没有修改过。
- 一些重放攻击可能隐藏在一个特定会话的范围内。例如，重放攻击不能利用登录数据(因为它会合并日期信息)，但它可以进行文件的上传或者请求的下载，只要对称会话密钥没有改变而且票据没有过期即可。为了消除这种可能性，所有的加密包都可以合并日期信息。
- 没有进行身份验证操作。因为我们没有使用 SSL 和证书，所以客户机无法确定它们正在和服务器通信。这就有可能发生中间人攻击。遗憾的是，目前还没有解决该问题的简单方法，因为这里没有可以验证证书信息的.NET API 。一种方法是客户机为它信任的服务器预先配置一系列公钥，并拒绝和其他服务器进行通信。

除了这些安全限制之外，还有许多可以改进代码的措施：

- 客户机应用程序用强名和 Authenticode 签名签署。这样可以减少客户机应用程序被恶意的 Trojan 应用程序代替的机会。
- 添加日志代码和审核代码。您可以不再利用 Debug 对象来编写错误消息，而是调用第三方组件中的 Log() 方法来完成，然后再把信息保存在事件日志或者数据库中。
- Windows 客户机还可能在许多方面得到改进。例如，您可以添加一些细节措施，在客户机还没有登录时禁用按钮，防止它企图获取一系列文件。

8.7　小结

最重要的应用程序设计问题和安全代码问题之一是其复杂性。为了实现加密，通常情况下必须添加冗长的附加代码，查找不常见的故障，甚至通过修改参数和返回值的数据类型来修改系统的接口。要想成功地实现安全的代码，需要谨慎地利用帮助类和专用的组件来控制这种复杂性。否则，随着应用程序中新功能和新方法的不断增多，就更加难以发现安全漏洞。

本章引用一个安全的、端对端的分布式应用程序进行说明。它利用多层设计模式把数据库和加密逻辑分别分离到单独的组件中，从而显著简化了客户程序和 Web 服务的代码。它从进行安全分析开始，演示了在设计一个新的应用程序时需要执行的一系列操作。

附录A 传输层安全

附录 A 简单讨论了保护在应用程序“下面”传输的数据的可选方法，也就是应用程序可以信赖于由传输机制保护的数据，而不是在应用程序中保护他们。使单个通道安全的一个常用方法就是使用 SSL(例如像一个 HTTPS 会话)，这已经讨论过了。尽管 SSLv3 非常安全，不过已经有了 SSL 的替代品，即传输层安全(TLS，Transport Layer Security)，它也用于保护单个通道。不过，这些技术都在某种程度上影响应用程序，像在 URL 中指定 HTTPS 而不是 HTTP，但它仍是一个建立安全通道的非常合理的方法。

在企业中尤其是通过 Internet，保护传输数据已越来越重要。通过 Internet 的安全通道是电子商务(例如)所需要的，因为客户不愿意发送他们的信用卡的其他信息。

说明：

需要一些基本的 TCP/IP 知识和定义才能完全理解本附录讨论的内容。Wrox 出版社的 Professional .NET Network Programming (ISBN 1-86100-735-3)很好地介绍了网络编程和术语。

本附录后面还将讨论其他技术，例如 IP 安全性(IPSec，IP Security)、第二层隧道协议(L2TP，Layer 2 Tunneling Protocol)或点对点隧道协议(PPTP，Point-to-Point Tunneling Protocol)来建立两个节点之间的安全“隧道”。这些“隧道”技术也叫作虚拟专用网(VPN，Virtual Private Networks)，将保护在该通道中的所有传输。这对于家庭到公司的网络来说是个很有吸引力的方案，其中雇员可能呆在家里，接着使用它们常用的 ISP 建立与公司网络的安全通信，在家就可获取可用的公司网络资源。

为了更好地理解这些技术的工作原理和作用，需要知道计算机网络通信的知识，下一节将作介绍。

A.1 参考模型

如今描述通信层的方法主要有两种：ISO/OSI 7 层参考模型和 TCP/IP 5 层参考模型。这一术语看起来有点让人混淆，TCP/IP 实际只表示 5 层模型中的两层，而开放式系统互连(OST，Open System Interconnection)是国际化标准化组织(ISO，International Standards Organization)中的一个工作组。

