ПРОГРАММИСТУ

П. Торстейнсон
Г. А. Ганеш

КРИПТОГРАФИЯ И БЕЗОПАСНОСТЬ В ТЕХНОЛОГИИ .NET

Лаборатория ЗНАНИЙ

КРИПТОГРАФИЯ И БЕЗОПАСНОСТЬ В ТЕХНОЛОГИИ .NET

EXPERT PRACTITIONERS. SEASONED INSTRUCTORS
.NET

.NET SECURITY AND CRYPTOGRAPHY

PETER THORSTEINSON
G. GNANA ARUN GANESH

Prentice Hall PTR, Upper Saddle River, NJ 07458
www.phptr.com

ПРОГРАММИСТУ

П. Торстейнсон, Г. А. Ганеш

КРИПТОГРАФИЯ И БЕЗОПАСНОСТЬ В ТЕХНОЛОГИИ .NET

Перевод с английского
В. Д. Хорева

под редакцией
С. М. Молявко

4-е издание, электронное

Москва
Лаборатория знаний
2020

УДК 004.7
ББК 32.973.202
Т61

Серия основана в 2005 г.

Торстейнсон П.

Т61 Криптография и безопасность в технологии .NET / П. Торстейнсон, Г. А. Ганеш ; пер. с англ. — 4-е изд., электрон. — М. : Лаборатория знаний, 2020. — 482 с. — (Программисту). — Систем. требования: Adobe Reader XI ; экран 10". — Загл. с титул. экрана. — Текст : электронный.

ISBN 978-5-00101-700-4

Подробно излагаются вопросы реализации на .NET-платформе симметричной и асимметричной криптографии, цифровых подписей, XML-криптографии, пользовательской безопасности и защиты кодов, ASP .NET-безопасности, безопасности Web-служб. Изложение построено на разборе примеров конкретных атак на системы безопасности, содержит большое количество текстов отлаженных программ.

Для программистов, занимающихся разработкой и настройкой систем безопасности на платформе .NET.

УДК 004.7
ББК 32.973.202

Деривативное издание на основе печатного аналога: Криптография и безопасность в технологии .NET / П. Торстейнсон, Г. А. Ганеш ; пер. с англ. — М. : БИНОМ. Лаборатория знаний, 2007. — 479 с. : ил. — (Программисту). — ISBN 978-5-94774-312-8.

ISBN 978-5-00101-700-4

Предисловие

За последние несколько лет криптография в частности и технологии обеспечения безопасности вообще непрерывно повышали свою значимость для пользователей Windows и производителей программного обеспечения. С другой стороны функции обеспечения безопасности (и криптографические функции в том числе) 32-разрядных систем семейства Windows стали сопоставимы с аналогичными функциями больших компьютерных платформ, где такие функции издавна имели высокий приоритет. Теперь же, с широким внедрением .NET, реализация функций обеспечения безопасности на персональных компьютерах стала делом более простым, чем когда-либо ранее. Разумеется, все еще требуются значительные усилия, для того чтобы понять и освоить базовые концепции, а также приобрести навыки, необходимые для обращения с соответствующими функциональными возможностями .NET. (Кстати, это как раз и есть тема данной книги.) И хотя многие из этих функций были доступны и раньше в скрытой форме библиотек Windows, именно появление платформы .NET делает программирование с использованием криптографии и других технологий обеспечения безопасности гораздо более простым, а его результаты – гораздо более мощными, чем когда-либо ранее. Платформа .NET Security Framework позволяет использовать широкий набор специальных классов, которые относительно нетрудно освоить, и все эти возможности мы исследуем в нашей книге.

Книга призвана исчерпывающе осветить все практические вопросы в реализации криптографических и иных, связанных с безопасностью, функциональных возможностей в приложениях .NET. Она представляет собой эффективное учебное пособие, содержащее множество ясных и наглядных примеров исходного кода.

Организация книги

Структура книги включает в себя десять глав и пять приложений. Глава 1 вводит читателя в тематику криптографии и безопасности на платформе .NET и содержит еще нетехнический обзор тех тем, которые гораздо детальней освещены в последующих главах. Также в этой первой главе приведены рассуждения о структуре книги и о том, как в ней соотносятся темы криптографических функций и других функций, связанных с безопасностью. Цель этой главы не состоит в том, чтобы достичь сколько-нибудь глубокого понимания или изучить примеры кода, но она должна привести читателя к концептуальному пониманию криптографических и

других, связанных с безопасностью, технологий, на платформе .NET. Глава 2 обеспечивает знакомство с теорией, достаточное для более глубокого понимания материала последующих глав. Главный ее посыл состоит в том, что все технологии безопасности основываются, в конечном счете, на криптографии, а для многостороннего понимания криптографии необходимо вначале освоить несколько базовых теоретических криптографических концепций. Главы 3, 4, 5 и 6 содержат детально проработанные примеры программирования в .NET, относящиеся к симметричным алгоритмам, асимметричным алгоритмам, цифровым подписям и XML-криптографии, соответственно. Главы 7 и 8 описывают программирование в .NET с использованием концепции идентификации пользователей и концепции прав доступа к коду, соответственно. Глава 9 введет вас в тему программирования на основе технологии ASP.NET, а глава 10 познакомит с программированием Web-служб на основе ASP.NET.

Все аспекты криптографии и безопасности в .NET рассматриваются в подходящем контексте и в должной последовательности в приложениях, где они в реальности чаще всего применяются. Приложения к книге описывают несколько дополнительных тем таких, как способы атаки на выполняющийся код и некоторые математические темы, связанные с криптографией.

Эта книга задумана, как практическое учебное пособие с множеством примеров программ, которые иллюстрируют конкретные вопросы и концепции. Мы здесь концентрируемся скорее на вопросах практического программирования задач безопасности в .NET, чем на системном администрировании. Эта книга обеспечивает достаточно общей информации, чтобы читатель понял, почему вопросы безопасности и криптография столь важны при разработке современного программного обеспечения. Цель книги состоит в том, чтобы научить читателя создавать серьезные приложения с использованием платформы .NET Security Framework. Эта книга является частью серии «The Integrated .NET Series» от Object Innovations и Prentice Hall PTR.

Примеры программ

Лучший способ изучить какую-то серьезную библиотеку классов (например, такую, как .NET Security Framework) состоит в том, чтобы изучить и написать на ней много программ. В этой книге содержится большое число небольших программ, которые иллюстрируют каждую из используемых на практике возможностей .NET Security Framework в отдельности, что облегчает их понимание. Программы (полностью или частично) приводятся в тексте книги (с переведенными комментариями и сообщениями), и все они имеются в программном приложении к книге (с оригинальными комментариями и сообщениями). Эти примеры программ представлены в виде самораспаковывающегося архивного файла на Web-сайте этой книги. После распаковки архива будет создана структура каталогов по пути (по умолчанию) C:\OI\NetSecurity. Все примеры про-

грамм (они начинают появляться в книге, начиная с главы 2) распределены по каталогам Chap02, Chap03 и так далее. Все примеры, относящиеся к одной главе, находятся в отдельных папках внутри каталога соответствующей главы. Имена папок ясно идентифицируют содержащиеся в них примеры программ.

Эта книга является частью серии «The Integrated .NET Series». Примеры программ для других книг этой серии находятся в своих каталогах внутри каталога \OI, поэтому все программы из всех книг серии будут находиться в одном месте. Эти программы предназначены только для целей обучения, и их нельзя использовать ни в каком программном продукте. Все программы, включая инструкции по их использованию, предоставляются на условиях «как есть», и любого рода претензии к ним не принимаются.

Web-сайт

Web-сайт для книг этой серии находится по адресу http://www.objectinnovations.com/dotnet.htm.

Ссылка на этот Web-сайт приведена для загрузки примеров программ для этой книги.

Благодарности

Питер Торстейнсон (Peter Thorsteinson)

Мы хотели бы поблагодарить Джилл Хэрри (Jill Harry) из Прентис Холл за ее поддержку в начинании этого проекта. Также мы благодарны редактору серии Роберту Обергу (Robert Oberg) за его ценную помощь.

Дж. Гнана Арун Ганеш (G. Gnana Arun Ganesh)

Я хотел бы поблагодарить моих родителей Дж. А. Гнанавел (G. A. Gnanavel) и Дж. Н. Вадивамбал (G. N. Vadivambal) за их безграничную любовь, терпение, поддержку и воодушевление. Также я благодарю мою сестру Дж. Дж. Сарадха (G. G. Saradha) за ее любовь, нежность и товарищескую поддержку. Моя глубочайшая благодарность адресована моему доброму ангелу, д-ру Роберту Обергу (Robert Oberg), который воодушевлял меня на протяжении всего этого замечательного проекта. Также я благодарен мистеру Анидо Ди (Anido Dey), мистеру Нарайана Рао Сурапанени (Narayana Rao Surapaneni) и мистеру Виноду Кумару (Vinod Kumar) за то, что они воодушевляли меня и побуждали двигаться вперед. Я хотел бы поблагодарить моего соавтора, Питера Торстейнсона (Peter Thorsteinson), за его руководство и помощь. Наконец, позвольте мне поблагодарить Всевышнего за то, что он предоставил мне такую возможность.

Мы хотели бы поблагодарить Эмили Фри (Emily Frey), Карен Гетман (Karen Gettman) и всех наших редакторов за их конструктивную помощь в усовершенствовании этой книги. Также мы хотели бы поблагодарить всех наших рецензентов за их детальные комментарии, которые во многом помогли обновить содержание книги.

Дж. Гнана Арун Ганеш (G. Gnana Arun Ganesh) является обладателем звания Microsoft .NET MVP (Most Valuable Professional – «наиболее ценный профессионал»), разработчиком, автором и консультантом по .NET. Также он ведет группу .NET Technology в Arun Micro Systems, которая занимается различными фазами технологии .NET. Он работал с технологией Microsoft .NET начиная с ее первоначальной бета-версии. Арун получил степень бакалавра электроники и коммуникационной техники в университете Бхаратьяра (Bharathiar), в колледже Kongo Engineering. Он является автором руководства .NET Reference Guide, опубликованного в InformIT. Он один из авторов Object Innovations (материалы для курса обучения фундаментальным программным технологиям). В качестве автора на темы .NET, он опубликовал более 50 статей о технологии .NET на различных Web-сайтах, посвященных этой теме. Как активный участник рецензирования в Prentice Hall, он написал множество технических рецензий, начиная с рецензии на книгу «C#: How to Program», написанную Харви и Полом Дейтел (Harry and Paul Deitel). Вот уже более двух лет, как администратор Arun Micro Systems, он обеспечивает онлайновое обучение .NET по всему миру.

4 июня 2003 года

Глава 1

Криптография и безопасность в .NET

Нечасто вы встретите книгу, в которой вопросы криптографии обсуждались бы одновременно с вопросами безопасности и при этом с одинаковым вниманием. Эти две темы, на первый взгляд, принадлежат к совершенно разным областям и обычно рассматриваются по отдельности. В конце концов, насколько важны для системного администратора проблемы криптографии, и как часто он задумывается над трудностями разбиения на множители произведения двух больших простых чисел? А как часто математику приходится думать о конфигурации системы с точки зрения безопасности, о таких, например, вещах, как управление доступом к ветвям реестра Windows или к виртуальным каталогам Internet Information Server (IIS)? Книги по криптографии, как правило, отличаются обилием математики и основываются на теоретических подходах. В противоположность им, книги по безопасности компьютерных систем ориентированы не на программиста, они посвящены практическим рецептам выполнения таких, например, операций, как установка сервера сертификатов, создание учетных записей и тому подобное. Между этими двумя крайностями мы видим программиста .NET, озабоченного проблемами, которые по своей природе не относятся ни к математике, ни к системному администрированию.

Тем не менее, программисты становятся все более заинтересованными во внедрении в свои программы как криптографических функций, так и функций, связанных с безопасностью системы. С одной стороны все функции обеспечения безопасности так или иначе основываются на криптографическом «фундаменте». В сущности, все реально применяемые протоколы и технологии безопасности такие, как Kerberos, шифрованная файловая система Windows, пакет Microsoft Certificate Server и все классы .NET Security Framework, полностью основываются на криптографических примитивах. С другой стороны любое программное обеспечение, так или иначе связанное с системой безопасности, в какой-то момент времени обязательно соприкасается с конфигурацией параметров безопасности конкретной системы, на которой оно работает. В этой главе мы рассмотрим вопросы криптографии и безопасности в технологии .NET и получим общее представление о том, как эти две темы связаны между собой с точки зрения .NET-программирования.

В следующих главах мы более детально изучим эти темы.

Природа этой книги

Эта книга написана специально для программистов, интересующихся вопросами криптографии и безопасности, но не для системных администраторов. Мы, следовательно, очень мало внимания уделим здесь тем навыкам, которые требуются профессиональному «сисадмину». Однако каждый программист должен иметь некоторое представление о задачах администрирования, если он хочет разрабатывать эффективные приложения, и программист, работающий в области безопасности, тут не исключение. Поэтому наша книга все же исследует некоторые аспекты администрирования в той мере, в какой они имеют отношение к .NET-программированию в области безопасности. С другой стороны, книга не адресована профессиональным математикам и криптографам[1] и не слишком углубляется в теоретические вопросы криптографии, однако уделяет все же некоторое внимание теории, поскольку даже ограниченное понимание теории криптографии весьма полезно для программиста.

В результате эта книга реализует смешанный подход к освещению предмета, сочетающий в себе основы теории криптографии и вопросы администрирования систем с точки зрения безопасности на платформе .NET. В первой главе мы начнем с обзора наиболее важных концепций безопасности и криптографии на платформе .NET, приведя примеры того, как реализуется система безопасности в приложениях .NET. В главе 2 мы рассмотрим теоретические основы криптографии, начиная с устройства шифра и принципов криптоанализа простейших шифровальных систем, известных с древних времен. Далее в главах 2, 4 и 5 мы продвинемся дальше, изучив технику программирования .NET применительно к трем основным криптографическим системам, используемым в наше время: симметричные и асимметричные системы, а также цифровая подпись. В этих трех главах приводятся развернутые примеры программного кода, реализующего все эти системы при помощи классов .NET Security. В главе 6 продолжится исследование вопросов, связанных с шифрованием и цифровой подписью, но уже в контексте технологии XML. Главы 7 и 8 продемонстрируют основные техники программирования, реализующие такие концепции .NET, как управление доступом на основе механизма ролей и управление доступом кода к ресурсам на основе свидетельств в программах .NET. Разумеется, реалии распределенных приложений и среды Internet делают особенно важными, с точки зрения безопасности, многие специфические аспекты программ, и в главах 9 и 10 мы рассмотрим вопросы безопасности технологии ASP.NET и Web-сервисов, основанных на .NET.

[1] Криптограф – это тот, кто разрабатывает и анализирует алгоритмы шифрования (но не тот, кто просто использует готовые библиотеки для встраивания в свою программу криптографических функций).

Опасность подстерегает повсюду

Если начать думать обо всех возможных опасностях, то можно стать параноиком. В конце концов, среднестатистический гражданин крайне редко становится объектом расследования ЦРУ или ФБР (насколько мы знаем, конечно...) или мишенью международного шпионского заговора. Если дать волю воображению, то многие потенциальные угрозы покажутся просто притянутыми за уши. В самом деле, почему бы ни обернуть свою голову алюминиевой фольгой для того, чтобы злонамеренные инопланетяне не смогли прочитать ваши мысли? Тем не менее, как бы странно это ни звучало, угрозы безопасности наших данных подстерегают повсюду: чем важнее и ценнее данные, тем большее значение приобретают аспекты безопасности. На самом деле, в компьютерном мире данные на удивленье часто подвергаются самым разнообразным угрозам и рискам.

ПОСТАВЬТЕ СЕБЯ НА МЕСТО ЗЛОУМЫШЛЕННИКА

Наверное, вам доводилось слышать совет опытного рыбака: чтобы поймать рыбу, нужно думать, как рыба. Лично мне этот совет всегда казался немного странным, поскольку непонятно все же, как мыслит рыба. Однако этот совет весьма полезен, если ваш противник – человек. Сказанное особенно верно, если речь идет о защите от потенциального злоумышленника[1], атакующего вашу систему: очень полезно изучить ситуацию с его точки зрения и попытаться представить себе возможный ход его мыслей.

Проблема заключается в том, что хорошим ребятам вроде нас с вами очень трудно мыслить так, как мыслит изобретательный злоумышленник, в то время как на его стороне оказываются неисчерпаемые ресурсы: изобретательность, время, энергия и знания. Зачастую все, что мы в состоянии сделать, это бежать вдогонку за мыслью злоумышленника. Шансы в этой игре неравны: одному единственному противнику достаточно найти одну единственную уязвимость, и вся система в опасности. В противоположность этому, измученный защитник должен предусмотреть все возможные варианты нападения и для каждого варианта предпринять достаточные оборонительные меры. Для того чтобы получить общее представление о возможных угрозах, давайте рассмотрим несколько примеров потенциальных угроз.

---

[1] Мы используем определение «злоумышленник, атакующий вашу систему» в отношении любого лица, которое стремится получить недозволенный доступ к системе с целью кражи, подделки или уничтожения данных. Такого злоумышленника часто называют «крэкер» (взломщик). Очень часто в качестве синонима для слова «крэкер» используют слово «хакер», что абсолютно неверно, и чему мы обязаны сообщениям современных СМИ на технические темы. Под словом «хакер», вообще говоря, имеется в виду компьютерный эксперт или энтузиаст, отличающийся обширными знаниями, как правило, приобретенными самостоятельно, и предпочитающий в своей деятельности «партизанские», нерегулярные методы.

ПРИМЕРЫ УГРОЗ И СООТВЕТСТВУЮЩИХ ИМ КОНТРМЕР

Вероятно, нет никаких пределов числу всех возможных трюков и уловок, которые может применить злоумышленник, атакующий вашу систему. Угрозы, с которыми имеет дело программист, концептуально идентичны тем угрозам, которым подвергается пользователь электронной почты (e-mail). Например, большинство людей не шифруют свои электронные сообщения, что можно представить, как отправку по обычной почте открытки вместо вложенного в конверт письма. В этом может заключаться определенный риск, поскольку такие электронные сообщения могут быть легко перехвачены на вашем почтовом сервере или на одном из маршрутизаторов на пути сообщения. Другой вариант угрозы – почтовый вирус может причинить вам немало неприятностей, просто разослав ваши же, ранее отосланные сообщения, по всем адресам в вашей адресной книге – это может поставить вас в неудобное положение. Если бы корреспонденция, по крайней мере, важная ее часть, шифровалась, то подобных проблем не возникло бы. От перехваченных сообщений злоумышленник не получил бы никакой пользы, а действия описанного выше вируса не привели бы ни к чему, кроме рассылки бессмысленных для получателей данных. Существуют почтовые вирусы, которые создают разделяемые ресурсы (общие папки) на вашем компьютере, предоставляя тем самым доступ к вашим папкам всем пользователям вашей локальной сети. Если бы папки были зашифрованы, то не было бы никаких неприятностей. Конечно, вы вовремя обновили свой антивирус, тщательно сконфигурировали почтовую программу и установили все нужные «заплатки», устраняющие известные уязвимости в программном обеспечении, для того, чтобы вообще избежать встречи с почтовым вирусом. Но даже в этом случае шифрование почты и файловой системы создаст дополнительный уровень защиты, который очень пригодится в случае, когда все остальные меры окажутся тщетными. Примеры, о которых шла речь, иллюстрируют важность шифрования в мире электронной почты. По аналогии с этим должно быть понятно, что шифрование критических данных имеет огромное значение вообще в любых программных технологиях.

Использование электронной подписи – еще один способ избежать излишних рисков. К сожалению, большинство пользователей электронной почты не используют электронную подпись в своей корреспонденции. Если вы не подписываете свои наиболее важные сообщения, то кто-нибудь может отправить сообщение от вашего имени, дискредитировав вас тем или иным образом. Если вы постоянно подписываете свои наиболее важные сообщения, то ваши корреспонденты будут ожидать наличия подписи и с подозрением отнесутся к любому фальшивому сообщению от вашего имени. Этот пример иллюстрирует роль цифровой подписи в электронной почте, но все сказанное можно распространить на любую область, в которой могут работать ваши программы.

ЛОЖНОЕ ЧУВСТВО БЕЗОПАСНОСТИ

К несчастью, людям свойственно считать само собой разумеющимся, что использование компьютера знакомым, привычным образом не заключает в себе никакой опасности, но это не так. Вот впечатляющий пример: летом 2002 года корпорация Microsoft в корейской версии пакета Visual Studio .NET непредумышленно распространила копию червя «Nimda». К счастью оказалось, что копия червя была включена в дистрибутив таким образом, что реальной опасности для чьей-либо системы он не представлял. Но кто вообще мог вообразить, что может быть что-то опасное в установке такого известного приложения, полученного от столь надежного поставщика? Эта новость послужила тревожным сигналом[1] для программистов во всем мире! Есть много примеров, доказывающих, что «само собой разумеющаяся» безопасность привычного программного обеспечения вовсе не разумеется сама собой. Как часто вам доводится слышать об обнаружении новых уязвимостей в программах и выпуске «заплат», ликвидирующих эти уязвимости? Увы, но подобные новости доходят до нас едва ли не ежедневно. А хорошая новость состоит в том, что .NET Security Framework и вообще вся платформа .NET способна обеспечить весьма эффективную защиту для программ и данных от множества потенциальных угроз. К сожалению, все проблемы безопасности невозможно решить раз и навсегда, однако технология .NET делает огромный шаг в этом направлении, позволяя создавать программы, гораздо более безопасные, чем когда-либо ранее.

Производители программного обеспечения, системные администраторы, программисты и пользователи – все мы должны проявлять бдительность и принимать необходимые меры предосторожности. Каждый должен помнить, что возникающее иногда чувство безопасности – ложное чувство. Совершенно очевидно, что вопросы безопасности играют важную роль, и это должны признать все люди, чья профессиональная деятельность связана с компьютерами. Особенно важным все это становится сейчас, когда компьютерные системы играют всевозрастающую роль во всех сферах жизни, и когда связи этих систем через Internet становятся все более глобальными.

[1] Много раньше, в истории развития системы UNIX, имел место еще один впечатляющий и убедительный пример подобного происшествия, когда стало ясно, что нельзя слепо доверять безопасности программного обеспечения, которое мы используем. Прочитайте на странице *http://www.acm.org/classics/sep95*, что сказал Кен Томпсон, «отец» UNIX, на эту тему.

Природа криптографии и других средств обеспечения безопасности

Центральной темой этой книги являются криптография и различные средства обеспечения безопасности на платформе .NET. Но если углубиться в эти темы, то легко потерять из виду главные вопросы, касающиеся самой природы криптографии и средств обеспечения безопасности:

- Почему криптография и средства обеспечения безопасности так важны?
- Что возможно и что невозможно сделать с их помощью?

Первый вопрос, в сущности, это вопрос типа «зачем нам это нужно?», а второй – это вопрос типа «что мы можем сделать при помощи этого?». Давайте, прежде чем углубиться в технические детали в последующих главах, рассмотрим эти два фундаментальных вопроса. И далее, когда вы будете читать остальную часть книги, не забывайте о существовании этих вопросов...

Почему криптография и средства обеспечения безопасности так важны

Все мы знаем много примеров из бизнеса, из истории войн, а порой даже из личного опыта, что встречаются ситуации, в которых немного более высокий уровень секретности очень помог бы делу и позволил бы избежать серьезных проблем. Можно вспомнить много случаев, когда дело обошлось бы без неприятностей и неудобств, если бы была проявлена чуть большая осторожность. Конечно, шифрование в состоянии обеспечить вам безопасность и секретность[1] в том, по крайней мере, что касается компьютерных данных. Существует четыре основных аспекта, в которых рассматривается безопасность данных: секретность, аутентификация, целостность и подтверждение обязательств (nonrepudiation). Очевидно, что секретность во многих случаях может иметь большое значение. Понятно, что секретность необходима в ситуациях, когда критическую информацию необходимо скрыть от противника. Вы без труда также представите себе ситуацию, когда очень важно точно знать – с кем именно вы имеете дело (проблема аутентификации). Не менее важно иногда увериться в том, что информация, которую вы получили от кого-то или кому-то отослали, не может быть изменена злонамеренным третьим лицом (проблема целостности). Наконец, вам может потребоваться уверенность в том, что некто, с кем вы договорились о чем-то, не откажется от своих слов (подтверждение обязательств). Для решения всех перечисленных

[1] В большинстве стран сокрытие или утаивание сведений, имеющих отношение к уголовному расследованию, является преступлением, поэтому в деле засекречивания данных необходим разумный подход: одно дело осторожность и предусмотрительность, но совсем другое – создание помех правосудию.

проблем (то есть секретности, аутентификации, целостности и подтверждения обязательств) можно реализовать соответствующие протоколы безопасности, использующие цифровые подписи и цифровые сертификаты, а так же симметричные алгоритмы шифрования, криптографические хеши и коды аутентификации сообщений (MAC – Message Authentication Codes).

ЗАЧЕМ БЕСПОКОИТЬСЯ, ЕСЛИ ВАМ НЕЧЕГО СКРЫВАТЬ?

Зачем беспокоиться о безопасности данных, если вам нечего скрывать? Этот вопрос иногда задают люди, которые наивно полагают, что в защите своей конфиденциальности нуждаются только преступники, террористы и шпионы, чьи грязные секреты нуждаются в сокрытии. Эта ошибочная точка зрения исходит из предположения, что обычному честному человеку мало, что необходимо скрывать в своей жизни, и что энергичная забота о конфиденциальности свидетельствует о нечистой совести. Здесь необходимо четко сознавать, что конфиденциальность – это совершенно нормальная потребность всех законопослушных граждан, что особенно верно, когда речь идет о гражданах государства, чьи власти далеки от идеала.

Чтобы немного яснее представить себе все аспекты этой проблемы, вообразите на минуту, что вам отказано в праве на конфиденциальность. Например, ваше правительство запретило гражданам отправлять по почте письма в закрытых конвертах и обязало публиковать медицинские и банковские данные всех людей – как бы вы чувствовали себя в подобной ситуации? Как бы вам понравился запрет на сохранение в тайне каких-либо сведений о себе, пусть даже с риском стать в результате жертвой преступников? Как бы вы чувствовали себя, если вся ваша переписка по электронной почте и все сведения о ваших посещениях сайтов в Web оказались бы доступными для всех интересующихся на поисковых сайтах, таких, например, как *www.google.com*? Конечно, вам придется признать, что право на конфиденциальность совершенно нормально и законно для вполне обычных, законопослушных граждан, и что определенная степень конфиденциальности является фундаментальным и неотъемлемым правом всех людей.

КАТЕГОРИИ БЕЗОПАСНОСТИ

Мы могли бы привести множество примеров из практики, однако, чтобы не повторяться и не путаться, рассмотрим общие категории, на которые можно разбить проблемы безопасности. Вам обязательно придут на ум из сообщений новостей или даже из собственной практики примеры каждой из этих категорий:

- утечки интеллектуальной собственности, нарушенные контракты, сорванные сделки;
- созданный злоумышленником программный код – почтовые вирусы, «логические бомбы», обычные программные вирусы и «троянские» программы;

- программные техники несанкционированного доступа такие, как, например, техника переполнения буфера;
- фальшивые сообщения[1];
- нарушение договорных обязательств;
- ошибки в коде, угрожающие безопасности данных.

При надлежащем планировании и применении функциональные возможности платформы .NET в плане криптографии и безопасности способны в огромной мере упростить решение всех этих проблем.

Что возможно и что невозможно сделать с помощью криптографии и средств обеспечения безопасности

Теперь мы приходим ко второму вопросу – а что вообще можно сделать при помощи криптографии и других средств защиты информации? Как ни важны все эти инструменты сами по себе, все же они – не панацея от всех проблем, связанных с безопасностью. Знание границ, в которые заключены возможности рассматриваемых средств, – необходимое знание в решении реальных проблем. Вначале давайте рассмотрим то, что можно сделать при помощи криптографии и других средств обеспечения безопасности данных.

ЧТО МОЖНО СДЕЛАТЬ ПРИ ПОМОЩИ КРИПТОГРАФИИ И ДРУГИХ СРЕДСТВ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ ДАННЫХ

Криптография, как и другие технологии, связанные с обеспечением безопасности данных, в состоянии защитить данные лишь в той части, что связана с устройством программного обеспечения, которое используют люди, но она бессильна в той части, что связана с поведением самих людей. Подобно тому, как в «традиционных» трагедиях вроде автомобильных и авиакатастроф, наиболее частой причиной является человеческая ошибка, неудача мер по безопасности данных в компьютерном мире также чаще всего имеет своей причиной ошибку программиста или пользователя. Средства, которые предоставляет платформа .NET в этом плане, позволяют реализовать следующие разновидности защиты:

- обеспечение конфиденциальности информации;
- аутентификация пользователей;

[1] Фальшивые сообщения могут принимать одну из многих форм. Например, перехват пакетов в сеансе TCP-соединения, при котором атакующая сторона получает не предназначенные ей пакеты с данными, модифицирует их с каким-то злым умыслом (например, изменение остатка на банковском счете) и пересылает их настоящему получателю. Другой пример – подмена IP-адреса, когда атакующая сторона подделывает адрес источника в IP-пакетах и отправляет их получателю, выдавая себя за кого-то другого. Самая простая и грубая форма фальшивого сообщения – это сообщение электронной почты e-mail, подделанное с целью ввести в заблуждение получателя.

- целостность информации;
- противодействие нарушению договорных обязательств;
- контроль доступа к ресурсам;
- доступность служб.

Конфиденциальность информации обеспечивается шифрованием, которое ограничивает доступ к информации, делая его возможным только для определенных пользователей. Аутентификация позволяет убедиться в том, что пользователь действительно является тем, за кого он себя выдает. Целостность информации означает, что только авторизованные пользователи могут создать или изменить сообщение, защищенное электронной подписью. Противодействие нарушению договорных обязательств заключается в том, что автор сообщения не сможет впоследствии отрицать факт отправки сообщения или спорить по поводу его содержания. Под «доступностью служб» имеется в виду с одной стороны обычная надежность (то есть среднее время бесперебойной работы), а с другой – средства противодействия атакам типа «отказ в обслуживании» (так называемые DOS-атаки).

Хотя механизмы квотирования, обычно встраиваемые в серверные приложения, еще не поддержаны явным образом в .NET, многие функциональные возможности среды выполнения («виртуальной машины») .NET позволяют значительно улучшить такую характеристику, как доступность служб.

ЧЕГО НЕЛЬЗЯ СДЕЛАТЬ ПРИ ПОМОЩИ КРИПТОГРАФИИ И ДРУГИХ СРЕДСТВ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ ДАННЫХ

Все распространенные алгоритмы шифрования давно уже проанализированы с точки зрения строгой математики, однако реальная криптография применяется в реальном мире. А в реальном мире у нас всегда имеется потенциально слабое звено, которое мы любовно называем «пользователем[1]»... Любые криптографические алгоритмы, любые меры по безопасности системы, все, что в состоянии придумать математики, программисты и системные администраторы, не в состоянии защитить систему от недобросовестного или ленивого пользователя. Не нами замечено, что программисты и системные администраторы довольно часто фокусируются на алгоритмах и протоколах, в то время как причиной нарушения безопасности сплошь и рядом оказываются обычные человеческие слабости. Например, вы можете применить для шифрования в своем приложении сколь угодно сильные алгоритмы, но если пользователь напишет пароль на бумажке и прилепит эту бумажку к монитору, чтобы не забыть пароль, как это нередко делают, то все ваши усилия окажутся напрасными: битва проиграна. Вот примерный список тех угроз

[1] Некоторые, не слишком добродушные, программисты имеют привычку неодобрительно отзываться о пользователях, называя их «юзерами» (от англ. «user») или «юзверями». Мы не поддерживаем подобную практику, поскольку не будь пользователей, не было бы работы и у нас, программистов.

для безопасности системы, которые связаны непосредственно с «человеческим фактором»:

- недостаточная подготовка, низкая дисциплина персонала;
- небрежность разного рода, например, плохой выбор пароля, компрометация ключей или пересылка нешифрованных данных;
- излишняя доверчивость, неопытность и наивность;
- атаки, основанные на «социальной инженерии» и артистизме;
- взятки, запугивание, шантаж;
- плохое качество программ, ошибки в коде.

Криптография и прочие средства обеспечения безопасности – это не ремни безопасности. Нет никакого смысла в самой причудливой защите, если эта защита не действует непрерывно и надежно. Очевидно, что незашифрованные данные не сохранят секретов, неподписанные данные легко подделать или подменить. А насколько защищено приложение с доступом по паролю, если пароль легко угадать или подобрать? Эффективные меры по обеспечению безопасности подразумевают дисциплину и бдительность. Другая проблема заключается в доверии: межсетевые экраны не защитят от собственного доверенного сотрудника, затаившего обиду на фирму. Для обеспечения реальной защиты необходима продуманная политика безопасности, четкие процедуры и эффективная подготовка пользователей.

В атаках на основе «социальной инженерии» используются психологические или эмоциональные трюки и уловки. Проще выражаясь, вы должны относиться очень скептически к заявлениям типа «вы можете мне доверять», поскольку у людей, действительно достойных доверия, редко бывает нужда в подобных заявлениях. Трудность здесь заключается еще и в том, что жертве «социальной инженерии», обманутой артистичным мошенником, потом трудно бывает признаться в факте компрометации защиты. В этой ситуации человеку легче все отрицать, чем признаться в собственной глупости или неосторожности. Поэтому никогда не следует терять бдительность, но если это все же случилось, то не позволяйте своему самолюбию испортить все окончательно.

Шантаж и угрозы, взятки и (не приведи, Господь!) пытки – это тоже своего рода атаки на основе «социальной инженерии», только возведенные в степень. Вы можете думать, что такие вещи случаются с людьми только в кино, но, увы, если ставки достаточно высоки, то все это может произойти и в жизни. Чтобы понять, почему это так, достаточно задуматься над экономическими аспектами раскрытия шифра. Предположим, что для раскрытия ключа к зашифрованным данным вам необходимо три месяца процессорного времени на суперкомпьютере – это, примерно, 50 тысяч долларов. Предположим далее, что нанять бандита, который за пару часов добудет тот же самый ключ, пользуясь собственными методами убеждения, стоит 2 тысячи долларов. Теперь, если вы человек, не обремененный моральными устоями (а такие люди существуют), какой путь вы изберете? Конечно вам следует держаться в рамках закона, тщательно выбирать свою компанию и избегать получения или дачи взяток. Но в том, что касается еще более мрачных вариантов, то вряд ли человеку, попавшему

в подобный переплет, помогут чьи-либо советы. К счастью, пытки в большинстве демократических стран – вещь весьма редкая, хотя есть немало стран, где это остается одной из тяжелых гуманитарных проблем[1].

Разумеется, нельзя во всем винить пользователей. В вопросах безопасности есть несколько аспектов, которые никак не зависят от пользователей, но в которых криптографические методы также бессильны. Всегда остаются проблемы физической безопасности и угроза побочной утечки информации. В том, что относится к безопасности, есть вещи, зависящие от толщины стен, крепости дверей и размера замков, а также от калибра винтовки. Под побочной утечкой мы понимаем здесь любые возможности, ведущие к снятию информации непредвиденным способом.

При помощи криптографических алгоритмов невозможно предотвратить физическое вторжение, такое, как проникновение со взломом и кража, и также неразумно ждать от обычного пользователя, чтобы он это предотвратил. Очевидно, что проблемы физической безопасности должны решаться физическими же методами. Возможно, вам не требуется такой уровень защиты, какой существует в центре управления системы NORAD в Шейенских горах, но какой-то минимальный уровень защиты требуется всегда. Вам не хотелось бы, чтобы вашу переписку по e-mail читала приходящая няня? А что, если ваш компьютер просто украдут? По всей вероятности, каждому требуется, по меньшей мере, парольная защита и запираемая дверь между его компьютером и внешним миром.

Изучая вопросы физической безопасности, помните, что кража или акт вандализма угрожает вашей системе не только со стороны пришельцев из внешнего мира. Внутренняя безопасность может быть сопряжена с точно такими же проблемами, что и безопасность внешняя. Любую компьютерную систему необходимо защищать в соответствии с тем, насколько ценный ресурс она собой представляет, и насколько она подвержена потенциальным угрозам.

Побочная утечка информации принимает самые причудливые формы, в которых физические аспекты функционирования компьютеров могут дать злоумышленникам шансы, которых вы не предвидели. Например, что может произойти с простыми текстовыми данными, копия которых осталась в системном файле подкачки или в дампе памяти, сгенерированном в результате системного сбоя? Если вы недостаточно предусмотрительны, то здесь найдутся возможности для злоумышленников. Другая возможность связана с радиочастотным излучением работающего компьютера, которое при использовании технологии Tempest[2] может стать источником утечки информации. Если представить себе миллионы электронных переключателей, которые срабатывают внутри компьютера каждую микросекунду, то кажется невероятным, что таким способом можно получить какую-то осмысленную информацию. Тем не менее, на практике уже было продемонстрировано, что изображение на экране монитора компьютера можно воспроизвести

[1] Более детально с положением дел в этой печальной области вы можете ознакомиться на сайте *http://www.amyesty.org/*.

[2] «Tempest» означает Transient Electromagnetic Pulse Emanation Standard (Стандарт электромагнитного импульсного излучения переходных процессов) – это набор секретных технологий, разработанных военными США для анализа паразитных электромагнитных излучений, исходящих от аналоговых и цифровых электронных устройств.

на другом компьютере при помощи только той информации, которая получена в результате анализа электромагнитного излучения. Широко известен случай, когда между компьютерами двух конкурирующих компаний, расположенных в соседних зданиях, обнаружилась никем не запланированная перекрестная связь, действовавшая через инфракрасное излучение между ИК-портами и ИК-клавиатурами разных компьютеров.

Побочные утечки информации могут возникнуть также в процессе шифрования данных. Знание промежутка времени, понадобившегося для шифрования, может представлять собой, в определенных случаях, утечку информации. Даже такие измерения, как измерение потребления электроэнергии компьютерным устройством, может дать представление о выполняемых устройством операциях, вплоть до конкретной последовательности микропроцессорных команд! Анализируя временные и энергетические показатели, противник может извлечь полезную для себя информацию, облегчающую расшифровку. Эти примеры наглядно показывают, какие невероятные возможности существуют в индустрии раскрытия секретов.

Следующая разновидность «побочных утечек» – это анализ маршрутизации сообщений. Даже если вы скрыли содержание сообщения, зашифровав его, сам факт его отсылки по определенному адресу может быть обнаружен. Конечно, в демократических странах об этом мало беспокоятся, но иной раз даже вполне законопослушному и респектабельному гражданину это может быть небезразлично. Если некто очень хочет знать все о ваших делах и располагает необходимыми ресурсами, то затруднить заинтересованному лицу задачу будет непросто. Собственно говоря, ваши усилия могут возыметь в такой ситуации обратный эффект[1]. Если вас беспокоит возможность подобного рода побочных утечек, то вам следует принять соответствующие меры. Электромагнитное излучение можно экранировать при помощи металлической сетки, электропитание компьютера можно фильтровать при помощи специальных устройств и так далее[2]. Однако эти ваши усилия, сами по себе, могут быть обнаружены и многое скажут о вас для постороннего взгляда. Скрыть маршруты сообщений в Internet можно, используя анонимные сетевые службы перенаправления почты (anonymous remailers), однако абсолютной анонимности они не гарантируют.

Мы рассмотрели несколько категорий угроз, исходя из предположения, что ничего незаконного не было совершено, но если это не так, то возникают и другие аспекты проблемы. Разумеется, криптография или иные средства обеспечения безопасности данных ничем вам не помогут, если вы дошли до такой жизни:

- свидетельские показания очевидцев или шпионов;
- уличающее поведение, например, экстравагантный стиль жизни или подозрительная поездка;

[1] Вам никогда не доводилось слышать о временах «охоты на ведьм» и о сенаторе Маккарти?

[2] Решение подобных задач требует участия квалифицированного инженера, располагающего необходимым опытом.

- физические улики – отпечатки пальцев, фотографии, финансовые записи или документы;
- расследования, проводимые правительственными организациями[1].

Безопасность в Windows: возраст зрелости

Безопасность данных и криптография с самого начала рассматривались, как важный фактор при работе в многопользовательской среде. Даже в самых ранних системах, «мейнфреймах» (больших ЭВМ) середины 60-х годов таких, как System/360[2], многопользовательская среда строилась с большим вниманием к аутентификации пользователей и изоляции задач друг от друга. Симметричные криптографические алгоритмы такие, как DES (Data Encryption Standard – Стандарт шифрования данных), активно применялись в банковских и правительственных приложениях уже в конце 70-х. Системы семейства UNIX[3] на всем протяжении своей истории учитывали проблемы безопасности как фактор первостепенной важности. В начале 90-х в UNIX-системах уже была внедрена симметричная и асимметричная криптография в различных протоколах и технологиях, как, например, в протоколе сетевой идентификации Kerberos.

В противоположность всему этому, история системы Windows начиналась с явного недостатка средств обеспечения безопасности. До некоторой степени это можно понять, учитывая тот факт, что в начале своего пути (особенно, еще в 16-разрядной версии), система Windows использовалась в качестве однопользовательского инструмента для некритичных приложений или вообще как игровая платформа. Но прошло время, и система Windows выросла, став целой отраслью промышленности и обеспечив мир эффективными и удобными приложениями. Тем не менее, проблемы аутентификации, конфиденциальности и секретности оставались незнакомыми большинству пользователей Windows, и производитель этой системы попросту продолжал угождать своему рынку. По сравнению с той одержимостью проблемами секретности и надежности, которая ха-

[1] Сдавайтесь, сопротивление бесполезно. Многие правительства используют компьютерные системы для сбора информации о преступной или подрывной деятельности. Согласно многим сообщениям, проект ФБР под названием «Carnivore» позволяет отслеживать сообщения электронной почты, а проект «Волшебный фонарь» позволяет внедрять в компьютер подозреваемых специальный вирус, предназначенный для раскрытия паролей.

[2] System/360 была разработана корпорацией IBM в 1964-м году. Главным архитектором этой операционной системы был Джин Амдал.

[3] Система UNIX была первоначально разработана в лабораториях Белла (Bell Labs), тогда частью компании ATT в 1969 г. Первую UNIX-систему написал на ассемблере Ken Tompson, а далее, на протяжении 30 лет, ее развивало множество людей. Многие компании имели отношение к этой системе, благодаря чему ныне существует целый ряд конкурирующих между собой реализаций, включая BSD, System V, Solaris, HP-UX, AIX, Linux и FreeBSD.

рактерна для компьютерных систем больших предприятий, пользователи Windows, в общем, терпимо относились к слабостям системы в этом отношении и больше ценили дружелюбный интерфейс и удобство для пользователей. Вот почему, к досаде компьютерных профессионалов старых времен, развитие Windows было сопряжено с проблемами надежности и безопасности, с утечками информации и уязвимостями различного рода. К счастью, теперь ситуация меняется, и тому есть ряд причин:

- пользователи персональных компьютеров стали гораздо опытней, они нуждаются в большей надежности и безопасности;
- корпорации осознали всю важность проблем безопасности данных в сети Internet;
- в Microsoft с недавнего времени проблемы безопасности и надежности стали рассматриваться, как проблемы стратегической важности;
- многие вычислительные задачи, требующие высокого уровня безопасности, переместились с «мейнфреймов» на персональные компьютеры;
- программный интерфейс Win32 обеспечивает мощную, но не слишком прозрачную поддержку функций, связанных с безопасностью;
- платформа .NET обеспечивает мощную и удобную поддержку всех задач, связанных с криптографией и иными средствами обеспечения безопасности;
- многие эксперты в этой области, включая Брюса Шнайера, провели большую работу по разъяснению всей важности обеспечения безопасности данных;
- снижение стоимости аппаратного обеспечения и рост производительности делают «накладные расходы» на обеспечение безопасности более приемлемыми;
- в январе 2000 года экспортные ограничения США на технологии, связанные с криптографией, были серьезно ослаблены[1];
- общественность теперь гораздо больше знает от таких опасностях, как вирусы, уязвимости, связанные с переполнением буфера[2], и тому подобным;
- увеличение объема критически важных служб, функционирующих в Web, делает проблемы безопасности по-настоящему жгучими.

---

[1] Криптографические продукты высокого уровня теперь разрешены к экспорту в большинство стран без специальной лицензии. На момент написания этой книги список стран, на которые все еще распространяется эмбарго, включал Кубу, Иран, Ирак, Ливию, Северную Корею и еще несколько стран. Актуальную информацию по этому вопросу можно получить на сайте Бюро Промышленности и Безопасности по адресу www.bxa.doc.gov.

[2] Как мы убедимся далее в этой книге, переполнение буфера является чрезвычайно опасной техникой, ее использовал, например, Internet-червь Code Red: данные из буфера переполняются в критически важную область памяти (в стек), и вредоносная программа получает управление.

Среда разработки .NET Framework и «виртуальная машина» CRL

Среда разработки .NET Framework и среда выполнения («виртуальная машина») CRL – Common Language Runtime позволяют программисту эффективно решать все те проблемы безопасности, что описываются в данной главе. Например, кражу информации можно предотвратить, снабдив свое приложение криптографическими функциями. Блокировать выполнение постороннего кода можно специальными приемами программирования и настройками контроля выполнения кода (Code Access Security – CAS).

Атаки, основанные на переполнении буфера, становятся практически невозможными благодаря жесткому контролю, осуществляемому средой выполнения .NET. Ошибки в коде, связанные с переполнениями или определениями типов, также не будут выполнены благодаря проверкам, которые осуществляет «виртуальная машина» CLR.

Следующие функциональные черты платформы .NET обеспечивают наиболее важные аспекты безопасности данных:

- возможность конфигурирования политики безопасности и контроля (административный контроль);
- CAS (контроль выполнения кода, основывающийся на конфигурации политики безопасности и контроля);
- концепция безопасности, основывающаяся на разделении ролей (управление доступом на основе идентификации пользователя и роли, к которой он принадлежит);
- управляемая в процессе выполнения верификация кода (проверка диапазонов адресов и контроль типов);
- домены приложений (изоляция приложений при выполнении);
- криптографические классы (доступ к мощным криптографическим алгоритмам).

Как .NET Framework упрощает решение проблем безопасности

Главная проблема в реализации функций безопасности на основе программного интерфейса Win32 API заключается в том, что он сложен для понимания и труден для использования. Для реализации простой операции, такой, как получение ключа от соответствующего сервиса операционной системы, требуется несуразно большое число строк кода. В результате многие разработчики просто игнорировали эти возможности, если могли без них обойтись. Тем же, кто нуждался в обязательной реализации функций,

связанных с безопасностью, при помощи интерфейса Win32 API, приходилось проделывать большую и непростую работу.

Платформа .NET Framework значительно упрощает решение подобных задач, играя роль оболочки для Win32 Security API и предоставляя объектно-ориентированный интерфейс. Многие операции, в том числе получение ключа, теперь выполняются автоматически при надлежащем использовании классов .NET Security. Кроме того, каждый класс .NET Security, который связан с выполнением критических с точки зрения безопасности задач, объявляется, как закрытый класс, и не может быть подделан или изменен с целью преодоления защиты.

Надежность и платформа .NET Framework

Прежде чем вы получите какую-то пользу от использования технологий криптографии, необходимо вначале убедиться в надежной работе приложения. Зачем нужна хорошо спланированная инфраструктура безопасности, если само приложение работает нестабильно и зачастую просто «падает»? Платформа .NET Framework делает многое для того, чтобы общая надежность приложений оставалась высокой. Первое, что необходимо здесь понимать, это то, что приложения .NET не компилируются в машинный код. Вместо этого они компилируются в некую промежуточную форму, именуемую Microsoft Intermediate Language (MSIL, промежуточный язык Microsoft), который для краткости называют также IL и который похож на байтовый код языка Java[1]. Это позволяет среде выполнения CRL и платформе .NET Framework решать многие задачи, связанные с безопасностью, автоматически:

- проверка границ в процессе выполнения кода делает невозможным переполнение стека или разрушение данных в памяти;
- проверка соответствия типов в процессе выполнения кода делает невозможными намеренные или случайные изменения типов;
- при вызове кода полномочия вызывающего кода верифицируются при помощи механизмов стека;
- автоматическая «сборка мусора» эффективно решает проблему утечек памяти;
- обработка исключений позволяет корректно обрабатывать все нештатные ситуации;
- выполнение кода, основанное на механизме разрешений, регулируется политикой безопасности, основывающейся на «свидетельствах».

[1] Существует много нововведений в .NET, которые стали новым словом в обеспечении секретности и безопасности. Справедливости ради, однако, следует признать, что наиболее эффективные из этих новых черт были первоначально реализованы на платформе Java, в классах Java Cryptography Extension (JCE).

Управляемый код и безопасность типов

Код, который используют службы CLR, называется управляемым (managed) кодом. Среда CLR предоставляет для использования набор служб таких, как проверка типов или автоматическая сборка мусора, что делает приложение более надежным и безопасным. Для того чтобы использовать эти службы, исполняемый код должен вести себя предсказуемо и упорядоченно. Безопасность типов является важным фактором общей безопасности и надежности приложения. Эта безопасность обеспечивается благодаря тому, что среда выполнения CLR «знает» абсолютно все обо всех типах данных, фигурирующих в приложении. Используя свое знание, среда CLR строго отслеживает использование типов.

Например, все типы данных, включая строки и массивы, обладают определенной структурой и обязаны подчиняться определенным правилам. Система общих типов (CTS – Common Type System) определяет правила для каждого типа данных, а также операции, которые CLR может выполнять с данными этого типа. Все эти ограничения определяются в CTS и реализуются в MSIL. Также CTS определяет возможные распределения памяти для типов и операции, допустимые в управляемом коде. Именно CTS ограничивает наследование классов и предотвращает выполнение небезопасных операций таких, как превращение целого числа в указатель или переход через границы массива. Код MSIL в период выполнения обычно компилируется в машинный код конкретной аппаратной платформы, на которой он выполняется, уже после выполнения этих проверок.

Для проверки типов на безопасность .NET использует в своих так называемых «сборках» (assemblies) описательные метаданные, которые определяют заключенный в сборке код и данные. Управляемые данные распределяются в «куче» автоматически. Это автоматическое распределение памяти называется сборкой мусора. Сборка мусора сокращает утечки памяти и способствует повышению надежности.

Каждый объект принадлежит к некоторому типу, и потому любая ссылка на объект есть ссылка на заранее распределенную область памяти с известной структурой. Поскольку произвольные манипуляции с указателями не разрешены (за исключением случая, когда они явно разрешены при помощи зарезервированного слова unsafe), единственный способ доступа к объекту состоит в обращении к его публичным элементам. Таким образом, CLR в состоянии проанализировать объект на предмет безопасности, прочитав его метаданные, содержащиеся в «сборке». Нет необходимости анализировать весь код объекта, для того чтобы убедиться в его безопасности.

В принципе возможно, используя язык C# и зарезервированное слово unsafe, создать небезопасный код, но другие языки такие, как VB.NET, в принципе могут создавать только управляемый код с безопасными типами. Слово unsafe требуется только для прямой работы с указателями. Такой код может потребоваться для вызова старых DLL, но он не может быть верифицирован CLR на безопасность типов.

MSIL определяет набор платформо-независимых инструкций[1], которые используют все компиляторы .NET. Этот язык реализует и использует правила проверки типов CTS непосредственно в процессе выполнения. После завершения всех проверок MSIL компилирует программу в машинный код конкретной платформы. Проверка типов выполняется по умолчанию, но ее можно отключить, если исполняемый код достоин доверия.

Программирование с использованием криптографии в .NET

Как мы убедимся в главах 3, 4 и 5, в .NET Framework доступно несколько криптографических классов. Эти классы поддерживают все основные криптографические алгоритмы, используемые в наше время. Мы познакомимся со всем этим очень подробно в последующих главах, пока же давайте сделаем краткий обзор основных функций, выполняемых криптографическими классами:

- симметричные алгоритмы шифрования DES, 3DES и RC2;
- криптографические потоки;
- асимметричный алгоритм RSA;
- цифровая подпись на основе RSA и DSA;
- хеш-алгоритмы, включая MD5, SHA1, SHA-256 и другие;
- коды аутентификации сообщений (MAC);
- алгоритм хеша с ключом;
- XML-шифрование;
- XML-подписи;
- генераторы псевдослучайных последовательностей[2];
- безопасность в ASP.NET;
- безопасность Web-служб.

[1] Независимость от платформы для CLR возможна только в теории. Однако возможно ли будет портирование CLR на многие платформы, как это произошло с Java Runtime Environment, покажет время.

[2] В .NET Framework имеется класс Random в пространстве имен System, который можно использовать для генерации последовательностей псевдослучайных чисел для игр, симуляторов и в некоторых статистических расчетах. Этот класс ни в коем случае нельзя использовать в целях криптографии, иначе вы серьезно ослабите свою криптографическую защиту. В целях криптографии можно использовать только специальный генератор (PRNG), как будет объяснено в последующих главах.

Программирование с использованием средств обеспечения безопасности в .NET

Как мы увидим в главах 7 и 8, в .NET Framework существует два возможных подхода к созданию безопасных программ: это CAS и безопасность, основанная на механизме ролей. Эти два подхода мы исследуем далее в контексте Internet и распределенных приложений в главах 9 и 10. Мы познакомимся с более полным определением этих терминов, а также с примерами кода, иллюстрирующими все это, но пока что просто бросим беглый взгляд на эти две концепции.

Безопасность, основанная на механизме ролей

У большинства людей есть, по крайней мере, интуитивное представление о пользователях и ролях, основанное на использовании таких операционных систем, как Windows 2000 и Windows XP. Просто вы можете управлять тем, как определенные пользователи получают доступ к тем или иным ресурсам, таким, как файлы, папки, реестр и так далее. Концепция безопасности, основанная на механизме ролей, использует два ключевых понятия: *аутентификация* и *авторизация*. Вопрос аутентификации – «кто вы такой», а вопрос авторизации – «позволено ли вам делать это».

В механизме ролей слово «вы» следует заменить объектом *principal*. Этот объект содержит в себе свойство *identity*, которое представляет идентификатор пользователя, от имени которого выполняется текущий поток. В главе 7 мы рассмотрим, как при помощи механизма ролей достигается безопасность в программах .NET.

CAS, свидетельства, политика и разрешения

Контроль выполнения кода (CAS) позволяет решать, какие действия коду разрешено выполнять, а какие – нет. CAS основывается на том, что вы можете назначить уровень доверия сборкам с кодом и ограничить разрешенные для этого кода операции, основываясь на заданных разрешениях. CAS тесно связан концепцией безопасности, основанной на *свидетельствах* (evidence). *Свидетельство* – это набор вспомогательной информации, которую CLR использует для принятия решений о том, к каким группам принадлежит код в сборке и, соответственно, какие действия этому коду разрешены. Свидетельства могут состоять, например, из информации о происхождении кода, из цифровой подписи на коде и так далее.

Политика безопасности – это набор правил, который конфигурируется администратором и используется средой CLR для принятия CAS-решений. Политика безопасности может быть задана на уровне предприятия,

машины, пользователя или приложения. Политика безопасности определяется в терминах разрешений. *Разрешение* – это объект, описывающий права и привилегии сборок, принадлежащих к определенной группе кода, на доступ к определенным ресурсами и на выполнение определенных действий. Собственно говоря, политика просто задает соответствие между свидетельствами и разрешениями.

Сборки могут программно или иным образом затребовать получение определенных разрешений. Политика безопасности определяет, какие разрешения могут быть предоставлены конкретной сборке. Политика безопасности основывается на правилах, которые определяет администратор, а CLR применяет эти правила, проводя соответствующую политику. Свидетельства, представленные идентификатором пользователя и, соответственно, его полномочиями, используются для выбора политики, которая будет применена к конкретной сборке кода. CLR решает на основе свидетельств сборок, какие разрешения им можно предоставить. Свидетельства могут включать в себя собственно идентификацию сборки, идентификацию того, кто ее подписал, ее происхождение (включая, например, URL-сайт и зону Internet). В главе 8 мы изучим все эти темы более подробно.

Итоги главы

В нашей первой главе мы рассмотрели соотношение между криптографией и другими средствами обеспечения безопасности. Познакомившись с общей организацией книги, мы рассмотрели те угрозы и риски, о которых должны помнить программисты, а также методы, которые можно использовать для защиты. Мы также совершили краткий обзор всех концепций, с которыми должны будут иметь дело программисты, реализующие криптографические и иные функции обеспечения безопасности на платформе .NET. Наконец, мы кратко описали главные черты техники программирования, характерные для .NET, все то, что мы подробно изучим в последующих главах.

Глава 2

Основы криптографии

В этой главе вы познакомитесь со многими фундаментальными концепциями, знание которых необходимо для понимания материала последующих глав. Здесь наша цель состоит в том, чтобы освоить базовые понятия криптографии на примере нескольких простых классических алгоритмов. Это поможет пониманию более сложных криптографических систем, которые встретятся нам далее.

Также в этой главе делается обзор криптографии в исторической перспективе. В остальной части книги содержится информация практического характера, охватывающая современные методы и технологии. Некоторые главы будут полностью посвящены практическим вопросам применения .NET Security Framework для реализации конкретных методик защиты. Если вы уже знакомы с базовыми понятиями криптографии, то эту главу можно безболезненно пропустить.

В заглавии этой книги сочетаются слова «безопасность» и «криптография», а эти понятия тесно связаны на фундаментальном уровне. Под «безопасностью» можно понимать множество самых разных вещей, однако исходная мысль здесь остается неизменной: не дать случиться чему-то опасному или нежелательному. Например, ваша цель может состоять в том, чтобы только авторизованные пользователи могли выполнить определенные действия в отношении определенных компьютерных ресурсов. Компьютерная безопасность самым непосредственным образом основывается на криптографической науке. Например, Kerberos[1], безопасный протокол сетевой аутентификации пользователей, и вместе с ним система безопасности .NET, основывающаяся на «свидетельствах» и обеспечивающая безопасность исполняемого кода, одинаково полагаются на мощные криптографические технологии, которые лежат в их основе.

---

[1] Система аутентификации Kerberos разработана в Массачуссетском технологическом институте (проект «Афины») и основывается на криптографической системе с симметричными ключами. Kerberos (Цербер) – это трехглавый сторожевой пес из греческой мифологии, охранявший вход в царство теней. Протокол с таким названием, надо полагать, просто обязан обеспечивать полную безопасность!

Чтобы секреты оставались секретами

Компьютерная безопасность – это искусство защиты различных ресурсов от доступа тех, кому вы не вполне доверяете. Разумеется, любые методы обеспечения безопасности вообще имеют смысл только тогда, когда вы в состоянии сохранить в секрете некоторые критические данные такие, как пароли, ключи и тому подобное. Без сохранения в секрете всего этого обеспечение безопасности невозможно. С другой стороны, криптография – это наука о сохранении секретов. В сущности, криптографию можно рассматривать как способ сохранения больших секретов (которые неудобно хранить в тайне из-за их размеров) при помощи секретов малых (которые прятать проще и удобней). Как мы увидим далее в этой главе, под «большими секретами» имеется в виду, как правило, так называемый *открытый текст*, а «малые секреты» обычно называют *криптографическими ключами*.

Основные термины криптографии

Шифром называют систему или алгоритм, трансформирующий произвольное сообщение в такую форму, которую не сможет прочитать никто кроме тех, кому это сообщение предназначено. Сущность шифрования состоит в задании соответствия между произвольным исходным сообщением и зашифрованным результирующим текстом, при этом должен существовать способ задать обратное соответствие, при помощи которого можно восстановить исходный текст. При шифровании и дешифровании используется *ключ (key)*, который и есть тот «маленький секрет». В криптографии принято полагать, что ключ должен быть единственным секретным элементом, притом, что алгоритм шифрования может быть известен[1]. Хороший алгоритм шифрования должен давать в результате сообщение, которое трудно отличить от случайного набора знаков, и из которого атакующая сторона может извлечь абсолютный минимум информации об исходном сообщении. *Пространством ключей* называют множество всех возможных ключей, доступных для использования в алгоритме.

Исходное, незашифрованное сообщение называют *открытым текстом (plaintext)*, при этом слово «текст» не должно наводить на мысль, что речь обязательно идет о словах, которые можно прочитать. Под текстом здесь могут разуметься любые данные, текстовые или двоичные, которые что-либо значат для человека или программы. *Зашифрованным текстом (ciphertext)*, соответственно, называют сообщение, полученное

[1] Это называют «принципом Керкхофа», в соответствии с которым надежность криптосистемы должна зависеть только от секретности ключа, будучи независимой при этом от секретности алгоритма шифрования.

в результате шифрования. Шифрованный текст используется везде, где необходимо пересылать секретные сообщения по открытым каналам связи или хранить секретную информацию на носителях, которые могут быть доступны атакующей стороне. Примерами первого и второго случаев, соответственно, являются протокол безопасного соединения SSL и шифрованная файловая система EFS (NTFS Encrypted File System). Под термином *шифрование* понимают процесс преобразования открытого текста в шифрованный текст. Соответственно, *дешифрование* – это процесс, обратный шифрованию, преобразующий зашифрованный текст в открытый. На рисунке 2.1 схематически изображен процесс шифрования конфиденциального сообщения при помощи симметричного шифра[1].

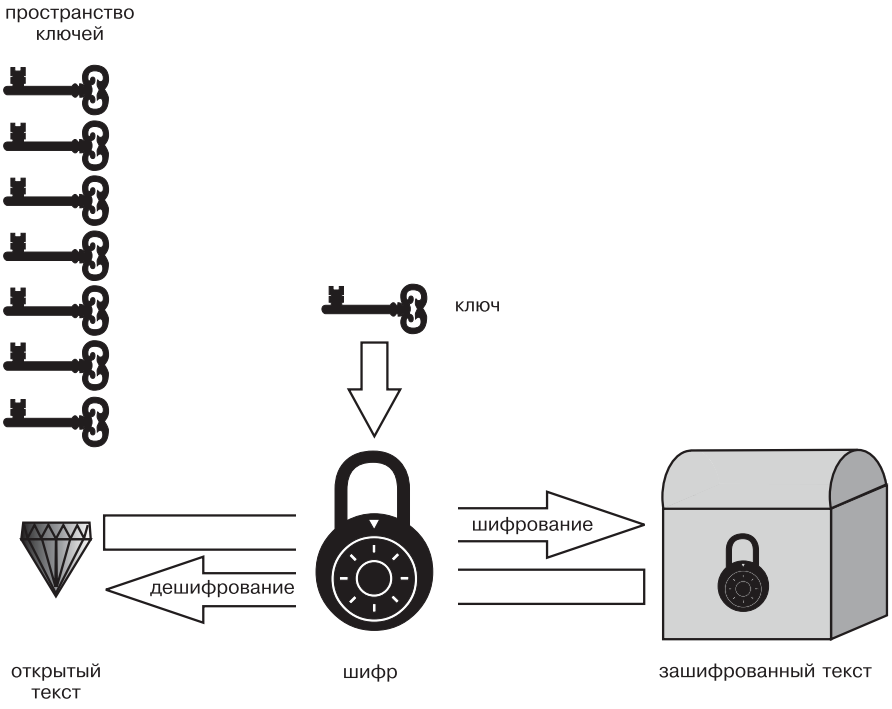


Рис. 2.1. Симметричное шифрование

Отправитель передает открытый текст сообщения для шифрования, после чего зашифрованный текст отправляется к получателю. Тем временем некто, кого мы будем называть *«нападающей»* или *«атакующей»* стороной, иными словами, *злоумышленник*, пытается получить *ключ к шифру* и *расшифровать сообщение*. Рисунок 2.2 иллюстрирует взаимоотношения между отправителем, получателем и атакующей стороной.

[1] Симметричные шифры обсуждаются в главе 3. Другая категория шифров, так называемые асимметричные шифры рассматриваются в главе 4.

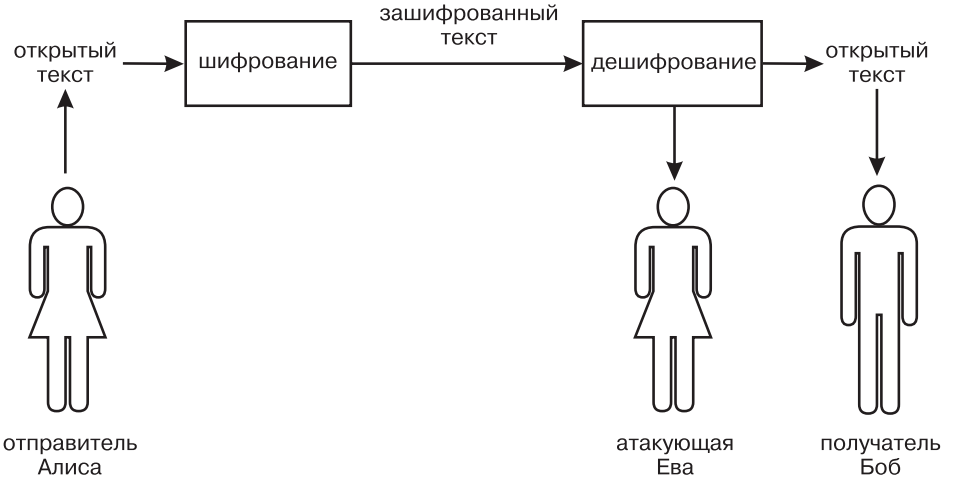


Рис. 2.2. Отправитель, получатель и атакующая сторона: Алиса, Боб и Ева

Для того чтобы сделать этот сценарий более жизненным, в криптографической литературе отправителя и получателя называют Алисой и Бобом, а атакующую сторону представляет злой персонаж по имени Ева. В более сложных сценариях вводятся дополнительные персонажи. При помощи всех этих персонажей описание криптографических технологий делается более живым и наглядным.

Разработку и применение шифров называют криптографией, в то время как науку о раскрытии шифров – *криптоанализом*. Поскольку проверка шифров на стойкость является обязательным элементом их разработки, криптоанализ также является частью процесса разработки. *Криптология* – это наука, предметом которой являются математические основания как криптографии, так и криптоанализа одновременно.

Криптоаналитической атакой называют использование специальных методов для раскрытия ключа шифра и/или получения открытого текста. Как уже говорилось, предполагается, что атакующей стороне уже известен алгоритм шифрования, и ей требуется только найти конкретный ключ.

Другая важная концепция связана со словом «взлом». Когда говорят, что некоторый алгоритм был «взломан», это не обязательно означает, что найден практический способ раскрытия шифрованных сообщений. Может иметься в виду, что найден способ существенно уменьшить ту вычислительную работу, которая требуется для раскрытия шифрованного сообщения методом «грубой силы», то есть простым перебором всех возможных ключей. При осуществлении такого «взлома» практически шифр все же может оставаться стойким, поскольку требуемые вычислительные возможности будут все еще оставаться за гранью реального. Однако, хотя существование метода «взлома» не означает еще реальной уязвимости алгоритма, обычно такой алгоритм более не используют.

Секретные ключи против секретных алгоритмов

Важный аспект эффективности шифра состоит в том, что надежность (стойкость) шифра должна полностью зависеть от сохранения в тайне ключа, но не зависеть от секретности алгоритма[1]. Это, на первый взгляд, противоречит интуитивным представлениям: кажется, секретный алгоритм мог бы сделать шифр более стойким. Однако на практике, очень трудно сохранить в секрете алгоритм шифрования, и несравненно легче сохранить в секрете конкретный использованный ключ. Представьте себе, например, несколько ситуаций. Что произойдет, если одно из ваших доверенных лиц перестанет быть доверенным? Поменять ключи в такой ситуации гораздо проще, чем сменить алгоритм. Что делать, если у вас просто появились подозрения относительно возможной компрометации алгоритма? Если бы такие подозрения появлялись в отношении ключа, то его смена просто «на всякий случай» не вызовет никаких трудностей, в то время как смена алгоритма шифрования выливается в большую проблему. Если вы до сих пор полагались на секретность алгоритма, то изменение этого алгоритма потребует изменения всего программного обеспечения и всей инфраструктуры, связанной с использованием этого алгоритма. Кроме того, вам придется разработать новый секретный алгоритм. И, напротив, для смены ключа вам достаточно сгенерировать новый случайный ключ и продолжать пользоваться той же самой инфраструктурой.

Идею использования секретного алгоритма часто называют «секретностью через скрытность», метод, при котором вы прячете ценный предмет в небезопасное, но скрытое место, например, под матрац. Секретность, основанная на хорошо известном (но притом мощном) алгоритме дает более высокую степень безопасности: вы при этом прячете свой ценный предмет в банковский сейф, который всем виден, но надежно защищен от взлома. Существует несколько сильных стандартных алгоритмов, которые были тщательнейшим образом исследованы. Никогда не используйте алгоритм, который вы изобрели сами (если, конечно, вы не криптограф с мировым именем), и никогда не используйте частные («фирменные») алгоритмы, предлагающиеся на рынке разными ловкими производителями. Если в рекламе подобных систем вам встретятся выражения вроде «невозможно взломать» или «абсолютная надежность», почти наверняка алгоритм попросту слаб, поскольку настоящие криптографы никогда не используют подобных выражений.

Используя хорошо известный, опубликованный стандартный алгоритм шифрования, вы получаете важное преимущество: этот алгоритм подвергался глубокому анализу и «атакам» со стороны множества исследователей в криптографическом сообществе. Данное обстоятельство дает вам возможность точно оценить криптографическую стойкость алгоритма, и, что еще

[1] Можно провести здесь аналогию: операционные системы с открытым кодом потенциально могут достичь более высокого уровня безопасности по сравнению с системами, исходный код которых неизвестен. Этот вопрос, впрочем, остается открытым и является предметом весьма оживленных дебатов.

важнее, быть уверенным в этой оценке. Наконец, здесь вы получаете все обычные преимущества, вытекающие из политики следования стандартам. У вас появляется широкий выбор в методах реализации, вы снижаете издержки и получаете совместимость с другими системами. Понятно, что при использовании стандартного алгоритма сам алгоритм не может быть секретом! Примеры стандартных алгоритмов, которые мы обсудим в последующих главах, включают в себя:

- DES (Data Encryption Standard – стандарт шифрования данных);
- «тройной» DES;
- AES (Advanced Encryption Standard – усовершенствованный стандарт шифрования);
- RSA (Rivest, Shamir и Adleman);
- DSA (Digital Signature Algorithm – алгоритм электронной подписи);
- SHA (Secure Hash Algorithm – хеширующий криптографический алгоритм).

Заметим, что алгоритм DES в техническом смысле «взломан», но все еще считается достаточно стойким. Мы увидим, что алгоритмы DES, «тройной» DES и AES относятся к симметричным системам, в то время как RSA и DSA представляют асимметричное шифрование и используются для обмена ключами и реализации цифровой подписи. Алгоритм SHA относится к категории хеширующих и используется в нескольких криптографических целях.

Классические методы сохранения тайны

На протяжении всей человеческой истории технологии сохранения и раскрытия секретов развивались параллельно, в непрерывной борьбе друг с другом. Сейчас мы рассмотрим несколько простых классических шифров. Хотя эти алгоритмы на практике давно уже не используются, и вряд ли вам доведется иметь с ними дело, знакомство с ними поможет вам освоить основные концепции криптографии, а также познакомиться с некоторыми математическими принципами.

ШИФР ЦЕЗАРЯ

Для того чтобы увидеть, как упоминавшиеся выше понятия выглядят в реальной ситуации, мы рассмотрим очень простой шифр, изобретение которого приписывается Юлию Цезарю. Уж если кто-то в истории и нуждался в криптографии, то это, конечно, Юлий Цезарь! По современным меркам это шифр тривиален и легко раскрывается, поэтому в реальности он применяться уже не может, но главное его достоинство заключается в простоте, и потому он послужит нам хорошим вступлением в мир шифров. Начиная со следующей главы, мы будем знакомиться с более современными и эффективными шифрами.

В шифре Цезаря[1] каждая буква в открытом тексте смещается на три позиции так, что буква «A», например, замещается буквой «D». Буква «B» замещается буквой «E» и так далее. Конец алфавита замыкается на его начало так, что буква «X» замещается буквой «A», а буква «Y» – буквой «B», «Z» – «C». В шифре Цезаря каждая буква смещается на 3 позиции, однако в более широком смысле этот шифр можно рассматривать, как смещение на некое целое число позиций (k), причем число k будет играть роль ключа. В этой главе мы будем рассматривать обобщенный вариант шифра Цезаря, пренебрегая исторической точностью. Поскольку каждый конкретный символ здесь замещается также конкретным одиночным символом, этот шифр относится к категории *моноалфавитных*. На рисунке 2.3 схематически изображено преобразование, выполняемое шифром Цезаря.

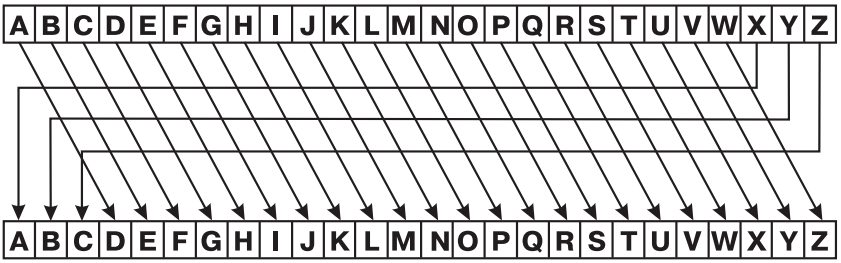


Рис. 2.3. Шифр Цезаря

```
Ключ:
  3
Открытый текст:
  P = HELLO CAESAR CIPHER
Зашифрованный текст:
  C = KHOOR FDHVDU FLSKHU
```

Давайте рассмотрим математическое определение шифра Цезаря. Прелесть математической нотации состоит в ее точности и компактности. Другое важное достоинство математического подхода состоит в следующем: доказав некую теорему, вы можете обращаться с доказанным, как с фактом, и потому можете использовать результат для доказательства другой теоремы. Единственная проблема – для некоторых людей математика ничем не лучше головной боли. Хотя для этого простого шифра математическое представление необязательно, иметь дело с более сложными

[1] У Юлия Цезаря (100-44 до н. э.) было множество секретов. Он вторгся в Британию и в Египет, а также почти во все остальные страны между этими двумя крайними точками, и потому у него было множество чисто военных секретов. Как у первого римского диктатора, у него было множество политических врагов, включая республиканцев Кассия и Брута, которые его, в конце концов, и убили. Вероятно также, что ему необходимо было хранить в секрете свою переписку с Клеопатрой. В добавление ко всему, у Клеопатры были близкие отношения с ближайшим соратником Цезаря Марком Антонием. Очевидно, у всех них было множество секретов, нуждающихся в сохранении.

шифрами можно, только используя математический аппарат. Поэтому, если вы намерены серьезно заниматься криптографией, вам следует обладать хотя бы минимальными познаниями в нескольких областях математики таких, как теория чисел и абстрактная алгебра. Так или иначе, но сейчас вам предстоит познакомиться с математическим описанием шифра Цезаря.

Определение: сдвиговый шифр (обобщенный шифр Цезаря)

При произвольном ключе k, где

$K \in Z_{26}$ (так обозначается множество целых чисел, $0 \le k \le 25$)

и произвольном открытом тексте p в виде кортежа[1], где

$p = (p_1, p_2, p_3, ..., p_m)$ и $p_i \in Z_{26}$ для $0 \le i \le m$,

результирующий шифрованный текст c будет представлен кортежем

$c = (c_1, c_2, c_3, ..., c_m)$ и $c_i \in Z_{26}$ для $0 \le i \le m$.

При этом функция шифрования для $E_k(p)$ для сдвигового шифра определяется следующим образом:

$c_i = E_k(p_i) = p_i + k \pmod{26}$ для $0 \le i \le m$.

А функция дешифрования $D_k(c)$ определяется, как

$p_i = D_k(c_i) = c_i - k \pmod{26}$ для $0 \le i \le m$.

Заметим, что шифр является обратимым, и мы должны доказать следующее:

$D_k(E_k(x)) = x$ для $c_i - k \pmod{26}$ для всех $x \in Z_{26}$

Как видите, математическое определение не обращается к таким деталям реального мира, как буквы. Вместо этого символы открытого текста, так же, как и символы текста зашифрованного, представляются целыми числами от 0 до 25. Прочие детали такие, как пунктуация, также игнорируются. В этом определении функция E_k является шифрующей функцией, а D_k – дешифрующей, при этом k играет роль ключа. Стандартная нотация Z_{26} представляет множество целых чисел {0, 1, 2, 3, ... , 25}. Выражение (mod 26) указывает, что мы используем арифметику по модулю 26 вместо обычной арифметики. Описание арифметики по модулю вы найдете в приложении B, посвященном математическим аспектам криптографии.

Если вас интересует реализация этого шифра на языке C#, посмотрите на пример **CaesarCipher**. Это реализация крайне проста, поэтому листинг мы здесь не приводим, но вы можете посмотреть на прилагаемый исходный код. Если вы запустите программу, то обнаружите, что она принимает символы от A до Z и пробел, игнорируя все прочие символы. Программа предлагает вам ввести ключ – число от 0 до 25 – и отвергает любые значения вне этого диапазона. Типичный сеанс работы программы **CaesarCipher** выглядит следующим образом:

[1] **Кортежем в линейной алгебре называют упорядоченный набор значений.**

```
Enter uppercase plaintext: VENI VIDI VICI[1]
Enter from 0 to 25: 3
Resulting ciphertext: YHQL YLGL YLFL
Recovered plaintext: VENI VIDI VICI
```

Если вы часто пользуетесь Usenet, то вам, возможно, знакома кодировка ROT13. Эта кодировка есть не что иное, как шифр Цезаря с ключом $k = 13$. Поскольку ROT13 чрезвычайно легко раскрыть, ее никогда не используют в реальной криптографии. Тем не менее, клиентское программное обеспечение Usenet[2] обычно предоставляет возможность отправлять сообщения в кодировке ROT13, для того чтобы скрыть ответ на загадку или избежать обид. Любой может разгадать этот шифр, но такое разгадывание потребует все же специальных усилий. Удобство кодировки ROT13 состоит в том, что один универсальный ключ 13 используется всеми для кодирования и декодирования.

АТАКА МЕТОДОМ «ГРУБОЙ СИЛЫ» НА ШИФР ЦЕЗАРЯ

Атакой методом «грубой силы» называют способ раскрытия шифра, при котором поиск ведется во всем возможном пространстве значений ключа до тех пор, пока не будет получен осмысленный результат. Для того чтобы проделать это с шифром Цезаря, вам необходимо задаться значением ключа 1 и продолжать перебирать все числа до 25, пока не будет получен осмысленный текст. Конечно варианты $k = 0$ и $k = 26$ будут бессмысленными, поскольку в этих случаях зашифрованный и открытый тексты будут идентичными. Пример программы **CaesarCipherBruteForceAttack** представляет собой реализацию этой атаки.

```
Class CaesarCipherBruteForceAttack
{
   static void Main(string[] args)
   {
   ...
            //последовательно проверять все пространство ключей
            for (int testkey=1; testkey <= 25; testkey++)
            {
                  Console.Write(
                        "testkey: {0,2} produced plaintext: ",
                        testkey);
                  StringBuilder plaintext =
                        Decrypt(ciphertext, testkey);
            }
   }
```

[1] Латинская фраза VENI VIDI VICI означает «пришел, увидел, победил» (эту фразу приписывает Юлию Цезарю римский историк второго века Светоний). Нет доказательств того, что Цезарь действительно произносил эти слова, однако если он это сделал, то секретом это, уж точно, не было!

[2] Usenet – это система телеконференций в Internet, объединяющая в себе тысячи конференций («групп новостей») на самые разные темы, какие только могут представлять интерес для разных людей, в том числе и на тему криптографии.

```
   ...
   static StringBuilder Decrypt(StringBuilder ciphertext,
              int key(
   {
              StringBuilder plaintext =
                     new StringBuilder(ciphertext.ToString());
              for (int index=0; index < plaintext.Length; index++)
              {
                     if (ciphertext[index] != ' ')
                     {
                            int character =
                            (((ciphertext[index]+26-'A')-key)%26)+'A';
                            plaintext[index] = (char)character;
                     }
              }
              return plaintext;
   }
}
```

Предположим, например, что у вас есть зашифрованный текст KHOOR, и вы знаете, что он зашифрован при помощи сдвигового шифра наподобие шифра Цезаря. Ниже приведен вывод программы **CaesarCipherBruteForceAttack**, где перечислены все возможные варианты расшифровки текста, полученные с использованием 25 возможных ключей. Как легко заметить, только вариант HELLO выглядит осмысленным текстом, и, таким образом, правильным ключом является число 3. Разумеется, в реальной атаке методом «грубой силы» перебор ключей прекращается, как только достигнут нужный результат, но мы привели здесь все варианты, для того чтобы лучше проиллюстрировать принцип.

```
Enter uppercase ciphertext: KHOOR
testkey:  1 produced plaintext: JGNNO
testkey:  2 produced plaintext: IFMMP
testkey:  3 produced plaintext: HELLO
testkey:  4 produced plaintext: GDKKN
testkey:  5 produced plaintext: FCJJM
testkey:  6 produced plaintext: EBIIL
testkey:  7 produced plaintext: DAHHK
testkey:  8 produced plaintext: CZGGJ
testkey:  9 produced plaintext: BYFFI
testkey: 10 produced plaintext: AXEEN
testkey: 11 produced plaintext: ZWDDG
testkey: 12 produced plaintext: YVCCF
testkey: 13 produced plaintext: XUBBE
testkey: 14 produced plaintext: WTAAD
testkey: 15 produced plaintext: VSZZC
testkey: 16 produced plaintext: URYYB
testkey: 17 produced plaintext: TQXXA
```

```
testkey: 18 produced plaintext: SPWWZ
testkey: 19 produced plaintext: ROVVY
testkey: 20 produced plaintext: QNUUX
testkey: 21 produced plaintext: PMTTW
testkey: 22 produced plaintext: OLSSV
testkey: 23 produced plaintext: NKRRU
testkey: 24 produced plaintext: MJQQT
testkey: 25 produced plaintext: LIPPS
```

Вам может придти в голову, разглядывая эти результаты, что в некоторых шифрованных текстах сразу несколько вариантов расшифровки может иметь осмысленный вид, что приведет к неопределенности относительно правильного ключа. Предположим, например, что шифрованный текст очень краток: WZR. Если применить к нему программу **CaesarCipherBruteForceAttack**, то сразу три варианта расшифровки будут являться осмысленными словами: TWO (ключ k=3), RUM (ключ k=5) и LOG (ключ k=11). Тем не менее, при увеличении длины шифрованного текста вероятность случайного появления осмысленного сообщения падает очень быстро. Эта концепция была формализована математически и получила название *расстояние единственности* шифра. Расстояние единственности определяет, насколько быстро приближается к нулю вероятность появления случайно подошедших ключей при атаке методом «грубой силы». Также эта величина показывает, насколько велик должен быть шифрованный текст для осуществления успешной дешифровки (речь идет, разумеется, о гипотетическом случае неограниченных вычислительных возможностей). Расстояние единственности зависит от статистических характеристик открытого текста и от алгоритма шифрования.

У вас может возникнуть вопрос: а что понимается под «осмысленным результатом»? Что если открытое сообщение записано на одном из множества иностранных языков? Любые результаты могут быть осмысленными на одном из языков, и это делает атаку на шифр более трудной[1].

Что если открытый текст перед шифрованием был сжат при помощи программы-архиватора? Тогда никакой из вариантов расшифровки не будет выглядеть, как «осмысленный текст». Тем не менее, в современной криптографии предполагается, что атакующей стороне все эти детали известны. В реальной жизни открытый текст может выглядеть осмысленным или нет, но при анализе шифра предполагается, что злоумышленник в состоянии опознать правильно расшифрованный текст. Также предполагается, что злоумышленнику известны все детали реализации шифра. Например, если открытый текст перед шифрованием сжат, то предполагается,

[1] Во время Второй Мировой войны во всем мире существовало около 50 тысяч людей, говоривших на языке индейцев Навахо. В корпусе морской пехоты США около 400 индейцев Навахо были использованы в качестве «переводчиков» для секретных разговоров в эфире. Немцы и японцы раскрыли многие из шифров союзников, однако коммуникации через переводчиков Навахо никогда не были раскрыты! Это был настоящий успех. Однако в наше время подобный подход, скорее всего, потерпел бы неудачу. Теперь скрытность уже не обеспечивает надежной безопасности.

что атакующей стороне известен полный процесс шифрования, включая сжатие. Если использован редкий язык, то предполагается, что злоумышленник им владеет. В общем предполагается, что злоумышленнику известно все, кроме ключа.

Как вы можете заметить, при использовании шифра Цезаря многократное шифрование нисколько не увеличивает защищенность текста. Например, если вы зашифруете текст CAT ключом $k = 6$ и получите IGZ, а затем зашифруете результат ключом 4, получив MKD, окажется, что вы всего лишь зашифровали исходный текст ключом $k = 4 + 6 = 10$. Здесь можно привести аналогию со сжатием файлов: повторное сжатие уже сжатых данных ни к чему не приведет. Многие шифры обладают этим свойством, однако существуют шифры (например, тройной DES[1], описываемый в главе 3), которые эффективно используют многократное применение одного алгоритма с разными ключами.

Без лишних объяснений понятно, что шифр Цезаря никуда не годится по современным понятиям, но, с другой стороны, можно ли много требовать от криптосистемы, чей возраст составляет два тысячелетия? При наличии только шифрованного текста атака требует лишь простой программы, перебирающей ключи из очень небольшого пространства в 25 вариантов. Если имеется также и открытый текст, задача становится вовсе тривиальной, поскольку простое сравнение шифрованного и открытого текстов легко выявляет ключ. Далее в этой главе мы познакомимся с немного более совершенным классическим шифром, который называют шифром Виженера, а также с некоторыми современными шифрами. Но прежде давайте рассмотрим еще один пример простого классического шифра, который все же отличается значительно более широким пространством ключей.

ПРОСТОЙ ПОДСТАНОВОЧНЫЙ ШИФР

Простой подстановочный шифр в свое время не помог королеве Марии[2]. В подстановочном шифре каждый символ заменяется заранее определенным символом подстановочного алфавита, что относит его, как и шифр Цезаря, к моноалфавитным подстановочным шифрам. Это означает, что существует однозначное соответствие между символами в открытом тексте и символами в тексте зашифрованном. Такое свойство шифра делает его уязвимым для атаки, основанной *на частотном анализе*. Принцип

[1] Алгоритмы DES и «тройной DES» были разработаны для эффективной аппаратной реализации и широко используются в машинах ATM.

[2] Мария Стюарт, королева Шотландии, родилась в 1542 году. Вместе с несколькими дворянами-католиками она была обвинена в заговоре с целью убийства Елизаветы I, королевы Англии. Она была признана виновной и казнена в 1587 году. Из предосторожности в переписке со своими сообщниками Мария использовала подстановочный шифр. Однако несколько ее писем были все же прочитаны и стали решающей уликой против нее в результате атаки на шифр, основанной на частотном анализе. Интересно, что задолго до времен королевы Марии, арабский криптограф девятого века по имени Ал-Кинди уже изобрел атаку, основанную на частотном анализе. Этот пример наглядно показывает, как важно быть в курсе современного состояния криптографии!

этой атаки нетрудно понять, если вспомнить от том факте, что частота появления символов в тексте индивидуальна для каждого символа и, по большому счету, неизменна. Кроме того, некоторые сочетания символов, такие, например, как «the» или «qu» в английском языке, дают в шифрованном тексте последовательности, которые очень легко обнаружить статистически. На рисунке 2.4 изображен принцип работы простого подстановочного шифра. Ключ, использованный в этом примере, является вариантом из довольно большого множества ключей.

Сколько может существовать вариантов ключа k? Для 26-буквенного алфавита существует 26! (факториал 26[1]) возможных перестановок. Это означает, что пространство ключей шифра соответствует, примерно, 88-битному ключу, поскольку 26! = 403 291 461 126 605 635 584 000 000, а это значение лежит между 288 и 289. Это гораздо больше, чем пространство ключей шифра Цезаря (всего 25 ключей), однако, как мы убедимся, подстановочный шифр дает на удивленье мало преимуществ по сравнению с шифром Цезаря. Причина заключается в том, что шифр этот легко поддается атаке, основанной на частотном анализе. Из этого обстоятельства вытекает важный вывод: большой размер ключа является необходимым, но отнюдь не достаточным условием стойкости шифра.

Чтобы понять, откуда взялось столь большое число ключей, представьте, что от вас требуется создать произвольную перестановку английского алфавита. Вам придется совершить последовательность шагов, выбирая на каждом шаге очередную букву. На первом шаге вы должны будете выбрать одну из 26 букв алфавита для того, чтобы задать первую букву перестановки.

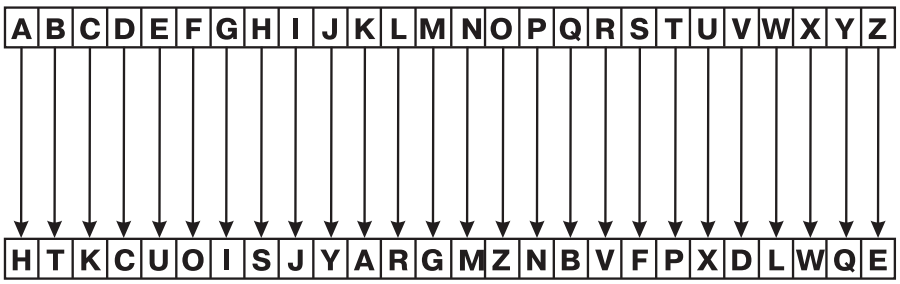


Рис. 2.4. Простой подстановочный шифр

```
Ключ:
  HTKCUOISJYARGMZNBVFPXDLWQE
Открытый текст:
  P = HELLO SIMPLE SUB CIPHER
Зашифрованный текст:
  C = SURRZ FJGNRU FXT KJNSUV
```

[1] В математической нотации факториал числа записывается, как число со знаком «!», например, «26!». Факториал числа 26 – это произведение всех целых чисел от 1 до 26, включая 26. Раздел математики, посвященный этим вопросам, называется комбинаторикой.

На втором шаге вы должны будете выбрать вторую букву уже из 25 оставшихся букв. На третьем шаге вы выбираете из оставшихся 24 букв и так далее. На последнем шаге у вас останется лишь одна буква. Таким образом, общее число возможных вариантов перестановки составит $26 \times 25 \times 24 \times \ldots \times 2 \times 1$, что в математической нотации записывается, как «26!». Вот определение простого подстановочного шифра, выраженное в математической нотации.

Определение: простой подстановочный шифр

Пусть K есть множество всех перестановок элементов в Z_{26}
$K = \{(k_1, k_2, k_3, \ldots, k_{26}):k_i \in Z_{26}\}$.

Выберем произвольный ключ k, где $k \in K$.

Выберем произвольный текст p, где $k \in K$
$p = (p_1, p_2, \ldots, p_m)$, где $p_i \in Z_{26}$ для $1 \le i \le m$.

Пусть результирующий шифрованный текст с есть:
$c = (c_1, c_2, \ldots, c_m)$, где $c_i \in Z_{26}$ для $1 \le i \le m$.

Далее определим функцию шифрования $E_k(p)$, как
$c_i = E_k(p_i) = k[p_i]$ для $1 \le i \le m$.

Функция дешифрования $D_k(c)$ определяется, как
$p_i = D_k(c_i) = E_k^{-1}(c_i)$ для $1 \le i \le m$.

Пример кода **SimpleSubCipher** иллюстрирует этот шифр. Код этот, опять же, очень прост и потому мы его здесь не приводим, но вы можете ознакомиться с ним в исходном тексте на языке C# в прилагающемся проекте. Сеанс работы программы приведен ниже:

```
Enter uppercase plaintext: HELLO SIMPLE SUB CIPHER
Enter 26 char key permutation: HTKCUOISJYARGMZNBVFPXDLWQE
Resulting ciphertext: SURRZ FJGNRU FXT KJNSUV
Recovered plaintext: HELLO SIMPLE SUB CIPHER
```

ЧАСТОТНЫЙ АНАЛИЗ: РАСКРЫТИЕ ПОДСТАНОВОЧНОГО ШИФРА

Для раскрытия простых подстановочных шифров обычно используют *атаку на основе частотного анализа*, в которой используются статистические методы. Здесь используется тот факт, что вероятность появления в открытом тексте определенных букв или сочетаний букв зависит от этих самых букв или сочетаний букв. Например, в английском языке буквы A и E встречаются гораздо чаще других букв. Пары букв TH, HE, SH и CH встречаются гораздо чаще других пар, а буква Q, фактически, может встретиться только в сочетании QU. Это неравномерное распределение вероятностей связано с тем, что английский язык (как и вообще все естественные языки) весьма избыточен. Эта избыточность играет важную роль: она уменьшает вероятность ошибок при передаче сообщений. Но с другой стороны избыточность облегчает задачу атакующей стороне.

Пример кода **SimpleSubCipherFrequencyAttack** демонстрирует принцип этой атаки. Пример программы достаточно сложен, однако основная идея, использованная здесь, проста. Если вы запустите программу, то увидите, что она загружает файл с образцом текста на английском языке. Затем она загружает файл с открытым текстом, генерирует случайный ключ, то есть случайную перестановку алфавита, и шифрует открытый текст. Далее на экран выводится зашифрованный текст, и отображаются статистические данные для шифрованного текста и текста-образца. Эта статистика позволяет вам сделать догадки о значении некоторых букв и буквосочетаний в шифрованном тексте. Фактический ключ и открытый текст также отображены на экране, но попробуйте расшифровать сообщение, не обращаясь к ключу и открытому тексту. Когда вы попытаетесь решить эту головоломку, то обнаружите, что ваш мыслительный процесс очень сходен с разгадыванием кроссвордов или с игрой в «Поле чудес». Программа интерактивна, и полагается она на вашу интуицию, просто предоставляя вам необходимую для догадок статистику. Эту программу можно усовершенствовать, сделав статистический анализ более сложным и применив эвристические методы – при этом процесс атаки можно будет полностью автоматизировать. Но в этом нет практического смысла, поскольку подобные шифры в наше время не используются. Тем не менее, принцип частотного анализа, использованный здесь, применяется при раскрытии гораздо более сложных и стойких шифров. Рисунок 2.5 иллюстрирует атаку на простой подстановочный шифр при помощи частотного анализа.

ШИФР ВИЖЕНЕРА

С изобретением телеграфа в середине 1800-х годов интерес к криптографии стал расти, поскольку ненадежность моноалфавитных подстановочных шифров была уже хорошо известна.

Решение, найденное в ту эпоху, заключалось в использовании шифра Виженера, который, как это ни странно, к тому моменту был известен уже на протяжении почти 300[1] лет. Этот шифр был известен во Франции, как *«нераскрываемый шифр»*, и это был действительно выдающийся шифр своего времени. Фактически, шифр Виженера оставался нераскрытым почти три столетия, с момента его изобретения в 1586[2] и до момента его

[1] Вас может удивить столь долгий срок, в течение которого шифр Виженера оставался невостребованным. Причина заключается в том, что шифр Виженера, несмотря на свою более высокую стойкость, труден в использовании «вручную». Ведь в те времена не было компьютеров, и не было даже механических шифрующих устройств.

[2] Ирония судьбы заключается в том, что шифр Виженера был изобретен в тот самый год, когда королева Мария была приговорена к смерти на основании улик, полученных в результате успешной расшифровки ее сообщений, которые основывались на слабом моноалфавитном подстановочном шифре. По странному совпадению Виженер предложил свой новый шифр королю французскому Генриху III, который был родственником королевы Марии. Если бы она использовала шифр Виженера, то, вероятно, не потеряла бы голову. Возможно, печальная судьба Марии побудила Генриха III внимательно прислушаться к идеям Виженера.

взлома в 1854, когда Чарльз Бэббидж[1] сумел, наконец, раскрыть его[2]. Интересно заметить, сколь недолгим оказался срок годности этого шифра, если отсчитывать его с момента широкого применения шифра в телеграфии. Здесь мы видим, как быстро криптографическая технология может стать неактуальной, если к методам ее взлома будет проявлен достаточно активный интерес. Тут поневоле забеспокоишься о безопасности современных зашифрованных данных. Как знать, быть может, какой-нибудь современный Бэббидж прямо в эту минуту изобретает новый способ взлома?

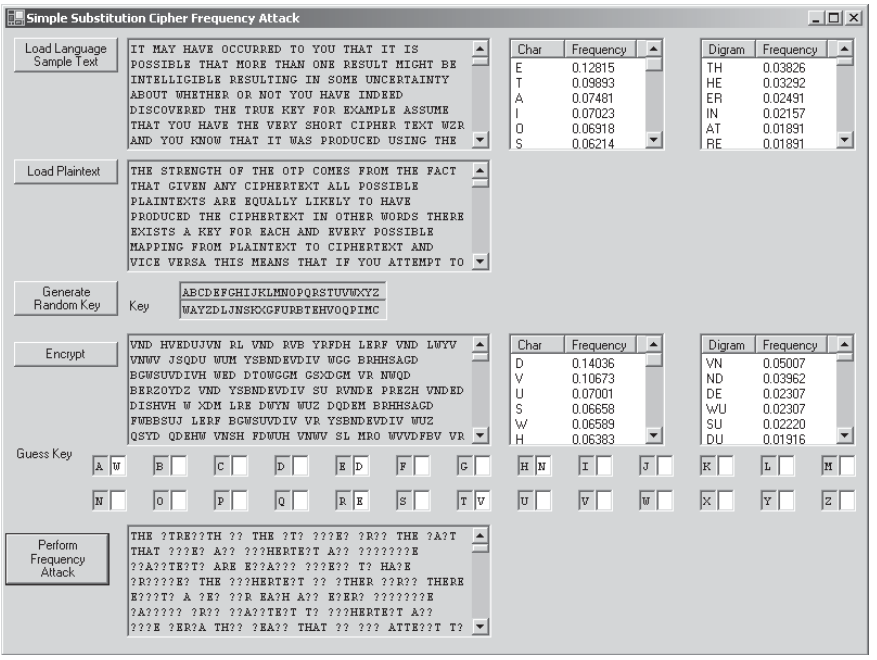


Рис. 2.5. Атака на простой подстановочный шифр при помощи частотного анализа

Шифр Виженера представляет собой *полиалфавитный* подстановочный шифр. Это означает, что для подстановки используются многие алфавиты, благодаря чему частоты символов в зашифрованном тексте не соответствуют частотам символов в тексте открытом. Следовательно, в отличие от моноалфавитных подстановочных шифров наподобие шифра Цезаря, шифр Виженера не поддается простому частотному анализу.

---

[1] Чарльз Бэббидж – легендарная фигура в мире компьютерных наук. Ему, кроме прочего, приписывается изобретение первого программируемого компьютера. Работа Бэббиджа по раскрытию шифра Виженера хранилась в тайне в течение многих десятилетий. Лишь в конце 1900-х годов датский историк Оле Франксен раскрыл этот секрет.

[2] Хотя шифр Виженера был раскрыт Бэббиджем, факт раскрытия был сохранен в секрете, и телеграфисты использовали этот шифр еще много десятилетий. Благодаря этому обстоятельству многие люди ошибочно считали, что их секретные сообщения в безопасности. Невольно напрашивается параллель с современностью: действительно ли надежны шифры, которые мы сейчас считаем надежными?

В сущности шифр Виженера меняет соответствие между открытыми и зашифрованными символами для каждого очередного символа. Он основывается на таблице, вид которой приведен на рисунке 2.6. Каждая строка этой таблицы – не что иное, как шифр Цезаря, сдвинутый на число позиций, соответствующее позиции в строке. Строка A сдвинута на 0 позиций, строка B – на 1 и так далее.

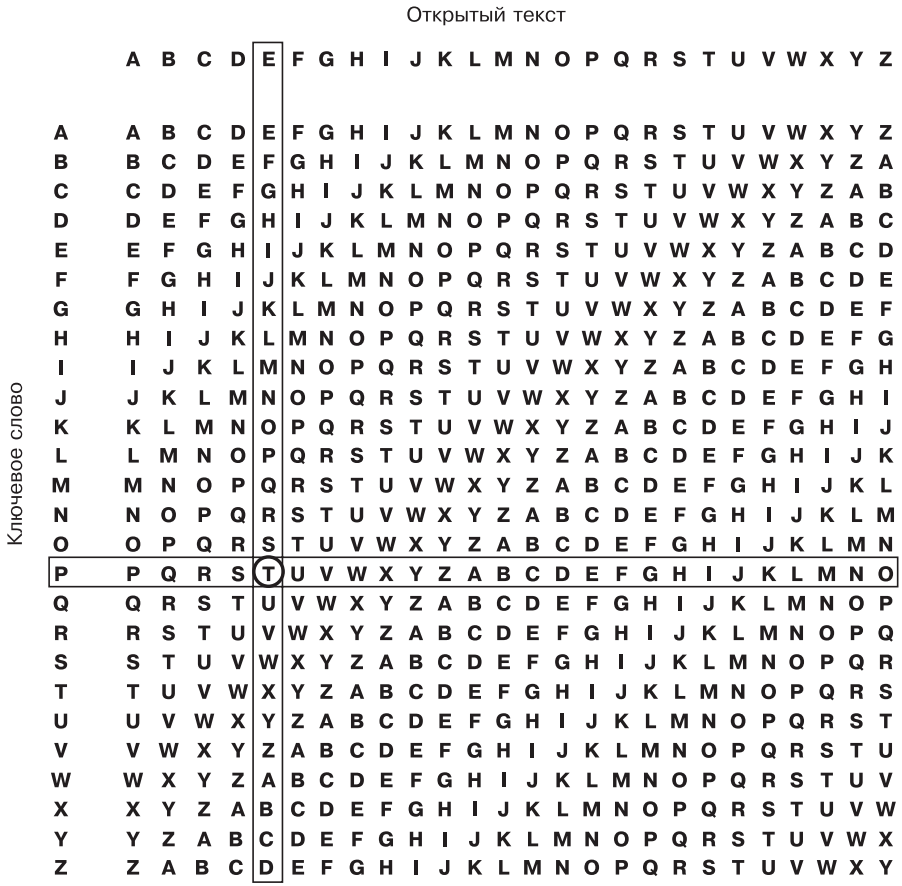


Рис. 2.6. Шифр Виженера

В шифре Виженера такая таблица используется в сочетании с ключевым словом, при помощи которого шифруется текст. Предположим, например, что нам требуется зашифровать фразу GOD IS ON OUR SIDE LONG LIVE THE KING при помощи ключа PROPAGANDA. Для шифрования вы повторяете ключ столько раз, сколько необходимо для достижения длины открытого текста, просто записывая символы под символами открытого текста. Затем вы получаете поочередно каждый символ зашифрованного текста, беря столбец, определенный по символу открытого текста, и пересекая его со строкой, определенной по соответствующему

символу ключа. Например, первый символ открытого текста *G* в сочетании с первым символом ключа *P*, как видно из рисунка 2.6, дадут столбец и строку, пересекающиеся на символе *V*, каковой и будет первым символом шифрованного текста. Все последующие символы определяются аналогичным образом.

```
Открытый текст  : GOD IS ON OUR SIDE LONG LIVE THE KING
Ключ            : PRO PA GA NDA PROP AGAN DAPR OPA GAND
Зашифрованный текст: VFR XS UN BXR HZRT LUNT OIKV HWE QIAJ
```

Для расшифровки текста запишите символы повторяющегося ключа под символами шифрованного текста. Каждый очередной символ открытого текста восстанавливается взятием строки, определенной символом ключа, и нахождением в этой строке столбца, в котором содержится соответствующий символ шифрованного текста. Затем, поднявшись к верхушке найденного столбца, вы получаете символ открытого текста.

```
Зашифрованный текст: VFR XS UN BXR HZRT LUNT OIKV HWE QIAJ
Ключ            : PRO PA GA NDA PROP AGAN DAPR OPA GAND
Открытый текст  : GOD IS ON OUR SIDE LONG LIVE THE KING
```

Процедура довольно утомительная, но ее все же можно проделать при помощи карандаша и бумаги. Возможно, вы предпочтете воспользоваться программой **VigenereCipher**, и заодно изучите ее исходный код. Вы убедитесь, что программа дает точно такой же результат, что получен выше на бумаге.