分层的模型的实现(有时是模型本身)也被叫作“栈”。栈中的每层底下都有一个相关性。您可能听说过 TCP/IP 栈这个术语，它多半指的就是系统中的 TCP/IP 实现。

A.1.1 ISO/OSI 7 层参考模型

这个模型本身决定了它实际应用于特定的情况下，如 TCP/IP。它也可以用于描述 TCP/IP，

但通常在描述协议时，它是用作基本模型。表 A-1 对该模型的 7 层作了介绍。

表 A-1

层	名　称	说　明
7	应用层	终端用户和终端应用程序协议，如 FTP、SMTP 和 HTTP 等
6	表示层	实际的协议表示就是在这儿实现的，如对数据封包和拆包的规则
5	会话层	该层负责保持会话一致性，确保所有的包在正确的装置间传输。例如，该层确保了包采取了正确的路由通过负载平衡的硬件
4	传输层	负责确保数据的可靠性和完整性(根据协议报头，并不防止数据被篡改)
3	网络层	进行地址指派和为包运输选路
2	数据链路层	将数据组织到框架中，通过物理层与它们通信
1	物理层	硬件在这里传输实际数据，通常通过电缆，但也可以通过光纤、空气或其他介质

A.1.2　TCP/IP 5 层参考模型

这一模型用于描述 TCP/IP 的特性，如表 A-2 所示。

表 A-2

层	名　称	说　明
5	应用层	终端用户和终端应用程序协议，如 FTP、SMTP 和 HTTP
4	传输层 (TCP)	负责确保数据可靠性和完整性(根据协议报头，并不防止数据被篡改)
3	网际层 (IP)	格式化包并指定路由
2	网络层	框架组织和传输规范
1	物理层	硬件在这里传输实际数据，通常通过电缆，但也可通过光纤、空气或其他介质

A.1.3　示例协议栈

SSL/TLS 和 IPSec 作用于不同层，可由图 A-1、A-2 和 A-3 所示的示例栈说明。

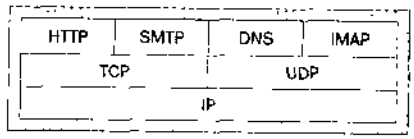

图　A-1

这是普通的 TCP/IP 栈，包括应用层协议如 HTTP、FTP、DNS 和 IMAP。

注意：

应用层协议可使用 TCP 和 UDP，因此可以通过该图大概知道协议之间的联系。

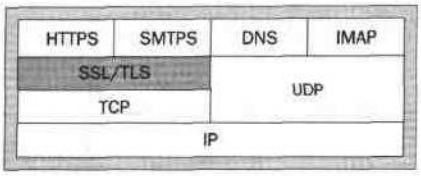

图 A-2

该示例显示了如何在 TCP 框上实现 SSL/TLS。

注意：

应用程序协议必须适合 SSL/TLS(由 HTTPS 和 SMPTS 的最后一个 S 示意)。

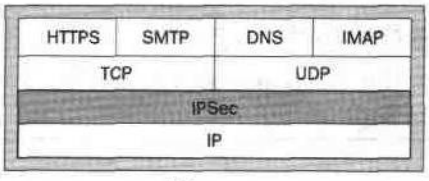

图 A-3

这个示例显示 IPSec 如何使上面的所有协议安全，而不作任何修改。

注意：

必须通过策略配置 IPSec；例如它并不自动将秘密提供给任何一个应用层协议。

使用 IPSec 也意味着 TCP/IP 栈实现必须支持它，而 SSL/TLS 实现并不影响 TCP/IP 栈，但要求对每个应用程序作修改。

说明：

如果没有作特别说明，本附录剩下部分指的都是 5 层模型。

A.2 传输层安全(TLS)

TLS 协议是在 RFC 2246 (http://www.ietf.org/rfc/rfc2246.txt)中定义的，构建于 SSLv3 规范之上。下面这段话是从 RFC 文档中摘录的，它对 TLS 规范的目的作了最好的阐述：