Вот определение шифра Виженера, сделанное в математической нотации.

Определение: шифр Виженера

Пусть даны произвольный ключ k, где

$k = (k_1, k_2, k_3, \ldots, k_n)$, где $k_i \in Z_{26}$ для $1 \le i \le n$,

и произвольный открытый текст p, где

$p = (p_1, p_2, \ldots, p_m)$, где $p_j \in Z_{26}$ для $1 \le j \le m$.

Пусть зашифрованный текст c представлен, как

$c = (c_1, c_2, \ldots, c_m)$, где $c_j \in Z_{26}$ для $1 \le j \le m$.

Тогда мы определим функцию шифрования E_{k_i}, как

$c_j = E_{k_i}(p_j)$, где $E_{k_i}(p) : p_j \rightarrow = p_j + k_i (\text{mod } 26)$,

а функция дешифрования $D_k(c)$ определяется, как

$p_j = D_{k_i}(c_j)$, где $D_{k_i}(c) : c_j \rightarrow c_j - k_i (\text{mod } 26)$.

Если вы запустите пример программы **VigenereCipher**, то сможете повторить наш пример:

```
Enter plaintext: GOD IS ON OUR SIDE LONG LIVE THE KING
Enter keyword: PROPAGANDA
Resulting ciphertext: VFR XS UN BXR HZRT LUNT OIKV HWE QIAJ
Recovered plaintext: GOD IS ON OUR SIDE LONG LIVE THE KING
```

АТАКА БЭББИДЖА: РАСКРЫТИЕ ШИФРА ВИЖЕНЕРА

Поскольку в полиалфавитном подстановочном шифре используется множественные подстановки, то нет отношения один-к-одному между символами открытого и зашифрованного текста. В результате между каждым символом в открытом тексте и символом в зашифрованном тексте существует отношение один-ко-многим. Хотя частотный анализ сам по себе не поможет раскрыть такой шифр, Бэббидж обнаружил, что сочетание анализа ключа с частотным анализом текста способно привести к успеху.

Как выполняется атака Бэббиджа? Прежде всего производится анализ ключа с целью выяснить длину ключа. В основном это сводится к поиску повторяющихся образцов в тексте. Для этого вы сдвигаете текст относительно самого себя на один символ и подсчитываете число совпавших символов. Затем должен следовать следующий сдвиг и новый подсчет. Когда эта процедура будет повторена много раз, вы запоминаете величину сдвига, давшую максимальное число совпадений. Случайный сдвиг дает небольшое число совпадений, но сдвиг на величину, кратную длине ключа приведет число совпадений к максимуму. Этот факт вытекает из того обстоятельства, что некоторые символы встречаются чаще других, и, кроме того, ключ повторен в тексте много раз с определенным интервалом. Поскольку символ совпадает с копией самого себя, зашифрованной тем же самым символом ключа, число совпадений будет немного увеличиваться при всех сдвигах, величина которых кратна длине ключа. Очевидно, что для выполнения этой процедуры требуется текст достаточно большого размера, поскольку расстояние единственности для этого шифра гораздо больше, чем для моноалфавитных подстановочных шифров.

После того как длина ключа будет, предположительно, определена, следующий шаг будет состоять в частотном анализе. При этом вы разделяете символы шифрованного текста по группам, соответствующим символам ключа, которые использовались для шифрования в каждой из групп, основываясь при этом на предположении о длине ключа. С каждой группой символов вы можете теперь обращаться, как с текстом, зашифрованным простым сдвиговым шифром наподобие шифра Цезаря, используя атаку методом «грубой силы» или частотный анализ. После того как все группы по отдельности будут расшифрованы, их можно собрать вместе и получить расшифрованный текст. Когда вы видите эту атаку в действии, то понимаете, что она не так уж сложна, и можно лишь удивиться тому, что данный шифр оставался непобежденным целых 300 лет. Другие, более сложные полиалфавитные шифры также уязвимы для подобной атаки, если длина ключа не слишком велика.

ЕДИНСТВЕННЫЙ НЕУЯЗВИМЫЙ ШИФР: ОДНОРАЗОВЫЙ ШИФРОВАЛЬНЫЙ БЛОКНОТ

Существует только один шифр, который теоретически безопасен на 100%. Это так называемый «шифровальный блокнот» или «одноразовый

блокнот» (One-Time Pad – OTP)[1]. Для достижения идеальной безопасности в методе «одноразового блокнота» применяются весьма строгие правила: ключи генерируются на основе настоящих случайных чисел, ключи сохраняются в строгом секрете и ключи никогда не используются повторно. В отличие от других шифров метод «одноразового блокнота» (OTP) так же, как и его математические эквиваленты, является единственной системой, неуязвимой для взлома. Метод OTP позволяет достичь идеальной безопасности, однако практическое его использование затруднено проблемой ключей. По этой причине метод «одноразового блокнота» применяют лишь в редких случаях, когда достижение абсолютной секретности важнее всего прочего, и когда требуемая пропускная способность невелика. Такие ситуации достаточно редки, их можно встретить, разве что, в военной области, в дипломатии и в шпионаже.

Давайте разберемся в этом методе. Для использования метода OTP необходимо сгенерировать новый, одноразовый случайный ключ, длина которого равна длине шифруемого сообщения. Затем байты сообщения комбинируются с байтами ключа побитно на основе обратимой логической операции «исключающее ИЛИ», в результате чего получается шифрованный текст. Операцию «исключающее ИЛИ» обозначают аббревиатурой XOR или математическим символом $\oplus$. В языке C# эта операция представлена символом ^. После этого преобразования ключ повторно для шифрования более не используется. Здесь необходимо понимать, что операция XOR обратима относительно себя самой, то есть $P = ((P \oplus K) \oplus K)$ для любых P и K, что делает ее очень удобной. Выражаясь проще, если вы примените операцию XOR к открытому тексту и ключу, а затем снова примените эту операцию к тому же самому ключу и результирующему шифрованному тексту, то вновь получите открытый текст. Определение операции XOR приведено в таблице 2.1.

Таблица 2.1. Операция XOR

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

[1] Клод Шеннон, изобретатель теории информации, доказал, что совершенный алгоритм шифрования требует, кроме прочего, чтобы длина ключа шифра была не меньше длины сообщения. Метод «шифровального блокнота» удовлетворяет этому требованию. Тем не менее, для большинства практических применений абсолютной секретности достичь трудно, и потому от стремления к абсолютной секретности отказываются в пользу секретности достаточной. В результате большинство криптоалгоритмов используют ключи, длина которых гораздо меньше длины сообщения, и потому идеальной теоретической стойкостью они обладать не могут.

Операция XOR – не единственная операция, которую можно использовать в методе «шифровального блокнота». Собственно говоря, подойдет любое обратимое преобразование. Рассмотрим, например, преобразование в шифре Цезаря[1], заключающееся в сложении ключевого байта с байтом открытого текста по модулю 256. Это в сущности то же самое преобразование, что и в шифре Цезаря, однако используется не один и тот же ключевой байт для всех байтов открытого текста: для каждого очередного байта текста генерируется новый случайный байт ключа. Небольшое преимущество операции XOR заключается в том, что для обращения преобразования применяется та же самая операция. В математическом же смысле никакой существенной разницы между этими двумя методами преобразования нет.

На рисунке 2.7 иллюстрируется метод «одноразового шифровального блокнота» (OTP). Слово *Elvis* кодируется в ASCII как 69, 108, 118, 105, 115 (в десятичной системе). Случайный ключ здесь – последовательность 169, 207, 195, 134, 172. В результате операции XOR получается 236, 163, 181, 239, 223. В правой части рисунка вы видите двоичное представление первого байта текста, ключа и шифрованного текста. Математически метод OTP описывается следующим образом.

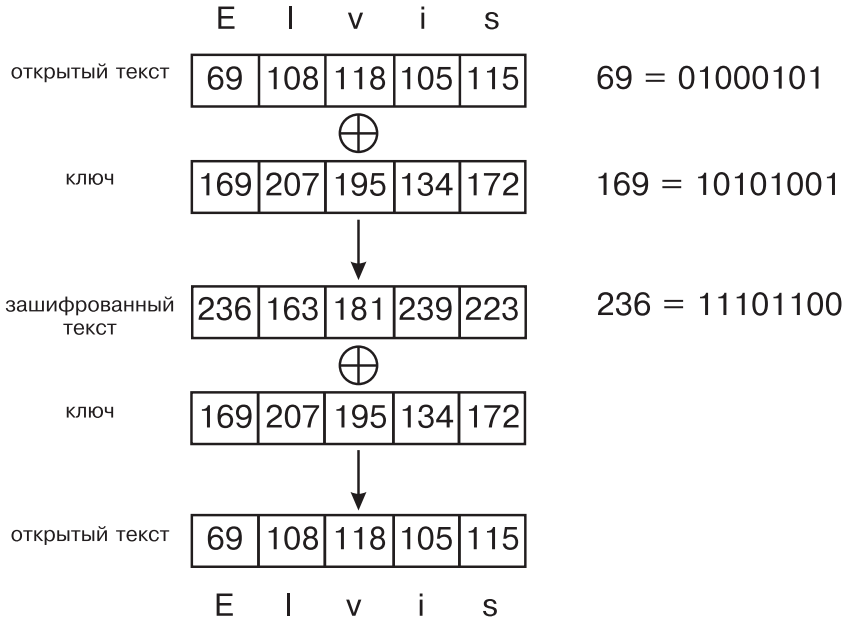


Рис. 2.7. Шифрование методом «одноразового блокнота» (OTP)

[1] В шифре Цезаря используется сложение по модулю 26, но мы расширяем здесь этот принцип до сложения по модулю 256, поскольку нас интересует шифрование байтов более, чем шифрование букв английского алфавита.

Определение: шифр OTP («одноразовый блокнот»)

Пусть даны произвольный ключ k, где
$k = (k_1, k_2, k_3, ..., k_m)$, где $k_j \in \{0, 1\}$ для $1 \le j \le m$,
и произвольный открытый текст p, где
$p = (p_1, p_2, ..., p_m)$, где $p_j \in \{0, 1\}$ для $1 \le j \le m$.

Тогда мы определим функцию шифрования $E_k(p)$ как
$E_k(p_i) = p_i \oplus k_i$ для $1 \le i \le m$,
а функция дешифрования $D_k(c)$ определяется, как
$D_k(c_i) = c_i \oplus k_i$ для $1 \le i \le m$,
тогда
$c_i = E_k(p_i)$ и $p_i = D_k(c_i)$.

Все вышесказанное реализовано в примере кода **OTP_XOR**. Метод **EncryptDecrypt** выполняет шифрование и дешифрование, поскольку операция XOR обратима относительно самой себя.

```
static void Main(string[] args)
{
    Console.Write("Enter plaintext: ");
    StringBuilder plaintext =
        new StringBuilder(Console.ReadLine());
    RandomNumberGenerator rng =
        new RNGCryptoServiceProvider();
    byte[] key = new Byte[100];
    rng.GetBytes(key);

    StringBuilder ciphertext =
        EncryptDecrypt(plaintext, key);
    Console.WriteLine("ciphertext: {0}",
        ciphertext);

    StringBuilder recoveredplaintext =
        EncryptDecrypt(ciphertext, key);
    Console.WriteLine("recovered plaintext: {0}",
        recoveredplaintext);
}
static StringBuilder EncryptDecrypt(
    StringBuilder data_in,
    byte [] key)
{
    StringBuilder data_oput =
        new StringBuilder(data_in.ToString());
    for (int index=0; index<DATA_IN.LENGTH; INDEX++)
    {
        int character =
            ((int)data_in[index] ^ key[index]);
        data_oput[index] = (char)character;
    }
    return data_oput;
}
```

Типичный сеанс работы кода **OTP_XOR** приведен ниже. При каждом запуске программы вы будете видеть новый шифрованный текст, поскольку ключ OTP каждый раз генерируется заново.

```
Enter plaintext: hello
ciphertext: IuE3c
recovered plaintext: hello
```

Сила метода OTP проистекает из того факта, что при любом заданном шифрованном тексте любые варианты исходного открытого текста равновероятны. Иными словами, для любого возможного варианта открытого текста найдется ключ, который в результате применения произведет этот шифрованный текст. Это означает, что если вы попытаетесь найти ключ методом «грубой силы», то есть просто перебирая все возможные ключи, то получите в результате все возможные варианты открытого текста. Здесь будет также и истинный открытый текст, но вместе с ним – все возможные варианты осмысленного текста, а это ничего вам не даст. Атака методом «грубой силы» на шифр OTP бесполезна и неуместна, вот, что вам следует помнить о методе «одноразового блокнота»! Надежда раскрыть шифр OTP возникает лишь в ситуации, когда ключ был использован несколько раз, для шифрования нескольких сообщений, или когда для генерации псевдослучайного ключа был использован алгоритм, дающий предсказуемую последовательность, или же когда вам удастся добыть ключ какими-то иными, не криптоаналитическими методами.

Вы можете убедиться в этом факте, рассмотрев следующую аргументацию. Предположим, ваш открытый текст состоит из единственного байта. Тогда, согласно определению OTP, ключ также должен состоять из одного байта. Выберите произвольный открытый текст из 256 возможных 8-битовых значений. Если вы затем выполните операцию XOR над этим значением и над 256 возможными ключами, то получите 256 отличающихся шифрованных текстов размером в один байт. Все варианты шифрованного сообщения будут отличаться друг от друга, поскольку операция XOR не вносит потерь с точки зрения теории информации. Отсюда следует, что, выбрав подходящий ключ, вы можете любой 8-битовый открытый текст преобразовать в любой 8-битовый шифрованный текст. Та же аргументация работает и в обратном направлении, и выбором подходящего ключа вы можете расшифровать любой байт в любой 8-битовый открытый текст. Поэтому способа определения, который из вариантов расшифровки является верным, не существует.

Однако в методе OTP есть и подводные камни, и если он реализован не надлежащим образом, то может оказаться уязвимым. Прежде всего, при повторном использовании ключа надежность перестает быть 100-процентной. Во-вторых, надежность метода зависит от технологии генерации случайных ключей. Детерминистические алгоритмы в принципе не могут сгенерировать настоящие случайные числа. Даже самые лучшие программные генераторы случайных чисел производят все же только псевдослучайные последовательности. И третье обстоятельство: длина ключа должна быть достаточно большой, а ключ должен генерироваться заново для каждого нового сообщения. Поскольку длина ключа не должна

быть меньше длины сообщения, а сообщения могут иметь произвольную длину, то способ генерации и распределения ключей вырастает в серьезную проблему. Распределение ключей подразумевает, что обе общающиеся стороны должны иметь копию одного и того же ключа. Возникает классический парадокс, как в романе «Уловка 22»: если у вас есть достаточно безопасный канал связи, по которому вы можете передать секретный ключ, то зачем вам вообще шифровать сообщение? А если у вас такого безопасного канала нет, то, как вы передадите ключ?

Что ж, метод «одноразового блокнота» бывает полезен в ситуациях, когда секретный канал существует только ограниченное время, в течение которого вы можете быстро передать то количество готовых ключей, которое, по вашему мнению, вам понадобится в последующем. Затем ключи просто хранятся каким-то безопасным способом[1]. Когда роскоши безопасного секретного канала связи больше не существует, вы шифруете сообщения, расходуя имеющиеся ключи. Единственное, о чем тут остается позаботиться, это порядок использования ключей: для того чтобы каждый раз отправитель и получатель пользовались одним и тем же ключом, они должны расходовать их в одном порядке. Во времена Второй Мировой войны нацисты использовали метод «одноразового блокнота» в дипломатической переписке[2]. Когда нацистские дипломаты возвращались в Берлин, они получали большой набор OTP-ключей, которые впоследствии, выехав за границу, использовали для коммуникаций по телефонным, телеграфным и радио-каналам. Англичане также широко использовали OTP-шифр[3].

Шифр «Enigma»

Мы рассмотрели несколько классических шифров, познакомившись на их примере с основными концепциями криптографии. Существует много других, не менее интересных классических шифров, но мы не рассматриваем их здесь, поскольку это не продвинет нас к нашей главной цели – изучению вопросов безопасности и секретности в мире .NET. Тем не менее, шифр «Enigma», использованный немцами во Второй Мировой войне, в историческом смысле представляет огромный интерес, поскольку являет собой переход от классических шифров к современной криптографии. Мы не обсуждаем в этой книге шифр «Enigma», однако в Internet доступно множество Java-апплетов, эмулирующих этот шифр. Многие из них снабжены исходным кодом, чтение которого позволит вам понять устройство шифра. Для того чтобы найти эти примеры, откройте поисковую машину *www.google.com* и выполните поиск по строке «Enigma java applets».

[1] Речь идет о физически безопасном хранении (дискета в сейфе) или (шутка) о записи файла ключа в зашифрованной файловой системе.

[2] Собственно, не только нацисты, но большинство правительств стран, основных участников войны, пользовались шифром OTP.

[3] В 1942 году нацистам удалось раскрыть шифр, который использовал британский флот. Это позволило германским подлодкам топить британские суда в феноменальных количествах. В конце 1942 года британский флот перешел на использование шифра OTP, и потери флота внезапно сократились во много раз.

Рабочий фактор атаки методом «грубой силы»

Объем вычислений, необходимый для достижения некоторой цели, называют *рабочим фактором*. Определение рабочего фактора раскрытия шифра – важнейший аспект анализа шифра на стойкость. Далее мы кратко рассмотрим, как вычисляется рабочий фактор атаки на шифр методом «грубой силы». Как мы убедились ранее, шифр Цезаря крайне слаб, поскольку его пространство ключей очень мало – нет нужды вспоминать о других его слабостях. Всего 25 возможных ключей – это соответствует, в терминах теории информации, всего лишь 6-битному ключу, поскольку 2 в степени 6 равняется 32, чего достаточно для представления 25 ключей. Однако существуют, как мы знаем, шифры с гораздо более объемным пространством ключей. Хорошая новость для Алисы и Боба (и плохая новость для Евы) заключается в том, что рабочий фактор атаки методом «грубой силы» с ростом размера ключа растет экспоненциально. Теоретически, для n-битного ключа рабочий фактор есть $W \propto 2^n$. Конечно, если Еве удастся найти метод атаки существенно более эффективный, чем метод «грубой силы», то положение может измениться.

Таким образом, более мощные шифры требуют для атаки на себя более изощренных методов, нежели метод «грубой силы». Причина заключается в том, что при достаточно большом размере ключа прямой перебор всех возможных ключей займет неприемлемо большое время. Например, если в шифре используется 128-битный ключ, то число всех возможных ключей будет равно 2 в степени 128. Это число чудовищно велико! В десятичной записи потребуется 39 знаков, чтобы записать это число. Запомните, что большой размер ключа является необходимым, но отнюдь не достаточным условием надежности шифра.

$$2^{128}=340\ 282\ 366\ 920\ 938\ 463\ 463\ 374\ 607\ 431\ 768\ 211\ 456$$

Если отбросить несущественные детали такие, как високосные годы, то можно принять, что число секунд в году составляет

$$60 \times 60 \times 24 \times 356 = 31\ 536\ 000.$$

Если мы зададимся крайне заниженной оценкой числа тактов процессора, необходимых для проверки одного ключа, и примем его равным 100, то на частоте 1 ГГц компьютер сможет проверить за одну секунду 100 000 000 ключей. Это означает, что за год будет проверено 315 360 000 000 000 ключей. Это очень большое число, однако, для 2^{128} ключей проверка займет 1 079 028 307 080 601 418 897 052 лет. Возраст Вселенной оценивают величиной в 15 миллиардов лет, поэтому наша атака методом «грубой силы» потребует промежутка времени, в 71 935 220 472 040 раз превышающего возраст Вселенной. Современные полупроводниковые технологии уже вплотную подошли к теоретическим пределам скорости переключения, определяемым квантовой механикой, и потому мы не можем надеяться на радикальное повышение тактовых частот.

Попытка решить эту проблему, распределив выполнение атаки на множество компьютеров, не намного облегчит задачу. Для того чтобы успешно завершить атаку в течение одного года, даже при условии идеальной масштабируемости, нам потребуется 1 079 028 307 080 601 418 897 052 процессоров. Даже если не принимать во внимание такие факторы, как физический объем, стоимость такого числа компьютеров намного превышает оборот всей экономики планеты Земля. Предположив, что каждый процессор занимает всего один кубический сантиметр, наш теоретический суперкомпьютер будет занимать объем, равный 5 процентам от объема Луны. Но это только процессоры! Память и соединительные шины, смонтированные на трехмерной материнской плате (суперплате – «матери» всех «матерей»), займут еще 20 процентов объема Луны. Эти размеры достаточны для возникновения гравитационных сил, действующих от периферии к центру устройства. Дальнейшие вопросы становятся абсурдными: сколько это стоит? Сколько времени займет постройка такого компьютера? Как его программировать? Сколько электроэнергии он будет потреблять? Очевидно, все эти вопросы попросту глупы, но они ясно иллюстрируют то обстоятельство, что по мере роста размера ключа потенциал для атаки методом «грубой силы» иссякает очень быстро.

Если вам любопытно посмотреть на программу, осуществляющую столь невероятные вычисления, изучите пример программы **BigIntegerFun**. Текущая версия .NET Framework еще не поддерживает многопроцессорную арифметику, поэтому данный пример написан на GnuMP[1], открытой и бесплатной библиотеке C, поддерживающей арифметику целых чисел произвольного размера и доступной на сайте *www.gnu.org*.

Арифметика произвольной точности

Криптография вообще и в особенности асимметричная криптография, описываемая в главе 4, очень часто и интенсивно использует арифметику произвольной точности. Вероятно вам никогда не потребуется иметь дело с подобной арифметикой напрямую, поскольку в .NET Framework реализовано большинство всех необходимых криптографических функций. Тем не менее, может встретиться ситуация, когда вам потребуется явным образом иметь дело с математической стороной криптографии, и для этого необходимо будет использовать специализированную математическую библиотеку, такую, как GnuMP. Такое может случиться, если, например, вы захотите поэкспериментировать с собственным асимметричным алгоритмом. На этот случай в приложении C вы найдете инструкции по загрузке, установке и применению библиотеки GnuMP в ваших программах на платформе .NET.

[1] Инструкции по загрузке и установке этой библиотеки приведены в приложении C.

Стеганография

Стеганографией называют искусство сокрытия информации таким образом, что сам факт сокрытия остается скрытым. В техническом смысле стеганографию не рассматривают в качестве разновидности криптографии, но все же она может эффективно использоваться для обеспечения секретности коммуникаций. Пример **Steganography** представляет собой простую программу, иллюстрирующую типичный прием стеганографии, в котором используется графическое изображение.

Мы не будем рассматривать здесь устройство этой программы, поскольку оно не имеет настоящего отношения к криптографии, но вы можете изучить прилагающийся исходный код. Здесь не используются криптографические классы .NET Security Framework, и потому при достаточном знании синтаксиса C# этот код нетрудно понять. Единственное, что нуждается здесь в пояснениях, это манипуляции с битами. Каждый 8-битовый байт исходного изображения участвует в изображении одного из пикселей. Для каждого пикселя определены три байта, представляющие красную, зеленую и синюю компоненты цвета пикселя. Каждый байт секретного сообщения разделяется на три поля размером 3, 3 и 2 бита. Этими 3-х и 2-х битовыми полями затем замещаются младшие, наименее значимые разряды трех «цветовых» байтов соответствующего пикселя.

Исходное изображение и картинка с внедренным секретным сообщением приведены на рисунке 2.8. Можете вы заметить разницу? Разница практически неразличима для глаза, поскольку сообщение заключено в наименее значимых разрядах тех байт, что представляют собой несколько первых пикселей изображения. Но если вы заполните скрытым сообщением всю картинку, то и тогда, скорее всего, отличие будет малозаметным.

Эта техника основывается на том факте, что младшие разряды в «цветовых» байтах очень слабо влияют на вид конкретного пикселя, и, соответственно, на общий вид изображения. Если проделать то же самое со старшими разрядами байтов, то изображение изменится разительно.

Значения младших разрядов в большинстве изображений носят случайный характер, поэтому скрытое сообщение в них можно выявить статистическими методами. То есть атакующая сторона может проанализировать ваше изображение на степень случайности значений младших разрядов и обнаружить закономерности, внесенные наличием сообщения. Чтобы этого избежать, вы можете преобразовать сообщение в последовательность, которая в статистическом смысле будет случайным «шумом». Например, вы можете использовать компрессию, которая не только улучшает статистическую «скрытность» сообщения, но уменьшает его размер. Кроме того, вы можете зашифровать сообщение перед внедрением его в изображение, улучшив тем самым его «случайность» и еще более затруднив его обнаружение статистическими методами.

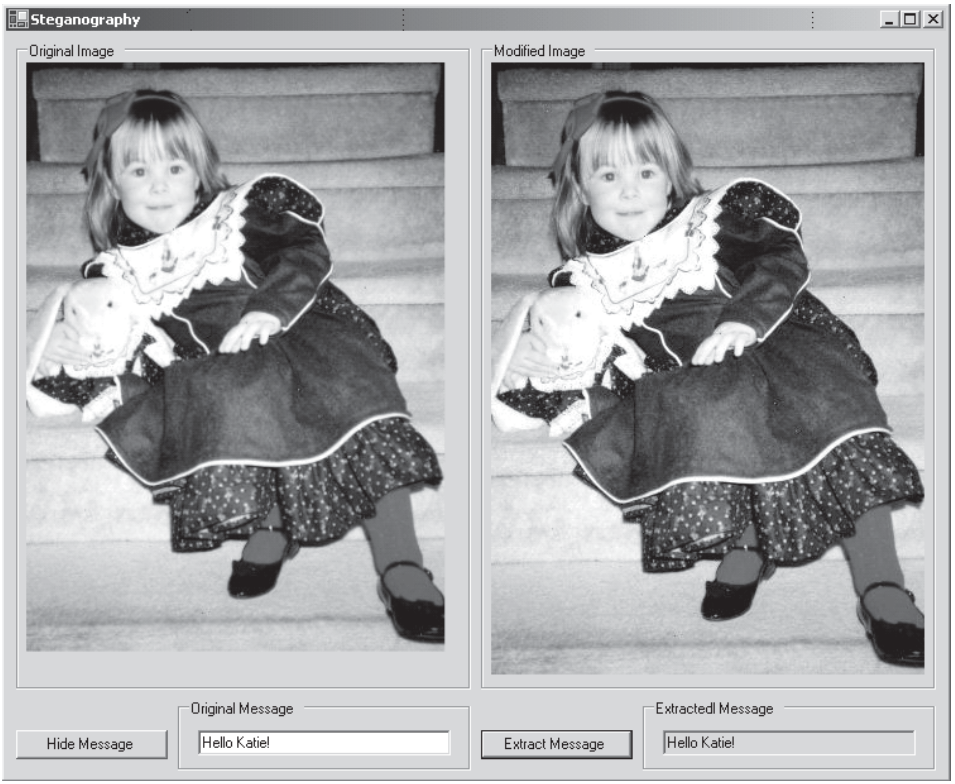


Рис. 2.8. Простой пример стеганографии

Вообще, в методах стеганографии все ограничивается лишь вашим воображением. Секретные сообщения можно прятать не только в изображениях, но и в аудиофайлах, в сообщениях электронной почты, в голосовой почте и так далее. Файлы, несущие в себе скрытые сообщения, могут распространяться абсолютно анонимно через группы новостей, Web-сайты, электронную почту или сетевые сервисы наподобие Napster.

Современные шифры

Современные шифры не оперируют алфавитом, состоящим из букв от A до Z, но, с другой стороны, и Цезарь не был знаком с такими понятиями, как бит или байт! В современных шифрах используются числа, причем иногда очень большие числа, которые представляют ключи или фрагменты открытого текста. Для шифрования здесь используются арифметические операции над числами, но это не всегда будет арифметика, которая знакома вам по средней школе.

Криптография и .NET Framework

Библиотека классов .NET Framework предусматривает пространство имен **System.Security.Cryptography**, в котором поддержано большинство основных симметричных и асимметричных шифров, а также несколько хеш-алгоритмов и генератор случайных чисел криптографического качества. Эта криптографическая основа может быть расширена, поскольку в форме «поставщиков услуг криптографии» сюда можно подключить модули сторонних производителей. Пространство имен **System.Security.Cryptography.XML** реализует стандарт W3C на цифровую подпись XML-объектов, а пространство имен **System.Security.Cryptography.X509Certificates** обеспечивает поддержку манипуляций с публичными сертификатами. Вот несколько базовых стандартов, реализованных в библиотеке классов **System.Security.Cryptography**. Мы изучим их более подробно в главах 3, 4 и 5.

- DES: Digital Encryption Standard (Стандарт цифрового шифрования – симметричный блочный шифр).
- 3DES: Тройной DES (симметричный блочный шифр, более сильная альтернатива DES).
- Rijndael: AES[1] (симметричный блочный шифр).
- RC2: Шифр, изобретенный Рональдом Ривестом (Ronald Rivest) – симметричный потоковый шифр.
- RSA: Шифр, созданный Ривестом, Шамиром и Адлеманом (Rivest, Shamir, Adleman) – асимметричный алгоритм, используемый, одновременно, как шифр, и как электронная подпись.
- MD5: хеш-алгоритм, алгоритм для создания электронного «отпечатка пальца», разработанный Ривестом.
- SHA-1, SHA-256, SHA-384, SHA-512: алгоритмы хеширования, разработанные институтом NIST (National Institute of Standards and Technology – Национальный институт стандартов и технологии) совместно с NSA, для использования совместно DSS (Digital Signature Standard – Стандарт электронной подписи).
- Генератор псевдослучайный чисел (PRNG).
- XML Signatures: цифровые подписи для XML-данных.
- X.509: стандарт публичных сертификатов.

Симметричная криптография

Подобно тому, как развитие телеграфии в 1800-х годах пробудило интерес к криптографии, развитие цифровых компьютеров (которые и изобретены то были, в сущности, в ответ на нужды криптографов времен Второй Мировой войны) породило огромный интерес к криптографическим

[1] Правительство США выбрало шифр Rijndael, как стандарт шифрования (стандарт AES – Advanced Encryption Standard, Усовершенствованный стандарт шифрования) в октябре 2000 г., заменив им стандарт DES.

алгоритмам. В результате на свет появились самые быстрые и сильные[1] методы криптографии – симметричные блочные шифры. Эти «плоды прогресса» явились к нам в виде DES, тройного DES, AES и ряда других стандартов.

Хорст Файстель (Horst Feistel), работавший в IBM в начале 1970-х, разработал симметричный блочный шифр, который впоследствии превратился в стандарт шифрования данных DES[2]. В стандарте DES одни и те же алгоритмы и 56-битовые ключи используются, как для шифрования, так и для дешифрования. Основу алгоритма DES составляют 16 повторяющихся циклов, каждый из которых включает в себя подстановку с последующей перестановкой в пределах 64-битового блока данных. Цель перестановок состоит в «перемешивании» и диффузии исходных данных, но все преобразования, разумеется, обратимы. Подстановка еще более усложняет зависимости между исходным и шифрованным текстами. В результате все статистические неоднородности исходного текста оказываются в шифрованном тексте сглаженными.

Поскольку входные данные, вообще говоря, не обязательно кратны размеру 64-битового блока, их разбивают на 64-битовые блоки, дополняя при необходимости незначащими байтами. Каждый цикл алгоритма DES состоит в разделении 64-битового блока входных данных на два 32-битовых подблока. Правая половина шифруется при помощи специальной функции[3], при этом используется подмножество ключа, уникальное для данного цикла. Затем зашифрованная правая половина объединяется с левой половиной операцией XOR, образуя в результате новую правую половину для следующего цикла. Прежняя правая половина в очередном цикле заменяется левой. Схематически одиночный цикл алгоритма DES изображен на рисунке 2.9. Мы более подробно рассмотрим алгоритм DES и другие симметричные блочные шифры в главе 3.

На протяжении 1990-х годов алгоритм DES, вероятно, исчерпал свою полезность. Для того чтобы сохранить обратную совместимость с имеющимся оборудованием и программным обеспечением, многие организации приняли для использования «тройной DES», алгоритм, сводящийся к трехкратному применению DES с тремя разными ключами. Тем не менее, недавно DES был официально заменен новым стандартом, симметричным блочным шифром под названием Rijndael в форме стандарта AES[4]. Шифр Rijndael (произносится «рейн дол» или «райн дал») был разработан бельгийскими криптографами Джоан Деймен (Joan Daemen) и Винсентом Райменом (Vincent Rijmen).

---

[1] Вы можете вспомнить метод «одноразового блокнота», теоретически, вообще несокрушимый шифр. Тем не менее, шифр OTP неприменим в большинстве реальных ситуаций из-за большого размера ключей и трудностей с их безопасным хранением и транспортировкой. Поэтому практически самыми сильными из доступных для реального применения шифров следует считать современные симметричные блочные шифры.

[2] DES был принят в качестве стандарта в NIST и опубликован в FIPS PUB 46 в 1977-м году.

[3] Эта функция описывается в главе 3.

[4] Более подробные сведения о стандарте AES можно найти по ссылке *http://csrc.nist.gov/encryption/aes/*.

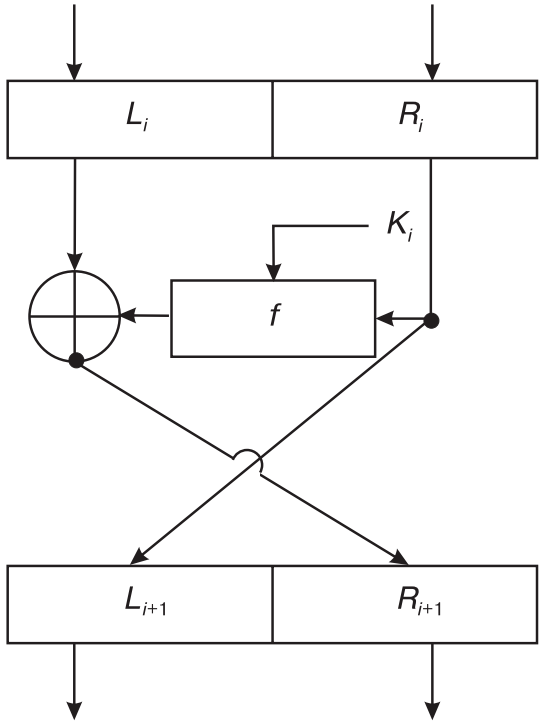


Рис. 2.9. Структура одиночного цикла алгоритма DES

В .NET Framework предусмотрены следующие классы для работы с симметричными алгоритмами:

- System.Security.Cryptography.**DES**;
- System.Security.Cryptography.**RC2**;
- System.Security.Cryptography.**Rijndael**;
- System.Security.Cryptography.**TripleDES**.

Асимметричная криптография

Асимметричная криптография, появившаяся на свет в конце 1970-х[1], представляет собой относительно новый поворот в истории криптографии.

Собственно говоря, можно только удивляться тому, что асимметричная криптография не была изобретена много раньше, учитывая тот факт, что она решает такие застарелые проблемы, как безопасность обмена ключами и возможность неподготовленного обмена секретными сообщениями. Эти проблемы раздражали многие могущественные правительства на

[1] Утверждают, что некоторые формы асимметричного шифрования использовались британской разведкой еще в 1950-х годах.

протяжении столетий, а многие большие корпорации – на протяжении десятилетий. Еще более удивительно то, что математический аппарат, на котором основывается асимметричная криптография, был хорошо известен уже несколько столетий. Тем не менее, асимметричная криптография родилась на свет лишь в тот момент, когда Уитфилд Диффи (Whitfield Diffie) и Мартин Е. Хеллман (Martin E. Hellman) опубликовали статью «Новые направления в криптографии» в 1976-м году.

Асимметричная криптография использует модульную арифметику и теорию простых чисел для того, чтобы создать два отдельных ключа. Один ключ при этом используется для шифрования, а второй – для дешифрования. Вместе они составляют пару ключей. Хотя оба ключа в математическом смысле очень глубоко между собой связаны, чрезвычайно трудно вычислить один ключ по другому. Один из ключей делается публично доступным («открытый ключ»), второй сохраняется в строжайшем секрете («секретный ключ»). Обычная секретность может быть достигнута, если зашифровать сообщение открытым ключом, а для расшифровки использовать секретный ключ. Но вот вторая сторона монеты: можно решить проблемы аутентификации, целостности и подтверждения обязательств, если использовать для шифрования секретный ключ, а расшифровывать сообщения открытым ключом.

В любом случае асимметричная криптография основывается на идее односторонней функции с «черным ходом». Подробней мы рассмотрим все это в главе 4, а пока мы опишем одностороннюю функцию, как любую функцию, которая легко вычисляется в прямом направлении и крайне трудно вычислима в направлении обратном. Иными словами, при заданном «x» легко вычисляется $y = f(x)$, однако по заданному «y» найти $x = f^{-1}(y)$ крайне трудно. Вероятно, вы легко найдете аналогии в повседневной жизни. Например, очень легко влить молоко в кофе, но трудно извлечь его обратно! Односторонняя функция с «черным ходом» – это такая односторонняя функция, которая внезапно становится легкой для вычисления в обратном направлении, если обеспечить ей дополнительную секретную информацию (например, ключ). В случае с молоком и кофе роль «черного хода» могут сыграть специальные знания из органической химии, позволяющие отделить молоко от кофе при помощи применения сложной последовательности химических реагентов и фильтрации с обработкой на центрифуге. Хотя я не уверен, что кому-то потом захочется выпить такое молоко!

Разработано несколько асимметричных алгоритмов: RSA, ElGamal и ECC[1], но самым широко используемым остается RSA. Мы подробно рассмотрим устройство алгоритма RSA в главе 4, а сейчас сделаем краткий обзор всего процесса использования этой технологии. Итак, Алиса хочет предоставить Бобу возможность отправить ей секретное сообщение.

1. Алиса случайным образом выбирает два больший простых числа p и q, которые она сохраняет в тайне, как секретный ключ (p, q).
2. Алиса перемножает эти числа, чтобы получить их произведение $n = p \cdot q$.

3. Алиса вычисляет функцию Эйлера (тотиент) $\phi(n) = (p - 1)(q - 1)$. Эта функция есть количество целых чисел, меньших n, и взаимно простых с n.
4. Алиса выбирает показатель e такой, что $1 < e < \phi(n)$, и наибольший общий делитель для e и $\phi(n)$ равен 1.
5. Алиса делает произведение n и показатель e общедоступными в виде открытого ключа (n, e).
6. Алиса вычисляет число d, инверсию по модулю n от e, такое, что для любого x выполняется: $x = (x^e)^d (\text{mod } n)$.

Далее события развиваются на стороне Боба.

1. Боб получает общедоступный открытый ключ (n, e).
2. Боб создает сообщение в открытом тексте и представляет его как число p.
3. Боб вычисляет зашифрованный текст, как число $c = p^e (\text{mod } n)$ и отправляет его Алисе.

Далее в ход событий вмешивается Ева – ее ждет разочарование.

1. Ева перехватывает шифрованный текст c.
2. Поскольку ей неизвестно d, она не в состоянии вычислить $p = c^d$.
3. Ей не удается вычислить d, даже зная открытый ключ (n, e), поскольку задача факторизации (разложение на сомножители) слишком трудна для вычисления.

А вот Алиса счастлива:

1. Алиса получает c.
2. Алиса прямо вычисляет открытый текст: $p = c^d$.

В .NET Framework предусмотрены следующие классы для работы с асимметричными алгоритмами:

- System.Security.Cryptography.**DSA**;
- System.Security.Cryptography.**RSA**.

Криптографические алгоритмы

Помимо главных симметричных и асимметричных алгоритмов шифрования, существует несколько важных алгоритмов, играющих вспомогательную роль таких, как генерация случайных чисел и алгоритмы хеширования. В последующих главах мы детально рассмотрим алгоритмы шифрования DES, RSA и так далее. Однако, поскольку генерация псевдослучайных чисел и хеширование имеют фундаментальное значение для всех аспектов криптографии, мы вкратце познакомимся с этими темами здесь. К вопросам генерации псевдослучайных чисел мы, практически, более не вернемся, но с алгоритмами хеширования мы еще встретимся в главе 5, при рассмотрении цифровой подписи и аутентифицирующих хешей.

ГЕНЕРАТОРЫ ПСЕВДОСЛУЧАЙНЫХ ЧИСЕЛ

Генераторы псевдослучайных чисел (PRNG) играют очень важную роль в криптографии. Мы уже видели, как радикально зависит надежность шифра OTP от степени случайности ключа. Фактически все современные шифры зависят от случайности своих ключей. Если последовательность чисел, генерируемая PRNG, недостаточно случайна, то среди ее чисел могут существовать зависимости, которые облегчат атаку на шифр. В случае успеха атакующая сторона может угадать закономерность, по которой генерируются ваши ключи, что будет означать полный крах криптосистемы. Симметричные блочные шифры в значительной мере полагаются на генератор псевдослучайных чисел, который дает им вектор инициализации, а также ключи.

К сожалению, при помощи детерминистических алгоритмов невозможно генерировать настоящие случайные числа. Если не используются специальные аппаратные решения, любая компьютерная программа будет работать по детерминистическому алгоритму, и лучшее, на что можно тут рассчитывать, это PRNG. Программный генератор псевдослучайных чисел хорошего качества отличается тем, что для него очень трудно предсказать очередное число на основе ранее выданных чисел. Конечный автомат, управляемый детерминистическим алгоритмом, неизбежно вернется в начальное состояние, и последовательность начнет повторяться, что по определению невозможно для настоящей случайной последовательности. Генераторы PRNG зависят от некоего начального, инициализирующего числа, которое называют семенем (*seed*). Начав работу с заданного значения *seed*, генератор производит детерминированную последовательность, которая будет повторена с начала, если задать ему такое же число *seed*. Конечно, одно и то же число *seed* ни в коем случае не следует использовать повторно.

Если наш источник случайных чисел несовершенен, то мы можем стремиться к тому, чтобы он был, по крайней мере, хорош. Хороший генератор PRNG обладает как можно более «плоской» кривой распределения, то есть на большом отрезке все числа встречаются с примерно одинаковой частотой. Также никакая последовательность чисел не встречается чаще других.

Если вас интересуют максимальные возможности, то существуют специальные устройства, которые используют случайность некоторых физических процессов таких, как квантовый шум резисторов или диодов, или шум радиоактивного распада, детектированный счетчиком Гейгера[1]. В

[1] Являются ли физические процессы действительно случайными или же нет – этот вопрос все еще дебатируется в квантовой физике. Гейзенберг и другие привели убедительные аргументы в пользу того, что истинная случайность действительно существует в природе. Тем не менее, Эйнштейн сделал свое знаменитое замечание: «Господь не играет в кости». Так или иначе степень случайности квантового шума или радиоактивного распада более чем достаточна для целей криптографии на все обозримое будущее!

этих целях используют также фракталы и хаотические системы[1] такие, как возмущения воздуха, нажатия клавиш на клавиатуре, парафиновой лампы[2] и другие.

ГЕНЕРАТОРЫ ПСЕВДОСЛУЧАЙНЫХ ЧИСЕЛ И .NET FRAMEWORK

Генераторы псевдослучайных чисел, которые обычно поставляются в составе платформы, как, например, генераторы в Windows или UNIX (функции **srand** и **rand**) не обладают достаточным качеством и непригодны для криптографических применений. Класс **Random**, доступный на платформе .NET, также недостаточно хорош. Класс **RNGCryptoServiceProvider** предоставляет доступ к высококачественному генератору, входящему в состав поставщика услуг криптографии (CSP – cryptographic service provider). Тем не менее, может оказаться, что вам никогда не потребуется обращаться к этому классу напрямую, поскольку криптографические классы .NET Framework неявно используют его для генерации ключей.

Абстрактный базовый класс **RandomNumberGenerator** производит лишь один дочерний класс с именем **RNGCryptoServiceProvider**. Класс **RandomNumberGenerator** поддерживает стандартный метод класса **Object**, а также методы **GetBytes** и **GetNonZeroBytes**, которые возвращают массив с байтами, содержащими криптографически сильную последовательность значений. Как понятно из его имени, метод **GetNonZeroBytes** возвращает массив случайных чисел без нулевых значений. Также класс **GetNonZeroBytes** поддерживает два переопределения метода **Create**, который создает экземпляр производной конкретной реализации криптографического генератора случайных чисел. Поскольку **RandomNumberGenerator** – абстрактный класс, вы не можете создать его экземпляр и не можете обращаться к этим методам напрямую. Вместо этого вы должны использовать конкретный производный класс. Давайте рассмотрим объявление этих функций.

```
public abstract void GetBytes(
   Byte[] data //массив для заполнения случайными байтами
);

public abstract void GetNonZeroBytes(
   Byte[] data // массив для заполнения ненулевыми случ. байтами
);
```

[1] Теория хаоса – относительно новая отрасль математики, занимающаяся нелинейными динамическими системами, которые с одной стороны детерминированы, но с другой стороны, поведение которых на достаточно большой период предсказать практически невозможно. Фондовый рынок и человеческий ум рассматриваются, как примеры хаотических систем. Теория фракталов имеет дело с математическим аппаратом, генерирующим огромные объемы информации из очень небольшого начального объема, что очень похоже на задачу PRNG.

[2] Либо само случайное число, либо seed для его генерации в PRNG получается вычислением криптографического хеша от цифрового образа непрерывно меняющегося контура в парафиновой лампе.

```
public static RandomNumberGenerator Create();
   //создает экземпляр реализации PRNG по умолчанию

public static RandomNumberGenerator Create(String);
   //создает экземпляр заданной реализации PRNG
```

Конкретный класс **RNGCryptoServiceProvider** можно создать и использовать напрямую. Следующий фрагмент кода иллюстрирует генерацию 128 случайных байт при помощи этого класса. Здесь использован конструктор **RNGCryptoServiceProvider**, но вы можете, если хотите, использовать вместо этого один из статических методов **Create**.

```
byte[] randomBytes = New Byte[128];
RNGCryptoServiceProvider rngcsp =
   new RNGCryptoServiceProvider();
rngcsp.GetBytes(randomBytes); //получение массива случайных чисел
```

КРИПТОГРАФИЧЕСКИЕ ХЕШИРУЮЩИЕ АЛГОРИТМЫ

Криптографические хеширующие алгоритмы получают на входе произвольный объем данных и на выходе уменьшают его до заданного размера (обычно это 128, 160 или 256 бит). Результат работы такого алгоритма называют «дайджестом сообщения» или «отпечатком пальца», и он, результат, в высокой степени идентифицирует исходное сообщение, подобно тому, как отпечаток пальца идентифицирует человека. В идеале криптографический хеширующий алгоритм должен удовлетворять следующим требованиям:

- трудно восстановить входные данные по выходным (то есть алгоритм должен быть односторонним);
- трудно подобрать такие входные данные, которые дали бы на выходе заранее заданный результат;
- трудно найти два варианта входных данных, которые дали бы одинаковые выходные результаты;
- изменение одного бита во входных данных приводит к изменению, примерно, половины битов в результате.

Хеш-алгоритм генерирует «отпечаток пальца» фиксированного размера для произвольного объема входных данных.

Результат работы хеш-алгоритма используется в следующих целях:

- с его помощью можно обнаружить изменения, внесенные во входные данные;
- он используется в алгоритмах, реализующих цифровую подпись;
- его можно использовать для трансформации пароля в такое секретное представление, которое можно безопасно передавать по сети или хранить на незащищенном устройстве;

- его можно использовать для трансформации пароля в ключ для использования в алгоритмах шифрования.

Наиболее часто с этой целью используют алгоритмы SHA-1 и MD5[1]. Алгоритм SHA-1 был введен в NIST и опубликован в стандарте криптографического хеширования Secure Hash Algorithm (SHS, FIPS 180-1). SHA-1 создает 160-битовый дайджест. За алгоритмом SHA-1 последовали SHA-250, SHA-384 и SHA-512, которые, соответственно, производят 256-, 384- и 512-битовые дайджесты. Более детальные сведения об этом можно найти по ссылке *http://csrc.nist.gov/encryption/tkhash.html*. Алгоритм MD5 производит 128-битовый дайджест, делая это быстрее, однако обладает меньшей стойкостью к атаке методом «грубой силы». MD5 был разработан Рональдом Ривестом в начале 1990-х и был принят в качестве RFC (см. *www.rfc.net/rfc1321.html*).

В библиотеке .NET Security Framework предусмотрены следующие классы для работы с хеширующими алгоритмами:

- System.Security.Cryptography.**KeyedHashAlgorithm**;
- System.Security.Cryptography.**MD5**;
- System.Security.Cryptography.**SHA1**;
- System.Security.Cryptography.**SHA256**;
- System.Security.Cryptography.**SHA384**;
- System.Security.Cryptography.**SHA512**.

Класс **KeyedHashAlgorithm** – это абстрактный класс, из которого производятся все классы, реализующие конкретные алгоритмы. Хеш с ключом (keyed hash) отличается от обычного криптографического хеша тем, что принимает в качестве дополнительных входных данных ключ. Таким образом, для верификации хеша необходимо знать ключ. Есть два производных класса, получаемых из **KeyedHashAlgorithm**, это **HMACSHA1** и **MACTripleDES.HMACSHA1**, они получают ключ произвольного размера и генерируют 20-байтовый «код аутентификации сообщения» MAC (Message Authentication Code), используя при этом алгоритм SHA-1. Буквы HMAC расшифровываются, как Keyed-Hash Message Authentication Code (код аутентификации сообщения при помощи ключевого хеша), это стандарт NIST (см. FIPS PUB 198). **MACTripleDES** генерирует код MAC при помощи «тройного DES», используемого в качестве хеширующего алгоритма. Он принимает ключи размером 8, 16 или 24 байта и генерирует 8-байтовый хеш. Алгоритмы хеширования с ключом полезны в схемах аутентификации и проверки целостности, фактически они являются альтернативой электронной подписи.