“此文档和 TLS 协议都基于 Netscape 发布的 SSL 3.0 协议规范。该协议与 SSL3.0 之间的区别并不大，但 TLS 1.0 和 SSL 3.0 并不实现互操作(尽管 TLS 1.0 合并了一个 TLS 实现通过它可以与 SSL3.0 联系的机制)。”

作为推荐使用的 SSLv3 的替代品，TLS 是在安全支持提供者接口(SSPI，Security Support Provider Interface)中实现的，它并不是直接在.NET Framework 中可用的，至少目前是这样。不过，可从 Microsoft 下载的两个示例实现了两个程序集(用 C++编写)，可帮助获取对.NET 中的 SSPI 的访问。从 http://msdn.microsoft.com/library/?url=/library/en-us/dndotnet/html/remsspi.asp 中可获取更多的信息并下载代码。

A.3　IPSec

IPSec(Internet Protocol Security)是由 Internet 工程任务组（IETF，Internet Engineering Task Force)设计的。Windows2000 及其后面的系统都包括了 IPSec，Microsoft 提供的基于 IPSec 的 VPN 客户机也可用于较老的系统；可参考下载中的 VPN 部分。IP 安全规范提供了加密的服务，如 IP 层的保密工作和身份验证——它在 TCP/IP 模型的第 3 层中工作。IPSec 由几个补充的安全协议组成：

- 身份验证头(AH，Authentication Header)——提供身份验证
- 封装安全负载(ESP，Encapsulating Security Payload) ——为包提供机密性的保证
- IP 负载压缩(IPCOMP ，IP payload compression) ——在 ESP 前提供压缩包的方法
- Internet 密钥交换(IKE ，Internet Key Exchange) ——确保密钥的安全交换(可选的)

Windows XP Professional 使用 MMC 管理单元包括了 IPSec 策略的本地管理，可以使用 Active Directory 和 Windows 2000 或.NET 服务器安装组策略。IPSec 本身并不可以直接用于较老(与 Windows 2000 相比)的平台，尽管它可用于本附录后面提到的一些 VPN 客户机(它们可用于较老的平台)。

A.3.1　网络地址转换(NAT)

在创建 IPSec 环境之前，需要知道一些与 Network Address Translation(即端口转换)相关的问题。NAT 服务作为局域网和 Internet 之间的网关，为请求将本地地址转换为 Internet 地址，或是为响应将 Internet 地址转换为本地地址。例如，Microsoft 的 Internet Connection Sharing(Internet 连接共享)技术使用 NAT，使得若干个计算机可以通过一个 IP 地址访问 Internet。如图 A-4 所示。

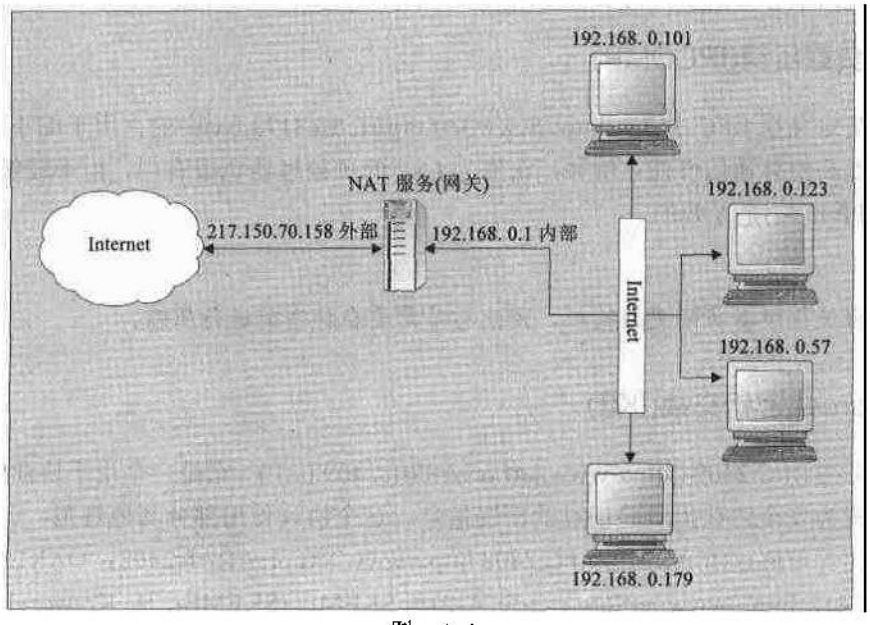

图　A-4

IPSec 基于 IP，因此使用它需要知道进行通信的计算机的相关 IP 地址。现在您可能知道 NAT 与 IPSec 是不兼容的，因为不可以将本地系统(网关内的)与网关外的计算机建立 IPSec 通信。

幸运的是，现在已经提出了解决这一问题的方案。参考 IETF 起草的 IPSec 包的 UDP 封装(http://www.ietf.org/internet-drafts/draft-ietf-ipsec-udp-encaps-04.txt)和 IKE 中的 NAT 转换协议(http://www.ietf.org/internet-drafts/draft-ietf-ipsec-nat-t-ike-04.txt)可获取更多信息。

Microsoft 在其 VPN 软件中实现了这些方案，这至少可以使用它的 L2TP/IPSec VPN 连接到您的公司，即便在家使用的是 ICS，只要连接到也实现了这些方案的服务器即可。

A.3.2　身份验证头(AH)

RFC 1826 (http://www.ietf.org/rfc/rfc1826.txt)中描述了身份验证报头，用于提供 IP 数据包的强大完整性和身份验证方案。AH 并没有提供网络分析中的机密性或保护性，而 ESP 却提供了。您可以按个人喜好组合这些安全协议。

AH 包含了一个对数据包中所有数据的强加密的检查和，包括所有报头中的静态字段。检查和是使用发送者和接收者前面已达成一致(例如使用 IKE)的算法和密钥计算的——在许多方面类似于 MAC。这样一个协议就叫作“安全协议”。参数是本地存储的，由安全参数索引(SPI，Security Parameter Index)引用。

A.3.3　封装安全负载(ESP)

封装安全负载是由 RFC 1927(http://www.ietf.org/rfc/rfc1927.txt)定义的，为 IP 数据包的完整性(像 AH)和机密性提供了一种机制。它可以加密传输层部分，如 TPC、UDP、ICMP 或 IGMP(传输或转接模式)或是一个完整的 IP 数据包(隧道模式)。

发送者和接收者必须事先对安全参数达成一致，对 AH 安全协议采取同样的方式。

A.3.4　IP 负载压缩(IPCOMP)

该协议是定义在 RFC 3173 (http://www.ietf.org/rfc/rfc3173.txt)中的，用于缩小 IP 数据包的尺寸，以便提高整体通信性能。例如，它作为 ESP 的预处理器也很有用，由于较低层的加密显示压缩(像 PPP 压缩)是无效的。

说明：

由于加密使得数据实际更为随机，因此总是需要在加密前进行压缩。

A.3.5　Internet 密钥交换(IKE)

IKE 定义在 RFC 2409 (http://www.ietf.org/rfc/rfc2409.txt)中，它是一个用于协商的混合协议，以保护的方式为安全协议提供验证过的密钥信息。这个协议使用部分其他规范，如 Internet 安全协议和密钥管理协议(ISAKMP, RFC 2408 http://www.ietf.org/rfc/rfc2408)、OAKLEY 密钥确定协议 (RFC 2412 http://www.ietf.org/rfc/rfc2412)和 SKEME (SKEME：H. Krawczyk 的一个用于 Internet 的通用安全密钥交换机制，http://www.research.ibm.com/security/oldpubl.html)。

IKE 通常作为一个单独的进程在服务器上执行，它并不直接影响 TCP/IP 栈。不过，它提供了创建 AH 和 ESP 使用的安全协议和安全参数索引的方法，这是运行 TCP/IP 栈的 IPSec 部分所必需的。

A.4　虚拟专用网(VPN)