---

[1] Иногда думают, что единственной целью атаки может являться шифр. Как ни странно, но хеширующие алгоритмы и генераторы случайных чисел также подвергаются нападениям. Атака на хеш-алгоритм может иметь своей целью получение входных данных по имеющемуся результату или нахождение второго варианта входных данных, который приводил к такому же результату. Атака на генератор случайных чисел означает нахождение закономерности в числах и предсказание их будущей последовательности. Такого рода атаки могут играть вспомогательную роль в более сложных атаках таких, как нападение на шифр или подделка электронной подписи.

Другие хеш-классы (реализующие функции MD5 и SHA) из предыдущего списка являются обычными хеш-алгоритмами и не принимают на входе ключей. Они пригодны в ситуациях, когда хеш используется в среде лиц, не разделяющих между собой никакой секретной информации.

Криптографические протоколы

Криптографические протоколы – это общепринятое соглашение, касающееся набора алгоритмов, последовательности действий и определения функций каждого из участников процесса.

Например, простой криптографический протокол, определяющий шифрование и дешифрование сообщений при помощи асимметричного алгоритма RSA и симметричного алгоритма Triple DES, мог бы выглядеть следующим образом[1]. Обратите внимание, алгоритм RSA работает слишком медленно, для того чтобы шифровать им данные заметного объема.

Поэтому он используется только для шифрования относительно небольшого секретного ключа для алгоритма Triple DES, который затем используется для шифрования основного объема данных.

1. Алиса и Боб генерируют каждый для себя пару ключей RSA (открытый и секретный ключи).
2. Они обмениваются открытыми ключами RSA, оставляя секретные ключи при себе.
3. Каждый из них генерирует собственный ключ Triple DES и шифрует этот ключ при помощи открытого ключа RSA, принадлежащего своему партнеру. Теперь расшифровать сообщение и получить ключ Triple DES можно только при помощи секретного ключа партнера.
4. Они пересылают друг другу зашифрованные ключи Triple DES.
5. Теперь, если Алисе или Бобу потребуется отправить секретное сообщение, каждый шифрует его при помощи ключа Triple DES своего партнера и отсылает его.
6. Партнер получает шифрованное сообщение и дешифрует его при помощи своего ключа Triple DES.

Другой пример протокола основывается на асимметричном алгоритме RSA и хеш-алгоритме SHA-1 и обеспечивает надежную идентификацию отправителя сообщения. Снова обратите внимание на то, что алгоритм RSA

[1] Приведенный здесь протокол достаточно прост для того, чтобы проиллюстрировать общую концепцию криптографических протоколов. Но эта схема уязвима для определенных атак. Например, Ева может перехватывать все сообщения, циркулирующие между Алисой и Бобом, подменяя их своими собственными сообщениями. Другую уязвимость этого протокола называют «атакой повтором» (replay attack), при которой Ева записывает циркулирующие сообщения, а затем повторяет их передачу, вводя Алису и Боба в заблуждение. Для противостояния подобным атакам протоколы делают значительно более изощренными.

работает очень медленно, и потому он применяется только к небольшому по размеру хешу («дайджесту», «отпечатку пальца») сообщения. Заметьте также, что этот протокол верифицирует только источник происхождения сообщения, но ничего не делает для защиты самого сообщения. Вам, вероятно, не составит труда додумать протокол дальше, чтобы обеспечить также и секретность сообщения.

1. Алиса и Боб генерируют каждый для себя пару ключей RSA (открытый и секретный ключи).
2. Они обмениваются открытыми ключами RSA, оставляя секретные ключи при себе.
3. При необходимости отправить сообщение своему корреспонденту каждый из них вычисляет хеш сообщения при помощи алгоритма SHA-1, затем шифрует этот хеш собственным секретным ключом RSA и отправляет сообщение вместе с зашифрованным хешем.
4. Когда Алиса или Боб получают сообщение, и если у них возникает необходимость убедиться в том, что отправителем является именно второй партнер, они расшифровывают присоединенный хеш при помощи открытого ключа RSA своего партнера. Затем они заново вычисляют хеш сообщения и сравнивают полученный результат с расшифрованным хешем. Если оба хеша совпадают, значит, отправителем является владелец использованного открытого ключа RSA.

В отличие от этих простых сценариев, криптографические протоколы могут подразумевать участие людей, которые не доверяют друг другу полностью, но тем не менее должны взаимодействовать каким-то образом. Например, это могут быть финансовые транзакции, банковские и торговые операции – везде используются специальные криптографические протоколы, учитывающие особенности конкретной среды.

Зачастую криптографические протоколы становятся компьютерными стандартами или конвенциями. Например, прокол Kerberos повсеместно используется для того, чтобы сервер и клиент могли надежно идентифицировать друг друга. Другой пример – это модель безопасного доступа к коду (CAS – Code Access Security) на платформе .NET, в которой исполняемый код снабжен цифровой подписью автора для верификации перед выполнением. Еще один пример: SSL – протокол защищенных сокетов (Secure Sockets Layer), используемый для безопасных коммуникаций через Internet. Есть много других примеров, включая PGP (Pretty Good Privacy – достаточно надежная секретность) для шифрования электронной почты или «соглашение о ключах Диффи-Хеллмана» для обмена сеансовыми ключами по незащищенному каналу и без предварительного обмена какой-либо секретной информацией. Существует ряд криптографических протоколов, которые реализованы и доступны в .NET Security Framework.

Криптоаналитические атаки

Существует общепринятая терминология, описывающая различные виды криптоаналитических атак. В приведенном ниже списке разновидности атак перечислены в порядке от самых трудных к самым легким в смысле аналитической трудности, в от самых вероятных к наименее вероятным в смысле возможностей, которые могут оказаться в распоряжении атакующей стороны.

- **Атака на основе только зашифрованного текста:** в распоряжении атакующей стороны имеется только некоторый, случайно выбранный шифрованный текст.
- **Атака с открытым текстом:** в распоряжении атакующей стороны имеется случайно выбранный открытый текст и соответствующий ему шифрованный текст.
- **Атака с выбранным открытым текстом:** в распоряжении атакующей стороны имеется выбранный открытый текст и соответствующий ему шифрованный текст.
- **Атака с выбранным зашифрованным текстом:** в распоряжении атакующей стороны имеется выбранный шифрованный текст и соответствующий ему открытый текст.
- **Адаптивная атака с выбранным открытым текстом:** атакующая сторона может многократно получать шифрованный текст, соответствующий заданному открытому тексту, основывая каждый *очередной выбор на предыдущих вычислениях*.

Атака на основе только зашифрованного текста – это самый очевидный сценарий, который легко себе представить. Когда вы только начинаете перехватывать шифрованные сообщения между обменивающимися сторонами, у вас вряд ли есть что-то большее, чем сами зашифрованные сообщения. К сожалению, при такой скудной начальной информации реализовать успешную атаку труднее всего. С течением времени у вас может появиться открытый текст одного из сообщений, или, по крайней мере, вы догадаетесь, что в сообщениях должны часто встречаться определенные слова. Например, во время Второй Мировой войны многие шифровки нацистов включали в себя легко предсказуемый текст. Это делало возможной атаку «с открытым текстом». Атаку с выбранным открытым текстом реализовать намного труднее, поскольку вы должны каким-то образом побудить отправителя зашифровать и отправить сообщение, текст которого вы выбрали сами. Например, если вы играете роль некоей доверенной стороны, вы можете прислать отправителю текст, который, как вы знаете, он по некоторым причинам зашифрует и перешлет получателю. Тщательным выбором текста вы можете значительно повысить свои шансы на раскрытие ключа.

Конечно, для всех этих хитроумных атак существуют некрасивые, совсем не технические альтернативы в виде, например, простой физической кражи ключа или подкупа, или шантажа в отношении владельца

ключа. Если оставить в стороне вопросы этики и законности, все эти способы страдают серьезными практическими недостатками, такими, например, что владелец ключа или другие стороны узнают, что ключ скомпрометирован, и результатом может быть смена ключа, или, еще хуже, намеренная передача дезинформирующих сообщений. Вы получаете огромное преимущество, если оказываетесь в состоянии не только раскрыть шифр, но и сохранить факт раскрытия в тайне.

Как уже упоминалось ранее, *атака методом «грубой силы»* представляет собой исчерпывающий перебор всех возможных ключей с применением этих ключей к шифрованному тексту. Если атака проводится только на шифрованный текст, пространство ключей должно проверяться до тех пор, пока не будет получен осмысленный текст, и в этом случае нет абсолютной уверенности в успехе (хотя практически можно положиться на предположение, что вряд ли осмысленный текст получился случайно). Если открытый текст заранее известен, то в правильности результата после нахождения ключа можно убедиться сравнением с известным текстом. После того, как ключ найден, можно расшифровать все остальные сообщения, открытый текст которых не был известен.

Человеческий фактор

В утопическом мире средства обеспечения безопасности и криптография были бы напрасной тратой времени и сил. Если бы вы могли доверять любому, и вам мог бы довериться любой, и все вместе знали бы, что никто никому не причинит сознательного вреда, то не было бы нужды в хранении секретов, в контроле доступа или идентификации личности. К сожалению, в реальности это не так. Следовательно, мы должны принимать меры предосторожности для защиты своих интересов от личностей и организаций, не достойных нашего доверия.

Риск и выигрыш

При любом взаимодействии с окружающим миром мы неизбежно имеем дело с возможными рисками и возможными выгодами. Очевидная стратегия состоит в том, чтобы принимать только такие риски, при которых ожидаемый выигрыш перевешивает ожидаемый ущерб. Вам хочется иметь своеобразный баланс, в котором вы что-то выигрываете от коммуникаций с другими людьми, но ограничиваете их возможности знать что-то о вас. Например, если вы хотите купить или продать что-то на сайте *www.ebay.com*, вам желательно убедиться в том, что информация о вашей кредитной карточке не попадет в ненадлежащие руки. Также вам требуется точно знать, с кем вы имеете дело.

Подобное балансирование может принимать разнообразные формы. Например, вы можете столкнуться с маловероятным риском, который связан с тяжелыми последствиями, и весьма вероятным, но скромным выигрышем. Такого рода решения вы принимаете каждый день, когда ведете машину. С другой стороны, маловероятная, но большая выгода может противостоять весьма вероятной или даже 100-процентной потере. Если вы когда-нибудь покупали лотерейный билет, то знакомы с таким балансом. Бывают ситуации, когда хорошего выбора у вас просто нет. Когда угроза огромна по ущербу и весьма вероятна, ее предотвращение может быть слишком дорого или вообще невозможно, и вы просто надеетесь на лучшее. Подумайте, например, о смерти.

Все эти крайние сценарии должны навести вас на мысль о взвешивании риска и потенциального выигрыша. Вам необходимо производить такое взвешивание всякий раз, когда вы занимаетесь настройкой своей системы с точки зрения безопасности или реализуете в программе криптографические функции. Помните, что если вы стремитесь к излишней безопасности, это может ухудшить главные, рабочие функции системы, а пользователей это побудит искать «обходные пути».

Другие важные концепции

Вот еще несколько важных концепций, связанных с безопасностью и криптографией.

- **Конфиденциальность** означает, что критическая информация защищена от доступа со стороны неавторизованных лиц. Это очень близко к концепции *секретности*.
- **Целостность** означает, что целостность данных подтверждена, и данные не могли быть кем-то изменены.
- **Аутентификация** – это процесс, удостоверяющий идентичность лица или системы.
- **Подтверждение обязательств** означает, что лицо не может предпринять какое-то действие, а впоследствии этот факт отрицать. Это тесно связано с концепцией *подтверждения получения* (proof of receipt).
- **Авторизация** означает, что политика управления доступом может быть сконфигурирована так, чтобы предоставлять определенным лицам права на выполнение каких-то операций или отказывать им в этих правах. Это тесно связано с концепцией *контроля доступа*.
- **Анонимность** означает, что лицо остается не идентифицированным.
- **Владение ресурсом** обеспечивает права на использование этого ресурса.
- **Сертификация** – это процесс удостоверения и подтверждения признанным авторитетом утверждений, заявленных лицом.
- **Свидетельство** – верификация одним лицом действий другого лица.

- **Верификация** – проверка действительности или истинности заявления, сделанного лицом. Сюда может входить проверка кода или данных на предмет безопасности или на соответствие каким-то правилам или стандартам.

Итоги главы

В этой главе вы познакомились с основными криптографическими терминами, концепциями и математическими элементами, необходимыми для понимания нескольких последующих глав. Для закрепления успеха мы изучили несколько классических алгоритмов шифрования. В следующих двух главах мы подробно изучим современные симметричные и асимметричные шифры, которые образуют основу безопасных коммуникаций и безопасного хранения информации, поддержки секретности и конфиденциальности, аутентификации, контроля целостности и подтверждения обязательств.

За пределами этой главы книга примет более прикладной характер, фокусируясь на конкретных функциях платформы Windows и .NET. Следовательно, мы не будем, практически, более возвращаться к освещению фундаментальных криптографических концепций. Для дальнейшего чтения на фундаментальные темы рекомендуем великолепную книгу *«The Handbook of Applied Cryptography»* Альфреда Дж. Менезеса (Alfred J. Menezes), Пола Си. Ван Ооршота (Paul C. van Oorschot) и Скотта А. Ванстоуна (Scott A. Vanstone), которая доступна в форматах PDF и Postscript по адресу *http://www.cacr.math.uwaterloo.ca/hac/*. Другая полезная книга – *«Applied Cryptography»* Брюса Шнайнера (Bruce Schneier). Великолепный исторический (не технический) очерк классической криптографии – книга *«The Codebreakers»* Девида Кана (David Kahn).

Глава 3

Симметричная криптография

Самыми фундаментальными задачами криптографии всегда были конфиденциальность, целостность и аутентификация. Традиционный подход к решению этих задач состоит в использовании решений, основывающихся на симметричных криптографических алгоритмах. В этой главе мы познакомимся с концепциями симметричной криптографии и увидим, как реализуются такие решения при помощи пространства имен **System.Security.Cryptography** в среде .NET Framework. В частности мы рассмотрим алгоритмы DES, «тройной» DES, Rijndael и RC2 и узнаем, как их использовать на практике.

Симметричные шифры

Симметричным называют шифр, в котором для шифрования и дешифрования используется один и тот же ключ (или разные ключи, которые связаны между собой в математическом смысле, то есть один ключ легко вычисляется из другого – фактически, такая ситуация все равно сводится к одному ключу). Поскольку единственный ключ полностью обеспечивает секретность шифра, он должен храниться в полной тайне. Если злоумышленник завладеет ключом, он сможет расшифровывать и зашифровывать сообщения, сведя все усилия обороняющейся стороны на нет. По этой причине симметричные шифры называют иногда шифрами «с секретным ключом» или «с разделяемым ключом».

В противоположность этому, в асимметричных шифрах используется пара ключей, между которыми существует математическая связь, однако выключить один ключ по другому крайне трудно. Как мы убедимся в главе 4, благодаря этому обстоятельству мы можем хранить в тайне лишь один из ключей, сделав второй публично доступным.

Симметричное шифрование и дешифрование математически представляются следующим образом (здесь E – функция шифрования, D – функция дешифрования, k – единственный ключ, M – открытый текст и C – шифрованный текст):

Шифрование: $C = E_k(M)$
Дешифрование: $M = D_k(C)$

На рисунке 3.1 проиллюстрирован принцип работы симметричной криптографии. Обратите внимание на тот факт, что отправитель и получатель (то есть Боб и Алиса) должны заранее договориться о секретном ключе и используемом алгоритме. Кроме алгоритма им надо согласовать также заранее ряд технических деталей таких, как вектор инициализации, режим вычислений и соглашение о заполнении пустых позиций. Все эти понятия мы вскоре рассмотрим.

Существует два основных типа симметричных алгоритмов: блочные шифры и потоковые шифры. Блочные шифры обрабатывают за один проход блок байт (обычно размером 64 или 128 бит). Потоковые шифры обрабатывают за один проход алгоритма один байт или даже один бит. Как мы убедимся, различия между блочными и потоковыми шифрами во многом условны, поскольку существуют режимы вычислений, в которых блочный шифр ведет себя подобно потоковому.

Рис. 3.1. Симметричная криптография

Как мы увидим в следующей главе, асимметричные шифры решают несколько важных криптографических задач, являясь, однако, менее безопасными, чем шифры симметричные (разумеется, при одинаковом размере ключа). Кроме того, симметричные алгоритмы, как правило, работают быстрее, что делает их более подходящими для шифрования больших массивов данных. Важно понимать, что симметричные и асимметричные шифры используются для решения разных задач, и их полезные свойства зачастую взаимодополняют друг друга. По этой причине в ряде важных криптографических протоколов используется сочетание симметричного и асимметричного шифров, что и позволяет достичь нужной цели.

На протяжении всей истории было изобретено множество симметричных алгоритмов, в то время как асимметричные шифры появились на свет сравнительно недавно. Симметричные шифры совершенствовались медленно и постепенно вплоть до окончания Второй Мировой войны, когда изобретение компьютеров дало мощный толчок развитию криптографии, и некоторые правительства тайно начали активную разработку шифров. К концу 1970-х уже многие частные организации начали пользоваться современными симметричными шифрами, ведущими свое происхождение из работ Хорста Файстеля (Horst Feistel), которые он вел в компании IBM[1] в 1967 г. В конце концов, алгоритм DES был принят правительством США в 1977 г. в качестве стандарта шифрования конфиденциальных (но не строго секретных) данных в правительственных учреждениях. Впоследствии DES использовался в частном секторе во многих приложениях, требующих секретности, таких, например, как банковские операции.

DES

Стандарт DES определен документом FIPS[2] PUB 46-3, также этот шифр документирован стандартом ANSI X9.32. Хотя алгоритм DES в наше время уже не рассматривается, как полностью безопасный[3], он все еще используется очень широко. Каждый раз, когда вы покупаете что-то в торговом автомате, вы, фактически, пользуетесь шифром DES. Этот алгоритм все еще можно использовать для сохранения секретов, не отличающихся крайней ценностью и чувствительностью, особенно, если речь идет об использовании уже имеющегося программного обеспечения, а в современных коммерческих приложениях этот шифр остается самым распространенным. В качестве временной рекомендации было предложено использовать более сильную модификацию «тройной DES», и, наконец, в 1990-х правительством США был принят алгоритм Rijndael, который и является сейчас правительственным стандартом симметричного шифрования.

DES является симметричным блочным шифром, преобразующим 64-битовые блоки данных при помощи 56-битового секретного ключа, причем преобразование включает в себя 16 циклов перестановок и подстановок. Как упоминалось в главе 2, подстановки, то есть замена битов данных,

---

[1] Хорст Файстель разработал в IBM шифр LUCIFER, являвшийся прямым предшественником алгоритма DES. В NSA активно интересовались проектом LUCIFER, и люди из NSA принимали участие в завершающих фазах разработки DES.

[2] FIPS – Federal Information Processing Standard (Федеральный стандарт обработки информации).

[3] Шифр DES уже несколько раз был взломан публично. Например, «Фонд электронного фронтира» (EFF – Electronic Frontier Foundation) построил специализированный компьютер для взлома DES, постройка обошлась в 250 тысяч долларов. Машина выиграла второй конкурс RSA DES Contest в 1998 году, раскрыв шифрованное сообщение методом «грубой силы» за 56 часов. Та же самая машина, действуя при поддержке ста тысяч персональных компьютеров через Internet (*http://www.distributed.net*), раскрыла DES всего за 22 часа.

делает связь между открытым и шифрованным текстами более сложной. Транспозиции, то есть перемещения битов внутри данных, приводит к диффузии данных, то есть информация распределяется по шифрованному блоку более равномерно, что затрудняет обнаружение статистических зависимостей. Более случайное в статистическом смысле распределение данных усложняет задачу криптоаналитика.

С математической точки зрения алгоритм DES задает обратимое преобразование множества всех 64-битовых чисел в само себя. Для прямого и обратного преобразования используется один и тот же 56-битовый ключ, что и делает DES симметричным алгоритмом. Выбором ключа задается одно из 2^{56} возможных преобразований, каждое из которых однозначно и обратимо.

DES имеет дело с 64-битовыми блоками данных, и обычный открытый текст перед шифрованием необходимо разбить на такие блоки, причем последний, скорее всего, неполный блок, придется дополнить. Существует несколько способов дополнения последнего блока до 64 бит. Каждый цикл DES получает на входе 64-битовый блок данных, который затем разделяется на два 32-битовых полублока. Правый полублок шифруется при помощи специальной функции, использующей в качестве ключа подмножество битов из 56-битового ключа DES, причем в каждом цикле используется свое, характерное для этого цикла подмножество. Зашифрованный правый полублок объединяется операцией XOR с левым полублоком, и результат становится новым правым полублоком для следующего цикла. В левый полублок для следующего цикла подставляется прежний правый полублок. На рисунке 3.2 показана логическая структура одного цикла DES.

Общая структура всех 16 циклов DES иллюстрируется рисунком 3.3. Мы не рассматриваем здесь все детали алгоритма. Например, 48-битовые субключи, используемые в каждом очередном цикле (от K1 до K16), генерируются из основного 56-битового ключа при помощи битовых манипуляций, зависящих от номера цикла. Мы также не углубляемся в такие детали реализации, как, например, специализированная шифрующая функция, которая применяется в каждом цикле и которая включает в себя S-подстановку и R-перестановку в каждом цикле. Для практического использования DES все эти детали не слишком важны, впрочем, если они вас все же интересуют, то ознакомиться с ними можно в документе FIPS 46-3[1].

Операционные режимы

DES является блоковым шифром и то, о чем говорилось в предыдущем разделе, относится 64-битовому блоку данных. Тем не менее, произвольный открытый текст не обязательно будет умещаться в 64-битовом блоке.

[1] Документ FIPS 46-3 доступен по следующему URL: *http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf*

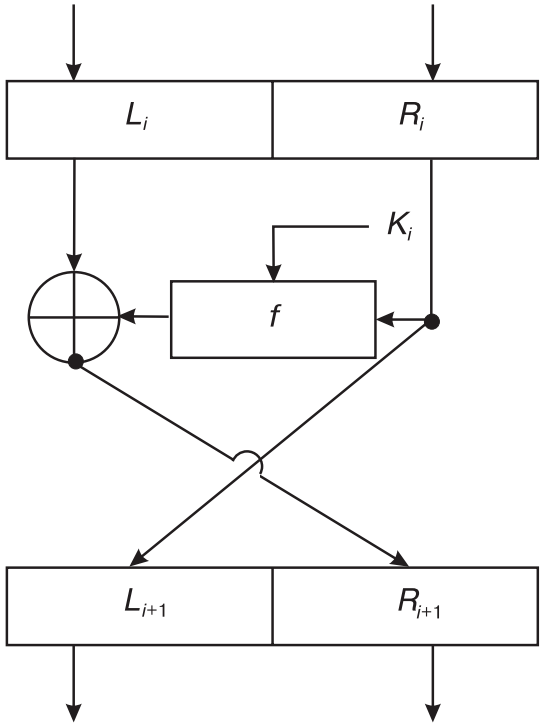


Рис. 3.2. Один цикл алгоритма DES

Следовательно, блоковые шифры (DES, Rijndael) должны иметь дело с последовательностью блоков входных данных. Для обработки последовательности блоков шифр должен оперировать блоками, используя некие правила, которые определяются операционным режимом. Существует несколько стандартных операционных режимов DES (ECB, CBC, CFB и OFB), определяемых в документе FIPS PUB 81[1]. В добавление к этому, для использования с шифром RC5 был разработан операционный режим CTS (описан в RFC 2040), и этот режим можно использовать также и с другими блоковыми шифрами.

Все пять операционных режимов определяются в .NET Framework при помощи перечисления **CypherMode** в **System.Security.Cryptography**. Вот полные названия этих режимов:

- электронная шифровальная книга (Electronic Codebook – ECB);
- сцепление шифрованных блоков (Cipher Block Chaining – CBC);
- шифрованная обратная связь (Cipher Feedback – CFB);
- обратная связь по выходу (Output Feedback – OFB);
- проскальзывание шифрованного текста (Cipher Text Stealing – CTS).

[1] Этот документ называется *DES Modes of Operation (Операционные режимы DES)*.

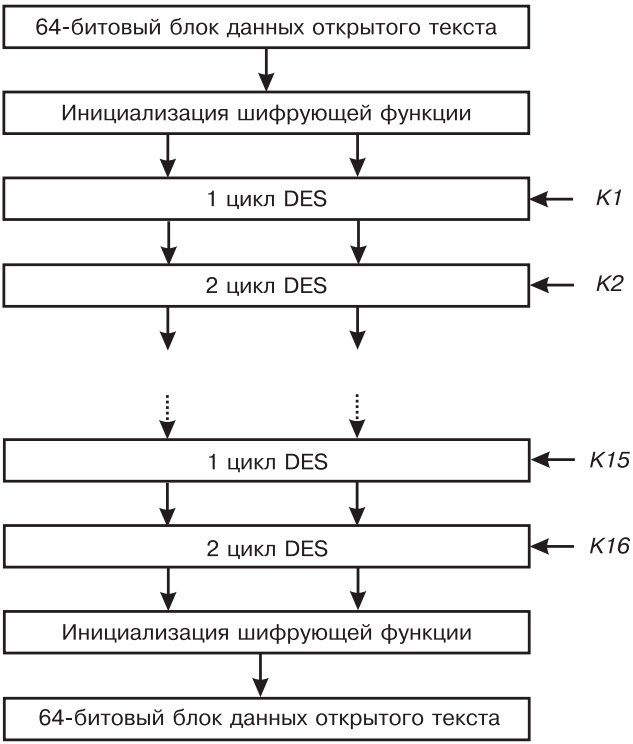


Рис. 3.3. Шестнадцать циклов DES

В текущей версии .NET Framework доступны только некоторые из этих режимов в применении к определенным алгоритмам шифрования. Режимы ECB, CBC и CBF в настоящее время работают с алгоритмами DES, тройной DES и RC2. Для алгоритма Rijndael доступны пока только режимы ECB и CBC. Режимы OFB и CTS еще не реализованы ни для одного алгоритма, хотя само их присутствие в перечислении **CypherMode** указывает на это, их поддержка в будущем все же предполагается. Интересно также отметить, что режим CTS был создан специально для алгоритма RC5, который в текущей версии .NET Framework вообще никак не представлен. Несмотря на недостающие части общей картины, мы сможем рассмотреть четыре операционных режима шифрования в последующих разделах с тем, чтобы получить о них достаточно полное представление.

Далее мы познакомимся с реальной программой .NET Framework, которая иллюстрирует применение этих режимов. Но вначале давайте рассмотрим их на концептуальном уровне. Для того чтобы понимать последующие описания, ознакомьтесь с нотацией, которая будет использоваться:

- M_i – ***i***-й 64-битовый блок открытого текста;
- C_i – ***i***-й 64-битовый блок шифрованного текста;
- E_k – функция шифрования DES, использующая ключ k;

- D_k – функция дешифрования DES, использующая ключ k;
- IV – 64-битовый вектор инициализации, используемый в некоторых режимах для имитации предыдущего блока данных в ситуации, когда обрабатывается самый первый блок.

РЕЖИМ ECB

Это самый простой операционный режим шифрования, в котором 16 циклов Файстеля, описанные выше, применяются к каждому очередному блоку данных индивидуально и безо всякой зависимости от других блоков. При заданном 56-битовом ключе режим ECB всегда дает однозначное соответствие между каждым блоком открытого текста и соответствующим ему блоком текста шифрованного, при этом ошибка в одном блоке не распространяется на последующие блоки. Тот факт, что каждый блок обрабатывается независимо от других, и дал название этому методу «электронной шифровальной книги», поскольку для любого ключа можно, теоретически, создать шифровальную книгу, пусть и очень большого размера. У режима ECB есть некоторые преимущества перед своими собратьями. Например, вы можете произвольно шифровать отдельные записи в файле или в базе данных, поскольку каждый блок шифруется независимо от других. Также, ECB позволяет обрабатывать блоки параллельно и одновременно, что при соответствующей аппаратной поддержке увеличивает пропускную способность всего процесса.

К сожалению, преимущества ECB перевешиваются его слабой криптостойкостью. Проблема этого режима состоит в том, что атакующая сторона может скомпилировать «шифровальную книгу» для конкретного ключа, основываясь на раскрытых или угаданных парах «открытый блок – шифрованный блок». Такие пары могут быть получены различными способами, основывающимися на повторяющейся природе многих частей текста таких, как заголовки или стандартные фразы. Интересно в этом методе атаки то, что раскрывать ключ в нем нет необходимости, просто по мере роста «шифровальной книги», то есть набора известных пар, раскрытие новых пар все более и более облегчается, давая в какой-то момент эффект снежного кома. Это похоже на складывание картинки из фрагментов, когда каждый новый правильно уложенный фрагмент облегчает дальнейшее продвижение. Схема функционирования режима ECB изображена на рисунке 3.4.

Как видно из рисунка 3.4, каждый блок открытого текста M_i преобразуется функцией E_k независимо от остальных блоков. Конечно, дешифрование в режиме ECB также осуществляется независимо для каждого блока шифрованного текста C_i при помощи функции дешифрования D_k. Математически режим ECB описывается следующим образом.

Шифрование в режиме ECB:

$$C_i = E_k(M_i).$$

Дешифрование в режиме ECB:

$$M_i = D_k(C_i).$$

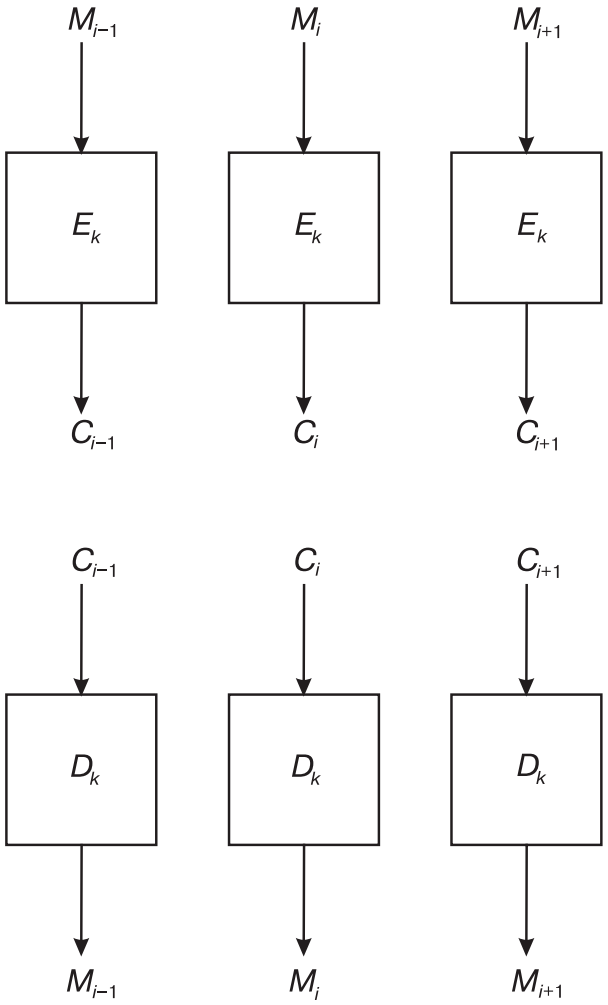


Рис. 3.4. Режим ECB

РЕЖИМ CBC

Режим CBC представляет собой более защищенную технологию, не позволяющую создать «шифровальную книгу». В режиме CBC перед началом 16 циклов Файстеля каждый блок открытого текста суммируется по операции XOR с предыдущим зашифрованным блоком. Самый первый блок суммируется по XOR с некоторым случайным 64-битовым вектором инициализации (*IV*), поскольку для первого блока открытого текста не существует предыдущего зашифрованного блока. Вы можете думать об *IV*, как о фиктивном блоке шифрованного текста, который, что интересно, вовсе не должен быть секретным (его можно безопасно передавать по открытому каналу). Схематически режим CBC изображен на рисунке 3.5.

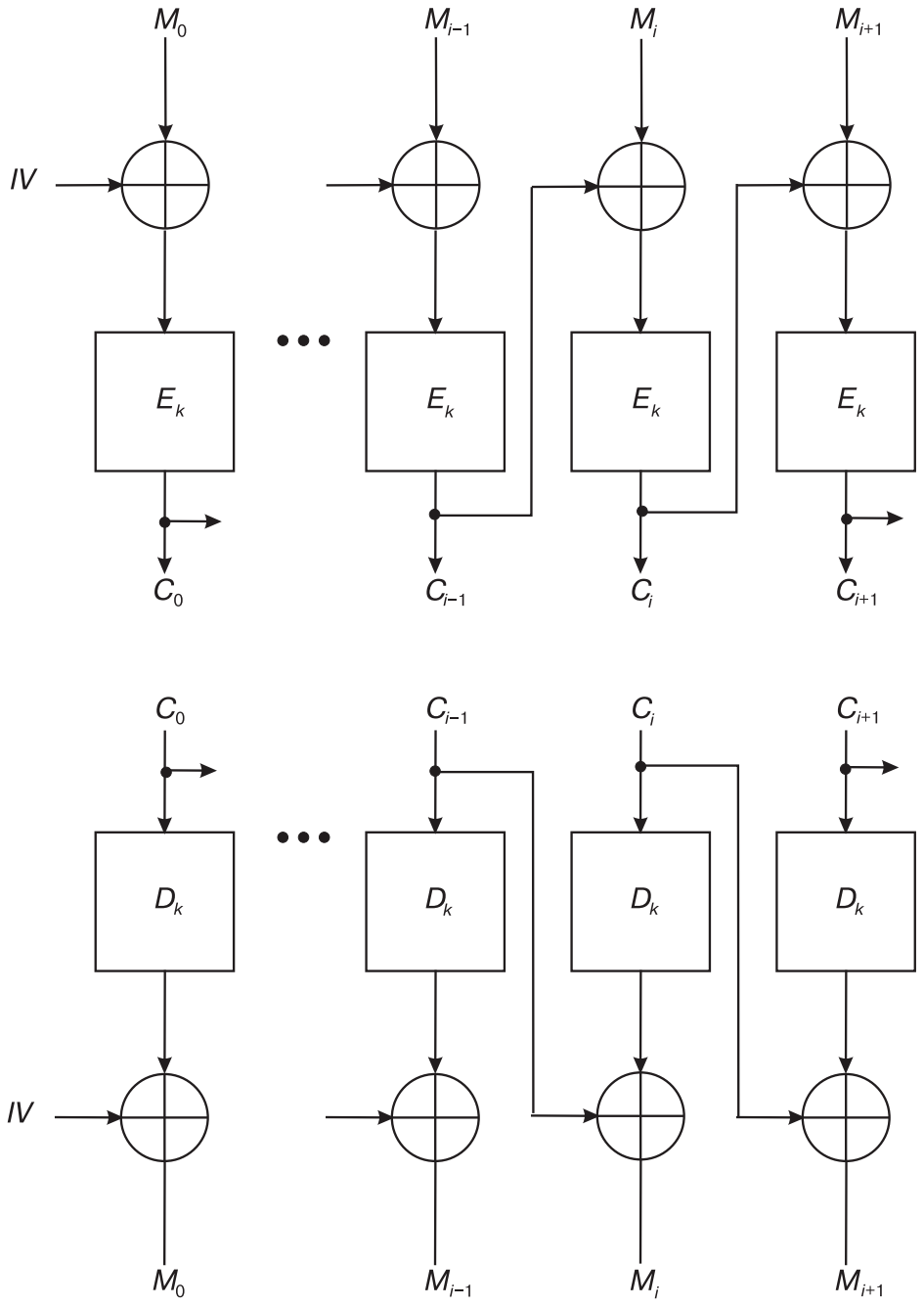


Рис. 3.5. Режим CBC

Как видно из рисунка, каждый блок открытого текста M_i суммируется по XOR с предыдущим блоком шифрованного текста C_{i-1}, и уже результат суммирования поступает на вход шифрующей функции E_k, на выходе которой образуется блок шифрованного текста C_i. При дешифровании в режиме CBC очередной блок шифрованного текста C_i преобразуется дешифрующей функцией D_k, а полученный результат суммируется по XOR с предыдущим шифрованным блоком C_{i-1}, что и дает расшифрованный блок M_i. Математически режим CBC можно описать следующим образом. Символ $\oplus$ обозначает здесь побитовую операцию XOR, а IV представляет собой вектор инициализации, используемый только в первом блоке.

Шифрование в режиме CBC:

$$C_0 = E_k(M_0 \oplus IV),$$

$$C_i = E_k(M_i \oplus C_{i\text{-}1}).$$

Дешифрование в режиме CBC:

$$M_0 = D_k(C_0) \oplus IV,$$

$$M_i = D_k(C_i) \oplus C_{i\text{-}1}.$$

РЕЖИМЫ CFB И OFB

Существует еще два стандартных режима для алгоритма DES, которые позволяют оперировать порциями открытого текста, меньшими стандартного 64-битового блока. Такие режимы полезны в качестве потоковых шифров, где алгоритм DES генерирует псевдослучайные биты, которые объединяются по XOR с потоковым открытым текстом[1]. Например, 8-битовый режим CFB можно использовать для шифрования потока двоичных байт или символов ASCII. Режимы ECB и CFB могут производить только 64-битовые блоки, а ведь существуют ситуации, в которых необходимо незамедлительно обрабатывать порции данных меньшего размера. Например, в радиолокационной системе необходимо передавать отдельные зашифрованные байты, по одному за раз, по мере их генерации, для управления системой в реальном времени. Такая система не терпит произвольных задержек, которые возникнут, если для продолжения шифрования алгоритм будет дожидаться формирования полного 64-битового блока. Режимы ECB и CBC непригодны в подобной ситуации, и тут как раз пригодятся режимы CFB и OFB.

При заданном ключе режимы CFB и OFB не дадут фиксированного соответствия между открытыми и шифрованными блоками, поскольку в них шифрование очередного блока зависит от предыдущего блока, и применяется вектор инициализации. Этот факт затрудняет криптоанализ, однако он также означает, что ошибка в любом бите данных при передаче шифрованного текста повредит не только текущий, но все последующие

[1] Здесь мы приводим несколько упрощенное описание режимов CFB и OFB, достаточное для понимания общего принципа. Детальные сведения обо всех режимах DES можно найти в документе FIPS PUB 81, доступном по адресу *http://csrc.nist.gov/publications/fips/fips81/fips81.htm*

блоки открытого текста. В режиме OFB потеря единственного бита при передаче способна полностью испортить весь текст. Преимущество режима CFB над OFB состоит в том, что он обладает свойством самосинхронизации, то есть при порче блока в результате ошибки передачи, правильность расшифровки будет восстановлена в уже следующем блоке.

Рисунок 3.6 иллюстрирует принцип режима CFB. Сдвиговый регистр позволяет использовать смещающееся подмножество ранее сгенерированных выходных битов для выполнения XOR с текущими битами открытого текста, подлежащими шифрованию. На диаграмме не видно, что для суммирования по XOR с текущим подмножеством используется только подмножество, состоящее из крайних левых зашифрованных битов из предыдущего прохода. Речь идет о 8 крайних левых битах, однако никакой специфики в этом числе нет, поскольку можно использовать любое подмножество, меньшее 64-битового блока.

Обрабатывая 8 бит за один проход, режим CFB, практически, превращает блочный алгоритм DES в потоковый. Математически операционный режим CFB можно описать следующим образом:

Шифрование в режиме CFB:

$$C_0 = M_0 \oplus E_k(IV),$$

$$C_i = M_i \oplus E_k(C_{i\text{-}1}).$$

Дешифрование в режиме CFB:

$$M_0 = C_0 \oplus D_k(IV),$$

$$M_i = C_i \oplus D_k(C_{i\text{-}1}).$$

На рисунке 3.7 изображено упрощенное представление о режиме OFB. Как видно из рисунка, основной принцип здесь почти совпадает с принципом работы CFB. Единственное отличие состоит в том, что в режиме CFB зашифрованные биты из предыдущего прохода попадают на вход сдвигового регистра уже после операции XOR. В противоположность этому, в режиме OFB биты попадают в сдвиговый регистр перед операцией XOR. Так же, как и CFB, режим OFB позволяет манипулировать порциями данных, меньшими 64-битового блока, на котором основывается алгоритм DES.

Математически операционный режим OFB можно описать следующим образом.

Шифрование в режиме OFB:

$$C_0 = M_0 \oplus E_k(IV),$$

$$C_i = M_i \oplus E_k(R_{i\text{-}1}),$$

где R_{i-1} – подмножество битов из предыдущего прохода до операции XOR.

Дешифрование в режиме OFB:

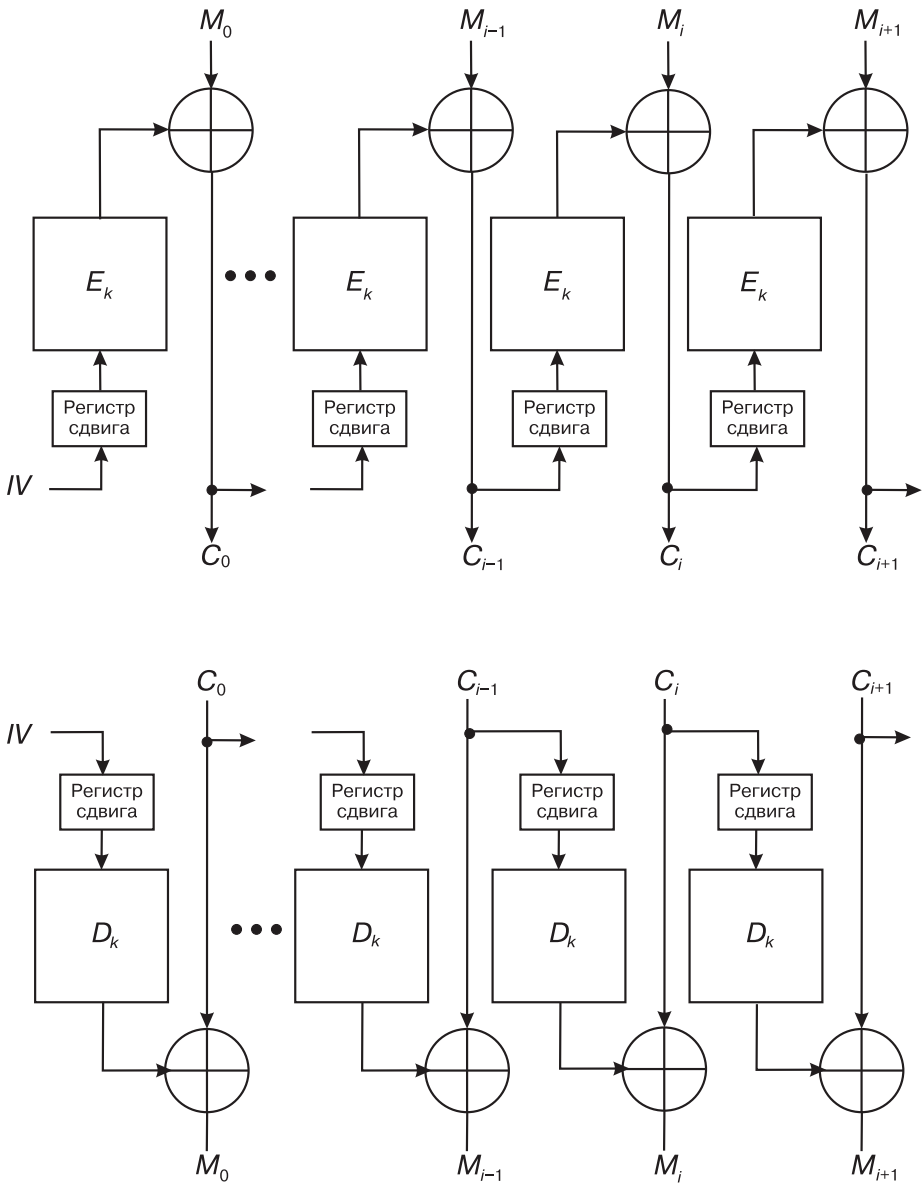


Рис. 3.6. Режим CFB

$$M_0 = C_0 \oplus D_k(IV),$$

$$M_i = C_i \oplus D_k(R_{i\text{-}1}),$$

где R_{i-1} – подмножество битов из предыдущего прохода до операции XOR.

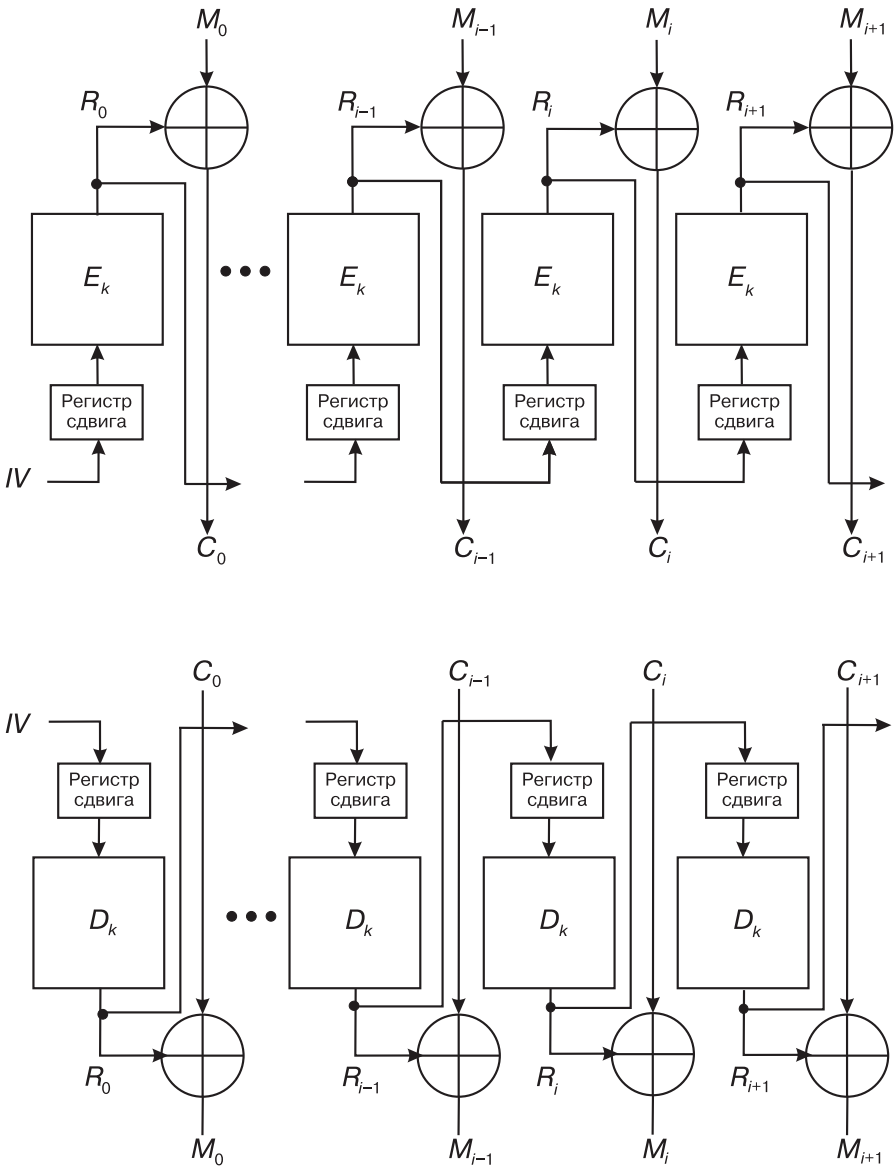


Рис. 3.7. Режим OFB

«Тройной» DES

В банковской сфере был принят стандарт ANSI X9.52[1], известный также под названиями Tripple DES («тройной» DES), TDES и 3DES. «Тройной» DES использовался в качестве временной альтернативы DES.

[1] Стандарты ANSI и ISO недоступны в свободном доступе и должны приобретаться (см. *webstore.ansi.org*).

Этот алгоритм основывается на обычном DES и обладает обратной совместимостью с его режимами. Усовершенствование заключается лишь в том, что каждый 64-битовый блок открытого текста шифруется три раза при помощи алгоритма DES с тремя разными ключами. Каждый блок открытого текста шифруется обычным алгоритмом DES первым ключом, полученный результат дешифруется вторым ключом, и, наконец, блок вновь шифруется третьим ключом. Этот алгоритм увеличивает эффективность DES трехкратным увеличением размера ключа, однако он также втрое увеличивает время, затрачиваемое на обработку.

Шифрование «тройным» DES:

$$C = E_{k_3}(D_{k_2}(E_{k_1}(M))).$$

Дешифрование «тройным» DES:

$$M = D_{k_1}(E_{k_2}(D_{k_3}(C))),$$

где k_1, k_2, k_3 – 56-битовые ключи.

Rijndael

Многие эксперты склоняются к мнению, что шифр DES приближается к окончанию своей полезной жизни. В конце 1970-х казалось, что 56-битовый ключ более чем достаточен для защиты чувствительных коммерческих данных, но оценка эта исходила из производительности и цены компьютеров того времени. К сожалению, в наше время, когда компьютеры стали значительно быстрее и дешевле, эта оценка уже неверна. Вообще говоря, вы всегда выбираете конкретную криптографическую технологию, исходя из тех затрат, которые должна будет понести атакующая сторона на преодоление защиты – уровень этих затрат должен делать атаку бессмысленной. Прошедшие десятилетия, в течение которых скорость переключения полупроводниковых цепей возрастала вдвое каждые 18 месяцев, а также прогресс криптоанализа, появление недорогой и производительной специализированной аппаратуры, возникновение моделей массовых параллельных вычислений через посредство сети Internet – все это радикально изменило ситуацию. «Тройной» DES немного поправил положение, но все же это была лишь временная «заплатка». Замена для DES была недавно найдена при помощи международного конкурса, и официальным преемником DES, известным под названием AES (Advanced Encryption Standard – Усовершенствованный стандарт шифрования), стал алгоритм Rijndael. Спецификацию Rijndael можно найти в документе FIPS-197[1]. Подобно своему предшественнику, этот шифр предназначен для защиты чувствительной, но не секретной информации. Ожидается, что со временем Rijndael начнет оказывать значительное влияние на коммерческие приложения, и в особенности – на финансовое программное обеспечение.

[1] Документ FIPS-197 доступен по адресу *http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf*

В отличие от DES, где размер ключа был фиксированным и равнялся 56 битам, алгоритм Rijndael способен использовать ключи размером 128, 192 или 256 бит (стандарты AES-128, AES-192 и AES-256, соответственно). В противоположность фиксированному 64-битовому блоку данных DES, Rijndael использует 128, 192 или 256-битовые блоки данных. Изначально Rijndael создавался для использования с другими размерами ключа и блока данных, но стандарт AES игнорирует все эти альтернативы. Число циклов шифрования в Rijndael зависит от размеров ключа и блока данных. Если размеры ключа и блока одновременно равны 128 битам, используется девять циклов. При размере ключа или блока, большем 128, если при этом ни один из них не больше 192, используется одиннадцать циклов. Наконец, если либо ключ, либо блок данных имеет размер 256 бит, то используется тринадцать циклов. Поскольку алгоритм предполагает еще один цикл в конце своего прохода, то речь, фактически, идет о 10, 12 или 14 циклах.