VPN 的作用就是建立一个安全的点对点的隧道，通过它可以将应用层的数据安全地通过公共(非安全的)网络。这是一个很好的方法，使得可以实现通过 Internet 的安全网络通信。这是建立从家到公司的安全链接的常用方法。像以往一样连接到 Internet，接着连接到公司 VPN 服务器等，就可以从家里安全访问公司网络了。

目前有两种流行的隧道协议：

- PPTP——点对点的隧道协议
- L2TP——2 层隧道协议

A.4.1　点对点的隧道协议(PPTP)

人们经常都说是 Microsoft 发明的 PPTP，但它实际上是由 Microsoft 所领导的一个联盟所开发的，包括 U.S. Robotics、ECI Telematics、3Com 和 Ascend Communications。不过，Microsoft 实现左右了 PPTP 整个市场。

PPTP 将 PPP(Point-to-Point Protocol，RFC 1661 http://www.ietf.org/rfc/rfc1661.txt)包封装在 GRE(Generic Routing Encapsulation，RFC 1701 http://www.ietf.org/rfc/rfc1701.txt 和 1702)中。PPTP 客户机可从 Windows 95 中获得，但由于安全弱点，较老的实现需要更新(查看 Windows Update 站点，从 http://windowsupdate.microsoft.com/中获取更新版本。)。

A.4.2　2 层隧道协议(L2TP)

该协议定义在 RFC 2661 (http://www.ietf.org/rfc/rfc2661.txt)中，合并了 PPTP 和 L2F(Cisco 的 Layer 2 Forwarding 规范)的最好功能。

目前，大部分的 PPTP 实现有下列缺点：

- 所有网络通信是加密的，使得它相对较慢(如果购买特殊的硬件来加速的话，代价会相对昂贵)。
- 与 IPSec(使用 Triple-DES)相比，大部分当前实现使用较弱的加密技术。
- 通常在每台计算机上手工建立安全协议；运用 IPSec 可以自动执行他们。

Windows 2000 中引入了 LT2P/IPSec VPN 软件，而 VPN 服务器包括在 W2K 服务器和.NET 服务器中。.NET 服务器的实现也包括了 IPSec 一节中提到的 NAT 遍历方案。用于 Windows 98、Windows Me (千禧年版本)和 Windows NT Workstation 4.0 的 L2TP/IPSec 客户机软件(包括支持 NAT 遍历)可从 Microsoft 的 http://www.microsoft.com/windows2000/server/evaluation/news/bulletins/l2tpclient.asp 上获得。

附录B　生成安全的随机数

随机数是密码术的一个重要部分。它们作为初始化向量用于密钥生成中，用于 SSL 和询问/响应协议中。

有许多测试根据数字计算给定数字序列的随机性。它们考虑给定数字在序列中出现的周期，作更细致的测量，包括相同数字或其他重复形式出现的周期。统计随机性的要求与加密随机性的不同。一个数字序列在统计上是随机的，但如果攻击者可以推算出数字的序列(通过了解使用的算法和随机种子值)，那么就是加密不安全的，如果我们想隐藏数据和使数据安全，那么这就不太好。

.NET Framework 包括了两个随机数字生成器：

- System.Random 类是个伪随机数字生成器，不适合于加密的应用程序。
- System.Security.Cryptography.RNGCryptoServiceProvider 类是个加密安全的随机数字生成器。这个随机生成器是使用 CryptoAPI CryptGenRandom()函数创建的。

本附录将讨论总是使用 RNGCryptoServiceProvider 而不是 Random(为密码术)的原因，演示如何使用 RNGCryptoServiceProvider 类。

B.1　伪随机数

对于一串随机的数字，最常见的描述就是没有从前一个数字推算出后一个数字的数学方法。最好的随机数是从物理过程(通过抛硬币来测定原子的放射性衰变)中获得的，因为实际物理过程才是真正随机的。事实上，一些随机数生成器使用硬件设置来实现，如音频输入或二极管。

从设计上来说，计算机是很确定的，因此不是生成随机数的最好选择。它们通常求助于一个生成统计上随机的数字串的算法。为了确定在该算法中使用的输入值，它们要求用户提供一个种子值，这通常来自于系统时钟，网络接口卡 MAC 地址以及其他不同的系统参数。