Алгоритм Rijndael существенно сложнее DES, который создавался в свое время с учетом простоты и эффективности аппаратной реализации. В алгоритме Rijndael используются сложные математически концепции такие, как алгебра полиномов с коэффициентами над полями Галуа GF(28), где обычные арифметические операции приобретают новое, причудливое значение. Такого рода детали выходят за рамки круга тем нашей книги, тем не менее, вы можете подробно с ними ознакомиться при помощи документа FIPS-197. К счастью, вам нет нужды вдаваться во все эти сложные вопросы, для того чтобы использовать алгоритм Rijndael в своих приложениях .NET, поскольку со всеми трудностями прекрасно справляется .NET Framework!

RC2

RC2 – это зарегистрированная торговая марка RSA Data Security, Incorporated. Алгоритм RC2 представляет собой симметричный блоковый шифр, разработанный Рональдом Ривестом (Ronald Rivest) в 1987 году, и использующий блоки входных данных размером 64 бита. Он был задуман, как будущая замена алгоритму DES, шифр с улучшенной производительностью и переменной длиной ключа (от 1 до 128 байт), что позволяет точно управлять криптографической стойкостью шифра. Регулировкой длины ключа шифр RC2 можно сделать как более, так и менее стойким, чем DES. Важным достоинством RC2 стал тот факт, что скорость его работы в типичных программных приложениях вдвое превышает скорость DES. Если верить людям из RSA Data Security, то аббревиатура RC расшифровывается, как «Ron's Code» (Код Рона) или «Rivest's Cypher» (Шифр Ривеста), но все это лишь неофициальные мнения.

RC2 был лицензирован для использования в нескольких коммерческих Internet-приложениях, включая Lotus Notes, Microsoft Internet Explorer, Outlook Express и Netscape Communicator. Безопасность сообщений электронной почты, согласно S/MIME (Secure/Multipurpose Internet Mail

Extensions – Безопасные многоцелевые расширения электронной почты), может обеспечиваться при помощи RC2, DES или Triple DES. Функции, которые RC2 выполняет в этих приложениях, сводятся к аутентификации посредством электронной подписи и конфиденциальности посредством шифрования.

RC2 был представлен в IETF в качестве RFC. Детальная спецификация RC2 содержится в документе RFC 2268, доступном по ссылке *http://www.ietf.org/rfc/rfc2268.txt*.

Программирование при помощи средств симметричной криптографии .NET

К вопросам безопасности данных корпорация Microsoft впервые обратилась в 1996-м году, анонсировав программный интерфейс Win32 Cryptography API (CryptoAPI) в системе Windows NT. Хотя интерфейс CryptoAPI обеспечивал полную и всестороннюю поддержку задач криптографии, пользоваться им было очень трудно. Для того чтобы понимать смысл множества параметров во множестве криптографических функций этого API, необходимо было хорошо разбираться в криптографии. К тому же интерфейс не являлся объектно-ориентированным, поскольку написан он был на обычном C, и для решения даже простейших задач необходимо было вызвать множество функций. К счастью, .NET Security Framework в огромной степени упрощает нашу работу, сводя все к одному, очень элегантному набору классов в пространстве имен **System.Security.Cryptography**.

Основные криптографические классы

В этой главе мы рассматриваем только программирование с использованием симметричной криптографии, однако, поскольку это наша первая встреча с .NET Security Framework, мы вкратце рассмотрим все общие классы, представленные в пространстве имен **System.Security.Cryptography**. Приведенный ниже список включает в себя классы, имеющие первостепенное значение, поскольку они составляют базу для всех аспектов программирования в области криптографии. Мы не рассматриваем здесь классы, имеющее отношение к безопасности доступа к коду (Code Access Security) или безопасности на основе механизма ролей (Role-Based Security). Классы, связанные с безопасностью и относящиеся к пространству **System.Security**, рассматриваются в главах 7 и 8.

- Классы, производные от **SymmetricAlgorithm**, инкапсулируют симметричные алгоритмы такие, как DES или Rijndael.
- Классы, производные от **AsymmetricAlgorithm**, инкапсулируют асимметричные (RSA и DSA).

- **CryptoStream** соединяет потоки исходных данных с криптографическим алгоритмом.
- **CspParameters** инкапсулирует параметры, специфичные для алгоритма, которые можно сохранить или загрузить посредством поставщика услуг криптографии (CSP).
- **HashAlgorithm** представляет базовый класс, из которого производятся все алгоритмы, связанные с хешами.
- **RandomNumberGenerator** представляет базовый класс для получения программных генераторов псевдослучайных чисел (PRNG).
- **ToBase64Transform** и **FromBase64Transform** используются для преобразования между потоком байт и представлением в виде 64-битовых блоков.
- **CryptographicException** инкапсулирует информацию об ошибках, возникающих при выполнении криптографических операций.

Поскольку все эти классы содержатся в пространстве имен **System.Security.Cryptography**, не забывайте объявлять это пространство при помощи оператора **using** в своих программах.

Класс SymmetricAlgorithm

Классы .NET Framework, реализующие симметричные алгоритмы, являются производными от класса **SymmetricAlgorithm**. Этот абстрактный класс обладает несколькими защищенными полями, которые напрямую недоступны для методов не производных классов. Тем не менее, эти защищенные поля доступны через посредство виртуальных свойств, реализованных в конкретном производном классе. Например, защищенное **int**-поле **BlockSizeValue** доступно посредством виртуального свойства **BlockSize** (способ доступа зависит от способа использования конкретного производного класса). Таким образом, попытка присвоить свойству значение, которое запрещено для конкретного используемого симметричного алгоритма, вызовет исключение **CryptographicException**, зависящее от этого алгоритма (т. е. от используемого производного класса). В любом случае защищенное поле и соответствующее ему публичное виртуальное свойство относятся к одному типу данных, а их имена отличаются только суффиксом Value.

Публичные виртуальные свойства, определенные в классе **SymmetricAlgorithm**, приведены в таблице 3.1.

Класс **SymmetricAlgorithm** снабжен только одним публичным конструктором, не требующим параметров. Этот конструктор инициализирует создаваемый экземпляр класса случайно выбранным секретным ключом. Конечно класс **SymmetricAlgorithm** поддерживает стандартные методы **Equals**, **Finalize**, **GetHashCode**, **ToString**, **GetType** и **MemberwiseClone**, определенные в базовом классе **Object**. В добавление к этому **SymmetricAlgorithm** поддерживает публичные методы, описание которых приведено в таблице 3.2.

Таблица 3.1. Публичные виртуальные свойства класса SymmetricAlgorithm

Публичное свойство	Описание
BlockSize	Возвращает или задает размер блока в битах для алгоритма, что соответствует количеству данных, подвергающихся шифрованию или дешифрованию за одну операцию. Сообщения, размер которых превышает это значение, разбиваются на несколько блоков. Последний блок при этом должен быть дополнен до достижения этого размера. Разрешенные варианты размера блока определяются свойством **LegalBlockSizes** для каждого из симметричных алгоритмов в отдельности. Значение этого свойства относится к данным типа **int**
FeedBackSize	Возвращает или задает размер данных обратной связи в битах для алгоритма. Это количество данных, возвращаемых в обратной связи при операциях шифрования и дешифрования. Обратная связь требуется для работы в операционных режимах OFB и CFB. Разрешенные размеры данных обратной связи зависят от конкретного алгоритма, однако не могут превышать размер блока. Это свойство относится к типу **int**
IV	Возвращает или задает вектор инициализации для симметричного алгоритма, что требуется в операционном режиме CBC. Значение этого свойства представляет собой массив значений типа **byte**
Key	Возвращает или задает секретный ключ для использования симметричным алгоритмом при шифровании или дешифровании. Значение этого свойства представляет собой массив значений типа **byte**
KeySize	Возвращает или задает размер секретного ключа в битах для использования симметричным алгоритмом при шифровании или дешифровании. Разрешенные варианты размера блока определяются свойством **LegalKeySizes** для каждого из симметричных алгоритмов в отдельности. Это свойство относится к типу **int**
LegalBlockSizes	Возвращает варианты размера блоков, поддерживаемые симметричным алгоритмом. Это свойство, доступное только для чтения, представляет собой массив значений типа **KeySizes** (отдельного типа BlockSizes не существует). Только такой размер блока, который совпадает с одним из элементов этого массива, допустим для данного алгоритма

Продолжение таблицы см. на следующей странице

Публичное свойство	Описание
LegalKeySizes	Возвращает варианты размера ключа, поддерживаемые симметричным алгоритмом. Это свойство, доступное только для чтения, представляет собой массив значений типа **KeySizes**. Для алгоритма допустим только такой размер ключа, который совпадает с одним из элементов этого массива
Mode	Возвращает или задает операционный режим для симметричного алгоритма. Это свойство относится к типу **CipherMode** и допускает значения ECB (каждый блок обрабатывается отдельно), CBC (вводится обратная связь между блоками), а также CFB или OFB (используется сдвиговый регистр, позволяющий обрабатывать порции данных размером меньше блока). Наконец, режим CTS является вариацией режима CBC
Padding	Возвращает или задает режим дополнения блоков для симметричного алгоритма, которое необходимо для расширения последнего, неполного блока данных до стандартного размера. Это свойство относится к типу **PaddingMode** и может принимать одно из трех значений: **PKCS7** соответствует байтам дополнения, каждый из которых равен общему числу таких байтов, **Zeros** соответствует дополнению нулями и **None** обозначает отсутствие дополнения (алгоритм шифрования при этом использует специальную схему дополнения, игнорируя данное свойство)

Классы, производные от SymmetricAlgorithm

Вы никогда не будете иметь дело напрямую с объектом типа **SymmetricAlgorithm**, поскольку это абстрактный класс, создание экземпляров которого невозможно. Вместо этого вы будете работать с производными классами, реализующими конкретные публичные свойства, а также абстрактные и виртуальные методы **SymmetricAlgorithm**, каждый раз в зависимости от используемого алгоритма. На рисунке 3.8 изображена иерархия классов симметричного алгоритма.

Из рисунка видно, что перечисленные ниже классы являются производными от **SymmetricAlgorithm**. Технику программирования с использованием этих классов мы вскоре рассмотрим.

- **DES** – абстрактный класс, инкапсулирующий симметричный алгоритм DES.
- **TripleDES** – абстрактный класс, инкапсулирующий симметричный алгоритм Triple DES, который обладает обратной совместимостью с DES, и значительно превосходит его по криптографической стойкости.

Таблица 3.2. Публичные методы класса SymmetricAlgorithm

Публичный метод	Описание
Clear	Этот метод вызывает метод **Dispose**, освобождая ресурсы, ранее занятые симметричным алгоритмом. Возвращает значение **void**
Create	Этот перегруженный статический метод используется для создания объекта, производного от **SymmetricAlgorithm**, который выполняет шифрование и дешифрование. Метод возвращает ссылку на объект типа **SymmetricAlgorithm**
Create-Decryptor	Этот перегруженный статический метод используется для создания объекта decryptor, используя при этом заданный ключ и явно или неявно заданный вектор инициализации. Метод возвращает ссылку на интерфейс **ICryptoTransform**, который можно использовать для трансформации блоков данных
Create-Encryptor	Этот перегруженный статический метод используется для создания объекта encryptor, используя при этом заданный ключ и явно или неявно заданный вектор инициализации. Метод возвращает ссылку на интерфейс **ICryptoTransform**, который можно использовать для трансформации блоков данных
Equals	Перегруженный виртуальный метод, наследуемый от **Object** и используемый для сравнения на равенство двух объектов, производных от **SymmetricAlgorithm**. Метод возвращает значение типа **bool**
GeneratedIV	Абстрактный метод, который будучи переопределен в производном классе, генерирует случайный вектор инициализации. Вызов метода возвращает **void**, однако, сгенерированный при этом вектор становится значением IV по умолчанию для объекта, производного от **SymmetricAlgorithm**
Generated-Key	Абстрактный метод, который будучи переопределен в производном классе, генерирует случайный ключ. Вызов метода возвращает **void**, однако, сгенерированный при этом ключ становится ключом по умолчанию для объекта, производного от **SymmetricAlgorithm**
GetHash-Code	Метод, наследуемый от **Object**. Создает значение хеша для объекта, производного от **SymmetricAlgorithm**. Возвращает значение типа **int**
GetType	Метод, наследуемый от **Object**. Возвращает тип объекта, производного от **SymmetricAlgorithm**. Возвращает значение типа **Type**
ToString	Виртуальный метод, наследуемый от **Object**. Возвращает объект **String**, представляющий объект, производный от **SymmetricAlgorithm**
ValidKeySize	Метод определяет, является ли допустимым заданный размер ключа для текущего алгоритма. Возвращает значение типа **bool**

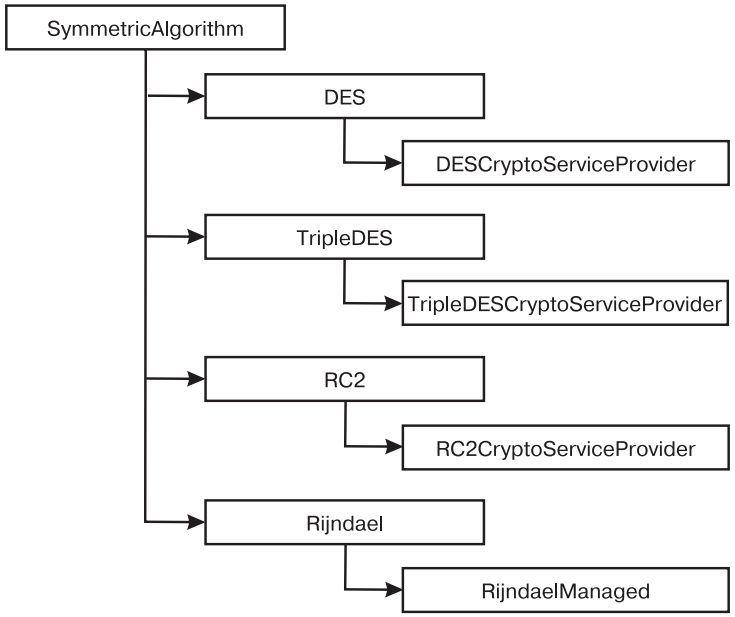


Рис. 3.8. Иерархия симметричных алгоритмов

- **Rijndael** – абстрактный класс, инкапсулирующий симметричный алгоритм Rijndael, являющийся новым стандартом, пришедшим на смену DES.
- **RC2** – абстрактный класс, инкапсулирующий симметричный алгоритм RC2, разработанный Рональдом Ривестом в качестве потенциальной замены для DES.

Из всех этих абстрактных классов далее производятся конкретные классы, реализующие все поддерживаемые симметричные алгоритмы. Платформа .NET Framework обеспечивает по одному конкретному классу на каждый алгоритм, тем не менее, провайдеры криптографических служб могут реализовать собственные классы для реализации собственных, частных расширений. В каждом случае реализуются публичные свойства, и, конечно, публичные методы, описанные для класса **SymmetricAlgorithm**.

По соображениям безопасности каждый конкретный класс «запечатан», то есть, иными словами, наследование из него запрещено. В стандартной поставке предусмотрено четыре конкретных класса:

- **TripleDESCryptoServiceProvider** – класс-оболочка, предоставляющий доступ к стандартной реализации поставщика услуг криптографии по алгоритму Triple DES, которая является частью Win32 CryptoAPI.
- **RC2CryptoServiceProvider** – класс-оболочка, предоставляющий доступ к стандартной реализации поставщика услуг криптографии по алгоритму RC2, которая является частью Win32 CryptoAPI.

- **DESCryptoServiceProvider**[1] – класс-оболочка, предоставляющий доступ к нижележащей стандартной реализации поставщика услуг криптографии по алгоритму DES, которая является частью Win32 CryptoAPI.
- **RijndaelManaged** – управляемый класс, реализующий алгоритм Rijndael. Этот класс не является оболочкой CryptoAPI.

Примеры программирования с использованием симметричных алгоритмов

В этом разделе мы рассмотрим пример программы **SymmetricAlgorithms**, прилагающийся к этой главе. Этот пример демонстрирует, как можно шифровать и дешифровать данные при помощи каждого из четырех производных от **SymmetricAlgorithm** классов. На рисунке 3.9 изображено окно программы, шифрующей и дешифрующей текстовое сообщение.

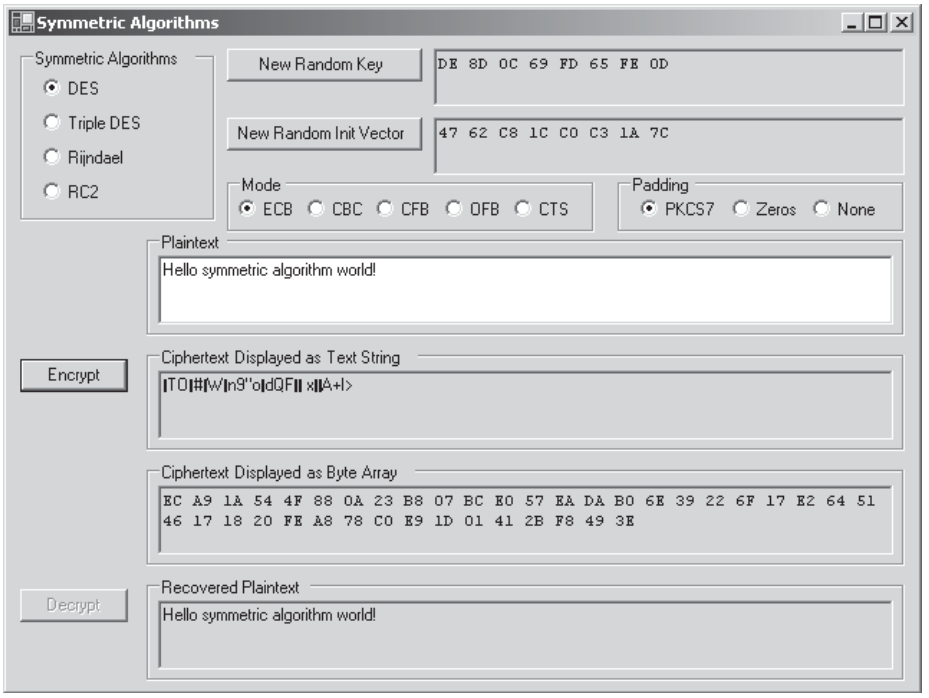


Рис. 3.9. Пример программы SymmetricAlgorithms

[1] Классы с именами, оканчивающимися на CryptoServiceProvider, представляют собой оболочки, использующие для своей реализации Win32 CryptoAPI. Как правило, это относится к старым алгоритмам. Классы с именами, оканчивающимися на Managed, являются полностью новыми реализациями, написанными на управляемом коде. Как правило, такие имена относятся к новым алгоритмам, хотя старые алгоритмы также могут быть реализованы в двух вариантах.

Давайте рассмотрим код программы **SymmetricAlgorithms**. Ее устройство немного противоречиво с точки зрения логики, поскольку она шифрует и дешифрует данные в пределах одной и той же экранной формы. Более реалистический сценарий потребовал бы передачи или промежуточного хранения шифрованного текста, но удобней и наглядней будет расположить все этапы процесса в одном месте. Для того чтобы сымитировать реальный сценарий, мы добавили пять полей к классу **SymmetricAlgorithm**, производному от **Form**.

Эти поля используются для передачи данных между шифрующими и дешифрующими методами. В нашем примере связью между участниками процесса будут, скорее эти поля, чем собственно данные. В реальной жизни ключ, вектор инициализации, режимы шифрования и дополнения должны быть заранее согласованы между корреспондентами, а через коммуникационный канал должны передаваться только зашифрованные данные. Поля, о которых идет речь – это байтовые массивы с именами **Key**, **IV**, **Mode**, **Padding** и **cipherbytes**.

```
//переменные, связывающие процессы шифрования и дешифрования
byte[] Key;
byte[] IV;
CipherMode Mode;
PaddingMode Padding;
byte[] cipherbytes;
```

При первоначальной загрузке формы для обработки этого события вызывается следующий метод. Поля Key и IV инициализируются случайными значениями ключа и вектора инициализации. Режим шифрования и соглашение о дополнении выбираются, согласно начальному состоянию переключателей.

```
private void Form1_Load(
  object sender, System.EventArgs e)
{
  //инициализировать ключ, вектор, режим и дополнение
  GenIV();
  GenKey();
  EstablishMode();
  EstablishPadding();
}
```

В программе имеется несколько методов (включая GenIV и GenKey), которые необходимы для создания объекта типа «симметричный алгоритм». В основном эта функция инкапсулирована в методе **CreateSymmetricAlgorithm**. Этот метод создает и возвращает производный от **SymmetricAlgorithm** объект, руководствуясь положением переключателя на форме. В зависимости от положения переключателя вызывается соответствующий статический метод **Create**.

```
SymmetricAlgorithm CreateSymmetricAlgorithm()
{
        //создания объекта типа "симметричный алгоритм"
        if (radioButtonRC2.Checked == true)
                return RC2.Create();
        if (radioButtonRijndael.Checked == true)
                return Rijndael.Create();
        if (radioButtonDES.Checked == true)
                return DES.Create();
        if (radioButtonTripleDES.Checked == true)
                return TripleDES.Create();
        return null;
}
```

Наиболее важную роль в программе играют те методы, которые собственно выполняют шифрование и дешифрование данных при помощи выбранного алгоритма, руководствуясь текущими значениями ключа и вектора инициализации. Фактически, эти методы являются обработчиками событий кнопок: это **buttonEncrypt_Click** и **buttonDecrypt_Click**. Обратите внимание, метод **CreateSymmetricAlgorithm** вызывается заново в обоих случаях. Классы **MemoryStream** и **CryptoStream**, которые также здесь используются, будут описаны немного ниже. Вот исходный код методов шифрования и дешифрования:

```
private void buttonEncrypt_Click(
        object sender, System.EventArgs e)
{
        //обслуживаем пользовательский интерфейс
        ...
        //задать симметричный алгоритм
        SymmetricAlgorithm sa =
                CreateSymmetricAlgorithm();

        //использовать текущие ключ и вектор
        sa.Key = Key;
        sa.IV = IV;

        // использовать текущие операционный режим и режим
        //  дополнения
        sa.Mode = Mode;
        sa.Padding = Padding;

        //задать поток шифрования
        MemoryStream ms = new MemoryStream();
        CryptoStream cs = new CryptoStream(
                ms,
                sa.CreateEncryptor(),
                CryptoStreamMode.Write);
```

```
        //записать байты открытого текста в поток шифрования
        byte[] plainbytes =
                Encoding.UTF8.GetBytes(textPlaintext.Text);
        cs.Write(plainbytes, 0, plainbytes.Length);
        cs.Close();
        cipherbytes = ms.ToArray();
        ms.Close();

        //отобразить зашифрованный текст в виде строки символов
        textCiphertext.Text =
        ...
        // отобразить зашифрованный текст в виде массива байтов
        ...

        //обслуживаем пользовательский интерфейс
        ...
}

private void buttonDecrypt_Click(
        object sender, System.EventArgs e)
{
        //задать симметричный алгоритм
        SymmetricAlgorithm sa =
                CreateSymmetricAlgorithm();

        //использовать текущие ключ и вектор
        sa.Key = Key;
        sa.IV = IV;

        // использовать текущие операционный режим и режим
        // дополнения
        sa.Mode = Mode;
        sa.Padding = Padding;

        //задать поток шифрования
        MemoryStream ms = new MemoryStream(cipherbytes);
        CryptoStream cs = new CryptoStream(
                ms,
                sa.CreateDecryptor(),
                CryptoStreamMode.Read);

        //читать шифрованные байты из потока
        byte[] plainbytes =
                new Byte[cipherbytes.Length];
        cs.Read(plainbytes, 0, cipherbytes.Length);
        cs.Close();
        ms.Close();
        //отобразить восстановленный открытый текст
        ...
        //обслуживаем пользовательский интерфейс
        ...
}
```

Криптографические потоки

Среда выполнения CLR поддерживает криптографические функции, ориентированные на потоки. Класс, инкапсулирующий криптографический поток, как нетрудно догадаться, называется **CryptoStream**. Любую криптографическую операцию, создающую объект **CryptoStream**, можно соединить с другими объектами **CryptoStream**. Чтобы выстроить цепочку из криптографических потоков, можно подключать выход одного объекта к входу другого и так далее, при этом не потребуется какого-либо промежуточного хранения данных. **CryptoStream** – это удобный класс, позволяющий читать и записывать данные через криптографический поток точно таким же образом, как если бы это был файл или сокет. Создать экземпляр класса **CryptoStream** можно для шифрования (режим записи) или для дешифрования (режим чтения). Поскольку в нашем примере **SymmetricAlgorithms** для представления шифрованных и дешифрованных данных используется массив байтов в памяти, для обеспечения операций ввода-вывода в отношении этих байтов используется класс **MemoryStream**. Вот конструктор объекта **CryptoStream**:

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

Передавая вновь созданный объект **Stream** и объект **ICryptoTransform**, ассоциированный с нужным нам симметричным алгоритмом, а также значение **CryptoStreamMode.Write**, задающее направление ввода-вывода, мы создаем в результате объект **CryptoStream**, который будет шифровать все записываемые в него данные.

```
MemoryStream ms = new MemoryStream();
CryptoStream cs = new CryptoStream(
       ms,
       sa.CreateEncryptor(),
       CryptoStreamMode.Write);
```

Задав вывод объекта **CryptoStream**, мы можем записывать в него байты, которые будут автоматически шифроваться объектом **ICryptoTransform** и записываться в ассоциированный с ним объект **Stream**, который в нашем примере представлен объектом **MemoryStream**, хотя это мог бы быть файл или сокет.

```
cs.Write(plainbytes, 0, plainbytes.Length);
cs.Close();
```

Передавая объект **MemoryStream**, который основан на массиве шифрованных байт в памяти, а также объект **ICryptoTransform** и признак режима **CryptoStreamMode.Read**, мы создаем в результате объект **CryptoStream**, расшифровывающий все записываемые в него данные.

```
MemoryStream ms = new MemoryStream(cipherbytes);
CryptoStream cs = new CryptoStream(
        ms,
        sa.CreateDecryptor(),
        CryptoStreamMode.Read);
```

Задав ввод объекта **CryptoStream**, мы можем считывать байты из объекта **Stream**, которые будут автоматически расшифровываться **ICryptoTransform** и помещаться в байтовый массив.

```
cs.Read(plainbytes, 0, cipherbytes.Length);
cs.Close();
```

Когда вы запустите пример программы, то увидите, что определенные сочетания режимов шифрования и дополнения доступны не для всех алгоритмов. Например, если вы выберете режим шифрования OFB и любой из четырех алгоритмов, то будет сгенерировано исключение **CryptographicException** с выдачей сообщения о том, что *режим с обратной связью по выходу (OFB) в данной реализации не поддерживается* (см. рисунок 3.10).

Другой пример проблем подобного вида: если вы попытаетесь использовать алгоритм Rijndael с режимом дополнения **Zeros** или **None**, то возникнет другое исключение **CryptographicException** с выдачей сообщения о том, что *во входном буфере недостает данных* (см. рисунок 3.11). Полезно увидеть такие исключения в учебной программе, однако на практике, конечно, подобных ситуация следует избегать.

Выбор надежных ключей

В классах **DES** и **TripleDES** имеется публичный статический метод **IsWeakKey**, который, получив на входе байтовый массив, представляющий собой ключ, возвращает булево значение. Как нетрудно догадаться по имени метода, он проверяет ключ на возможную криптографическую слабость.

Как также нетрудно догадаться, под «слабым» ключом понимается такой ключ, при котором раскрытие шифра будет облегчено. Слабый ключ опознается по следующему признаку: если зашифровать открытый текст слабым ключом, то повторное шифрование уже зашифрованного текста тем же ключом даст в результате открытый текст. Поскольку 3DES полностью основывается на DES, то слабый ключ DES приводит также к слабости ключа 3DES. В алгоритмах Rijndael и RC2 понятие слабого ключа отсутствует, поэтому для классов **Rijndael** и **RC2** необходимости в методе **IsWeakKey** нет.

Было обнаружено четыре слабых ключа DES, и, очевидно, использования этих ключей следует избегать. Впрочем, важность проблемы слабых ключей не стоит преувеличивать. Для 56-битового ключа DES возможно 256 вариантов, в то время как известных слабых ключей всего 4, то есть 22. Таким образом, вероятность получить слабый ключ при случайном его выборе составляет 2-54, а это крайне ничтожная величина. Известны также 12 ключей, считающихся «полуслабыми», но вероятность случайно набрести на такой ключ также пренебрежимо мала.

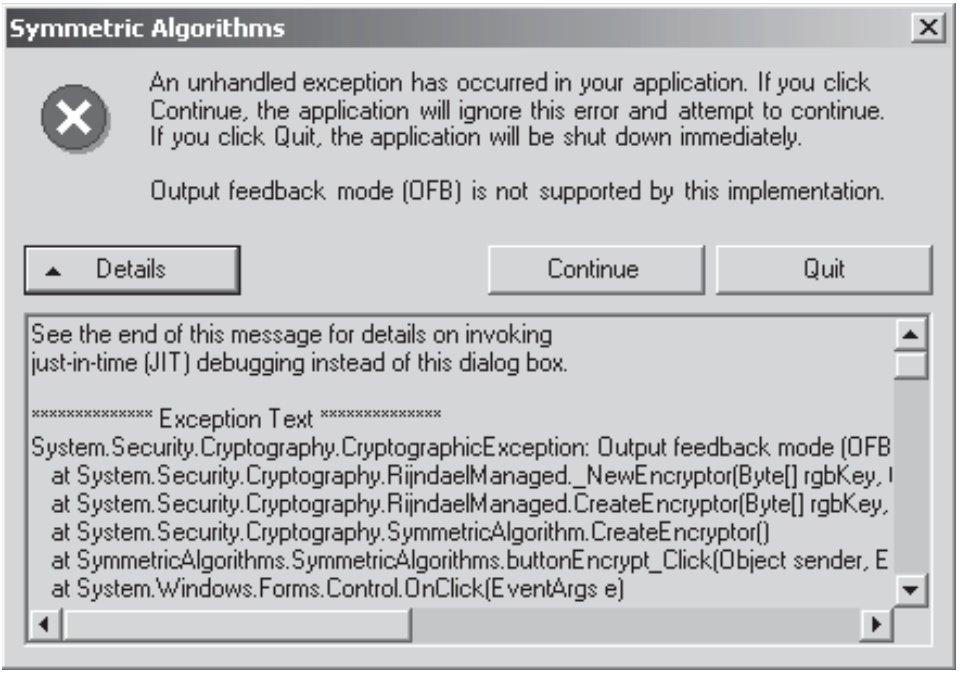


Рис. 3.10. Метод OFB в текущей реализации не поддерживается

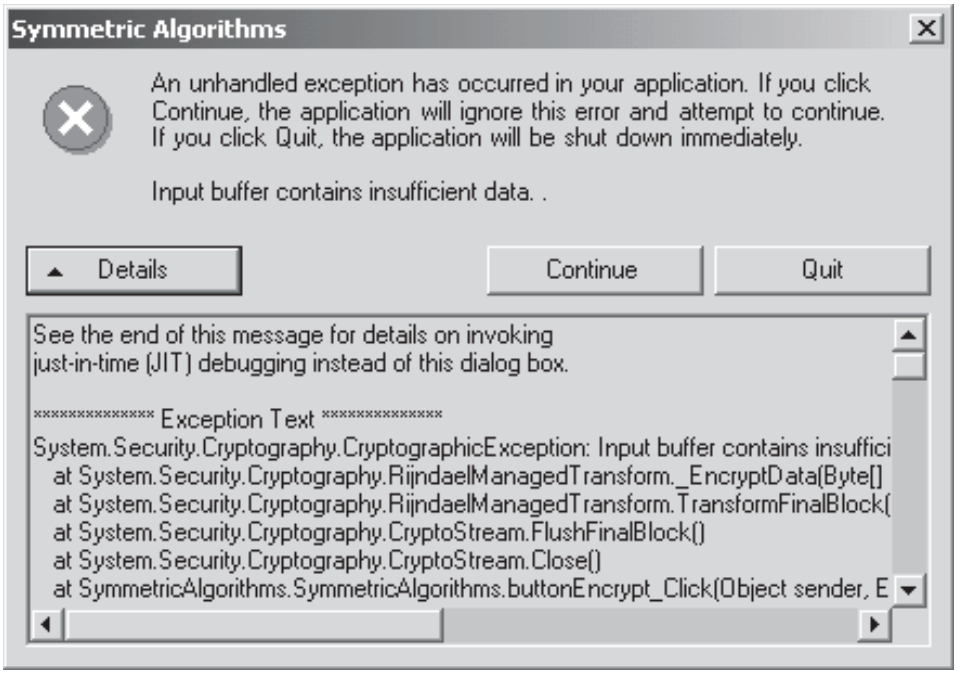


Рис. 3.11. Алгоритм Rijndael несовместим режимами дополнения Zeros и None

В любом случае большинство программ используют секретный ключ, который сгенерирован случайно конструктором того объекта, который реализует используемый симметричный алгоритм. При этом слабый ключ сгенерирован заведомо не будет. Если вы создаете ключ вызовом метода **GenerateKey**, то вы также гарантированы от получения слабого ключа. В тех случаях, когда вы пользуетесь ключом, полученным каким-то иным образом, например, ключом от провайдера CSP или иного внешнего источника, классы **DES** и **TripleDES** все равно сгенерируют исключение **CryptographicException** при попытке шифрования слабым ключом. Лишь для тех, кто любит все проверять сам, предусмотрен метод **IsWeakKey**, позволяющий проверить ключ на «слабость».

Проблемы передачи ключей

Одна из проблем, связанных с использованием симметричных алгоритмов, состоит в том, что для каждого корреспондента на компьютере должен иметься отдельный ключ, который вместе со своей второй копией на машине корреспондента обеспечивает секретность коммуникаций между двумя корреспондентами. Это означает, что с ростом числа корреспондентов быстро растет число секретных ключей, которые необходимо хранить. Если коммуникации включают в себя N участников-корреспондентов, то максимальное потребное число ключей составит сумму всех чисел от 1 до $N - 1$. На рисунке 3.12 показано, как выглядит потребность в ключах для случаев $N = 1$ и $N = 4$. Обратите внимание на тот факт, что число потребных ключей растет быстрее числа сообщающихся корреспондентов. В следующей главе, посвященной асимметричным алгоритмам, мы увидим эффективное решение этой проблемы, такое, что число необходимых секретных ключей остается равным числу участников.

Есть и другая проблема, связанная с симметричной криптографией. Несмотря на неудобства, связанные с быстрым ростом числа необходимых ключей, симметричные алгоритмы остаются мощным, быстрым и, в целом, наиболее подходящим средством обеспечения секретности: особенно их превосходство над асимметричной криптографией заметно при обработке больших массивов данных.

Следовательно, нам необходима возможность обмениваться симметричными сеансовыми ключами. Проблема состоит в том, что передача сеансового ключа требует секретного канала, позволяющего одному участнику передать ключ своему корреспонденту. Это классическая ситуация из «Уловки-22[1]», поскольку нам нужен секретный канал для того, чтобы

[1] Если вы способны сохранить в секрете передаваемый ключ, то у вас уже есть секретный канал, и симметричная криптосистема, вам, собственно, уже не нужна, поскольку вы можете просто передавать вместо ключа свои данные по этому секретному каналу. Возможные решения состоят в использовании секретного канала, который существует лишь небольшое время, в передаче ключа доверенным курьером, или в использовании протокола обмена асимметричными ключами, такими, как RSA или протокол обмена ключами Диффи-Хеллмана.

установить секретный канал! К счастью, для этой головоломки существуют вполне приемлемые решения.

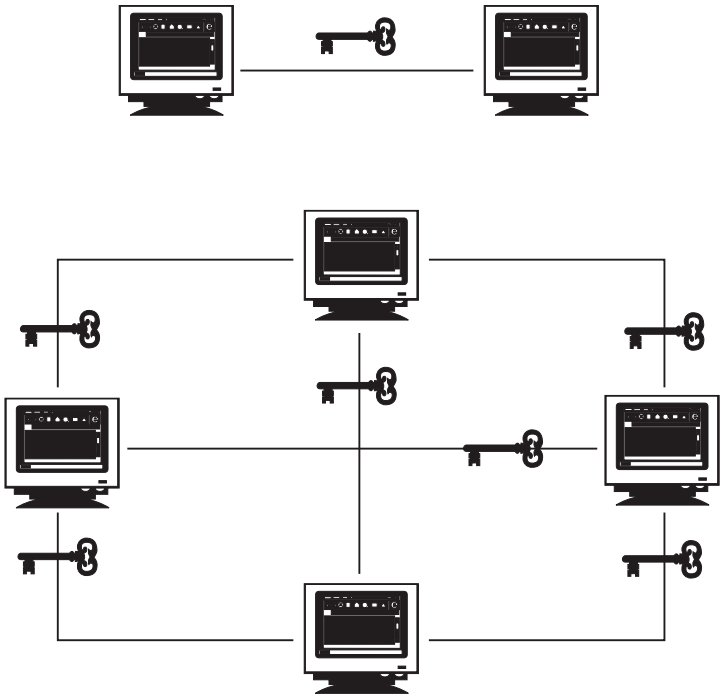


Рис. 3.12. Рост числа необходимых ключей, сопровождающий рост числа участников коммуникаций в симметричной криптографии

Пример программы **NaiveKeyExchange** (исполняемые файлы **Sender.exe** и **Receiver.exe**) иллюстрирует наивный подход к проблеме обмена ключами, в котором ключи передаются через нешифрованный дисковый файл. Аналогичным образом можно установить сетевое соединение через обычные гнезда (сокеты) и передавать ключи через это соединение. Проблема очевидна: секретный ключ в таких обстоятельствах не может оставаться секретным. Фактически, он доступен для злоумышленника. Чтобы усовершенствовать эту схему обмена ключами, необходимо принять какие-то меры к обеспечению секретности, например, зашифровать ключ. И опять мы получаем парадокс «Уловки-22»[1]! Решение мы найдем в следующей главе, использовав классы **RSACryptoServiceProvider** и **CspParameters** для шифрования сеансовых симметричных ключей очень удобным и надежным образом.

[1] Этот парадокс проистекает из необходимости зашифровать ключ, который должен использоваться для шифрования данных. Подразумевается, что для безопасной передачи необходимо его зашифровать каким-то другим ключом, который, в свою очередь, для безопасной передачи также нужно зашифровать каким-то другим ключом, и так далее до бесконечности.

Шифрованные хеши и целостность сообщения

Основными задачами криптографии являются обеспечение конфиденциальности, обеспечение целостности и аутентификация. Мы увидели, как можно обеспечить конфиденциальность при помощи симметричной криптографии. Как насчет целостности и аутентификации? Обеспечение целостности подразумевает, что несанкционированное изменение сообщения будет предотвращено или, по крайней мере, обнаружено. Аутентификация означает, что идентичность источника сообщения надежно подтверждена. Как оказалось, для решения этих задач можно использовать как асимметричные, так и симметричные алгоритмы. Асимметричные алгоритмы мы будем рассматривать в 4-й главе, а сейчас посмотрим, как в этих целях можно использовать симметричную криптографию.

Предположим, Алиса и Боб являются единственными людьми, располагающими значением некоторого секретного ключа. Допустим также, что Алиса и Боб доверяют друг другу в том, что ключ не получит какая-либо третья сторона. Если один из них получит шифрованное сообщение, которое после расшифровки этим секретным ключом превратится в действительный открытый текст, то этот факт будет означать, что сообщение отправлено вторым владельцем ключа. Причина кроется в том, что без знания секретного ключа создать такое сообщение крайне трудно.

Единственная неясность здесь – это точное значение выражения «действительный открытый текст». Наиболее эффективный способ подтвердить действительность текста состоит в вычислении криптографического хеша. До тех пор пока сообщение не изменено, повторное вычисление хеша будет давать такое же значение. Если вычисленное значение хеша не изменилось, то это с крайне высокой вероятностью подтверждает неизменность текста.

Разумеется, недостаточно просто вычислить хеш, поскольку атакующая сторона также может вычислить хеш измененного сообщения и подменить им оригинальный хеш. Это не обеспечит ни целостности, ни аутентификации. Однако, если Алиса вначале вычислит хеш, а затем зашифрует получившееся значение секретным ключом, то проблема будет решена. Алиса отправляет Бобу сообщение (зашифрованное или открытое – проблему конфиденциальности мы здесь не рассматриваем) и сопровождает его зашифрованным хешем сообщения. Боб дешифрует полученный хеш и вычисляет хеш полученного сообщения заново, сравнивая оба значения. Если вычисленный хеш совпадет с расшифрованным хешем, это с весьма высокой вероятностью подтвердит аутентичность автора и целостность сообщения. Атакующая сторона оказывается в незавидном положении. Если мы попытаемся, изменив сообщение, вычислить заново его хеш, то зашифровать этот хеш без секретного ключа мы не сможем, а оригинальный зашифрованный хеш не совпадет после расшифровки со вновь вычисленным хешем подмененного сообщения.

Здесь необходимо сказать несколько слов о понятии криптографический хеш. Давайте начнем с немного более раннего понятия. Хеш-функцией называют любую функцию, которая преобразует входные данные произвольного размера в выходные данные стандартного, обычно, относительно небольшого размера. Функция вычисления циклической контрольной суммы CRC32 – простой пример хеш-функции, которая давно используется с целью обнаружения ошибок передачи и хранения. Криптографическая хеш-функция в своей основе устроена так же, но обладает при этом некоторыми дополнительными свойствами. Например, крайне трудно найти два разных варианта входных данных, которые дадут одинаковые хеши на выходе функции (для циклической контрольной суммы CRC32 это не так). Также это должна быть односторонняя функция, то есть преобразование, которое она выполняет, должно быть необратимым: легко вычислить хеш сообщения, но восстановить сообщение по его хешу невозможно. Хорошая криптографическая хеш-функция должна также производить непредсказуемые и значительные изменения в хеше при любом, даже однобитовом изменении в тексте входного сообщения. Криптографические хеши используют для такого представления данных произвольного размера, которое в высокой степени характеризует эти данные и которое трудно подделать, что делает хеш аналогом обычной человеческой подписи.

Перечисленные ниже классы криптографических хешей .NET Framework являются производными абстрактного класса **HashAlgorithm**. Везде слева приводится имя абстрактного класса, а справа – имя производного конкретного класса. Буквы MD в имени «MD5» расшифровываются, как «Message Digest» – дайджест сообщения, что является синонимом криптографического хеша. Аббревиатура SHA означает «Secure Hash Algorithm» – алгоритм безопасного хеша – этот алгоритм был первоначально разработан в NIST и в NSA для использования в алгоритме электронной подписи DSA.

- **KeyedHashAlgorithm** производит **HMACSHA1** и **MACTripleDES**;
- **MD5** производит **MD5CryptoServiceProvider**;
- **SHA1** производит **SHA1Managed** и **SHA1CryptoServiceProvider**;
- **SHA256** производит **SHA256Managed**;
- **SHA384** производит **SHA384Managed**;
- **SHA512** производит **SHA512Managed**.

Пример программы **EncryptedHash** показывает, как использовать шифрованный криптографический хеш сообщения для обеспечения его целостности. В этом примере открытый текст вначале обрабатывается хеш-алгоритмом MD5, а затем полученный хеш шифруется симметричным алгоритмом DES. В реалистическом сценарии само сообщение и его зашифрованный хеш передавались бы по некоторому каналу другой стороне, которая расшифровала бы хеш и сверила бы его с вновь вычисленным хешем сообщения. Для простоты и наглядности в этом примере функции передающей и принимающей стороны совмещены в одной программе.

```
...
//Создать хеш сообщения по алгоритму MD5
   MD5CryptoServiceProvider md5 =
   new MD5CryptoServiceProvider();
byte[] hashBytes =
   md5.ComputeHash(plaintextBytes);
...
//зашифровать хеш при помощи DES
SymmetricAlgorithm sa = DES.Create();
MemoryStream msEncrypt =
        new MemoryStream();
CryptoStream csEncrypt = new CryptoStream(
        msEncrypt,
        sa.CreateEncryptor(),
        CryptoStreamMode.Write);
csEncrypt.Write(hashBytes,
        0, hashBytes.Length);
csEncrypt.Close();
byte[] encryptedHashBytes = msEncrypt.ToArray();
msEncrypt.Close();
...
//дешифровать хеш при помощи DES
MemoryStream msDecrypt =
        new MemoryStream(encryptedHashBytes);
CryptoStream csDecrypt = new CryptoStream(
        msDecrypt,
        sa.CreateDecryptor(),
        CryptoStreamMode.Read);
byte[] decryptedHashBytes =
        new Byte[encryptedHashBytes.Length];
csDecrypt.Read(decryptedHashBytes,
        0, encryptedHashBytes.Length);
csDecrypt.Close();
msDecrypt.Close();
...
//сравнить оригинальный и дешифрованный хеши
bool match = true;
for (int i=0; i<hashBytes.Length; i++)
{
        if(hashBytes[i] != decryptedHashBytes[i])
        {
                match = false;
                break;
        }
}
if(match)
        Console.WriteLine(
                " Значения хешей совпали!");
else
        Console.WriteLine(
                "Значения хешей не совпадают! ");
```

Хеш-алгоритмы с ключом и целостность сообщения

Хеш-алгоритм с ключом или «ключевой хеш-алгоритм» – это такая хеш-функция, которая при хешировании исходных данных в качестве входного параметра получает также ключ. Это обстоятельство означает, что вам не требуется делать явным образом второй шаг и шифровать полученный хеш, поскольку функция выдаст уже шифрованный хеш.

Абстрактный класс **KeyedHashAlgorithm** представляет все реализации хеш-алгоритмов с ключами. В данное время существует только две реализации: классы **HMACSHA1** и **MACTripleDES**. Эти конкретные классы можно использовать для проверки целостности сообщения, если исходить из предположения, что ключ известен только отправителю и получателю. В нашем примере отправитель вычисляет хеш HMAC[1] для данных, используя при этом секретный ключ, а затем отправляет его вместе с исходными данными. Получатель заново вычисляет хеш HMAC по полученным данным, используя при этом тот же самый секретный ключ, и сравнивает два значения, удостоверяясь в целостности сообщения. HMACSHA1 здесь принимает ключ произвольного размера и вычисляет 20-байтовое значение хеша. Здесь также использован упрощенный подход, объединяющий обе стороны в одной программе. На практике отправитель и получатель просто вычисляют ключевой хеш независимо друг от друга, и получатель имеет возможность проверить целостность дошедших до него данных. Единственное приготовление, которые они оба должны сделать, – это договориться об используемом секретном ключе.

```
//получить от пользователя открытый текст
Console.WriteLine("Enter a plaintext string:");
String plaintextString = Console.ReadLine();

//преобразовать строку в массив байтов
Byte[] plaintextBytes =
        (new UnicodeEncoding()).GetBytes(
        plaintextString);

//создать ключевой хеш
byte[] key = new byte[16];
HMACSHA1 hmac = new HMACSHA1(key);
CryptoStream cs = new CryptoStream(
        Stream.Null, hmac, CryptoStreamMode.Write);
cs.Write(
        plaintextBytes, 0, plaintextBytes.Length);
cs.Close();
byte[] keyedHashBytes = hmac.Hash;
```

[1] HMAC представляет собой механизм аутентификации посредством криптографической хеш-функции. *См. RFC 2104*.

```
//отобразить байты хеша
Console.WriteLine(
        "Байты ключевого хеша для строки открытого текста:\n" +
        BitConverter.ToString(keyedHashBytes));
```

Итоги главы

Эта глава была посвящена симметричным алгоритмам и тем классам .NET Framework, которые реализуют эти алгоритмы. Мы рассмотрели здесь алгоритмы DES, TripleDES, Rijndael и RC2, а также сопутствующие использованию этих алгоритмов детали такие, как режимы шифрования и дополнения. Мы увидели, как можно использовать для сохранения секретов классы, производные от **SymmetricAlgorithm**. Также, воспользовавшись подвернувшейся возможностью, мы познакомились с классом **CryptoStream** и с классами, представляющими криптографические хеши. Еще мы мимоходом посмотрели на проблему обмена симметричными ключами.

Глава 4

Асимметричная криптография

Современный мир компьютерных коммуникаций породил огромную потребность в удобных, надежных и производительных технологиях шифрования. Симметричные алгоритмы такие, как DES или Rijndael, обеспечили эффективное решение многих задач, в особенности тех, где речь идет о шифровании больших массивов данных. Однако в определенных ситуациях симметричные алгоритмы наталкиваются на свои внутренние ограничения: проблему распределения ключей и проблему доверия. В данной главе мы рассмотрим эти проблемы и узнаем, как они решаются при помощи асимметричных алгоритмов. Мы рассмотрим затем работу асимметричного алгоритма на концептуальном уровне, подробно остановившись на действии односторонней функции с «черным ходом». Далее мы детально изучим работу алгоритма RSA, который является в настоящее время наиболее популярным асимметричным алгоритмом. Наконец, мы увидим, как RSA реализуется на практике, в типичной программе с использованием соответствующих классов .NET Security Framework.

Здесь мы сосредоточимся на основополагающей идее асимметричного алгоритма и изучим конкретный его образец – алгоритм RSA – с точки зрения технологий шифрования и дешифрования. В главе 5 мы займемся асимметричными алгоритмами RSA и DSA уже с точки зрения аутентификации и контроля целостности, включая технологию электронной подписи. Более подробное рассмотрение алгоритма RSA с математической точки зрения вы найдете в приложении B.

Проблемы, связанные с использованием симметричных алгоритмов

Одна из главных проблем в практическом использовании симметричных алгоритмов состоит в организации распределения ключей (мы назвали это «уловкой-22»). Другая проблема состоит в том, что две общающиеся

стороны, обладающие одним и тем же секретным ключом, вынуждены доверять друг другу. Проблема доверия возникает, когда шифрование используется для аутентификации и проверки целостности. Как мы видели в главе 3, симметричный ключ можно использовать для удостоверения аутентичности отправителя, однако при этом абсолютно необходимо, чтобы одна сторона доверяла другой.

Проблема распределения ключей

Проблема распределения ключей проистекает из того факта, что обе общающиеся стороны должны получить секретный ключ, прежде чем они смогут обмениваться сообщениями, и обе стороны должны обеспечить секретность этого ключа.

В некоторых ситуациях возможен прямой обмен ключами, однако заметим, что огромный по объему обмен коммерческими данными в наше время происходит между сторонами, которые никогда заранее между собой не договаривались и не могли заблаговременно позаботиться о ключах. Эти стороны, как правило, не могут доверять друг другу так, чтобы симметричное шифрование между ними могло использоваться в целях аутентификации. В условиях взрывообразного расширения Internet все шире проявляется необходимость в аутентификации сообщающихся сторон, которые ранее вообще не слышали о существовании друг друга. К счастью, все эти проблемы можно решить при помощи асимметричных алгоритмов[1].

Проблема доверия

Очень важно бывает иметь возможность убедиться в целостности сообщения и в личности его отправителя. Например, если передаваемые данные представляют собой финансовые транзакции, то на карту может быть поставлено очень многое. В какой-то степени эти проблемы могут быть актуальны даже для простой личной переписки по электронной почте, поскольку уголовные расследования очень часто зависят от улик наподобие «кто что знал» и «когда он это узнал». Для удостоверения идентичности отправителя можно использовать симметричный ключ, однако в этой схеме аутентификации неизбежно возникает проблема доверия.