这些随机数字很适合于计算机游戏中的示例数据或建模物理过程。不过，它们不适用于加密。它的弱点包括下列几点：

- 伪随机数是周期性的。最终将重复数字序列。
- 如果使用相同的种子值，将接收到序列完全一样的“随机”数。因此，随机序列与种子值一样多。
- 随机数可使用逆向工程。运用算法知识，强力攻击会立即猜测种子值。如果种子值和时间之间有相关性，攻击者将会推算出所有后面的“随机”数。

在创建 Random 类实例时，可以指定种子值(使用 32 位整数)，或是不进行指定(那将使用当前系统计时器中的值)。接着可以调用 NextBytes()方法来用伪随机数据序列填充字节数组，或是带有指定值范围的参数的 Next()方法。下面的代码显示了一些用 Random 类创建伪随机数的不

同方法。

```
// Create a random generator and specify the seed value
Random rand = new Random(DateTime.Now.Millisecond);

// Create a random generator and use the default
// (current millisecond) seed.
rand = new Random();

// Generate an integer number between 0 and 10.
int number = rand.Next(10);

// Generate an integer number between 1 and 10.
number = rand.Next(1, 10);

// Fill a byte array with random data.
byte[] data = new byte[100];
rand.NextBytes(data);
```

下面的循环中是个使用 Random 类的简单控制台示例。它演示了其中的一个弱点。

```
class RandomNumberTest
{
    [STAThread]
    static void Main(string[] args)
    {
        do
        {
            Random rand = new Random();
            Console.Write(rand.Next(1, 10));
        } while (true);
    }
}
```

由于每次实例化 Random 类时，都是用当前时间作为种子值的，因此如果快速(在毫秒内)实例化该例，那么将接收到同样的的数字：

```
8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888881111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111133333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333333355555555
55555 ...
```

可以在循环外通过实例化 Random 来纠正这一问题。在这里，每次调用 Next()将移至算法生成的下一个伪随机数。这就是在使用它检索随机数之前需要实例化 Random 类的原因——它

是个可以记起它的“位置”的多状态对象。不过，如果使用的种子值相同，那么将重复随机数序列。

伪随机数是出现许多臭名昭著的攻击的主要原因。有一种攻击就是针对赌博应用程序，这种应用程序使用一个随机数种子值来对纸牌进行排序，而洗牌的可能性是有限的。在看完开始的几张牌后，用户可以将当前发的牌与某种可能的洗牌序列匹配，来确定剩下牌的顺序。另一个著名的例子就是 Netscape Navigator 早期版本中的取决于时间的随机数字生成器，它泄露了动态生成的用于加密运用 SSL 的会话中数据的密钥。如果您对此感兴趣，那么可以从 http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html 中获取对该弱点的详细描述。

B.2 加密的随机数

为了说明一连串的随机数字是加密安全的，必须使得用户不可能通过计算重新生成同样序列的随机数。遗憾的是，运用伪随机数字，可以很容易地重新生成同样的序列。用户需要知道的知识就是伪随机数生成器算法(或.NET Framework 副本)和种子值。

通过加密保护数据基于加密算法和更为随机的种子值。.NET 为名为 RandomNumberGenerator 的加密的随机数生成器定义了一个抽象基类，它定义了 Create()和 GetBytes()方法。它还包括了一个派生类，即 RNGCryptoServiceProvider，它使用 CryptoAPI 的 CryptGenRandom 函数。

为了构成种子值，需要用不同的值组合成一个系统范围内的种子值。这些值包括调用的应用程序可以提供的位，例如鼠标或键盘动作之间的用户反应时间、像进程 ID 和线程 ID 这样的系统和用户数据、系统时钟、系统计数器、自由磁盘群集数和散列的用户环境块。接着使用 SHA-1 散列这个值，输出用于创建一个随机数据流(用于更新系统种子值)。这可以起作用，是因为散列值生成了看似随机的数据，只改变源文档(种子值)中的一个位，任何两个输出的散列共享它们 50%的位，尽管两个输出只有一位之差。当然，从理论上讲，有些过程还是周期性的。例如磁盘搜索时间看似随机的，实际取决于易于确定的因素，可以被推测出来。为了获取更好的随机数生成，可将 CryptoAPI 用于硬件生成器，像 Intel 的随机数生成器(查看 http://www.intel.com/design/security/rng/rngppr.htm)。