Как вы помните из главы 3, в этой схеме создается хеш данных, который затем шифруется при помощи симметричного ключа, разделяемого обеими сторонами. К получателю, обладающему ключом, данные приходят вместе с присоединенным зашифрованным хешем. Получатель расшифровывает хеш и вычисляет хеш данных заново. Сравнив два хеша, он убеждается как в целостности сообщения, так и в личности его отправителя, поскольку зашифровать хеш мог лишь обладатель ключа.

---

[1] Асимметричные алгоритмы называют также «алгоритмами с открытым ключом». Это название может сбивать с толку, поскольку на самом деле речь идет о двух ключах: открытом и секретном. Выражение *«алгоритм с открытым ключом»* просто подчеркивает контраст с идеей симметричного алгоритма, где фигурирует только секретный ключ.

Эта схема прекрасно работает, если у вас есть такая роскошь, как возможность заблаговременно распределить секретные ключи, но даже в этом случае возникнет еще одна проблема. Как быть, если вы не можете доверять второй стороне, с которой вы разделяете один секретный ключ? Суть проблемы состоит в том, что вы не можете различить двух корреспондентов, обладающих одинаковыми ключами. Например, один ваш корреспондент, имеющий ваш ключ, может отправить сообщение другому вашему корреспонденту, выдавая себя за вас. Он может послать самому себе подделанное сообщение от вас. При отсутствии полного доверия между сторонами данная схема аутентификации становится бесполезной. Другая проблема может возникнуть, если ваш корреспондент «поделится» ключом с кем-то еще, не ставя вас об этом в известность. Вы можете внезапно оказаться в ситуации, когда в возникших разногласиях вам не на что опереться. Например, ваш партнер отказался выполнять ранее взятые на себя обязательства, утверждая, что сообщение для вас подписал кто-то другой, пользуясь секретным ключом, полученным от вас. Проблема подтверждения обязательств нередко возникает в общении между сторонами, не вполне доверяющими друг другу. Суть проблемы заключается в том, что при использовании любой симметричной схемы необходимо полное доверие между сторонами, что очень часто попросту нереально.

К счастью, асимметричные алгоритмы в состоянии решить все эти проблемы, выполняя те же самые криптографические операции, но используя для шифрования хеша секретный ключ, который находится в распоряжении только одной стороны. Благодаря этому любой может расшифровать хеш при помощи общедоступного открытого ключа и верифицировать хеш сообщения. Тем самым решаются проблемы доверия и подтверждения обязательств[1].

Эту технику называют цифровой подписью и она будет главной темой следующей главы.

Идея асимметричной криптографии

В 1970-х годах Мартин Хеллман и Уитфилд Диффи, а также, независимо от них, Ральф Меркль (Ralph Merkle) изобрели красивую криптографическую технологию. Идея технологии состояла в том, чтобы избавиться от проблем симметричной криптографии (доверие и распределение ключей), заменив единственный секретный ключ парой математически связанных ключей, один из которых будет общедоступным (открытым), а второй –

[1] Строго говоря, одного этого для решения всех проблем недостаточно. Для полной надежности необходимо участие третьей стороны, которую называют «сертификационным центром» (certificate authority), которая берет на себя формальную ответственность за подтверждение идентичности своих клиентов. Более подробно о сертификационных центрах и использовании асимметричных алгоритмов в целях аутентификации мы поговорим в главе 5.

секретным, доступным только тому, кто сгенерировал данную пару. Преимущества новой технологии очевидны. Во-первых, нет необходимости в секретном и заблаговременном распределении ключей, поскольку доступным второй стороне нужно сделать только открытый ключ, который нет необходимости хранить в секрете и можно передавать и публиковать по открытым каналам. Во-вторых, в противоположность симметричной криптографии, где секретный ключ должны были хранить в тайне как минимум две стороны, здесь только одна сторона должна обеспечить тайну своего ключа. Это, как легко понять, гораздо проще и создает гораздо меньше проблем. В-третьих, во многих сценариях проблема доверия к другой стороне вообще не встает, поскольку без знания вашего секретного ключа (а его не знает никто кроме вас) злоумышленники не могут сделать свое дело, например, подписать за вас документ или раскрыть ваши секреты.

Асимметричная криптография не отменяет криптографию симметричную. Наоборот, очень важно понимать достоинства и недостатки обеих технологий, чтобы правильно использовать их в комплексе так, чтобы они дополняли друг друга. Симметричные алгоритмы, как правило, работают значительно быстрее, особенно на больших массивах данных. Кроме того, при том же размере ключа они обладают большей стойкостью. С другой стороны, необходимость иметь две заранее распределенные копии секретного ключа у обеих общающихся сторон в больших сетях приводит к взрывообразному росту числа пар ключей, которыми нужно распоряжаться. Кроме того, поскольку секретный ключ передается и хранится более чем в одном месте, его необходимо менять достаточно часто, возможно, даже после каждого сеанса связи. Наконец, хотя симметричный алгоритм, в принципе, можно использовать для аутентификации в форме шифрованного хеша сообщения, полнофункциональная цифровая подпись все же требует асимметричной техники, такой, как RSA или DSA. Как мы увидим в следующей главе, симметричный алгоритм ключевого хеша можно использовать для реализации MAC (Message Authentication Code – Код аутентификации сообщения), что обеспечит контроль целостности и аутентификацию, однако не обеспечит подтверждения обязательств. В противоположность этому, асимметричный алгоритм цифровой подписи обеспечивает целостность, аутентификацию и подтверждение обязательств и притом позволяет воспользоваться услугами центра сертификации (CA – Certificate Authorities).

Использование асимметричной криптографии

Для того чтобы воспользоваться симметричной криптографией, Боб случайным образом генерирует пару ключей[1]. Он предоставляет свой открытый

[1] В реальности пара ключей генерируется автоматически, криптографическим программным обеспечением, скрывающим детали от пользователя. Например, Microsoft Outlook генерирует ключевую пару при помощи соответствующей системной службы Windows, для того чтобы зашифровать сообщение электронной почты и подписать его. Свободно распространяемое программное обеспечение PGP (Pretty Good Privacy) работает сходным образом, однако генерирует ключи самостоятельно (это мультиплатформенный инструмент).

ключ всем желающим, включая Алису. Когда Алисе потребуется отправить секретную информацию Бобу, она зашифрует данные его открытым ключом при помощи соответствующего асимметричного алгоритма. Затем она отправит зашифрованное сообщение Бобу. Любой, у кого нет секретного ключа Боба, должен будет преодолеть колоссальные трудности для того, чтобы дешифровать сообщение. Однако у Боба есть секретный ключ (тот самый «черный ход» к односторонней функции), и он легко получит из сообщения открытый текст. Схема использования асимметричной криптографии приведена на рисунке 4.1.

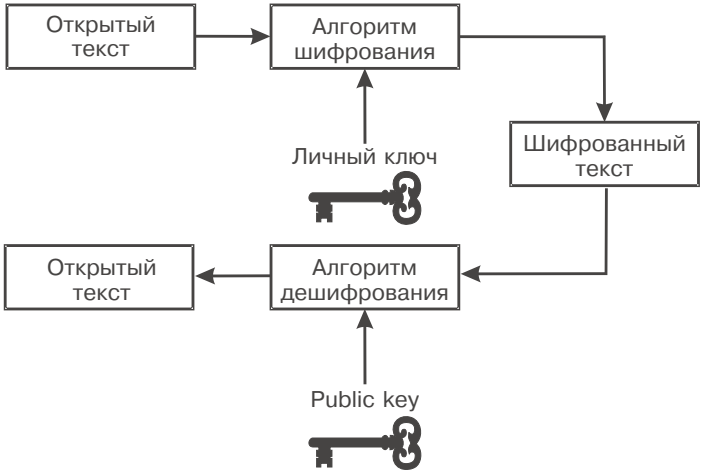


Рис. 4.1. Как работает асимметричная криптография

Аналогия с кодовым замком

Традиционный симметричный шифр можно представить себе, как сейф с кодовым замком, причем одна и та же комбинация запирает и отпирает замок[1]. В этом случае аналогией для асимметричного шифра будет служить довольно странное устройство. У замка сейфа есть две кодовых комбинации, одна для запирания и другая для отпирания. Сохранив одну комбинацию в секрете, и сделав другую общедоступной, вы можете эффективно контролировать либо помещение в сейф предметов, либо их изъятие. У вас появляются два полезных сценария: секретность без предварительного распределения ключей и подтверждение целостности.

СЕКРЕТНОСТЬ БЕЗ ПРЕДВАРИТЕЛЬНОГО РАСПРЕДЕЛЕНИЯ КЛЮЧЕЙ

Вот первый сценарий. Если вы знаете общеизвестную комбинацию для запирания замка, но не знаете секретный отпирающий код, то вы можете по-

[1] Придется немного напрячь воображение. Конечно, в реальности большинство кодовых замков запираются простым закрытием дверцы. В нашей аналогии мы считаем, что для запирания замка тоже нужно набрать его код.

местить что-то в сейф и запереть его, при этом никто кроме владельца секретного кода[1] не сможет получить этот предмет. Этот пример иллюстрирует принцип спонтанной конфиденциальности, то есть секретности, достигнутой без предварительного распределения ключей. Таким образом, у нас есть решение для проблемы распределения ключей, описанной выше.

ПОДТВЕРЖДЕНИЕ ЦЕЛОСТНОСТИ

Противоположный сценарий возникает, если запирающая комбинация сохранена в секрете, а запирающий код сделан общедоступным. Кто угодно может отпереть сейф, но никто не сможет запереть в нем новое содержимое. Таким образом, изменить содержимое сейфа и запереть его вновь в состоянии только владелец секретного запирающего кода. Вы можете подумать, что эту проблему легко обойти, ведь злоумышленник может сгенерировать новую пару ключей и запереть сейф новым секретным ключом. Однако отпираться сейф будет теперь новым открытым ключом, а старый, настоящий открытый ключ перестанет действовать. Таким образом, если вы знаете исходный открытый ключ, то обмануть вас таким образом невозможно. Поскольку здесь попытки подлога легко обнаруживаются, этот сценарий демонстрирует способ обеспечения целостности данных.

Односторонняя функция с «черным ходом»

Существует несколько асимметричных алгоритмов, основывающихся на этой великолепной идее, и все они имеют общие математические принципы. В основе каждого асимметричного алгоритма лежит математическая функция особенного типа, предложенная впервые Диффи и Хеллманом, которую называют *односторонней функцией «с черным ходом»*. Эта односторонняя функция, при условии наличия некоторой дополнительной информации, допускает значительно более легкое вычисление обратной функции.

Прежде чем перейти к обсуждению односторонней функции «с черным ходом», давайте рассмотрим односторонние функции вообще[2].

Односторонней функцией называют математическую функцию, которая обладает сильной асимметрией с вычислительной точки зрения. Такую функцию легко вычислить в прямом направлении, но обратную к ней функцию вычислить крайне трудно. Подобные функции основываются на *трудных* математических задачах таких, как разложение на сомножители больших составных чисел, являющихся произведением ряда простых чисел (факторизация), задача дискретных логарифмов или задача об укладке ранца («ранцевая задача»). Трудность решения подобных

[1] Обратите внимание, этого не сможете сделать даже вы. Поскольку вы сами запирали сейф, и потому знаете его содержимое, то разница невелика. Однако в некоторых случаях эта небольшая тонкость может сыграть свою роль.

[2] Односторонние функции, как принято считать повсеместно, существуют, и несколько функций, считающихся односторонними, широко используются в криптографии. Однако формального доказательства существования односторонних функций, к сожалению, не существует!

задач является предметом специальных областей математики и выходит за рамки тем данной книги. Достаточно сказать, что вычислительная трудность решения этих задач стремительно растет с увеличением размерности используемых чисел (этот размер можно соотнести с размером ключа в асимметричных алгоритмах).

При поиске решения математической задачи, основанной на односторонней функции, очень легко проверить предполагаемое решение на правильность. Вообразите, что вы складываете картинку-головоломку, и хотите найти место для очередного фрагмента. Поиск правильного места может занять много времени, но в любой момент, когда у вас появляется правдоподобная версия, вы можете проверить ее очень легко. Если перевести наш пример в математическую плоскость, представьте, что вы решаете задачу нахождения простых сомножителей[1] большого составного числа. Эта задача считается очень трудной[2], однако когда у вас уже есть простые сомножители, то все сводится к тому, чтобы их перемножить и сравнить результат с исходным составным числом. Чтобы понять, как это выглядит на практике, попробуйте найти все простые сомножители числа 3431, пользуясь только карандашом и бумагой. Без помощи компьютера большинству людей требуется несколько часов на решение этой задачи. Однако если попросить вас проверить, не являются ли числа 47 и 73 решением этой задачи, то перемножите вы их и получите результат 3431 очень быстро. Примите теперь во внимание, что число 3431 можно представить при помощи всего лишь двенадцати двоичных разрядов. Что, если попробовать решить эту задачу для числа, состоящего из 1024 двоичных разрядов? Поиск простых сомножителей – это задача, трудность которой растет экспоненциально с ростом разрядности числа (подумайте о суперкомпьютерах и миллионах лет), в то время как трудность проверки готового решения растет очень умеренно (микросекунды или миллисекунды на персональном компьютере).

Само выражение «односторонняя функция» немного сбивает с толку, поскольку, по определению, шифр может основываться только на обратимой математической функции – это означает, что обратная функция

---

[1] Простое число – это положительное целое число больше 1, которое делится без остатка только на единицу и само на себя, например, 2, 3, 5, 7, 11 и так далее. Составное число, также положительное и целое, не является простым и делится без остатка также и на другие числа, например, 4, 6, 9, 10 и так далее.

[2] Обратите внимание на слово «считается». Это слово означает, что никто еще не продемонстрировал публично быструю технику решения подобной задачи, и есть много соображений в пользу того предположения, что задача действительно трудна. Однако все это мнения и предположения, а не математическое доказательство. Как можно быть уверенным в этом без твердого математического доказательства? История математики полна примеров того, как разные ловкие парни находили хитроумные способы решить задачи, до того считавшиеся трудными. Известна история о Гауссе: когда ему было 7 лет, учитель задал всему классу задачу – просуммировать все целые числа от 1 до 100. Другие дети решали задачу очень долго, и мало кто решил ее правильно. Юный Гаусс решил задачу в мгновение ока, сразу назвав ответ: 1050. Он сообразил, что искомая сумма будет эквивалентна 50-ти суммам пар чисел, причем каждая такая пара равна 101, и просто умножил 101 на 50. Он нашел решение быстрее, чем можно было бы записать его на бумаге.

должна действительно существовать. В конце концов, если бы шифр основывался на необратимой функции, мы смогли бы лишь зашифровать данные, но не расшифровать их. Тем не менее, на практике обратить одностороннюю функцию настолько трудно в вычислительном отношении, что простой факт существования обратной функции мало в чем помогает. Такая функция теоретически является «двусторонней», но на практике она – «односторонняя».

Теперь мы приходим к идее «черного хода». Черный ход позволяет облегчить задачу, если вы знаете один маленький, но важный секрет. Это как книжная полка в доме с привидениями, которая поворачивается и открывает вход в подземелье. Но для того чтобы этим входом воспользоваться, вы должны знать о его существовании и должны знать также, что полка поворачивается трехкратным нажатием на канделябр. Односторонняя функция, которая обладает тем дополнительным свойством, что ее обратная функция вычисляется очень легко при наличии некоей дополнительной информации (секретный ключ), и называется односторонней функцией с «черным ходом». Если вам нравится, можете представлять себе такую функцию, как одностороннюю функцию «с привидениями».

В математике известно множество односторонних функций, но не так-то просто найти среди них такие, которые позволяют использовать механизм «черного хода». Было найдено лишь несколько функций, которые пригодны для использования в криптографии. Поиск таких алгоритмов, которые были бы достаточно удобны в практическом применении и обладали разумными размерами ключей, еще более сузил круг пригодных функций. Примерами удачных односторонних функций с «черным ходом» являются дискретные логарифмы, на которых основывается алгоритм DSA и задача о разложении больших чисел на простые сомножители (основа алгоритма RSA).

Преимущества асимметричного подхода

При использовании асимметричного подхода только один ключ надо держать в секрете, и этот ключ должен храниться в секрете только у одной стороны. Это огромное преимущество во многих ситуациях, особенно тогда, когда общающиеся стороны не могут заранее договориться о ключах. Тем не менее, для того чтобы этим обстоятельством можно было пользоваться на практике, подлинность опубликованного открытого ключа должна быть подтверждена какой-то третьей стороной, такой, как сертификационный центр. Поскольку секретный ключ должен храниться только у одной стороны, он никогда не передается по сети. Следовательно, асимметричная пара ключей может использоваться без изменений на протяжении многих сеансов, возможно, многие годы. Еще одно преимущество асимметричной схемы состоит в том, что ее можно использовать для электронной подписи, в том числе с механизмом подтверждения обязательств. Наконец, поскольку каждой стороне для общения с любым корреспондентом необходима только одна пара ключей, общее число потребных ключей в сети с большим числом участников получается гораздо меньшим, чем в случае симметричной схемы.

Сочетание асимметричных и симметричных алгоритмов

Поскольку при использовании асимметричных алгоритмов нет необходимости в предварительном обмене ключами, у вас может появиться искушение решить все проблемы симметричных алгоритмов, просто заменив их асимметричными. Но это означало бы выплеснуть из ванночки ребенка вместе с водой. Мы по-прежнему нуждаемся в таких качествах симметричных алгоритмов, как скорость работы и криптостойкость, поэтому мы объединим вместе два (или даже более) алгоритма, чтобы совместить их достоинства.

Например, в почтовых программах Microsoft Outlook и Netscape Communicator шифрование почтовых сообщений реализуется при помощи спецификации S/MIME (Secure/Multipurpose Internet Mail Extensions – Безопасные многоцелевые расширения электронной почты). Эта спецификация является стандартом IETF, который поддерживает одновременно аутентификацию при помощи цифровой подписи и конфиденциальность посредством шифрования, причем для шифрования может использоваться один из симметричных алгоритмов, включая DES, 3DES и RC2. Обмен симметричными ключами и электронная подпись реализуются при помощи алгоритма RSA, также используются алгоритмы хеширования MD5 и SHA-1.

Другой пример – популярное программное обеспечение PGP, разработанное Филлипом Циммерманном (Philip Zimmermann), которое обеспечивает криптозащиту почты и файлов при помощи нескольких алгоритмов, реализующих необходимый набор протоколов[1]. Здесь используются шифрование почтовых сообщений и электронные подписи на основе широкого набора симметричных, асимметричных и хеш-алгоритмов. Для транспортировки сеансового симметричного ключа можно использовать асимметричные RSA и ElGamal, 3DES – здесь лишь один из возможных вариантов массового шифрования данных. Цифровую подпись в PGP можно реализовать при помощи RSA или DSA, с использованием MD5 или SHA-1 для создания дайджеста сообщения.

Существуют и другие протоколы, построенные, как гибрид нескольких симметричных и асимметричных алгоритмов, например, IPSec (IP Security Protocol – Безопасный IP) или SSL (Secure Sockets Layer – Протокол защищенных сокетов). IPSec является стандартом IETF, который обеспечивает аутентификацию, целостность и конфиденциальность на уровне датаграмм, позволяя создавать «виртуальные частные сети» (virtual private network – VPN). Протокол SSL, разработанный компанией Netscape, обеспечивает аутентификацию и конфиденциальность в Internet-соединениях, главным образом, в соединениях по протоколу HTTP.

[1] Подробнее об этом см. OpenPGP Message Format RFC 2440 по адресу *http://www.ietf.org/rfc/rfc2440.txt*

Существующие асимметричные алгоритмы

Вспомним, что единственная информация, которую необходимо секретно передать другой стороне перед использованием симметричного шифрования – это секретный ключ. Поскольку такие ключи, как правило, малы и не превышают по размеру 256 бит, есть смысл использовать асимметричный алгоритм только для передачи симметричного ключа, а затем уже использовать этот симметричный ключ для шифрования и дешифрования больших объемов данных. В этом сценарии секретный симметричный ключ принято называть *сеансовым* ключом.

В наши дни известно несколько асимметричных алгоритмов, включая RSA, DSA, ElGamal и ECC. До сих пор самым популярным из них остается RSA, названный так в честь своих изобретателей (Rivest, Shamir, Adelman). Алгоритм RSA основывается на задаче разложения большого составного числа на простые сомножители. RSA можно использовать для шифрования, для обмена сеансовыми симметричными ключами, а также для реализации электронной подписи. Алгоритм DSA (Digital Signature Algorithm – алгоритм электронной подписи), предложенный институтом NIST в 1991 году, не так гибок и может применяться только для реализации электронной подписи. Алгоритм ElGamal, изобретенный Таером Элгамалем (Taher ElGamal), основывается на задаче вычисления дискретного логарифма над конечным полем. Аббревиатура EEC расшифровывается, как Elliptic Curve Cryptography (Криптография на основе эллиптической кривой), этот алгоритм предложен Нилом Коблицом (Neal Koblitz) и Виктором Миллером (Victor Miller) в 1985 г. В сущности, ECC – это не алгоритм как таковой, а, скорее, альтернативная алгебраическая система, предназначенная для реализации алгоритмов (таких, например, как DSA), основанная на особенных математических объектах, которые называют эллиптическими кривыми над конечными полями. Алгоритмы ElGamal и EEC в настоящее время не поддержаны в .NET, однако среда .NET Security Framework предусматривает возможность расширения при помощи сторонних реализаций.

Некоторые алгоритмы такие, как RSA и ElGamal, можно использовать как для шифрования, так и для реализации электронной подписи. Другие же (DSA) полезны только в одном качестве, как электронная подпись. Как принято считать, при одинаковом размере ключа асимметричные алгоритмы работают значительно медленнее и обладают меньшей стойкостью в сравнении с алгоритмами симметричными. По этой причине для получения эффективного результата асимметричные алгоритмы используют с ключами большого размера и только для шифрования небольших объемов данных – сеансовых симметричных ключей и хешей сообщений, которые отличаются небольшими размерами в сравнении с основным объемом полезных данных.

RSA: самый распространенный асимметричный алгоритм

Наиболее широко используемым асимметричным алгоритмом в настоящее время является RSA, и он в полной мере поддержан в среде .NET Security Framework. Рон Ривест, Ади Шамир и Леонард Адлеман (Ron Rivest, Adi Shamir, Leonard Adleman) изобрели шифр RSA в 1978 году в ответ на идеи, выдвинутые Хеллманом, Диффи и Меркелем (Hellman, Diffie, Merkel). Далее в этой главе мы познакомимся с высокоуровневой реализацией RSA, обеспечиваемой средой .NET Security Framework. Но вначале давайте рассмотрим RSA на концептуальном уровне.

Основания RSA

Понимание основ устройства RSA поможет вам использовать его более осмысленно. Здесь мы сосредоточимся на концепциях RSA, а в приложении B мы рассмотрим два примера реализации этого алгоритма «с нуля». Первый пример, **TinyRSA**, представляет собой игрушечную версию алгоритма, ограниченную 32-битовой арифметикой, другая же, **BigRSA**, является полномасштабной реализацией. Вам вряд ли придется когда-либо создавать реализацию алгоритма RSA «с нуля», поскольку в большинстве криптографических библиотек, включая NET Security Framework, имеются прекрасные реализации (по всей вероятности лучше, чем моя). Тем не менее, примеры в приложении B помогут вам понять внутреннее устройство RSA с тем, чтобы осмысленно его использовать.

Вот, как работает RSA. Вначале мы генерируем случайную пару ключей. Как и вообще, в криптографии очень важно сгенерировать ключи самым случайным и, следовательно, самым непредсказуемым способом. Затем мы шифруем данные открытым ключом при помощи алгоритма RSA. Наконец, мы дешифруем сообщение при помощи секретного ключа и убеждаемся в совпадении полученного результата с исходным сообщением. Обратите внимание на тот факт, что шифруем мы открытым ключом, а дешифруем – секретным. Здесь наша цель – конфиденциальность. В следующей главе мы познакомимся с противоположным подходом, когда для шифрования используется секретный ключ, а для дешифрования – открытый. Целью здесь будет аутентификация и проверка целостности.

Вот основная последовательность шагов, необходимая для создания пары ключей RSA.

1. Выберем случайно два простых числа p и q. Для того чтобы алгоритм работал правильно, эти два числа не должны быть равны друг другу. Для того чтобы шифр оказался стойким, эти числа должны быть большими, и для их записи должна использоваться

форма целых чисел произвольной точности с размером, по меньшей мере, 1024 бита[1].

2. Вычислим произведение этих чисел: $n = p \cdot q$.
3. Вычислим функцию Эйлера[2] (тотиент) для этих двух простых чисел, которая представляется греческой буквой φ. Функция эта вычисляется по формуле $\varphi = (p - 1) \cdot (q - 1)$.
4. Теперь, когда у нас есть значения n и φ, сами числа p и q нам более не нужны. Однако мы должны побеспокоиться о том, чтобы они не попали в чьи-то руки. Их необходимо уничтожить, стерев все следы, поскольку атакующая сторона может по этим числам легко воссоздать нашу пару ключей.
5. Далее мы случайно выбираем число e, большее 1, меньшее φ и относительно простое к φ. О двух числах говорят, что они относительно просты друг к другу, если у них нет общих простых сомножителей. Обратите внимание, что e не обязательно должно быть простым. Значение e совместно со значением числа n будет представлять собой открытый ключ, используемый для шифрования.
6. Затем мы вычисляем уникальное значение d (оно будет использоваться для дешифрования), которое удовлетворяет следующему условию: если произведение d и e разделить на φ, то остаток от деления будет равен 1. Математически это выражается уравнением $d \cdot e = 1 \pmod{\varphi}$. На математическом жаргоне это звучит так: d есть мультипликативная инверсия от e по модулю φ. Значение d теперь должно храниться в секрете. Если вам известно значение φ, то значение d легко вычислить из e при помощи метода, называемого алгоритмом Евклида. Если вы знаете n (а это открытое значение), но не знаете p и q (которые уничтожены), то вычислить φ чрезвычайно трудно. Значения d и n вместе составляют секретный ключ.

Сгенерировав свою пару ключей, мы можем теперь зашифровать сообщение при помощи открытого ключа, предприняв следующие действия.

1. Возьмем положительное число m, представляющее собой фрагмент открытого текста. Для того чтобы алгоритм работал правильно, число m должно быть меньшим, чем модуль числа n, которое ранее было вычислено, как $p \cdot q$. По этой причине длинные сообщения необходимо разбивать на такие фрагменты, что каждый из них может быть представлен целым числом заданного битового размера, затем каждый фрагмент шифруется индивидуально.

---

[1] На практике существуют и другие критерии, которые необходимо учитывать при выборе чисел p и q. Например, даже если оба этих числа велики, их произведение будет относительно легко разложить на множители (метод, известный, как алгоритм факторизации Ферма), если разница между ними мала.

[2] Функция Эйлера, обозначаемая греческой буквой «фи», представляет собой число положительных целых чисел, меньших или равных n, которые относительно просты к n (т. е. не имеют с n общих простых сомножителей). Число 1 считается относительно простым ко всем целым числам.

2. Мы вычисляем шифрованный текст c при помощи открытого ключа, состоящего из e и n. Формула, которой мы пользуемся: $c = m^e \pmod{n}$.

Наконец, расшифровать сообщение мы можем при помощи следующих действий.

1. Мы вычисляем открытый текст при помощи шифрованного текста, а также секретного ключа, состоящего из d и n. Формула, которой мы пользуемся: $m = c^d \pmod{n}$.
2. Сравниваем полученное m с исходным m и убеждаемся, что они равны, поскольку дешифрование у нас – операция, обращающая шифрование.

Миниатюрный пример RSA

Вот пример алгоритма RSA, который прост почти настолько, что его можно выполнить при помощи карандаша и бумаги. По своему масштабу он близок к программе **TinyRSA**, которая уже упоминалась. Битовый размер чисел в этом примере смехотворно мал и не обеспечивает надежного шифрования, однако на концептуальном уровне этот пример полностью иллюстрирует работу алгоритма RSA. Преимущество такого изучения состоит в том, что на миниатюрном уровне «карандаша и бумаги» сущность алгоритма понять гораздо легче. В конце концов, не все люди способны перемножать в уме 1024-битовые числа! Даже имея дело с 32-битовыми числами, операция возведения в степень легко переполнит 32 разряда, если вы не будете достаточно аккуратны в своей реализации[1].

Следуя концептуальным шагам, описанным выше, мы начнем с выбора двух простых чисел n и q, не равных друг другу[2]. Поскольку мы намеренно выбираем очень малые числа, мы можем не принимать во внимание возможность переполнения 32-битовой арифметики. Это также позволяет нам использовать обычный калькулятор, встроенный в Windows.

1. Пусть мы выбрали два простых числа p и q:

$$p = 47$$
$$q = 73$$

2. Произведение n этих чисел мы вычислим, как

$$n = p \cdot q = 3431$$

3. Функция (тотиент) Эйлера двух простых чисел находится, как

[1] Для того чтобы избежать переполнения, нельзя использовать операцию возведения в степень (потенцирование) напрямую. Вместо этого вы должны выполнять циклическое умножение, нормализуя на каждом шаге результат.

[2] Понятно, что в реальной жизни эти детали скрыты от пользователя. Криптографические программы автоматически выбирают эти два случайных простых числа и выполняют все необходимые действия с ними. Тем не менее, программисту бывает иногда необходимо знать, как реализуется такой алгоритм «с нуля».

$$\phi = (p - 1) \cdot (q - 1) = 3312$$

4. Теперь, когда у нас есть n и ϕ, мы можем отбросить p и q, позаботившись о том, чтобы никаких следов их существования не осталось.
5. Далее мы случайным образом выбираем число e, большее 1, меньшее n, и относительно простое к ϕ. Конечно, здесь возможны многие варианты, и каждого кандидата следует протестировать при помощи метода Евклида[1]. Предположим, мы выбрали следующее значение для e:

$$e = 425$$

6. Теперь мы вычисляем инверсию от e по модулю ϕ:

$$d = 1769$$

7. Полученное d мы сохраняем в секрете, а числа e и n делаем общедоступными.

Теперь, когда у нас есть секретный и открытый ключи, мы можем заняться шифрованием и дешифрованием данных. Как вы уже понимаете, эти данные должны быть представлены в виде целых чисел, для того чтобы над ними можно было выполнить необходимые операции. На практике открытый текст обычно представляет собой секретный ключ для симметричного алгоритма или хеш сообщения, но, в сущности, это могут быть любые данные. В какой бы форме ни находились данные, их необходимо представить в виде последовательности целых чисел, размер которых ограничен размером используемого ключа. Мы не будем вдаваться в вопросы представления и интерпретации данных, сосредоточившись вместо этого на принципах работы алгоритма. По этой причине мы используем в нашем сценарии в качестве открытого текста просто одно небольшое целое число.

1. Предположим, что наш открытый текст представляет собой просто число:

$$\text{открытый текст} = 707$$

2. Зашифрованный текст мы вычислим по формуле $c = m^e \pmod{n}$:

$$\text{шифрованный текст} = 707 \text{ ^ } 425 \pmod{3431} = 2142$$

3. Зашифрованное значение не удастся легко преобразовать в исходный вид без знания d. При этом с ростом разрядности чисел трудность решения задачи растет экспоненциально. Однако если вы посвящены в тайну и знаете, что $d = 1769$, то открытый текст вы получите по формуле $m = c^d \pmod{n}$:

[1] Метод Евклида – это эффективный способ нахождения наибольшего общего делителя для двух целых чисел.

открытый текст = 2142 ^ 1769 (mod 3431) = 707

Если вы откомпилируете приведенный ниже код, то сможете проверить наши вычисления. Изучая код, надо помнить, что полноценные реализации алгоритма RSA используют гораздо большие разрядности чисел, а реальный открытый текст, как правило, разбивается на целые числа, гораздо большие 707.

```
int m = 707;  //открытый текст
int e = 425;  //показатель шифрования
int n = 3431; //модуль
int c = 1;    //шифрованный текст

//шифрование: c = m^e(mod n)
for (int i=0; i<e; i++) //используем цикл,
                        //чтобы избежать переполнения
{
   c = c*m
   c = c%n //нормализуем к модулю
}
//теперь зашифрованный текст равен 2142

int d = 1769; //показатель дешифрования
m = 1; //открытый текст


//дешифрование: m = c^d(mod n)
for (int i=0; i<d; i++) //используем цикл,
                        //чтобы избежать переполнения
{
   m = m*c
   m = m%n //нормализуем к модулю
}
//открытый текст m = 707, что соответствует исходному значению
```

Предостережение: вопросы вероятности

Каждый асимметричный алгоритм основывается на некоторой односторонней функции с «черным ходом». И здесь неизбежно возникает один важный вопрос. Как можно быть уверенным в том, что функция эта действительно – односторонняя? Лишь потому, что никто еще публично не продемонстрировал способ, позволяющий относительно легко вычислить обратную функцию, надежность алгоритма не может считаться доказанной.

Если кто-то открыл подобный способ и сохранил свое открытие в тайне, он может раскрывать множество шифрованных текстов каждый день, и никто об этом не узнает. Возможно, это напоминает паранойю, однако список «подозреваемых» возглавляют крупнейшие правительства, поскольку именно у них в распоряжении находится большое число крупных математиков и обширные вычислительные ресурсы, а именно для крупного правительства очень заманчиво было бы обрести такое тайное преимущество перед остальным миром.

Самый убедительный путь к тому, чтобы убедиться в надежности шифра, состоит в строгом математическом доказательстве того факта, что не существует легкого способа вычислить обратную функцию без знания «черного хода». К сожалению, такого доказательства не существует. Имеются и были опубликованы лишь доказательства для некоторых частных (не очень существенных) случаев.

Несмотря на отсутствие строгих доказательств, большинство экспертов считает, что распространенные асимметричные алгоритмы такие, как RSA, вполне надежны при достаточном размере ключа. Но заметьте, что это лишь мнение экспертов, которое строгого математического доказательства не заменяет полностью. Конечно, помня о множестве шифров на протяжении всей истории, которые считались надежными лишь до тех пор, пока не находился метод успешной атаки, нельзя быть уверенным в этом до конца.

В случае RSA уверенность основывается на широко распространенном убеждении, что крайне трудно вычислить значения d, p или q, если известен лишь открытый ключ, то есть значения n и e. Разумеется, стойкость шифра быстро растет, когда вы увеличиваете размерность значений p и q. Встроенные целочисленные типы языка C# ограничены максимальным размером 64 бита, что далеко не удовлетворяет потребностей реальных асимметричных задач. На самом деле нам требуются целые числа размером 1024 бита и более. Другой источник беспокойства должен заключаться в том, что стойкость шифра основывается на факте, который строго до сих пор не доказан. Все исходят из предположения, что не существует иного способа атаковать RSA, кроме разложения числа pq на простые сомножители p и q. Но никто еще не доказал, что способ, не требующий разложения pq на простые сомножители, в принципе не может существовать. Если некто изобретет подобный способ, он сможет стать самым богатым человеком в мире, но жизнь его станет очень опасной, потому что многие захотят его убить.

Программирование при помощи .NET Asymmetric Cryptography

В этом разделе мы рассмотрим примеры программ **RSAAlgorithm** и **SavingKeysAsXml**, которые предназначены для иллюстрации материалов этой главы. Эти два образца кода покажут, как следует шифровать и дешифровать при помощи алгоритма RSA, а также как сохранять и извлекать информацию о ключах в формате XML. Пример RSAAlgorithm использует конкретный класс **RSACryptoServiceProvider**. На рисунке 4.2 показано место этого класса в иерархии классов, принадлежащей абстрактному классу **AsymmetricAlgorithm**. Другой конкретный класс, **DSACryptoServiceProvider**, будет обсуждаться в главе 5, когда мы займемся вопросами электронной подписи.

Пример использования алгоритма RSA

Пример RSAAlgorithm использует метод **Encrypt** класса **RSACryptoServiceProvider**. Этот метод получает на входе два параметра, первый из которых должен представлять собой байтовый массив, содержащий входные данные. Второй параметр, булево значение, должен указывать на режим дополнения, который должен использовать метод. Дополнение требуется потому, что входные данные, в общем случае, могут не соответствовать требуемому размеру в битах. Поскольку алгоритм требует фиксированного размера блока входных данных, для достижения этого размера необходимо будет дополнять реальные данные. Если второму параметру присвоено значение True, то для дополнения будет использоваться технология OAEP[1]. В противном случае используется традиционный режим дополнения PKCS#1 v1.5. Режим PKCS#1 v1.5 наиболее часто используется для дополнения блоков данных при шифровании RSA. Тем не менее, рекомендуется, чтобы приложения RSA, вновь разрабатываемые на платформах, где поддерживается OAEP, использовали именно OAEP. Заметим, что дополнение OAEP доступно в системах Microsoft Windows XP и Windows 2000 при установке пакета шифрования.

К сожалению, более ранние версии Windows не поддерживают OAEP, и по этой причине вызов метода **Encrypt** со вторым параметром, равным True, приведет к возникновению исключения **CryptographicException**. Метод **Encrypt** возвращает результирующие зашифрованные данные в виде байтового массива.

---

[1] OAEP (Optimal Asymmetric Encryption Padding – оптимальное дополнение для асимметричного шифрования) – эта техника дополнения, разработанная Меиром Белларом (Mihir Bellare) и Филом Роуджуеем (Phil Rogaway) в 1993 году специально для RSA. Техника OAEP обеспечивает дополнение, гораздо более качественное с точки зрения безопасности, в сравнении с обычно используемой техникой PKCS#1 v1.5.

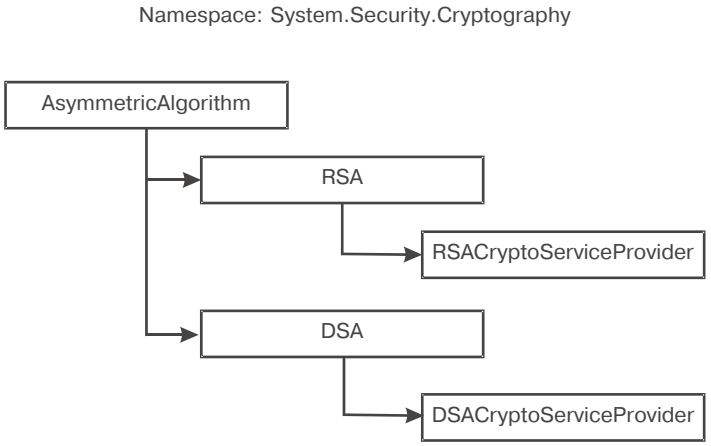


Рис. 4.2. Иерархия класса AsymmetricAlgorithm

Вот синтаксис обращения к этому методу:

```
public byte[] Encrypt(
   byte[] rgb,
   bool fOAEP
);
```

Противоположную операцию выполняет метод **Decrypt**. Вы без труда догадаетесь, как он работает. Первый параметр – байтовый массив, содержащий зашифрованные данные. Второй параметр имеет точно такой же смысл, что и в методе **Encrypt**. Возвращаемое значение представляет собой байтовый массив с расшифрованными данными.

Вот синтаксис обращения к методу **Decrypt**:

```
public byte[] Decrypt(
   byte[] rgb,
   bool fOAEP
);
```

На рисунке 4.3 приведен вид программы **RSAAlgorithm**, которая предназначена для шифрования и дешифрования сообщений. Вы можете ввести текст сообщения в текстовое поле в верхней части формы. После этого, щелчком на кнопке **Encrypt**, вы зашифруете сообщение и заполните все поля формы кроме последнего поля (в остальных полях появятся параметры RSA). Наконец, щелчок на кнопке **Decrypt** расшифрует данные, которые отобразятся в самом нижнем поле. Разумеется, расшифрованное сообщение окажется идентичным исходному открытому тексту.

Давайте теперь рассмотрим исходный код программы **RSAAlgorithm**. Метод **buttonEncrypt_Click** вызывается по щелчку пользователя на кнопке **Encrypt**. Эта функция шифрует содержимое текстового поля с открытым текстом при помощи заданного открытого ключа RSA. Пара ключей

RSA генерируется программой при запуске автоматически, но ключи можно сгенерировать заново при помощи кнопки **New RSA Parameters**. В коде есть несколько участков, имеющих отношение к манипуляциям с элементами управления, эти и подобные им участки кода несущественны с точки зрения изучения функций RSA. Здесь мы будем игнорировать фрагменты кода, не относящиеся непосредственно к нашей теме. Если вас интересуют детали пользовательского интерфейса, пожалуйста, изучите соответствующие фрагменты.

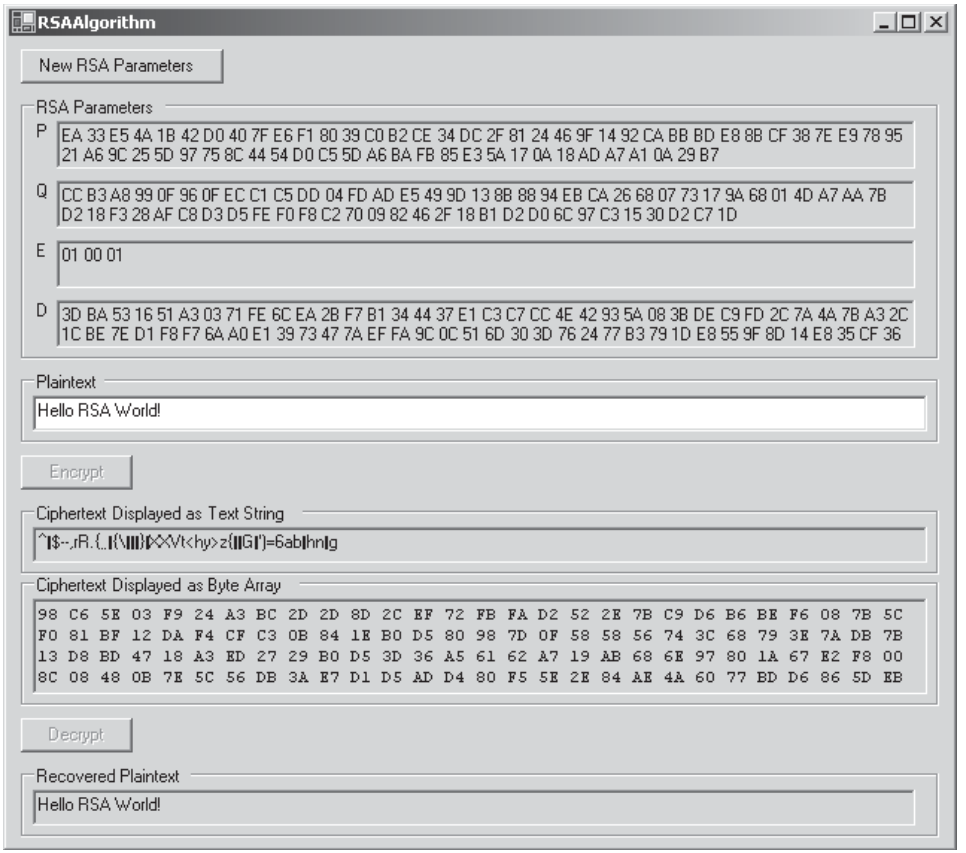


Рис. 4.3. Пример программы RSAAlgorithm

Впервые параметры RSA генерируются при запуске программы, вызовом метода **GenerateNewRSAParams** внутри метода **RSAAlgorithm_Load**. Также метод **GenerateNewRSAParams** вызывается при каждом щелчке на кнопке **New RSA Parameters** (эти щелчки обрабатываются методом **buttonNewRSAParams**).

Метод **GenerateNewRSAParams** устроен очень просто. Он создает объект класса **RSACryptoServiceProvider**, извлекает его внешние и внутренние параметры при помощи метода **ExportParameters** класса RSA, и

отображает эти параметры в пользовательском интерфейсе. Фактически, эти параметры хранятся в полях типа **RSAParameters**. Поле этого типа с именем **rsaParamsExcludePrivate** содержит значения открытых параметров RSA (т. е. только модуль и показатель), которые необходимы для шифрования, выполняемого методом **buttonEncrypt_Click**. Другое поле типа **RSAParameters** с именем **rsaParamsIncludePrivate** получает набор открытых и секретных параметров, которые необходимы для дешифрования (метод **buttonDecrypt_Click**). Рассмотрим метод **GenerateNewRSAParams**. Обратите внимание на тот факт, что метод **ExportParameters** здесь вызывается дважды. Первый раз методу задается аргумент True, а во второй – False. Значение True указывает методу, что в информацию необходимо включить все, в том числе секретный ключ. При вызове с аргументом False метод экспортирует только информацию, относящуюся к открытому ключу. Мы разделяем здесь ключевую информацию затем, чтобы проиллюстрировать и подчеркнуть тот факт, что для шифрования используется только открытая информация, в то время как дешифрование требует, одновременно, открытого и секретного ключей. Этот момент чрезвычайно важен для понимания асимметричной криптографии. Возможно, лучше было бы вообще разбить задачу на два отдельных приложения, но в нашем примере единственное приложение сделано максимально простым, что облегчает изучение темы. Вам было бы полезно потратить немного времени на то, чтобы убедиться: функции шифрования и дешифрования в нашей программе действительно используют только необходимую для них информацию о параметрах RSA, используя для этого метод **ImportParameters**.

```
private void GenerateNewRSAParams()

{
        //установить асимметричный алгоритм RSA
        RSACryptoServiceProvider rsa =
                new RSACryptoServiceProvider();

        //извлечь открытые и секретные параметры RSA
        rsaParamsIncludePrivate =
                rsa.ExportParameters(true);

        // извлечь только открытые параметры RSA
        rsaParamsExcludePrivate =
                rsa.ExportParameters(false);
```

Когда мы создаем экземпляр класса **RSACryptoServiceProvider**, то получаем, фактически, реализацию RSA, предоставляемую провайдером услуг криптографии (CSP). Этот класс напрямую производится из класса RSA. Класс **RSA** предусматривает возможность иметь несколько реализаций алгоритма RSA, существующих как производные классы, однако в настоящее время доступна только реализация CSP.

Два поля, в которых сохраняется информация о параметрах RSA при вызове метода **ExportParameters**, объявлены, как поля типа **RSAParameters** (см. фрагмент кода ниже). Поле **rsaParamsExcludePrivate** используется при шифровании, а поле **rsaParamsIncludePrivate** – при дешифровании.

```
//открытые значения модуля и показателя,
//необходимые для шифрования
RSAParameters rsaParamsExcludePrivate;

//открытые и секретные параметры RSA для дешифрования
RSAParameters rsaParamsIncludePrivate;
```

В методе **buttonENcrypt_Click** мы создаем новый экземпляр класса **RSACryptoServiceProvider**, инициализируя его сохраненной информацией открытого ключа при помощи метода **ImportParameters** объекта **RSA** (при этом в качестве аргумента используя поле **rsaParamsExclude-Private**). Далее мы получаем открытый текст в виде байтового массива с именем **plainbytes**. Наконец, мы выполняем главное действие, вызывая метод **Encrypt** объекта **RSA**. Этот вызов возвращает нам байтовый массив с именем **cipherbytes**. Это не локальная переменная, а свойство конкретного экземпляра объекта, поскольку нам потребуется передать этот массив методу, выполняющему дешифрование.

```
private void buttonEncrypt_Click(
        object sender, System.EventArgs e)
{
        //работаем с пользовательским интерфейсом
        ...

        //установить асимметричный алгоритм RSA
        RSACryptoServiceProvider rsa =
                new RSACryptoServiceProvider();

        // извлечь только открытые параметры RSA для шифрования
        rsa.ImportParameters(rsaParamsExcludePrivate);

        //прочитать открытый текст и зашифровать его
        byte[] plainbytes =
                Encoding.UTF8.GetBytes(textPlaintext.Text);
        cipherbytes =
                rsa.Encrypt(
                plainbytes,
                false); // использование fOAEP требует
                        // пакета шифрования

        //отобразить зашифрованный текст
        ...
```

```
        // отобразить зашифрованный текст
        //в шестнадцатеричном формате
        ...

        //работаем с пользовательским интерфейсом
        ...
}
        ...
//переменная для передачи шифрованного текста
byte[] cipherbytes;
```

Метод **buttonDecrypt_Click** вызывается по щелчку пользователя на кнопке **Decrypt**. Снова, точно так же, создается объект **RSA**. Информация в этот объект загружается при помощи его метода **ImportParameters**, однако на этот раз аргументом служит поле **rsaParamsIncludePrivate**, поскольку для дешифрования необходим как открытый, так и секретный ключи. Открытый текст мы получаем обращением к методу **Decrypt** объекта **RSA**. Поскольку для шифрования и дешифрования использовались ключи RSA, принадлежащие к одной паре, результирующий дешифрованный текст в точности совпадет с исходным.

```
private void buttonDecrypt_Click(
        object sender, System.EventArgs e)
{
        //установить асимметричный алгоритм RSA
        RSACryptoServiceProvider rsa =
                new RSACryptoServiceProvider();

        //импортировать открытый и секретный ключи
        rsa.ImportParameters(rsaParamsIncludePrivate);

        //прочитать шифрованный текст и дешифровать его
        byte[] plainbytes =
                rsa.Decrypt(
                cipherbytes,
                false); //использование fOAEP требует
                        //пакета шифрования

        //отобразить полученный открытый текст
        ...

        //работаем с пользовательским интерфейсом
        ...

}
        ...
//переменная для передачи шифрованного текста
byte[] cipherbytes;
```

Сохранение ключей в формате XML

Не всегда возможно передать содержимое объекта **ExportParameters** напрямую между двумя приложениями, тем более, если речь идет о разных платформах или хотя бы о разных криптографических библиотеках. В конце концов, класс **ExportParameters** специфичен для Microsoft и для .NET. Более удобный и универсальный способ передачи открытого ключа заключается в использовании XML-потока[1]. Пример программы **SavingKeysAsXml** показывает, как следует записывать и считывать ключи в формате XML. Этот пример почти идентичен предыдущей программе. Главное отличие состоит в том, что для хранения и передачи открытого ключа между методом шифрования и методом дешифрования мы используем XML вместо объекта **ExportParameters**. Еще одно отличие заключается в том, что информация о параметрах RSA не отображается, а вместо этого отображается содержимое XML-потока.

Ради простоты и наглядности мы вновь используем монолитное приложение, объединяющее обе функции на одной экранной форме. Не составит никакого труда разделить программу на две части, выполняющие, по-отдельности, шифрование и дешифрование. Для того чтобы сделать пример более реалистичным, XML-данные здесь записываются в файл. Этим мы имитируем сценарии реального мира, где подобную информацию приходится записывать и передавать при помощи какого-то внешнего носителя или сетевого соединения. С точки зрения программиста наиболее серьезные изменения по отношению к предыдущей программе заключаются в том, что вызовы методов **ExportParameters** и **ImportParameters** класса **RSACryptoServiceProvider** заменены вызовами методов **ToXmlString** и **FromXmlString**, принадлежащих тому же классу. Здесь также используется логический аргумент, указывающий, нужно ли включать в информацию секретный ключ.