说明：

创建加密安全的随机数需要更多时间，这意味着如果需要快速地在一个短时间内生成大量(例如数百万)随机数是不适合的。在一个简单测试中，Random 类在不到 1 秒的时间内生成一百万个随机数，而 RNGCryptoServiceProvider 做同样的事情用了约 8 秒的时间。

下列所示为使用 RNGCryptoServiceProvider 用随机数据填充字节数组的代码。注意，编程人员并没有指定种子值。

```
byte[] randomBytes = new byte[100];
RandomNumberGenerator rand = RandomNumberGenerator.Create();
rand.GetBytes(randomBytes);
```

与 Random 类不同的是，RNGCryptoServiceProvider 和 RandomNumberGenerator 都没有为生成在某一特定范围的数提供任何便利的方法。不过，可以使用像 BitConverter 这样的类模拟这些计算，这个类可以将二进制数据转换成另一种数据类型。下列所示为创建一个随机的 32 位的整数：

```
// Four bytes are needed for a 32-bit integer.
byte[] randomBytes = new byte[4];

RandomNumberGenerator rand = RandomNumberGenerator.Create();
rand.GetBytes(randomBytes);
int randomInt = BitConverter.ToInt32(randomBytes, 0);
```

类似的技术可用于用数据填充字符串，甚至是通过 BitConverter.ToString()或 Convert.ToBase64String()方法应用 Base64 编码。

下面这个代码示例使用 RNGCryptoServiceProvider 在一个控制台应用程序中生成随机数字。这等同于前面使用 Random 类的那个示例。

```
class RandomNumberTest2
{
    [STAThread]
    static void Main(string[] args)
    {
        byte[] randomBytes = new byte[0];

        do
        {
            RandomNumberGenerator rand;

            rand = RandomNumberGenerator.Create();
            rand.GetBytes(randomBytes);

            // Convert the random byte into a decimal from 1 to 10.
            Console.Write(
             (int)(((decimal)randomBytes[0] / 256) * 10) + 1);
      } while (true);
    }
}
```

为了将随机字节值(从 0 到 256)转换为从 1 到 10 的数字，可以使用下列公式：

```
value = (int)((decimal)RandomByte / 256 * MaxValue) + 1
```

该输出比前面的测试更随机，将为随机性传递统计测试。更为重要的是，有恶意的用户无法猜出后面的随机数，即使他们知道用于生成这些数字的技术，并且拥有最近生成的值的列表。

```
10111323918197167928612236551913232617147255352654410268874695827839127231113104741031
```

094354161042691104878739910937788487107148109699184367397834616261091552622271085618991154
75644693167835952351622104542910646462935939444989532462836587828621059 4 ...

通过用一个真正随机的数字作为 Guid 类的种子值，可以使用类似的技术创建加密的随机 GUID：

```
byte[] randomBytes = new byte[16];
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
rng.GetBytes(randomBytes);
Guid randomGuid = new Guid(randomBytes);
```

记住，在通过创建 SymmetricAlgorithm 或 AsymmetricAlgorithm 类的新实例动态生成密钥时，.NET 将使用 RNGCryptoServiceProvider 生成安全的随机值，接着将该值用于填充密钥字段和初始值向量。

说明：

使用用 RNGCryptoServiceProvider 生成的数字作为 Random 类的种子值并不够，尽管这将提高一点安全性。这一技术确保无法猜测出随机数序列的起点。不过，如果给定 Random 类生成的数字序列，攻击者仍然可以确定后面的数字。

附录C 支持、勘误表和代码下载

我们一贯重视读者的意见，并想知道每位读者对本书的看法，包括读者喜欢和不喜欢的内容，以及读者希望我们下一次完善的地方。您可以通过发送电子邮件(地址为 feedback@wrox.com)来向我们反馈意见。请确保反馈信息提到本书的书名。

C.1 如何下载本书的示例代码

当您访问 Wrox 公司站点(地址为 http://www.wrox.com/)时，通过 Search 工具或书名列表，可以方便地定位需要的书目。然后单击本书的详细页面中的 Download Code 超链接，就可以下载相应的范例代码。

从我们的站点上下载的文件都是使用 WinZip 压缩过的文档。保存文件件到本地磁盘上的文件夹中后，需要使用 WinZip 或一个兼容的解压程序来解压文件。在这个 Zip 文件中有一个文件夹结构和一个 HTML 文件，其中 HTML 文件解释这个文件结构并给出进一步的信息，同时包括一些 e-mail 支持和其他相关读物的链接。

C.2 勘误表

我们已经尽最大努力确保本书中的文本和代码没有错误，但是错误仍然在所难免。如果您发现本书存在错误，例如拼写错误或不正确的代码片段，请反馈信息给我们，我们将不胜感激。勘误表的发送可以节约其他读者学习本书的时间，而且能够帮助我们提供更高质量的信息。请通过 e-mail 将您的信息发送到 support@wrox.com，您的反馈信息将被检查，如果正确，将被粘贴到本书的勘误页面上。

要在我们的站点上找到勘误表，请访问 http://www.wrox.com/ACON1.asp?ISBN=1861007396，单击本书的详细页面中的 Book Errata 超连接即可。

C.3 E-Mail 支持

如果您希望直接向详细了解本书的专家咨询本书中问题，可以发送电子邮件到 support@wrox.com，要求在邮件的主题字段中带上本书的书名和 ISBN(国际标准图书编号)的后 4 位数字。一个典型的电子邮件应包括下面的内容：

- 在 Subject 框中必须有本书的书名、ISBN 的后 4 位数字和问题所在的页数。
- 信息的主体应包括读者的名字、联系信息和问题。

我们将不返回您的无用邮件，因为我们仅仅需要有用的详细资料，以便节约您和我们的时间。当您发送一个电子邮件信息时，它将得到下面一系列支持：

- 用户支持：首先，您的信息将被递送到我们的用户支持人员手中，并由他们阅读。他们备有常见问题的文件，并将立即回答有关本书或者 Web 站点的任何常见问题。
- 编辑支持：接着，一些有深度的问题将被送到负责本书的技术编辑手中，他们在程序设计语言或者特定的产品上有着丰富的经验，能够回答相关主题的详细技术问题。
- 作者支持：最后，如果编辑不能回答您的问题，他们将请求本书的作者。我们将尽量保护作者免受干扰，以便不影响其写作。然而，我们也非常高兴转寄给他们一些特殊的问题。所有 Wrox 公司的作者都为他们的书提供技术支持。作为回应，他们将发送电子邮件给用户和编辑，进而使所有的读者受益。

Wrox 公司的支持部门仅仅对那些与我们出版的书目内容直接相关的问题提供支持，对于超出常用书目支持的问题，您可以从 http://p2p.wrox.com/ forum 中的 P2P 公共列表中获得支持信息。

C.4　p2p.wrox.com 站点

为了便于作者和其他人讨论，请加入到 P2P 站点的邮件列表中，除了一对一的邮件支持系统外，我们独特的系统将 programmer to programmer™(由程序员为程序员而作)的编程理念与邮件列表、论坛、新闻组等其它服务相联系。如果您向 P2P 发送一个问题，应该相信登录邮件列表的 Wrox 公司作者和其他相关专家一定会检查到它。无论您是在阅读本书，还是在开发自己的应用程序，都可以在 http://p2p.wrox.com/上找到许多对自己有所帮助的邮件列表。

按照下面的步骤可以预定一个邮件列表：

(1) 登录 http://p2p.wrox.com/。

(2) 从左边的主菜单栏选择一个适当的种类。

(3) 单击希望加入的邮件列表。

(4) 按照说明订阅并填写自己的邮件地址和密码。

(5) 回复您收到的确认邮件。

(6) 使用预定管理程序加入更多的邮件列表并设置自己的邮件首选项。