Ниже приведен код метода **GenerateNewRSAParams**, который выполняет здесь те же функции, что и в предыдущей программе. Разница состоит в том, что данные мы сохраняем в двух XML-файлах **PublicPrivateKey.xml** и **PublicOnlyKey.xml**. Эти два файла затем используются функциями шифрования и дешифрования.

```
private void GenerateNewRSAParams()
{
        //установить асимметричный алгоритм RSA
        RSACryptoServiceProvider rsa =
                new RSACryptoServiceProvider();
```

[1] Способ передачи открытого ключа (объект **ExportParameters** или XML-поток) не имеет отношения к вопросам безопасности. Разумеется, вам не следует передавать секретный асимметричный ключ или сеансовый симметричный ключ по открытому каналу. Для обмена чувствительной информацией следует зашифровать ключ шифрования. На первый взгляд это похоже на рекурсию, однако на самом деле в таком шифровании есть смысл. В главе 6 мы увидим, как передавать зашифрованные ключи при помощи XML, используя криптографические стандарты XML.

```
        //открытый и секретный ключи RSA
        StreamWriter writer =
                new StreamWriter("PublicPrivateKey.xml");
        string publicPrivateKeyXML =
                rsa.ToXmlString(true);
        writer.Write(publicPrivateKeyXML);
        writer.Close();

        //только открытый ключ RSA
        writer =
                new StreamWriter("PublicOnlyKey.xml");
        string publicOnlyKeyXML =
                rsa.ToXmlString(false);
        writer.Write(publicOnlyKeyXML);
        writer.Close();

        //отобразить оба ключа RSA
        textBoxPublicKeyXML.Text = publicPrivateKeyXML;

        //работаем с пользовательским интерфейсом
        ...
}
```

Наконец, метод **buttonDecrypt_Click** создает собственный экземпляр класса **RSACryptoServiceProvider**, но инициализирует его методом **FromXmlString**, используя файл **PublicPrivateKey.xml**, в котором содержится полная пара ключей, необходимая для дешифрования.

```
private void buttonDecrypt_Click(
        object sender, System.EventArgs e)
{
        //установить асимметричный алгоритм RSA
        //при помощи ключа из XML-файла
        RSACryptoServiceProvider rsa =
                new RSACryptoServiceProvider();

        //открытый и секретный параметры RSA для дешифрования
        StreamReader reader =
                new StreamReader("PublicPrivateKey.xml");
        string publicPrivateKeyXML = reader.ReadToEnd();
        rsa.FromXmlString(publicPrivateKeyXML);
        reader.Close();

        //прочитать шифрованный текст и дешифровать его
        byte[] plainbytes =
                rsa.Decrypt(
                cipherbytes,
                false); // использование fOAEP требует
                        //пакета шифрования
```

```
        //отобразить полученный открытый текст
        ...

        //работаем с пользовательским интерфейсом
        ...
}
```

На рисунке 4.4 изображено окно программы **SavingKeysAsXml**, используемой для шифрования и дешифрования сообщений. На форме, кроме прочего, отображается содержимое файла **PublicPrivateKey.xml**, которое используется для дешифрования. Довольно трудно разобрать содержимое XML-элементов, слитых в один непрерывный поток, но если присмотреться, вы увидите там значения параметров RSA. Метод шифрования использует только значения модуля и показателя.

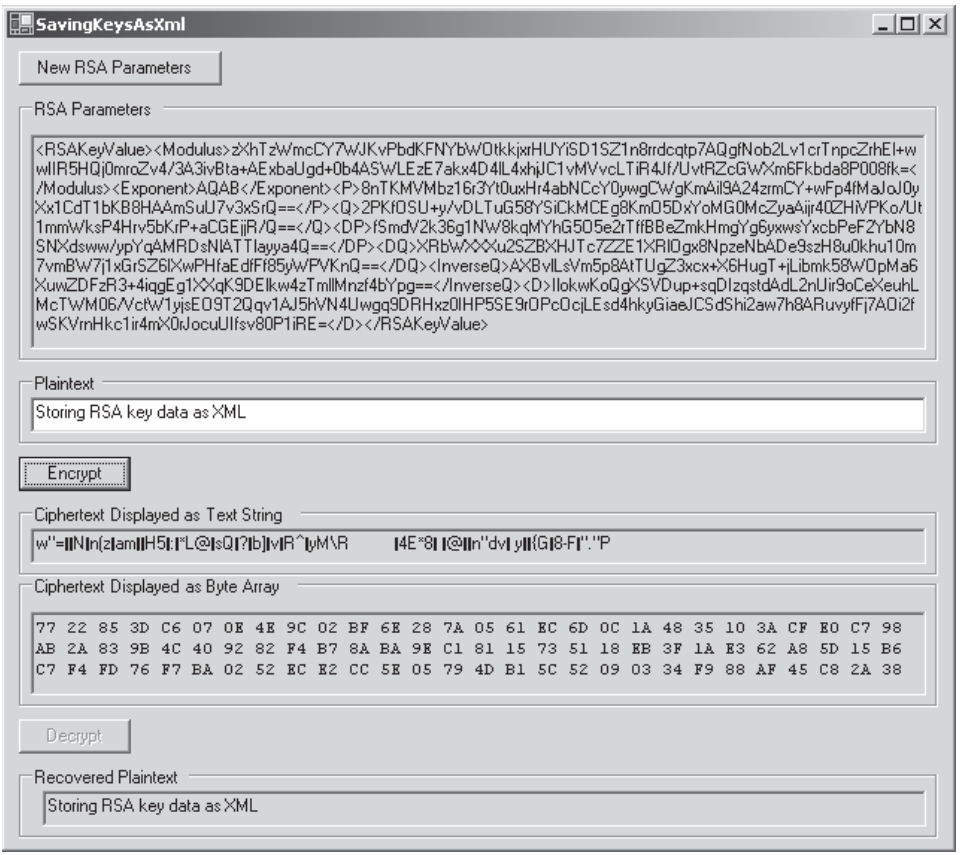


Рис. 4.4. Пример программы SavingKeysAsXml

Цифровые сертификаты

Для того чтобы практически использовать такой асимметричный алгоритм, как RSA, необходим способ публикации открытых ключей. Распространить открытый ключ можно и вручную, но идеальное решение заключается в использовании центра сертификации (CA), где открытый ключ будет содержаться в цифровом сертификате, также называемым цифровым удостоверением. Цифровой сертификат – это документ, который удостоверяет вашу личность в сообщениях или транзакциях через посредство Internet. Вы можете получить цифровой сертификат при помощи доверенной организации, такой, например, как Verysign, или же вы можете установить собственный сервер сертификатов (CA-сервер). В программе Microsoft Outlook вы можете получить доступ к центру сертификации и сгенерировать сертификат, открыв диалог Сервис | Параметры, выбрав вкладку Безопасность и щелкнув на кнопке Получить удостоверение. Существует множество коммерческих центров сертификации, а сама сертификация существует во многих вариантах, отличающихся ценой и уровнем доверия. В разной степени и разными способами разные центры сертификации попытаются удостовериться в том, что вы тот, за кого себя выдаете. В случае положительного результата будет сгенерирован документ, содержащий в себе открытый ключ, который вы представляете на сертификацию. Затем центр сертификации подпишет этот документ при помощи своего собственного секретного ключа. Разумеется, никто кроме данного центра не владеет этим ключом, а ваш секретный ключ также недоступен никому, в том числе и центру сертификации. В результате получится документ, именуемый цифровым сертификатом или цифровым удостоверением, который центр сертификации внесет в свою базу данных и сделает доступным для всех, кто желает иметь с вами дело, используя средства шифрования и аутентификации. Любой ваш корреспондент сможет обратиться к центру и получить ваш сертификат. Проверив достоверность документа при помощи открытого ключа центра сертификации, ваш корреспондент извлечет из него ваш открытый ключ, при помощи которого зашифрует сообщение для вас или проверит вашу электронную подпись.

Итоги главы

В этой главе мы познакомились с асимметричными алгоритмами, в основном, на примере RSA. Мы увидели, как при помощи асимметричных алгоритмов решаются проблемы, присущие алгоритмам симметричным. Мы также рассмотрели принципы работы алгоритма RSA и узнали, как использовать его в программах при помощи класса **RSACryptoServiceProvider** в среде .NET Framework. Наконец, мы узнали, как можно сохранять и передавать параметры шифрования в формате XML так, чтобы они были доступны другим участникам коммуникаций. В следующей главе мы продолжим изучение асимметричных алгоритмов, переместив свое внимание на вопросы, связанные с электронной подписью, что также является важным аспектом в применении асимметричных алгоритмов.

Глава 5

Цифровая подпись

В предыдущей главе мы увидели, как асимметричные алгоритмы, и в особенности RSA, используются для достижения конфиденциальности. Однако, как оказывается, есть еще одна сфера применения асимметричных алгоритмов – это цифровая подпись, при помощи которой решаются задачи аутентификации, контроля целостности и подтверждения обязательств. В этой главе мы узнаем, как работает цифровая подпись, изучим работу алгоритмов RSA и DSA при их использовании для реализации цифровой подписи. Оба этих алгоритма, RSA и DSA, полностью поддержаны в .NET Framework, поэтому мы рассмотрим законченные примеры программирования с их использованием.

Технология цифровой подписи требует использования еще одного криптографического примитива, известного, как «криптографический хеш». Наиболее часто используются хеш-алгоритмы SHA-1 и MD5. Алгоритм SHA-1 создает 160-битовый хеш, а алгоритм MD5 – 128-битовый. Для достижения более высоких степеней безопасности могут использоваться SHA-256, SHA-384 и SHA-512, создающие, соответственно, значения размером 256, 384 и 512 бит. Все эти хеш-алгоритмы поддерживаются в .NET без установки каких-либо дополнительных компонентов. Поскольку эти алгоритмы необходимо использовать еще до использования самой цифровой подписи, мы начнем изучение этой темы с них. Затем, рассмотрев вопросы создания программ, реализующих цифровую подпись, мы узнаем, как генерируются, хранятся, импортируются и экспортируются ключи.

Хеш-алгоритмы

Как мы вскоре увидим, криптографические хеш-функции используются в технологии цифровой подписи потому, что они позволяют эффективно выявлять нарушения целостности сообщения. Хеш – это функция, которая ставит в соответствие небольшой, фиксированного размера объем двоичных данных произвольному, сколь угодно большому объему входных данных. Хеш называют также *дайджестом сообщения* или *отпечатком пальца*.

Характеристики хорошей хеш-функции

Хорошая хеш-функция должна обладать очень низкой вероятностью того, что два разных варианта входных данных дадут в результате одинаковые хеши. Это позволяет использовать небольшой по размеру хеш в качестве «представителя» объектов данных произвольного размера. Здесь можно провести аналогию с отпечатком человеческого пальца, который мал по размеру, однако настолько характерен для своего обладателя, что с его помощью можно идентифицировать человека. Для любого конкретного хеша крайне трудно подобрать другие входные данные такие, что давали бы в результате точно такой же хеш, этот факт и является защитой сообщения от подделки[1].

Простые хеш-функции широко используются в программном обеспечении, не связанном с криптографией, например, для обнаружения ошибок или для быстрого поиска объектов по хеш-таблице. Виртуальный метод **GetHashCode** объекта **Object** – это пример простой хеш-функции, не используемой для криптографических нужд. Чем же отличается криптографическая хеш-функция от такой, например, хеш-функции, как **GetHashCode**? Криптографическая хеш-функция обладает дополнительными свойствами, которые необходимы при использовании ее в криптографических целях таких, как хеширование паролей или цифровая подпись. Например, для криптографической хеш-функции невозможно вычислить входные данные по их хешу. Также (практически) невозможно при заданных входных данных подобрать другой вариант, который давал бы в результате такой же хеш.

Кроме того, для криптографической функции важно, чтобы она легко вычислялась. Здесь должен иметь место обычный компромисс между производительностью и достигаемым уровнем безопасности. К счастью, алгоритмы MD5, SHA-1, так же, как и SHA-256, SHA-384 и SHA-512 довольно эффективны, и их выбор позволяет установить баланс между производительностью и безопасностью. Самым быстрым алгоритмом является MD5, и он, конечно, обеспечивает самую низкую безопасность.

Желательно также, чтобы небольшие изменения в битах входных данных приводили к большим и непредсказуемым изменениям в вычисленном хеше. Это затрудняет задачу восстановления входных данных по выходным. Итак, идеальная криптографическая хеш-функция должна обладать следующими свойствами:

- входные данные могут обладать произвольными размерами;
- выходные данные всегда обладают небольшим, фиксированным размером, вытекающим из используемого алгоритма;

[1] В настоящее время не обнаружено никаких коллизий для MD5, SHA-1, SHA-256, SHA-384 или SHA-512. Для алгоритма MD5, который в режиме 128 бит является самым слабым из перечисленных, с использованием публично известных техник атаки методом «грубой силы» рабочий фактор пропорционален 2^{64}, что выходит за границы возможностей любой потенциальной нападающей стороны. Для алгоритма SHA-1 с его 160 битами рабочий фактор пропорционален 2^{80}, а в наше время такая атака недоступна никому. Если говорить об алгоритмах SHA-256, SHA-384 и SHA-512, то они, по всей вероятности, будут неуязвимы еще очень долго.

- функция быстро вычисляется;
- ее трудно обратить (то есть это односторонняя функция);
- вероятность возникновения коллизий невелика.

Последний пункт означает, что трудно найти два варианта входных данных, которые давали бы на выходе один и тот же хеш. Это может показаться странным на первый взгляд. Если вы можете иметь на входе данные произвольного размера, то число всевозможных вариантов входных данных бесконечно. Если при этом размер вычисляемого хеша фиксирован, то число возможных вариантов выходных данных конечно. Таким образом, очевидно, что должно существовать бесконечное число вариантов входных данных, дающих одинаковые хеши. Это похоже на парадокс. Однако, несмотря на бесконечное число вариантов, приводящих к коллизии, на самом деле найти хотя бы два варианта, дающих одинаковые хеши, чрезвычайно трудно! Таким образом, тот факт, что коллизии трудно найти, вытекает не из малого числа существующих коллизий (поскольку это число бесконечно велико). Коллизии трудно находить потому, что число вариантов сообщений, не приводящих к коллизиям, радикально больше. Безопасность хеша основывается на чрезвычайной трудности нахождения даже всего лишь одной пары вариантов сообщения, приводящей к коллизии!

Хеш-алгоритмы, поддерживаемые в .NET

Две наиболее часто используемые криптографические хеш-функции – это SHA-1 (Secure Hash Algorithm, хеширующий криптографический алгоритм), опубликованный NIST в середине 1990-х, и MD5 (Message Digest, дайджест сообщения), разработанный Р. Ривестом в начале 1990-х. В добавление к этому было опубликовано несколько новых версий SHA. Также для целей, связанных с аутентификацией сообщений, важную роль играет алгоритм ключевого хеша. Все упомянутые алгоритмы поддерживаются в .NET Framework в форме классов, производных от **Hash- Algorithm**[1]:

- **MD5**;
- **SHA1**;
- **SHA256**;
- **SHA384**;
- **SHA512**;
- **KeyedHashAlgorithm**.

[1] Здесь необходимо заметить, что метод **GetHashCode** класса **Object** создает 32-битовый хеш объекта, однако эта функция не предназначена для использования в криптографии. Ее назначение состоит в более простых операциях, таких, например, как поиск объекта по хеш-таблице и тому подобное.

На рисунке 5.1 изображена иерархия классов хеш-алгоритмов. Иерархию возглавляет абстрактный класс **HashAlgorithm**, производный от класса **Object**. Производные абстрактные классы **KeyedHashAlgorithm**, **MD5**, **SHA1**, **SHA256**, **SHA384** и **SHA512** представляют часто используемые криптографические хеш-алгоритмы.

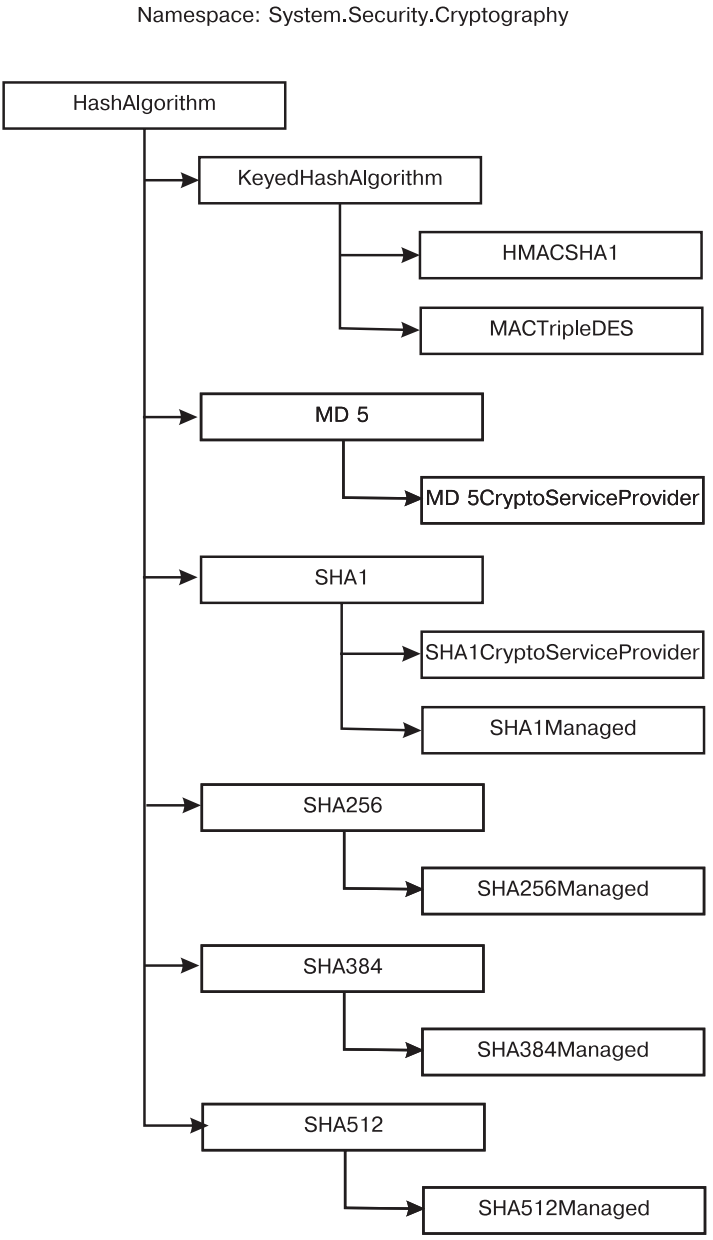


Рис. 5.1. Иерархия хеш-алгоритмов

Поскольку все это абстрактные классы, их невозможно использовать напрямую, создавая на их основе экземпляры объектов. Из каждого такого класса производятся конкретные классы реализации, которые уже используются напрямую. Классы, имена которых заканчиваются на CryptoServiceProvider, реализованы с использованием интерфейса CryptoAPI, предоставляемого операционной системой. Классы, имена которых заканчиваются на Managed, реализованы полностью средствами контролируемого C#-кода, без использования CryptoAPI.

Класс **HMACSHA1** производит ключевой хеш (Keyed-Hash Message Authentication Code – код аутентификации сообщения при помощи ключевого хеша или HMAC) при помощи хеш-функции SHA-1. Класс **MACTripleDES** производит ключевой хеш HMAC при помощи шифрования Triple DES[1], использованного в качестве хеш-функции. Ключевой хеш HMAC похож на цифровую подпись в том смысле, что его тоже можно использовать для верификации аутентичности и целостности сообщения, однако его отличие заключается в том, что для подтверждения обязательств его использовать нельзя. В то время как цифровая подпись представляет собой асимметричную схему, в которой только один из ключей хранится в секрете, ключевой хеш HMAC требует секретного хранения симметричного ключа у отправителя и получателя одновременно. Поскольку ключ известен более чем одной стороне, то любая третья сторона не сможет определить, кто из хранителей ключа создал хеш. Впрочем, хеши MAC меньше по размерам и вычисляются быстрее, чем цифровая подпись.

Хотя среда .NET допускает свободное расширение, и в ней можно использовать частные хеш-алгоритмы, реализованные в форме классов, производных от **HashAlgorithm**, мы настоятельно рекомендуем этого не делать! Для того чтобы достоверно подтвердить безопасность алгоритма, необходимо, чтобы он подвергся тщательному анализу многих экспертов на протяжении длительного времени. Широко распространено мнение, что любой выигрыш в секретности, достигнутый использованием частного, неизвестного публике алгоритма, радикально перекрывается проигрышем в безопасности, следующим из вероятности, что этот алгоритм может обладать уязвимыми местами. По этой причине лучше использовать известный, но хорошо проверенный алгоритм, такой, как MD5 или один из вариантов SHA.

Кроме хеш-алгоритмов, поддержанных в .NET, не существует других алгоритмов, которые использовались бы в промышленности сколько-нибудь широко[2]. Реализованные в Microsoft Windows интерфейсы CryptoAPI и CSP (поставщик услуг криптографии) поддерживают хеш-алгоритмы

---

[1] Некоторые криптографические хеш-алгоритмы, как, например, MD5 и SHA-1, были разработаны «с нуля», но есть и такие алгоритмы, что были созданы на основе существующих генераторов псевдослучайных чисел или симметричных алгоритмов шифрования. Класс **MACTripleDES** представляет собой пример хеша, основанного на симметричном алгоритме Triple DES. К сожалению, эта техника генерации хеша отличается малым быстродействием.

[2] Имеется ряд алгоритмов, предложенных к использованию (например, GOST, HAVAL, RIPE-MD, SNERFU и TIGER). Однако ни один из них не получил такого широкого распространения и не подвергся столь тщательной проверке, как MD5 или SHA-1.

MD4, MD5 и SHA-1. В среде .NET Framework не поддерживается MD4, поскольку поддержана его усовершенствованная версия MD5 (в MD4 была обнаружена критическая уязвимость, устраненная в MD5). Среда .NET Framework поддерживает алгоритмы SHA-256, SHA-384 и SHA-512, которые не поддержаны в CryptoAPI, поскольку являются относительно новыми стандартами.

Класс HashAlgorithm

Класс **HashAlgorithm** обладает публичным свойством **Hash**, которое представляет собой байтовый массив, содержащий в себе вычисленный хеш. Публичное свойство **HashSize** содержит значение размера хеша в битах. Наиболее важный публичный метод класса **HashAlgorithm** – метод **ComputeHash**, возвращающий значение хеша в виде байтового массива, вычисленное для заданных во входном параметре – байтовом массиве – входных данных. В следующем коде иллюстрируется применение класса **HashAlgorithm** на примере конкретного производного класса, инкапсулирующего алгоритм SHA-1. Предполагается, что входные данные уже существуют в форме байтового массива **messageByteArray**.

```
HashAlgorithm sha1  = new SHA1CryptoServiceProvider();
byte[] sha1Hash = sha1.ComputeHash(messageByteArray);
```

Классы MD5 и SHA

Класс **MD5** инкапсулирует алгоритм MD5, который производит 128-битовый хеш из входных данных произвольного размера. Алгоритм «дайджеста сообщения» MD5 определен стандартом RFC 1321[1]. Этот алгоритм первоначально предназначался для приложений цифровой подписи, в которых хеш (то есть дайджест сообщения) шифруется секретным ключом RSA. MD5 является расширением алгоритма MD4 (опубликованного Ривестом в 1990), при этом, работая немного медленнее, он обеспечивает более высокий уровень безопасности.

Класс **SHA1** инкапсулирует алгоритм SHA-1. Алгоритм SHA-1 может обрабатывать входные данные размером не более 264 бит, производя при этом хеш размером 160 бит. Этот дайджест сообщения можно использовать в алгоритме DSA[2], как мы увидим чуть ниже. Алгоритм SHA-1 был принят в качестве стандарта NIST под названием SHS (Secure Hash Standard – стандарт безопасного хеша), который опубликован в документе FIPS PUB 180-1[3]. Интересно отметить, в документе FIPS PUB 180-1 утверждается, что алгоритм SHA-1 основан на принципах, сходных с принципами алгоритма дайджеста сообщения MD4. Таким

[1] Подробней см. *http://www.faqs.org/rfcs/rfc1321.html*.
[2] Алгоритм DSA основывается на задаче дискретных логарифмов, первоначально предложенной Шнорром (Schnorr) и ЭльГамалем (ElGamal).
[3] Подробней см. *http://www.csrc.nist.gov/publications/fips/fips180-1.pdf*.

образом, две наиболее широко используемых хеш-функции, MD5 и SHA-1, тесно связаны друг с другом.

Классы **SHA256**, **SHA384** и **SHA512** инкапсулируют набор родственных хеш-алгоритмов, которые производят хеши размером 256, 384 и 512 бит. Эти новые алгоритмы, определенные документом FIPS PUB 180-2[1] в 2002 году, ведут свое происхождение от SHA-1. Они были созданы, поскольку со времени первоначальной публикации SHA-1 в 1993 году возникли опасения, что размер хеша в 160 бит не обеспечивает высоконадежной защиты на длительную перспективу. В то время как атака методом «грубой силы» на n-битовый симметричный алгоритм измеряется рабочим фактором, пропорциональным 2^n, такая же атака на n-битовый криптографический хеш, измеряется рабочим фактором, пропорциональным $2^{n/2}$. Причина кроется в том, что атака «по дню рождения»[2] (birthday attack) на хеш считается успешной, если найдены всего два варианта входных данных, дающих одинаковые хеши. Следовательно, рабочий фактор атаки на 160-битовый алгоритм SHA-1 пропорционален, практически, всего лишь 2^{80}. Это достаточно большое число, и такая атака, вероятно, недоступна почти для всех современных противников, но для критически важных случаев и в расчете на длительный период этот уровень безопасности, вероятно, уже недостаточен.

Интересно отметить, что новый стандарт симметричного шифрования AES, известный также под названием Rijndael, предусматривает три варианта, отличающихся длиной ключа: AES-128, AES-192 и AES-256. Вы можете заметить, что размеры хеша в алгоритме SHA (256, 384 и 512) в точности равны удвоенным длинам ключей в AES (128, 192, 256). Поскольку хеш уязвим для атаки методом «грубой силы» с рабочим фактором, равным половине рабочего фактора для атаки на симметричный шифр, такие размеры хеша точно уравнивают стойкость шифра и хеша. Это обстоятельство очень важно в случае, когда речь идет о криптографическом протоколе, использующим симметричный шифр и хеш одновременно. Стойкость протокола определяется стойкостью его самого слабого звена. Например, если вы используете очень сильный симметричный алгоритм в сочетании со слабым хешем (или наоборот), противник сможет атаковать ту часть протокола, которая обладает наименьшей стойкостью. Фактически, вы напрасно потратили процессорное время и другие ресурсы на реализацию самого сильного звена. И напротив, если вы сочетаете в протоколе алгоритмы равной стойкости, то без лишних затрат получаете точно рассчитанный уровень безопасности.

---

[1] Подробней см. *http://www.csrc.nist.gov/publications/fips/fips180-2.pdf*. Документ FIPS 180-2, который, кроме версий 256, 384 и 512, описывает также алгоритм SHA-1, заменяет документ 180-1.

[2] Выражение «атака по дню рождения» описывает метод «грубого» поиска коллизий для хеш-функции. Выражение это происходит из сценария в теории вероятности, который называют «Проблемой дня рождения». Сценарий относится к вычислению вероятности того, что в случайной группе людей найдется два человека с одинаковыми днями рождения. Хотя, интуитивно, такая вероятность кажется достаточно низкой, на самом деле минимальный размер группы, в которой эта вероятность будет равна 50%, составляет всего 23 человека.

Класс KeyedHashAlgorithm

Класс **KeyedHashAlgorithm** представляет собой интересную вариацию на основе концепции обычного хеша, в которой хеш вычисляется не только на основе входных данных, но также на основе дополнительной информации, играющей роль ключа. Алгоритм ключевого хеша, таким образом, представляет собой зависящую от ключа, одностороннюю хеш-функцию. Этот механизм удобен для аутентификации сообщения, поскольку создать или верифицировать такой хеш может лишь владелец ключа. Таким образом, алгоритм ключевого хеша обеспечивает одновременно контроль целостности и аутентификацию между доверяющими друг другу сторонами, которые заблаговременно обменялись ключами. Класс **KeyedHashAlgorithm** является абстрактным классом, из которого производятся классы **HMACSHA1** и **MACTripleDES**. Эти классы инкапсулируют алгоритмы ключевого хеша, основанные на алгоритмах SHA-1 и TrippleDES, соответственно.

Идентификаторы объектов

Программистам иногда требуются системы (конвенции) по наименованию объектов, которые позволяют давать однозначные и всем понятные имена множеству различных протоколов, форматов данных и алгоритмов. Международный стандарт ASN.1[1] OIDs (Object Identifiers – идентификаторы объектов) определен и поддерживается многими организациями, включая институт стандартов ANSI, и служит для формирования уникальных идентификаторов для компьютерных форматов, логически организованных в иерархию имен. Существует большое число идентификаторов OID, идентифицирующих конкретные протоколы, алгоритмы и форматы данных. В частности, большинство криптографических алгоритмов признаны институтом ANSI и получили уникальные идентификаторы OID. Например, некоторые идентификаторы приведены в таблице 5.1. Как мы убедимся, некоторые из этих идентификаторов OID необходимо использовать в определенных методах классов .NET Security Framework таких, как методы **SignHash** и **.VerifyHash** классов **RSACryptoServiceProvider** и **DSACryptoServiceProvider**.

В приведенном ниже фрагменте кода показано, как идентификатор OID используется в качестве параметра метода **SignHash** класса **RSACryptoServiceProvider**. Разумеется, здесь предполагается, что переменная (байтовый массив) **hashbytes** уже создана вызовом метода **ComputeHash** класса **SHA1**.

[1] ASN.1 (Abstract Syntax Notation number One – абстрактная синтаксическая нотация номер один) – это международный стандарт, специфицирующий данные, используемые в коммуникационных протоколах.

```
//создать объект RSA с ключом по умолчанию
RSACryptoServiceProvider rsa =
   new RSACryptoServiceProvider();

//подписать хеш при помощи OID для алгоритма SHA-1
signaturebytes =
   rsa.SignHash(hashbytes, "1.3.14.3.2.26");
```

Таблица 5.1. Идентификаторы ASN.1 OID для некоторых криптографических хеш-алгоритмов

Криптографический хеш-алгоритм	OID
MD5	1.2.840.113549.2.5
SHA-1	1.3.14.3.2.26
SHA-2	562.16.840.1.101.3.4.2.1
SHA-3	842.16.840.1.101.3.4.2.2
SHA-5	122.16.840.1.101.3.4.2.3

Как работает цифровая подпись

На рисунке 5.2 изображена общая схема применения цифровой подписи. В верхнем левом углу этой диаграммы мы берем исходное сообщение и создаем 160-битовый хеш (дайджест сообщения) при помощи алгоритма SHA-1. Затем дайджест сообщения шифруется при помощи секретного ключа, известного только его владельцу, то есть отправителю сообщения. Обратите внимание, это шифрование не имеет своей целью обеспечение секретности, поскольку используется секретный ключ, и любой желающий может расшифровать хеш, пользуясь общедоступным открытым ключом. Собственно, именно это и должен сделать получатель сообщения.

Результат этого шифрования и называют цифровой подписью. Никто другой, кроме владельца секретного ключа, не сможет создать такую подпись, даже располагая оригиналом исходного сообщения. По тем же причинам никто не сможет изменить сообщение или создать поддельное сообщение так, чтобы обман не раскрылся.

В правом верхнем углу диаграммы подписанное сообщение формируется объединением исходного сообщения, его цифровой подписи и открытого ключа, соответствующего тому секретному ключу, которым шифровался хеш. В таком виде подписанное сообщение пересылается получателю. Из диаграммы также видно, что открытый ключ должен быть публично доступен. Эффективный способ обеспечения такого доступа – использование центра сертификации.

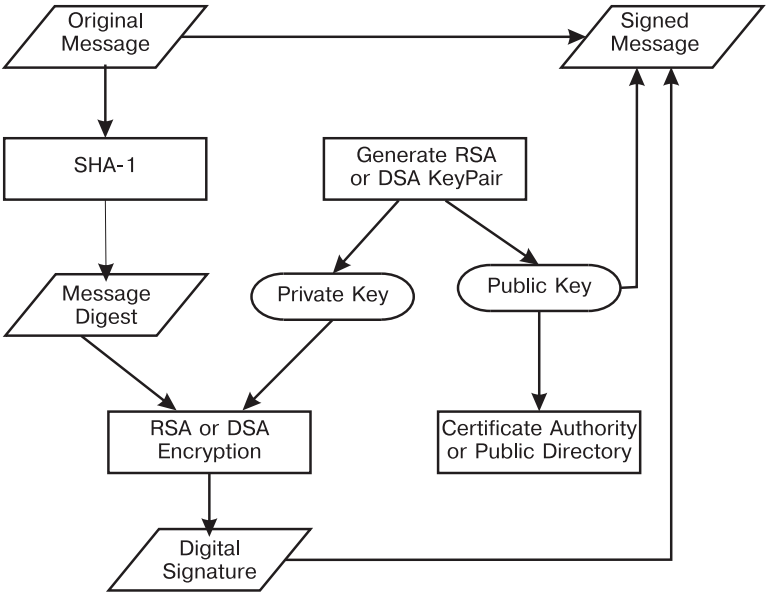


Рис. 5.2. Так работает цифровая подпись

Другой вариант нашей истории изображен на рисунке 5.3, где подписанное сообщение верифицируется получателем. Получатель хочет убедиться в том, что сообщение действительно отправлено отправителем, а не кем-либо еще. Также получатель хочет убедиться в том, что по пути следования сообщение не было кем-то изменено.

В левом верхнем углу диаграммы полученное сообщение разбивается на три компоненты – на исходное сообщение, открытый ключ и цифровую подпись. Для того чтобы сравнить шифрованный хеш с сообщением, его необходимо заново вычислить. Если вновь вычисленный хеш совпадет с расшифрованным хешем, то можно быть уверенным (в весьма высокой степени), что сообщение не изменено с момента наложения подписи. И наоборот, если хеши не совпадут, это будет означать, что исходное сообщение повреждено или изменено каким-то образом.

Итак, вычислив заново хеш сообщения, мы нуждаемся теперь в исходном хеше для сравнения. В правом верхнем углу диаграммы мы начинаем манипуляции с цифровой подписью. Хеш зашифрован секретным ключом отправителя, и для расшифровки мы можем использовать прилагающийся открытый ключ. Получатель извлекает открытый ключ и проверяет его – по крайней мере, в первый раз, – при помощи центра сертификации, играющего роль доверенной третьей стороны. Если ключ верифицирован успешно, он используется для дешифрования цифровой подписи и получения хеша, который теперь можно сравнить с хешем, который вычислен заново. Если хеши совпали, сообщение аутентично и не изменено. Предполагая, что секретный ключ отправителя никому более не доступен, можно сделать также вывод о подтверждении обязательств отправителя, вытекающем из его доказуемого авторства.

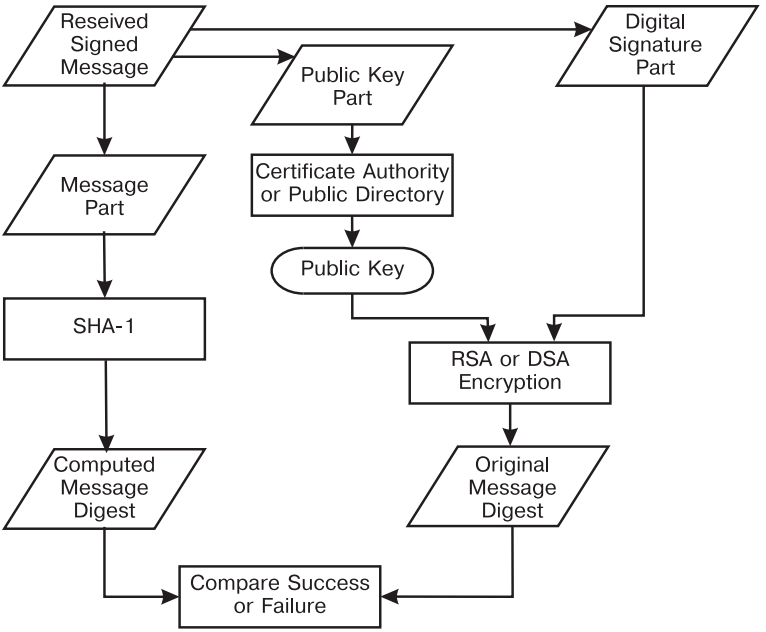


Рис. 5.3. Как верифицировать цифровую подпись

Стандарт DSS (Digital Signature Standard – Стандарт цифровой подписи), определенный документом FIPS PUB 186-2[1], предусматривает три алгоритма подписи: DSA, RSA (как указано в стандарте ANSI X9.31) и Elliptic Curve DSA (DSA с эллиптическими кривыми). Поскольку алгоритм Elliptic Curve DSA не поддержан в настоящее время в библиотеке классов .NET Security, мы рассмотрим здесь только DSA и RSA.

RSA в качестве алгоритма цифровой подписи

Алгоритм RSA можно использовать, как для обеспечения секретности, так и в целях цифровой подписи. В главе 4 мы видели, как RSA используется для обеспечения секретности (в особенности он удобен для обмена секретными симметричными сеансовыми ключами), здесь же мы рассмотрим его применение для реализации цифровой подписи. Собственно говоря, это просто другая сторона монеты: в любом шифровании открытым ключом с последующим дешифрованием секретным ключом достаточно перевернуть все вверх ногами, и получится цифровая подпись, для этого достаточно шифровать секретным ключом и дешифровать ключом открытым.

[1] См. *http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf*.

С точки зрения цифровой подписи нет необходимости в шифровании сообщения целиком. Вполне достаточно сгенерировать хеш исходного сообщения и зашифровать этот небольшой по размеру объект своим секретным ключом. Любой, располагающий соответствующим открытым ключом (то есть кто угодно), сможет расшифровать хеш и верифицировать его. Если дешифрованный хеш совпадет с вновь вычисленным хешем, получатель может увериться в том, что сообщение отправлено именно владельцем соответствующего секретного ключа и с момента наложения подписи никем не изменено. Уверенность эта проистекает из того обстоятельства, что крайне трудно подобрать второе сообщение, которое дало бы при вычислении точно такой же хеш.

Пример программы с использованием подписи RSA

Пример программы **RSASignature** показывает, как создать и верифицировать для сообщения подпись RSA. На рисунке 5.4 изображено окно программы **RSASignature**, в котором генерируется и верифицируется цифровая подпись. Вы можете создать цифровую подпись для сообщения щелчком на кнопке Create Signature, а затем, не меняя исходного сообщения, вы можете проверить подпись щелчком на кнопке Verify Signature. При этом появится окно сообщения, удостоверяющее, что подпись подлинна.

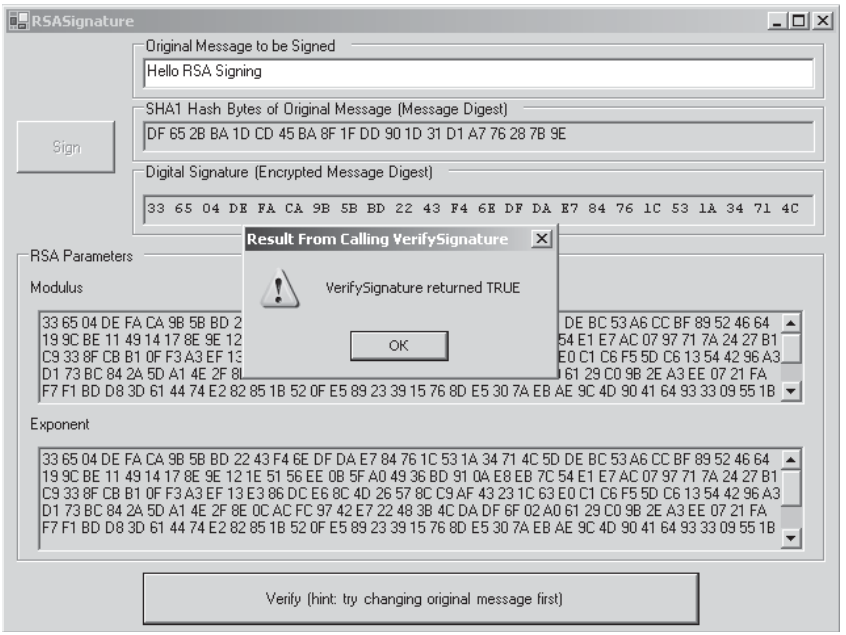


Рис. 5.4. Пример программы RSASignature

Если вы снова создадите цифровую подпись щелчком на кнопке Create Signature, а затем измените («подделаете») сообщение в текстовом поле перед тем, как щелкнуть на Verify Signature, то появится сообщение о недействительности подписи. Таким образом, программа обнаружит факт подделки сообщения.

Это, разумеется, достаточно искусственный пример, поскольку генерация подписи и ее проверка выполняются в пределах одного окна. Здесь это сделано для удобства, чтобы продемонстрировать все аспекты работы с цифровой подписью в одном компактном приложении. В реальной жизни применение цифровой подписи больше похоже на сценарий «клиент-сервер» или на распределенное приложение.

Метод **buttonSign_Click** (его код приведен ниже) накладывает подпись на сообщение. Код, относящийся к пользовательскому интерфейсу, мы игнорируем, поэтому можем здесь сосредоточиться на криптографических аспектах программы.

```
private void buttonSign_Click(
   object sender, System.EventArgs e)
{
   //получить исходное сообщение в виде байтового массива
   byte[] messagebytes = Encoding.UTF8.GetBytes(
           textOriginalMessage.Text);

   //создать дайджест сообщения алгоритмом SHA1
   SHA1 sha1  = new SHA1CryptoServiceProvider();
   byte[] hashbytes =
           sha1.ComputeHash(messagebytes);

   //отобразить значение хеша в шестнадцатеричном формате
   ...

   // создать объект RSA с ключом по умолчанию
   RSACryptoServiceProvider rsa =
           new RSACryptoServiceProvider();

   //создать хеш при помощи OID для алгоритма SHA-1
   signaturebytes =
           rsa.SignHash(hashbytes, "1.3.14.3.2.26");

   //извлечь параметры RSA, необходимые для верификации
   rsaparams = rsa.ExportParameters(false);

   //отобразить значение подписи в шестнадцатеричном формате
   ...

   //отобразить параметры RSAв шестнадцатеричном формате
   ...
   //обработать пользовательский интерфейс
   ...
}
```

Код этой процедуры генерирует хеш, который затем подписывается вызовом метода **SignHash** объекта **RSA**. Метод **ExportParameters** используется для сохранения открытого ключа в объекте **RSAParameters**, что позволит использовать его для верификации. Для верификации необходим только открытый ключ, а полная пара – открытый и секретный ключи – нужна лишь отправителю[1]. Поэтому подписывающая сторона создает объект **RSACryptoServiceProvider** и экспортирует затем только открытый ключ при помощи метода **ExportParameters** с параметром false.

Посмотрим теперь на метод **buttonVerify_Click**, который верифицирует цифровую подпись сообщения. Здесь заново вычисляется хеш сообщения, для того чтобы его можно было сравнить с подписью. Затем создается объект **RSA**, но на этот раз автоматически сгенерированный открытый ключ заменяется вызовом метода **ImportParameters** с использованием того объекта **RSAParameters**, что был создан ранее подписывающей процедурой. Благодаря этому параметры RSA будут идентичны параметрам, использованным при создании подписи. Хотя код, работающий с пользовательским интерфейсом, здесь не приведен, программа отображает результат вызова метода **buttonVerify_Click** при помощи окна сообщения.

```
private void buttonVerify_Click(
   object sender, System.EventArgs e)
{
   //получить сообщение, возможно, измененное,
   //в виде байтового массива
   byte[] messagebytes = Encoding.UTF8.GetBytes(
           textOriginalMessage.Text);

   // создать дайджест исходного сообщения алгоритмом SHA1
   SHA1 sha1  = new SHA1CryptoServiceProvider();
   byte[] hashbytes =
           sha1.ComputeHash(messagebytes);

   // создать объект RSA с импортированными параметрами
   RSACryptoServiceProvider rsa =
           new RSACryptoServiceProvider();
   rsa.ImportParameters(rsaparams);

   // верифицировать хеш, используя идентификатор OID для SHA-1
   bool match = rsa.VerifyHash(
           hashbytes, "1.3.14.3.2.26", signaturebytes);

   //вывести сообщение с результатом верификации
   ...
   //обработать пользовательский интерфейс
   ...
}
```

[1] Вспомните, что говорилось в главе 4 о шифровании с целью достижения секретности – этот сценарий противоположен описываемому.

Для передачи информации между этими двумя методами мы используем два поля. Первое из них – уже упоминавшийся объект **RSAParameters**. Этот объект инкапсулирует информацию открытого ключа. Второе поле – байтовый массив, содержащий цифровую подпись сообщения, сгенерированную первым методом. Хотя фактический текст сообщения тоже должен передаваться каким-то образом от подписывающего метода к методу, проверяющему подпись, здесь нет отдельного поля для него. Вместо этого текст хранится и извлекается обоими методами прямо в текстовом поле пользовательского интерфейса, благодаря чему пользователь может его изменять. Такая организация работы программы дает возможность наглядно убедиться в последствиях любой подделки сообщения. Разумеется, в реальном сценарии текст должен передаваться, как данные, между двумя приложениями.

```
//переменные для передачи информации между двумя процедурами
RSAParameters rsaparams;
byte[] signaturebytes;
```

Алгоритм цифровой подписи DSA

Алгоритм DSA является федеральным стандартом NIST, используемым совместно со стандартом SHA (Secure Hash Algorithm – хеширующий криптографический алгоритм). Институт NIST опубликовал первую версию алгоритма DSA, как составную часть стандарта DSS (Digital Signature Standard – Стандарт цифровой подписи, FIPS 186) в мае 1994 года. DSA основывается на задаче дискретных логарифмов, которая описывается в следующих двух разделах.

Как вы помните из главы 4, асимметричный алгоритм всегда основывается на некоторой односторонней функции. Одностороннюю функцию, на которой основывается алгоритм DSA, называют задачей дискретных логарифмов. Эта задача использует область, которую в абстрактной алгебре называют теорией групп. Группы – это достаточно сложная и обобщенная концепция, выходящая за пределы обычной школьной арифметики. Тем не менее, в наших целях достаточно понимать теорию групп с элементарной точки зрения. Давайте рассмотрим вкратце, что такое теория групп, а затем перейдем к задаче дискретных логарифмов, чтобы вновь далее вернуться к программированию.

Математическое основание: теория групп

Упрощенно группу можно рассматривать, как непустое множество (назовем его G) в сочетании с некоторой бинарной операцией, которую мы по традиции будем называть умножением.

По определению группа G должна подчиняться следующим четырем аксиомам:

- **Замыкание:** если a и b являются элементами G, то $a \cdot b$ также является элементом G.
- **Ассоциативность:** $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ верно для любых a, b и c, входящих в G.
- **Тождество:** G содержит в себе уникальный элемент, традиционно обозначаемый 1, такой, что $(a \cdot 1) = (1 \cdot a) = a$ для каждого элемента a, входящего в G. Этот элемент называют единичным.
- **Инверсия:** для каждого элемента a в G существует элемент a^{-1} такой, что $a \cdot a^{-1} = a^{-1} \cdot a = 1$. Элементы a^{-1} и a называют мультипликативными инверсиями.

Эти аксиомы могут напомнить вам элементарную арифметику с рациональными числами[1], и действительно множество рациональных чисел вместе с операцией умножения представляет собой группу согласно этим аксиомам. Однако заметим, что элементарная целочисленная арифметика не создает группу, поскольку инверсия для целых чисел не дает целое число. Например, инверсия 3 равна $1/3$, а это число уже не относится к целым. Существование инверсии критически важно для конструирования алгоритмов шифрования, поскольку дешифрование есть инверсия шифрования. При отсутствии инверсии вы можете зашифровать сообщение, но не сможете его расшифровать, а это делает шифрование бесполезным.

Однако для целых чисел еще не все потеряно! Если вы немного подправите идею целого числа, ограничив множество конечным диапазоном, а также измените идею операции умножения так, что результат умножения циклически повторяется внутри конечного диапазона, то сможете сконструировать группу на основе конечного множества целых чисел. Это называется модульной арифметикой или арифметикой по модулю.

Задача дискретных логарифмов использует конечную группу, над которой мы повторяем повторяющееся умножение при помощи модульной арифметики. Конкретно мы используем мультипликативную группу целых чисел по модулю, являющемуся простым числом. Причина, по которой модуль должен быть простым числом, заключается в следующем: это обстоятельство гарантирует, что для каждого элемента группы существует уникальная мультипликативная инверсия. Если выбрать модулем составное число, то для некоторых элементов множества инверсии могут совпасть, что приведет к коллизиям, которых следует избегать. Далее мы приведем таблицу умножения по модулю 7 (а это, заметим, простое число). Обратите внимание на тот факт, что каждая строка в этой таблице, кроме первой, представляет собой перестановку, и ни одно произведение

[1] Обратите внимание, мы говорим здесь о рациональных числах в строгом математическом смысле, а не о типах данных (плавающие, двойной точности), используемых в программировании для аппроксимации рациональных и вещественных чисел. К сожалению, плавающие числа и числа двойной точности не формируют группу, и потому для целей криптографии бесполезны.

в строке не равно другому произведению в этой же строке. Это означает, что для каждого числа в группе существует мультипликативная инверсия. Разумеется, наш простой пример ничего не доказывает для случая произвольных простых чисел. Мы не будет приводить здесь доказательство, поскольку оно потребовало бы много вспомогательных материалов, однако, если эта тема вам интересна, вы найдете доказательство в одной из множества книг по теории чисел или абстрактной алгебре.

Все, на чем мы здесь сконцентрируем внимание, это тот факт, что если модуль является простым числом, то мультипликативная инверсия действительно существует для каждого элемента в группе. Важность этого обстоятельства объясняется тем, что для использования в криптографической односторонней функции инверсия обязательно должна существовать. Вспомните, односторонняя функция отличается не отсутствием инверсии, а тем, что ее инверсию крайне трудно вычислить, однако существовать эта инверсия должна обязательно, иначе функция будет для нас бесполезна.

```
*|0 1 2 3 4 5 6 (модуль равен 7 – простое число)
-+-------------
0|0 0 0 0 0 0 0
1|0 1 2 3 4 5 6
2|0 2 4 6 1 3 5
3|0 3 6 2 5 1 4
4|0 4 1 5 2 6 3
5|0 5 3 1 6 4 2
6|0 6 5 4 3 2 1
```

В качестве противоположного примера давайте рассмотрим таблицу умножения по модулю, не являющемуся простым числом. Здесь мы приведем пример для числа 8, которое, конечно, не является простым. Если вы посмотрите на строки этой таблицы, то заметите, что некоторые произведения повторяются в одной строке неоднократно. Это говорит о том, что для некоторых значений мультипликативная инверсия не является уникальной, и потому не может быть использована в качестве основы для асимметричного алгоритма. Курсивом выделены числа, имеющие повторы в своей строке. Эти повторы показывают, что строка не является перестановкой, и, следовательно, инверсия неуникальна.

```
*|0 1 2 3 4 5 6 7 (модуль равен 8 – составное число)
-+---------------
0|0 0 0 0 0 0 0 0
1|0 1 2 3 4 5 6 7
2|0 2 4 6 0 2 4 6
3|0 3 6 1 4 7 2 5
4|0 4 0 4 0 4 0 4
5|0 5 2 7 4 1 6 3
6|0 6 4 2 0 6 4 2
7|0 7 6 5 4 3 2 1
```

Задача о дискретных логарифмах

Вы можете припомнить из элементарной математики, что многократное умножение числа само на себя называется возведением в степень или потенцированием, а обратную операцию называют логарифмированием и обозначают, как log. Например, следующие арифметические выражения иллюстрируют противоположность между операциями возведения в степень и логарифмом числа. Рассмотрите: выражение *10 в степени 2 равно 100*, и *логарифм 100 по основанию 10 равен 2*. В математической нотации это записывается следующим образом:

$$10^2 = 100$$

$$\log_{10}100 = 2$$

Вы уже, быть может, почувствовали одностороннюю природу этой схемы: вычислить степень числа довольно просто, поскольку это просто умножение, повторенное несколько раз. Вычисление логарифма, очевидно, гораздо более трудная задача. Оказывается, что разность в трудности вычисления в этих двух взаимообратных направлениях становится очень сильной с ростом размера чисел. В специальном разделе математики, теории сложности, говорится, что рабочий фактор вычисления степени числа растет полиномиально, а рабочий фактор вычисления логарифма числа растет экспоненциально. Поскольку размерность чисел мы можем менять произвольно, то можно достичь той точки, где трудность задачи с экспоненциальным ростом начинает взрывообразно опережать трудность задачи с полиномиальным ростом. Ура! Мы получили одностороннюю функцию.

Давайте обсудим задачу в алгебраических терминах, используя неизвестную величину x. Например, вам сказали: «10 в степени x равно 100. Найдите x». В математической нотации задача записывается следующим образом:

$$10^x = 100$$

Решение задачи:

$$x = \log_{10}100$$

Рассмотрим теперь эти же операции над элементами конечной модульной арифметической группы, которую обозначим Z_p[1], и которая содержит целые числа от 0 до $p - 1$ (подразумевается, что p – простое число, и что над группой определена арифметика по модулю p). Любой элемент g, выбранный из Z_p и возведенный в степень x, представляется при помощи традиционной нотации g^x. Это означает, что значение g умножено само на себя x раз по модулю p. Число g называется генератором группы, поскольку все элементы группы могут быть получены возведением

[1] Выше мы обозначали группу (в очень широком смысле) буквой G, но здесь нотация Z_p указывает на более узкое понятие – конечную группу целых чисел с арифметикой по модулю. Существуют и другие группы, не относящиеся к этой категории.

g в соответствующую степень. Задача о дискретных логарифмах теперь сводится к тому, чтобы при заданных элементах a и b, входящих в Z_p, найти целое число x такое, что $a^x = b$. Например, пусть $p = 19$, $a = 5$ и $b = 7$, найдем теперь x, исходя из условия:

$$5^x = 7(\text{mod } 19)$$

Решение для x будет выглядеть следующим образом:

$$x = \log_5 7(\text{mod } 19)$$

Это уравнение выражает x, как логарифм по модулю. Здесь использованы крохотные числа, но в случае большой разрядности чисел не существует известного способа решить эту задачу быстро и напрямую. Самые быстрые из известных способов решения обладают рабочим фактором, растущим экспоненциально по мере роста битового размера чисел.

В нашем примере вы можете использовать простой перебор, подставляя вместо x все числа от 0 до 18 и проверяя каждый раз уравнение. Затратив немного времени, вы обнаружите, что решением является число 6, поскольку, действительно, $56 = 7(\text{mod } 19)$. В нашем примере атака методом «грубой силы» не займет много времени, поскольку модуль 19 очень мал, и, следовательно, число возможных вариантов ничтожно. В конце концов, модуль 19 можно представить в виде 5-битового двоичного числа, но, что если размер модуля будет измеряться сотнями битов? Примите во внимание тот факт, что алгоритм DSA использует ключи размером от 512 до 1024 битов. Рабочий фактор атаки методом «грубой силы» удваивается с добавлением каждого бита, обеспечивая нам хороший запас прочности.

Хотя это до сих пор не доказано формально, задача о дискретных логарифмах, видимо, является очень эффективной односторонней функцией. Подобно тому, как в RSA используется задача разложения на простые сомножители, в алгоритме DSA используется задача о дискретных логарифмах. Однако параллель эта не полна, поскольку RSA можно использовать как для шифрования, так и как электронную подпись, в то время как DSA используется только как электронная подпись.

Заметьте, что секретный ключ известен только его владельцу, и если для генерации электронной подписи используется секретный ключ, то наложить подпись может только владелец ключа. Проверить же подпись может любое заинтересованное лицо, поскольку соответствующий открытый ключ общедоступен.

Подробности внутреннего устройства алгоритма DSA довольно сложны[1], но суть алгоритма состоит в том, что сообщение, рассматриваемое как число произвольной точности, возводится в степень, которая хранится в секрете. Верификация состоит в выполнении обратной операции, в которой вычисляется логарифм от подписи по модулю, при этом наличие секретного ключа ускоряет процесс. Поскольку отправитель посылает

[1] Основная идея дискретных логарифмов в роли односторонней функции довольно проста. Сложность алгоритма DSA проистекает из того факта, что для обеспечения «черного хода» к односторонней функции, что необходимо для существования секретного ключа, в него встроены дополнительные механизмы.

вместе с подписью исходное сообщение, получатель может выполнить верификацию и обнаружить подделку сообщения, если она имела место.

Так же, как мы видели в случае RSA, при большом размере сообщения неудобно и неэффективно подписывать все сообщение. Мы можем создать хеш (дайджест) сообщения и наложить подпись на него. Крайне трудно будет так подделать сообщение, чтобы его новый хеш совпал с исходным.

Как работает DSA

Здесь мы приведем «кровавые подробности» внутреннего устройства DSA. Как вы увидите, здесь используется не только задача о дискретных логарифмах сама по себе. Мы не будем отвлекаться на объяснения по поводу того, зачем нужен тот или иной шаг, и почему они работают именно таким образом. Наше описание только обрисует общую схему, по которой выполняются фактические реализации этого алгоритма. По описанию вы поймете, что реализация алгоритма DSA – совсем не тривиальная задача, и нам остается лишь порадоваться тому факту, что у нас есть готовые библиотеки (такие, как .NET Framework), где вся работа уже проделана!

ГЕНЕРАЦИЯ КЛЮЧА

1. Выберите простое число p с битовым размером 512, 576, 640, 704, 768, 832, 896, 960 или 1024.
2. Выберите 160-битовое простое число q, на которое число $p - 1$ делится без остатка.
3. Выберите числа g и h такие, что $g = ((h\hat{}((p - 1)/q)) \bmod p) > 1$ и $1 < h < p - 1$.
4. Выберите случайным образом секретный ключ x в диапазоне $0 < x < q$.
5. Вычислите $y = (g\hat{}x) \bmod p$.
6. Результирующий открытый ключ будет набором чисел (p, q, g, y).

НАЛОЖЕНИЕ ПОДПИСИ

1. Получите открытый текст m.
2. Выберите случайное число s такое, что $1 < s < q$.
3. Вычислите $r = ((g\hat{}s) \bmod p) \bmod q$.
4. Вычислите $s = ((H(m) - r \cdot x) \cdot s\hat{}(-1)) \bmod q$, где $H(m)$ – это хеш, вычисленный при помощи алгоритма SHA-1.
5. Результирующая подпись есть набор чисел (r, s).

ПРОВЕРКА ПОДПИСИ

1. Вычислите $w = (s\hat{}(-1)) \bmod q$.
2. Вычислите $u1 = (H(m) \cdot w) \bmod q$.
3. Вычислите $u2 = (r \cdot w) \bmod q$.
4. Вычислите $v = (((g\hat{}u1) \cdot (y\hat{}u2)) \bmod p) \bmod q$.
5. Подпись действительна, если $v = r$.

Иерархия класса AsymmetricAlgorithm

Иерархия наследования класса асимметричного алгоритма, располагающаяся в пространстве имен **System.Security.Cryptography**, приведена на рисунке 5.5. На рисунке не показано, что **AsymmetricAlgorithm** производится из класса **Object**. Как вы помните по главе 4, класс **AsymmetricAlgorithm** является абстрактным классом, и из него производятся классы **DSA** и **RSA**, которые также являются абстрактными. Из классов **DSA** и **RSA** затем производятся классы **RSACryptoServiceProvider** и **DSACryptoServiceProvider**, которые и обеспечивают реализацию алгоритмов RSA и DSA. Оба они используют поддержку CryptoAPI. Вы можете самостоятельно реализовать RSA или DSA или воспользоваться сторонней реализацией, но все же в большинстве случаев рекомендуется использовать реализацию .NET.

Давайте рассмотрим теперь публичные методы и свойства класса **DSACryptoServiceProvider**. Как и вообще для любого симметричного или асимметричного алгоритма, конструктор этого класса автоматически генерирует ключевую информацию в момент создания экземпляра.

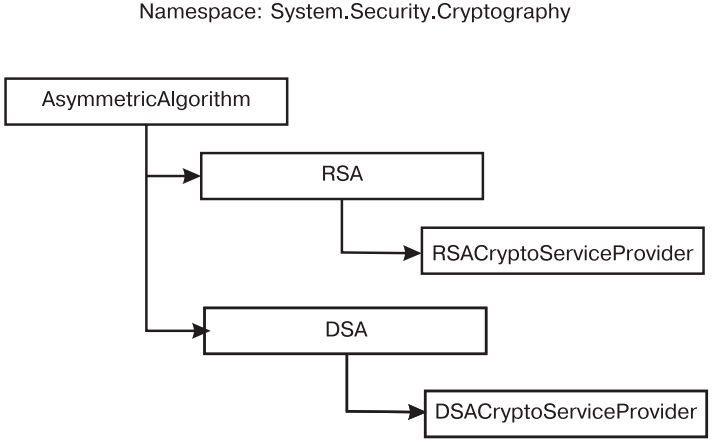


Рис. 5.5. Иерархия класса асимметричного алгоритма

Класс DSACryptoServiceProvider

Вот публичные свойства класса **DSACryptoServiceProvider**:

- **KeyExchangeAlgorithm** получает имя алгоритма обмена ключами;
- **KeySize** получает размер ключа в битах;
- **LegalKeySizes** получает разрешенные размеры ключей;
- **PersistKeyInCsp** определяет, должен ли ключ быть приоритетным в CSP;
- **SignatureAlgorithm** получает имя алгоритма цифровой подписи.

Также класс **DSACryptoServiceProvider** обладает несколькими публичными методами, которые могут вам понадобиться. В функциях, обеспечиваемых этими методами, наблюдается некоторая избыточность, а это означает, что одну и ту же задачу вы можете решить разными способами. Вот наиболее важные из публичных методов класса **DSACryptoServiceProvider**:

- **CreateSignature** создает подпись DSA для заданного сообщения;
- **VerifySignature** верифицирует подпись DSA для заданного сообщения;
- **SignData** вычисляет хеш сообщения и подписывает результат;
- **VerifyData** верифицирует заданную подпись, сравнивая ее с подписью, вычисленной для заданного сообщения;
- **SignHash** вычисляет подпись для заданного значения хеша;
- **VerifyHash** верифицирует заданную подпись, сравнивая ее с подписью, вычисленной для заданного хеша;
- **ToXmlString** создает и возвращает XML-представление текущего объекта DSA;
- **FromXmlString** создает объект DSA из XML-данных;
- **ExportParameters** экспортирует параметры DSA в объект **DSAParameters**;
- **ImportParameters** импортирует параметры DSA из объекта **DSAParameters**.

Пример программы с использованием DSA

Пример программы **DSAAlgorithm** демонстрирует, как можно создавать и верифицировать подпись DSA. Вы увидите здесь много черт сходства с примером программы **RSASignature**, который рассматривался выше. В конце концов, хотя объекты **DSA** и **RSA** реализуют разные алгоритмы, они используются для решения одной и той же задачи. На рисунке 5.6 приведен вид окна программы **DSAAlgorithm**, которая генерирует и верифицирует цифровую подпись для заданного сообщения.

Как видите, пользовательский интерфейс устроен аналогично интерфейсу предыдущей программы. Вы вводите сообщение, создаете цифровую подпись, а затем, изменив сообщение или оставив его неизменным, верифицируете подпись. Появившееся окно сообщения укажет на результат проверки.

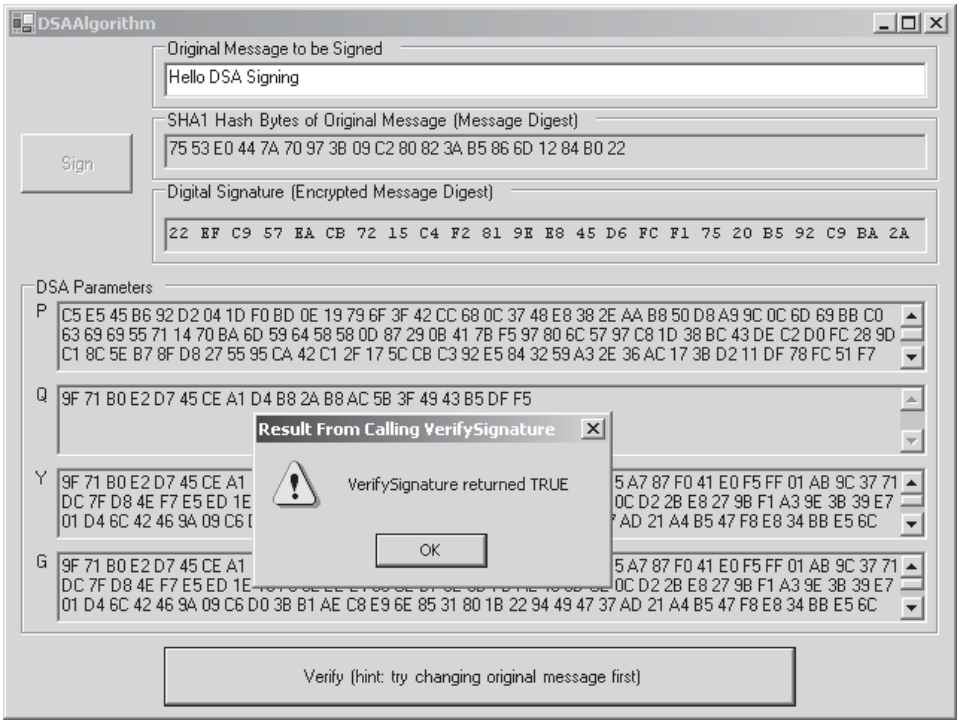


Рис. 5.6. Пример программы DSAAlgorithm

Метод **buttonSign_Click**, подписывающий сообщение, приведен ниже. Здесь также полностью игнорируется пользовательский интерфейс. Пример кода DSA в основном устроен аналогично коду RSA.

```
private void buttonSign_Click(
   object sender, System.EventArgs e)
{
   //получить исходное сообщение в виде байтового массива
   byte[] messagebytes = Encoding.UTF8.GetBytes(
       textOriginalMessage.Text);

   //создать дайджест сообщения алгоритмом SHA1
   SHA1 sha1  = new SHA1CryptoServiceProvider();
   byte[] hashbytes =
     sha1.ComputeHash(messagebytes);

   //отобразить значение хеша в шестнадцатеричном формате
   ...

   //создать объект DSA с ключом по умолчанию
   DSACryptoServiceProvider dsa =
     new DSACryptoServiceProvider();
```

```
   //подписать хеш при помощи OID для алгоритма SHA-1
   signaturebytes =
     dsa.SignHash(hashbytes, "1.3.14.3.2.26");

   //извлечь параметры RSA, необходимые для верификации
   dsaparams = dsa.ExportParameters(false);

   //отобразить значение подписи в шестнадцатеричном формате
   ...
   //обработать пользовательский интерфейс
   ...
}
```

Метод **buttonVerify_Click** выполняет верификацию подписи для заданного сообщения. Он также устроен аналогично своему предшественнику в предыдущей программе. После вычисления хеша сообщения метод выполняет его верификацию.

```
private void buttonVerify_Click(
   object sender, System.EventArgs e)
{
   //получить сообщение, возможно, измененное,
   //в виде байтового массива
   byte[] messagebytes = Encoding.UTF8.GetBytes(
      textOriginalMessage.Text);

   //создать дайджест исходного сообщения алгоритмом SHA1
   SHA1 sha1  = new SHA1CryptoServiceProvider();
   byte[] hashbytes =
      sha1.ComputeHash(messagebytes);

   //создать объект DSA с импортированными параметрами
   DSACryptoServiceProvider dsa =
      new DSACryptoServiceProvider();
   dsa.ImportParameters(dsaparams);

   // верифицировать хеш, используя идентификатор OID для SHA-1
   bool match = dsa.VerifyHash(
      hashbytes, "1.3.14.3.2.26", signaturebytes);

   //вывести сообщение с результатом верификации
   ...
   //обработать пользовательский интерфейс
   ...
}

//переменные для передачи информации между двумя процедурами
DSAParameters dsaparams;
byte[] signaturebytes;
```

Мы разобрались на примерах программ **DSAAlgorithm** и **RSASignature**, что способы работы с цифровой подписью через посредство классов **DSACryptoServiceProvider** и **RSACryptoServiceProvider** практически идентичны. Именно такой однородности и следует ожидать от столь хорошо организованной библиотеки классов, как .NET. Вы просто вызываете метод **SignHash**, для того чтобы подписать хеш, а затем вызываете метод **VerifyHash**, чтобы выполнить верификацию, и нет никакой разницы, работаете ли вы с RSA или с DSA.

Если сообщение было изменено после наложения подписи, то метод **VerifyHash** этот факт обнаружит. Если секретный ключ, который сгенерировался при вызове **SignHash**, не соответствует открытому ключу, используемому при верификации подписи, метод **VerifyHash** также обнаружит этот факт. Таким образом, цифровая подпись обеспечивает аутентификацию и целостность сообщения.

Если вы посмотрите на программный код в этих двух примерах, то заметите, что не предпринимается никаких специальных мер для явной генерации пары ключей и для передачи открытого ключа. Фактически, пара ключей генерируется автоматически в момент создания экземпляра **DSACryptoServiceProvider** или **RSACryptoServiceProvider**. Затем открытый ключ передается между отправителем и получателем при помощи методов **ExportParameters** и **ImportParameters** объектов **DSACryptoServiceProvider** или **RSACryptoServiceProvider**.

Мы продемонстрировали лишь одну из нескольких основных техник. Например, вместо явного создания байтового массива для хеша и последующего обращения к методам **SignHash** и **VerifyHash**, можно достичь того же результата, вызвав метод **SignData**, который автоматически создает новый хеш и подписывает его (проверить хеш далее можно вызовом **VerifyData**). Методы **SignData** и **VerifyData** не требуют предварительного создания хеша в явном виде, и им не нужно указывать идентификатор OID, поскольку они «понимают» простые имена хеш-алгоритмов такие, как «**SHA1**» или «**MD5**».

В случае DSA возможен еще один подход, заключающийся в вызове метода **CreateSignature**, который создает байтовый массив с цифровой подписью напрямую из заданного байтового массива с хешем. В этом случае обращение к методу **VerifySignature**, которому необходимо передать в байтовых массивах подпись и оригинальный хеш, вернет логическое значение, указывающее на результат проверки подписи.

Между RSA и DSA есть два небольших отличия, которые необходимо отметить. В отличие от класса **DSACryptoServiceProvider**, класс **RSACryptoServiceProvider** не поддерживает методы **CreateSignature** и **VerifySignature**. Второе различие проистекает из того факта, что алгоритм DSA не может использоваться для шифрования данных. По этой причине класс **DSACryptoServiceProvider** не поддерживает методы **Decrypt** и **Encrypt**, которые имеются в классе **RSACryptoServiceProvider**. За исключением этих двух различий оба класса обладают идентичными наборами публичных свойств и методов.

Итоги главы

Эта глава познакомила вас с цифровой подписью, основанной на криптографическом хеше и асимметричном алгоритме. Подробно мы рассмотрели алгоритмы RSA и DSA. Вкратце были изучены математические основания этих алгоритмов, также мы увидели в действии примеры программ, реализующих эти алгоритмы с использованием классов **DSACryptoServiceProvider** и **RSACryptoServiceProvider**. В главе 6 мы продолжим тему цифровой подписи, сосредоточившись теперь на криптографии XML и ее использовании в сочетании с цифровой подписью.

Глава 6

Криптография и XML

Расширяемый язык разметки[1] XML (Extensible Markup Language) в последнее время стал, вероятно, самой распространенной из новых технологий в компьютерном мире. Этот язык можно обнаружить в самых неожиданных областях – от издательского программного обеспечения до обработки распределенных транзакций на основе SOAP. Фактически, XML доказал свою полезность во всех ситуациях, где данные структурируются в некий стандартный формат. В любом подобном приложении XML делает данные независимыми от языка программирования и от платформы. В сущности, XML представляет собой универсальные структурированные данные, для обработки которых можно создавать многократно используемые программные библиотеки.

Поскольку криптография также очень широко используется при обмене данными, нет ничего удивительного в том факте, что между криптографическими технологиями и технологией XML имеется множество областей взаимного интереса. Самый очевидный подход заключается в применении криптографических алгоритмов к XML-данным, например, в целях шифрования или электронной подписи. Результатом этого подхода явились два новых важных стандарта консорциума W3C, именуемые XML Encryption (*шифрование XML*) и XML Signatures (*XML-подпись*), которые и являются темой данной главы.

XML Encryption – шифрование XML

Одна из самых привлекательных черт XML – данные на этом языке гораздо лучше подходят для восприятия человеком, чем данные в наиболее распространенных частных или традиционных двоичных форматах. Просто откройте рабочий лист Excel в текстовом редакторе Блокнот и попробуйте понять что-нибудь в нем. В документах XML все теги разметки, структурирующие документ, сами помогают читателю интерпретировать значение размеченных данных. Это поможет вам, при необходимости, разобраться в структуре документа даже весьма сложного, такого, который

[1] В этой главе предполагается, что вы в какой-то степени уже знакомы с технологией XML. Для справок вы можете использовать сайт *www.xml.org*. Более подробные и исчерпывающие сведения вы найдете в Рекомендации W3C по XML 1.0, которая доступна по адресу *http://www.w3.org/TR/REC-xml*.

используется в высокоструктурированных данных, относящихся к сложным приложениям. Легкость чтения данных XML особенно полезна для программистов, которым часто приходится читать входные и выходные данные при разработке и отладке программ. Разумеется, легкость чтения является достоинством лишь в том случае, если вы предполагаете, что данные будут читаться людьми. В большинстве случаев это не так, особенно, если приложение для этих данных уже разработано, эксплуатируется и обрабатывает реальные данные. Для наиболее чувствительных элементов данных в XML-документах можно применить выборочное шифрование. В этом случае отдельные элементы в документе станут недоступны для человеческого восприятия.

XML Encryption против SSL/TLS

Уже существуют протоколы шифрования и защиты в Internet такие, как протокол защищенных сокетов SSL, протокол защиты транспортного уровня TLS[1] и IPSec[2], которые можно использовать для обеспечения конфиденциальности и защиты целостности данных между двумя сообщающимися приложениями. В шифровании XML-данных есть два аспекта, отличающие этот процесс от шифрования вышеупомянутыми протоколами. Первое отличие состоит в том, что в XML-документе можно зашифровать лишь отдельные элементы документа, оставив прочие элементы открытыми. Второе отличие связано с использованием данных – при шифровании XML-данных не имеет значения, передаются ли эти данные в сеансе связи между двумя приложениями или же хранятся на некотором носителе (например, в дисковом файле или в записях базы данных), где к ним получают доступ в разное время разные приложения. В противоположность этому протоколы SSL, TLS и IPSec защищают сеанс соединения между приложениями, как единое целое, но не могут использоваться для шифрования хранимых данных. Шифрование XML-данных не отменяет и не заменяет упомянутые протоколы, оно используется для решения совершенно другого класса задач безопасности. Вот две главные задачи, в которых используют шифрование XML:

- выборочное шифрование отдельных элементов документа;
- шифрование структурированных данных, хранящихся на носителе и предназначенных для разделяемого доступа со стороны многих приложений.

---

[1] Протокол SSL был разработан компанией Netscape для обеспечения конфиденциальности и защиты целостности данных, которыми обмениваются два приложения в сети. Протокол SSL составил основу протокола TLS, который определен стандартом RFC2246. Протоколы SSL и TLS в основе своей одно и то же, и потому их часто обозначают, как «SSL/TLS».

[2] Протокол IPSec обеспечивает криптозащиту на уровне пакетов, при этом подразумевается реализация протокола VPN.

Спецификация шифрования XML

Подробное описание шифрования XML можно найти в официальной спецификации XML Encryption Syntax and Processing (Обработка и синтаксис шифрования XML)[1]. Здесь содержится общее описание с примерами, а также XML-схема[2], определяющая синтаксис шифрования XML. Во время написания этой книги статус спецификации остается на уровне Предложения[3], и это означает, что в итоговую Рекомендацию могут быть еще внесены изменения. Фактически в .NET пока не предусмотрена поддержка этой спецификации. По этой причине в настоящее время нам придется написать собственный код для реализации такой поддержки.

Создание библиотеки классов .NET, реализующей спецификацию в полной мере – крупная задача, которую мы не будем пытаться здесь решить. Вместо этого мы прямо в коде программы реализуем функции, необходимые для шифрования отдельных элементов в простом XML-документе. Мы сделаем это, используя имеющиеся классы для работы с XML и классы для шифрования. К счастью, как мы убедимся далее в этой главе, спецификация подписи XML уже приняла завершенный вид, и потому в .NET для нее предусмотрена более полная поддержка.

Что обеспечивает шифрование XML

Шифрование XML обеспечивает стандартизованные средства для шифрования структурированных данных и представляет результат в виде XML-документа. Это позволяет вам шифровать любые данные, будь то XML-документ целиком, отдельные элементы XML-документа или вообще произвольные внешние данные, не представленные в формате XML. Результатом будет шифрованный XML-элемент, либо непосредственно содержащий данные, либо ссылающийся на них. В любом случае зашифрованные данные будут доступны для приложения, поддерживающего технологию XML, не зависящую от платформы.

Общие концепции криптографии, с которыми мы познакомились в предыдущих главах, включая симметричное и асимметричное шифрование, остаются в силе и при шифровании XML-данных. Разница заключается в том, что мы здесь используем теги разметки XML для представления такой информации, как ключи, алгоритмы и, конечно, собственно данные, которые содержит или на которые ссылается XML-документ.

---

[1] Эта спецификация W3C доступна по адресу *http://www.w3.org/TR/xmlenc-core/*.

[2] Более подробные сведения об этой XML-схеме можно найти по адресам *http://www.w3.org/ TR/2001/REC-xmlschema-1-20010502/* и *http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/*.

[3] Предложение (Предложенная Рекомендация) концерна W3C – это проект, отвечающий всем требованиям, предъявленным соответствующей Рабочей Группой, однако не одобренный еще для повсеместного применения. Такой документ рассматривается как черновик, который может быть еще изменен, заменен или вообще снят с повестки дня в любой момент.

Синтаксис XML Encryption

Синтаксис шифрования XML ясно и недвусмысленно определен спецификацией W3C, и потому мы не будем здесь полностью дублировать эту информацию. Вместо того, чтобы подробно описывать каждый тег и каждое правило, мы сделаем краткий обзор ключевых элементов этого синтаксиса и приведем простой пример, иллюстрирующий его использование.

ПРОСТРАНСТВО ИМЕН XML ENCRYPTION

Пространства имен[1] используются в XML по той же причине, что и во многих языках программирования, как C#, Java и C++. Пространства имен решают проблему коллизий имен, обеспечивая универсальные имена, область действия которых выходит за рамки отдельного XML-документа.

Каждая спецификация синтаксиса XML определяет набор тегов (то есть элементов и атрибутов), который называется словарем синтаксической разметки. Однако зачастую приложения работают с документами, созданными при помощи различных словарей. Следовательно, неизбежно возникает проблема конфликта имен, относящихся к разным словарям синтаксической разметки. Благодаря тому, что каждый словарь разметки определен в собственном пространстве имен, коллизий удается избежать. По этой причине спецификация шифрования XML определяет собственное пространство имен:

```
Xmlns:xenc="http://www.w3c.org/2001/04/xmlenc#"
```

В добавление к тому, что они определяют область действия синтаксиса, пространства имен используются в качестве префикса для идентификаторов алгоритмов. Вот несколько примеров идентификаторов алгоритмов, определенных спецификацией. Вы должны помнить, что любой алгоритм, такой как RSA, 3DES, SHA01 и тому подобный, должен быть идентифицирован.

```
http://www.w3.org/2001/04/xmlenc#rsa-1_5
http://www.w3.org/2001/04/xmlenc#tripledes-cbc
http://www.w3.org/2001/04/xmlenc#aes128-cbc
http://www.w3.org/2001/04/xmlenc#aes256-cbc
http://www.w3.org/2001/04/xmlenc#aes192-cbc
http://www.w3.org/2001/04/xmlenc#rsa-1_5
http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p
http://www.w3.org/2000/09/xmldsig#sba1
http://www.w3.org/2001/04/xmlenc#sba256
http://www.w3.org/2001/04/xmlenc#sba512
http://www.w3.org/2001/04/xmlenc#ripemd160
```

[1] Более подробные сведения о пространствах имен XML можно найти по адресу *http://www.w3.org/TR/1999/REC-xml-names-19990114/*.

ЭЛЕМЕНТЫ ШИФРОВАНИЯ XML

Главным синтаксическим компонентом, используемым в шифровании XML, является элемент EncryptedData. Все остальные элементы являются его потомками. В этом разделе мы кратко рассмотрим элементы шифрования XML, включая элемент EncryptedData и производные от него элементы.

- Элемент **EncryptedData** является самым общим элементом, заключающим в себе (фактически или при помощи ссылки) как собственно шифрованные данные, так и вспомогательную информацию, необходимую для дешифрования, такую как секретный сеансовый ключ и указание на использованный алгоритм. Если этот элемент является корневым элементом документа, то зашифрованным оказывается весь документ. В документе может содержаться несколько экземпляров этого элемента, но ни один из таких элементов не может содержать в себе другой элемент EncryptedData или сам содержаться в нем.
- Элемент **EncryptionMethod** является потомком элемента EncryptedData и содержит в себе указание на алгоритм шифрования (используется один из идентификаторов алгоритмов, упомянутых в предыдущем разделе). Если этот элемент отсутствует, то алгоритм должен быть известен приложению благодаря каким-то другим средствам.
- Элемент **ds:KeyInfo** является потомком элемента EncryptedData и содержит в себе симметричный сеансовый ключ, используемый для шифрования и дешифрования. Если этот элемент в документе отсутствует, то ключ должен быть передан каким-то другим способом. Элемент ds:KeyInfo можно использовать в элементе EncryptedData, содержащем шифрованные данные, однако можно также создать XML-документ с элементами EncryptedData и ds:KeyInfo, где содержится только ключ. Схема XML для элемента ds:KeyInfo определена в отдельной спецификации, посвященной цифровой подписи XML.
- Элемент **EncryptedKey** является потомком элемента ds:KeyInfo, используемым для хранения симметричного сеансового ключа. Элемент EncryptedKey сходен с элементом EncryptedData, только используется он для хранения ключевой, а не произвольной информации.
- Элемент **AgreementMethod** является потомком элемента ds:KeyInfo, он служит для задания метода разделения секретного сеансового ключа. Если этот элемент отсутствует, то соглашение о разделении ключей должно быть реализовано каким-то другим образом.
- Элемент **ds:KeyName** является потомком элемента ds:KeyInfo, и предназначен для доступа к секретному сеансовому ключу по имени, которое может прочесть человек.

- Элемент **ds:RetrievalMethod** является потомком элемента ds:KeyInfo, он обеспечивает ссылку URI на другой элемент ds:KeyInfo, где содержится секретный сеансовый ключ.
- Элемент **CipherData** является обязательным элементом, потомком элемента EncryptedData, который содержит собственно шифрованные данные или ссылку на них. Если он содержит данные, то они заключаются в элементе CipherValue, если же он ссылается на данные, то ссылка заключается в элемент CipherReference. Это единственный потомок элемента EncryptedData, который является обязательным. Это легко понять, поскольку нет смысла в элементе EncryptedData, в котором не содержится никаких данных.
- Элемент **CipherValue** заключает в себе шифрованные данные.
- Элемент **CipherReference** заключает в себе ссылку на шифрованные данные, хранимые в другом месте.
- Элемент **EncryptionProperties** содержит вспомогательную информацию, специфичную для конкретного приложения, такую, как время и дата шифрования и тому подобное.

Давайте рассмотрим теперь, как все эти элементы используются в шифрованном XML-документе. Самым общим элементом является EncryptedData, который может содержать в себе свои необязательные дочерние элементы: EncryptionMethod, ds:KeyInfo и EncryptionProperties. Также в нем должен содержаться один обязательный элемент CipherData. В элементе ds:KeyInfo могут содержаться различные необязательные элементы с информацией, касающейся сеансового ключа. В элементе CipherData может содержаться один из двух элементов: CipherValue или CipherReference. Синтаксис элемента EncryptedData описывается следующей структурой, где знак ? соответствует ни одному или одному экземпляру, а знак * соответствует ни одному или нескольким экземплярам[1].

```
<EncryptedData Id? Type? MimeType? Encoding?>
   <EncryptionMethod/>?
   <ds:KeyInfo>
              <EncryptedKey>?
              <AgreementMethod>?
              <ds:KeyName>?
              <ds:RetrievalMethod>?
              <ds:*>?
   </ds:KeyInfo>?
   <CipherData>
              <CipherValue>?
              <CipherReference URI?>?
   </CipherData>
   <EncryptionProperties>?
</EncryptedData>
```

[1] DOM – Document Object Model, объектная модель документа. Эта спецификация доступна по ссылке *http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20021022/*.

Чтобы этот синтаксис был понятней, давайте посмотрим, как элемент EncryptedData используется в реальном примере. Это несложный XML-документ, который будет использоваться в примере программы, который ожидает нас впереди. Рассмотрим этот документ до и после шифрования. Вот содержимое исходного документа до шифрования:

```
<invoice>
   <items>
            <item>
         <desc>Deluxe corncob pipe</desc>
                  <unitprice>14.95</unitprice>
                  <quantity>1</quantity>
            </item>
   </items>
   <creditinfo>
            <cardnumber>0123456789</cardnumber>
            <expiration>01/06/2005</expiration>
            <lastname>Finn</lastname>
            <firstname>Huckleberry</firstname>
   </creditinfo>
</invoice>
```

Как видим, перед нами обычный счет, где приведен список купленных товаров – эта информация не является чувствительной и не нуждается в шифровании. Однако далее следует информация о номере кредитной карточки, и она, конечно, в шифровании нуждается. Далее приведено содержимое документа после шифрования чувствительной информации:

```
<invoice>
   <items>
            <item>
                  <desc>Deluxe corncob pipe</desc>
                  <unitprice>14.95</unitprice>
                  <quantity>1</quantity>
            </item>
   </items>
   <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element">
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <KeyName>My 3DES Session Key</KeyName>
      </ds:KeyInfo>?
      <CipherData>
            <CipherValue>tu7ev6nuRCYUgZpN0ZABz+VJvL...
            </CipherValue>
      </CipherData>
   </EncryptedData>
</invoice>
```

Этот документ отличается от оригинала тем, что элемент с информацией о счете полностью заменен элементом EncryptedData. Элемент CipherValue содержит зашифрованные данные, представляющие собой элемент creditinfo из исходного документа. В элементе ds:KeyInfo содержится имя сеансового ключа, который необходимо использовать для дешифрования.

Если вы посмотрите на атрибут Type в определении элемента EncryptedData, то обнаружите, что речь идет скорее об элементе, а не о содержимом (#Element). Это означает, что мы шифруем не содержимое элемента, а весь элемент целиком. Вы сами должны решить в своем приложении, что больше вам подходит. В зависимости от характера вашей информации, может быть достаточно скрыть только содержимое элемента, но, возможно, потребуется скрывать весь элемент. Понятно, что безопаснее скрывать элемент полностью, поскольку в этом случае у атакующей стороны не будет даже возможности проявить интерес к содержимому. Однако специфика приложения может требовать, чтобы теги разметки элемента были открыты, и в этом случае шифровать нужно будет только его содержимое.

Важно понимать, что информация о счете в нашем примере – не единственное, что подлежит шифрованию. Информация о счете зашифрована при помощи сеансового симметричного ключа, однако сам ключ также должен быть зашифрован открытым асимметричным ключом так, чтобы только получатель сообщения, владелец секретного асимметричного ключа, мог прочесть информацию о счете. Мы могли бы внедрить в документ собственно сам сеансовый симметричный ключ, для этого потребовалось бы добавить элемент CipherValue с зашифрованными данными симметричного сеансового ключа. Вместо этого мы указали лишь имя ключа. В нашем примере мы используем для размещения ключа отдельный документ, листинг которого приведен ниже. Обратите внимание, в этом документе элементом самого верхнего уровня является EncryptedKey, а не EncryptedData.

```
<EncryptedKey CarriedKeyName="My 3DES Session Key">
   <EncryptionMethod Algorithm="...xmlenc#rsa-1_5" />
   <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
               <KeyName>My Private Key</KeyName>
   </ds:KeyInfo>?
   <CipherData>
               <CipherValue>bYgmKUZIXzwt2te9dmONF7Mj...
               </CipherValue>
   </CipherData>
</EncryptedKey>
```

Как работает шифрование XML

Как и во многих схемах шифрования общего назначения, в XML Encryption используется сочетание симметричного и асимметричного алгоритмов. Симметричный алгоритм используется для массового шифрования

данных в элементах XML, а при помощи асимметричного алгоритма обеспечивается безопасный обмен симметричными ключами. Вот типичная схема обмена шифрованными XML-данными.

1. Получатель генерирует асимметричную пару ключей, сохраняя один ключ в секрете и делая другой общедоступным. Благодаря этому любой желающий может зашифровать данные так, что они будут доступны лишь владельцу секретного ключа, то есть получателю.
2. Отправитель получает доступ к открытому ключу получателя посредством прямого, незащищенного соединения, или через посредство центра сертификации, если необходимо убедиться в аутентичности открытого ключа.
3. Отправитель генерирует секретный симметричный ключ.
4. Отправитель шифрует этим ключом нужные элементы в своем XML-документе.
5. Отправитель шифрует свой секретный ключ при помощи открытого ключа получателя.
6. Отправитель внедряет в документ зашифрованные данные и зашифрованный симметричный ключ, при необходимости добавляя вспомогательные данные такие, как параметры алгоритма, и получая в результате новый, зашифрованный XML-документ. Все это делается в соответствии со стандартным синтаксисом, определенным в спецификации XML Encryption.
7. Отправитель пересылает получателю зашифрованный XML-документ.
8. Получатель извлекает из документа зашифрованные данные и зашифрованный симметричный ключ.
9. Получатель дешифрует симметричный ключ при помощи своего асимметричного секретного ключа, используя соответствующий алгоритм.
10. Получатель дешифрует данные при помощи дешифрованного секретного ключа.

Классы, используемые в XML Encryption

Хотя в .NET Framework предусмотрена высокоуровневая поддержка спецификации XML Signatures, но, как мы убедимся в дальнейшем, в настоящее время поддержка шифрования XML Encryption отсутствует. По этой причине мы самостоятельно реализуем необходимые функции при помощи доступных классов из пространств имен **System.Xml** и **System.Security.Cryptography**. Эти пространства имен обеспечивают общие функции синтаксического анализа XML и криптографии.

Мы не будем здесь всесторонне изучать эти два пространства имен. Мы уже познакомились с основными криптографическими классами в предыдущих главах, а широкие возможности синтаксического анализа

XML выходят за рамки тем этой книги. Все, что нам необходимо знать, это несколько методов из этих классов, которые используются в очередной учебной программе.

КЛАСС XMLDOCUMENT

Класс **XmlDocument** обеспечивает навигацию по XML-документу и его редактирование так, как это предусмотрено объектной моделью документа (DOM8) концерна W3C. Класс **XmlDocument** определен в пространстве имен **System.Xml** и поддерживает большое число методов и свойств. Класс **XmlDocument** является потомком класса **XmlNode**, от которого он наследует многие из своих частей. Тот факт, что документ в целом также является элементом, содержащим в себе другие субэлементы, имеет смысл, поскольку XML-документ строится по принципу наследственной иерархии.

В следующих абзацах описываются те методы, которые особенно полезны для понимания очередного примера программы, который ждет нас впереди.

Существует четыре перегруженных версии метода **Load**, каждая из которых загружает XML-документ из заданного источника. Один из этих методов **Load** получает на входе строковый параметр, представляющий собой URL файла с документом. В следующем примере программы таким параметром будет просто имя файла на диске.

```
public virtual void Load(
   string filename
);
```

Метод **LoadXml** аналогичен методу **Load** с той разницей, что он загружает XML-документ непосредственно из заданной в параметре строки. Здесь мы передаем строку символов, представляющую собой XML-данные.

```
public virtual void LoadXml(
   string xml
);
```

Существует также три перегруженных версии метода **CreateElement**. В каждом случае эти методы создают и добавляют в документ новый элемент XML с заданным именем. Мы воспользуемся двумя такими методами.

```
public XmlElement CreateElement(
   string name
);

public virtual XmlElement CreateElement(
   string prefix
   string localName
   string namespaceURI
)
```

Метод **AppendChild**, полученный наследованием из **XmlNode**, добавляет узел в конец списка потомков текущего узла. Мы используем его для добавления узлов в XML-документ.

```
public virtual XmlNode AppendChild(
   XmlNode NewChild
)
```

Метод **CreateAttribute** располагает тремя перегруженными версиями, которые используются для создания атрибутов с заданным именем. Мы используем лишь одну из них, принимающую единственный строковый параметр.

```
public XmlAttribute CreateAttribute(
   string name
)
```

Метод **SelectSingleNode**, унаследованный из **XmlNode**, располагает двумя перегруженными версиями. Мы используем вариант, принимающий единственный строковый параметр.

```
public XmlNode SelectSingleNode(
   string xpath
)
```

Метод **Save** сохраняет XML-документ в заданном месте. Доступны четыре перегруженных версии этого метода. Мы используем ту из них, которая принимает в качестве единственного строкового параметра имя файла, в котором должен быть сохранен документ.

```
public virtual void Save(
   string filename
);
```

КЛАСС XMLELEMENT

Класс **XmlElement**, определенный в пространстве имен **System.Xml**, представляет XML-элемент в составе документа. Класс **XmlElement** производится из класса **XmlLinkedNode**, который, в свою очередь, производится из класса **XmlNode**. Класс **XmlElement** поддерживает множество методов и свойств, однако мы воспользуемся только методом **AppendChild** и свойством **InnerText**.

Метод **AppendChild**, унаследованный из **XmlNode**, добавляет заданный узел в конец списка узлов-потомков текущего узла. Мы используем его для добавления узлов-потомков в XML-элемент. Этот метод аналогичен одноименному методу из класса **XmlDocument**, который описывался выше.

```
public virtual XmlNode AppendChild(
   XmlNode newChild
)
```

Свойство **InnerText** используется для получения или задания полной строки символов, представляющей весь элемент вместе с его потомками.

```
public override string InnerText
{get; set;}
```

КЛАСС XMLATTRIBUTE

Класс **XmlAttribute**, определенный в пространстве имен **System.Xml**, поддерживает большое число полезных свойств и методов. Мы используем только свойство **Value**, определяющее значение узла.

```
public override string Value
{get; set;}
```

КЛАСС RSACRYPTOSERVICEPROVIDER

Мы уже видели в главах 4 и 5, как используется класс **RSACryptoServiceProvider**. Вы, возможно, помните, что этот класс определен в пространстве имен **System.Security.Cryptography**. Мы воспользуемся знакомыми уже методами **Encrypt** и **Decrypt**. Нам также потребуются методы **FromXmlString** и **ToXmlString**. Метод **Decrypt** дешифрует данные при помощи алгоритма RSA. Метод **Encrypt**, наоборот, их шифрует.

```
public byte[] Decrypt(
   byte[] rgb,
   bool fOAEP
);
public byte[] Encrypt(
   byte[] rgb,
   bool fOAEP
);
```

Метод **ToXmlString** создает и возвращает XML-строку RSA-объекта. Параметр **includePrivateParameters** определяет, включается ли в информацию секретный ключ. Если этот параметр равен true, то в результирующий XML будет включен как открытый, так и секретный ключ.

```
public override string ToXmlString(
   bool includePrivateParameters
);
```

Метод **FromXmlString** создает RSA-объект из заданной строки XML-данных.

```
public override void FromXmlString(
   string xmlString
);
```

КЛАСС CONVERT

Имея дело с шифрованием, мы чаще всего оперируем данными, представленными в форме байтовых массивов. Однако при работе с XML-данными нам придется иметь дело с текстовыми строками. Чтобы преодолеть это противоречие, мы воспользуемся статическими методами **ToBase64String** и **FromBase64String**, принадлежащими классу **Convert**. Метод **ToBase64String** преобразует произвольный массив беззнаковых 8-битовых целых в эквивалентное строковое представление в базисе base64.

```
public static string ToBase64String(
   byte[] inArray
);
```

Метод **FromBase64String** преобразует строку символов в базисе base64 в соответствующий массив беззнаковых 8-битовых целых.

```
public static byte[] FromBase64String(
   string s
);
```

Передача асимметричных ключей

Вам не потребуется ничего делать для того, чтобы сгенерировать асимметричную пару ключей, поскольку классы асимметричных алгоритмов (как RSA, так и DSA) автоматически генерируют случайную пару ключей каждый раз, когда создается экземпляр класса. Поскольку отправитель и получатель создают свои экземпляры RSA-объекта независимо друг от друга, сгенерированные при этом пары ключей, очевидно, не будут совпадать. Проблема решается передачей информации об открытом ключе, которая в случае алгоритма RSA сводится к значениям модуля и показателя.

Такая задача решается при помощи пары методов **ToXmlString** и **FromXmlString** или **ExportParameters** и **ImportParameters**. Методы **ToXmlString** и **FromXmlString** позволяют сохранять и извлекать объект в XML-строке, а методы **ExportParameters** и **ImportParameters** позволяют делать то же самое, используя в качестве хранилища объект **RSAParameters**. Использование XML-строк привлекает своей простотой, однако, поскольку объект **RSAParameters** также может преобразовываться в строку, его можно использовать для передачи ключевой информации. Например, если получатель создает экземпляр объекта RSA, он может извлечь полученный открытый ключ и передать его отправителю, не шифруя. Отправитель получает открытый ключ и использует его для шифрования секретных данных, которые и отправляет получателю. Получатель, используя оставшийся у него секретный ключ, дешифрует данные.

В следующем примере программы мы используем методы **ToXmlString** и **FromXmlString** класса RSA. Но вначале, давайте рассмотрим сходные методы: **ExportParameters** и **ImportParameters**. Метод **ExportParameters** обладает тем же параметром, что и **ExportParameters**, а именно – параметром **includePrivateParameters**, который определяет, будет ли сохраняться секретный ключ. При значении true этого параметра сохраняется полная асимметричная пара ключей. При значении false сохраняется только открытый ключ. Этот параметр необходим, поскольку в информации, которую вы разделяете с другой стороной, должен присутствовать только открытый ключ. Секретный ключ должен сохраняться только в некоторых случаях, и при этом вы должны использовать безопасное хранилище, такое, например, как файл шифрованной файловой системы (EFS – Encrypted File System)[1]. Альтернатива заключается в хранении секретного ключа в хранилище ключей CryptoAPI (CAPI). Один из конструкторов **RSACryptoServiceProvider** принимает в качестве параметра объект **CspParameters**. Класс **CspParameters** включает в себя поле **KeyContainerName**, которое можно использовать для хранения пары ключей.

Далее приведен синтаксис обращения к методам **ExportParameters** и **ImportParameters** классов **RSA** и **DSA**. Вместо сохранения ключей в формате XML, здесь они сохраняются в объекте, специфичном для используемого алгоритма, **RSAParameters** для **RSA** и **DSAParameters** для **DSA**. В обоих случаях для метода **ExportParameters** предусмотрен параметр **includePrivateParameters**, играющий ту же роль, что и в **ToXmlString**.

```
public abstract RSAParameters ExportParameters(
   bool includePrivateParameters
);
public abstract void ImportParameters(
   RSAParameters parameters
);
public abstract DSAParameters ExportParameters(
   bool includePrivateParameters
);
public abstract void ImportParameters(
   DSAParameters parameters
);
```

Пример программы XmlEncryption

Пример программы XmlEncryption иллюстрирует шифрование и дешифрование XML-данных. Исходный код программы приведен ниже.

[1] Это может превратиться в «бесконечную историю». Симметричный сеансовый ключ зашифрован для безопасной передачи асимметричным ключом, при этом асимметричный ключ хранится в шифрованной файловой системе (например, EFS). При этом файл зашифрован симметричным ключом, который, в свою очередь, зашифрован асимметричным ключом, который ассоциируется с шифрованной версией вашего пароля. Никто и не говорит, что криптография – это просто...

Работа программы начинается с создания объектов «отправитель» (sender) и «получатель» (receiver). Моделируется типичный сценарий, где одна программа шифрует и передает данные, а вторая получает их и дешифрует. Чтобы вместить весь сценарий в одну программу, мы представляем общающиеся стороны двумя раздельными классами. Разумеется, в реальном мире это были бы две разные программы.

Затем «получатель» задает параметры RSA, которые будут использованы для шифрования сеансового симметричного ключа при помощи метода **EstablishXmlRSAParameters**. Именно так и происходит в реалистичных сценариях, поскольку именно получатель должен сгенерировать асимметричную пару ключей с тем, чтобы передать открытый ключ отправителю.

Затем вызовом метода **CreateOriginalXmlDocument** отправитель создает новый XML-документ, подлежащий шифрованию.

```
//XMLEncryption.cs

//ПРИМЕЧАНИЕ: необходимо добавить ссылку на этот проект
//в System.Security

using System;
using System.IO;
using System.Text;
using System.Xml;
using System.Security.Cryptography;
using System.Security.Cryptography.Xml;

class XMLEncryption
{
   static void Main(string[] args)
   {
              //создать участников
              Sender sender = new Sender();
              Receiver receiver = new Receiver();

              //задать открытый и секретный ключи RSA
              receiver.EstablishXmlRsaParameters(
                     "RsaIncludePrivateParams.xml",
                     "RsaExcludePrivateParams.xml");

              //создать исходный XML-документ, подлежащий
              // шифрованию
              sender.CreateOriginalXmlDocument(
                     "OriginalInvoice.xml");

              //создать сеансовый ключ и зашифровать его открытым
              // ключом RSA
              byte [] IV = sender.CreateAndEncryptXmlSessionKey(
                     "RsaExcludePrivateParams.xml",
```

```
                "SessionKeyExchange.xml");

            //зашифровать исходный XML-документ сеансовым ключом
            sender.EncryptOriginalXmlDocument(
                "OriginalInvoice.xml",
                "RsaExcludePrivateParams.xml",
                "SessionKeyExchange.xml",
                "EncryptedInvoice.xml");

            //дешифровать исходный XML-документ сеансовым ключом
            receiver.DecryptXmlDocument(
                "EncryptedInvoice.xml",
                "RsaIncludePrivateParams.xml",
                "SessionKeyExchange.xml",
                "DecryptedCreditInfo.xml",
                IV);
    }
}

class Sender
{
    public void CreateOriginalXmlDocument(
            String originalFilename)
    {
            //создать исходный XML-документ
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.PreserveWhitespace = true;
            xmlDoc.LoadXml(
                "<invoice>\n" +
                "  <items>\n" +
                "    <item>\n" +
                "      <desc>Deluxe corncob pipe</desc>\n" +
                "      <unitprice>14.95</unitprice>\n" +
                "      <quantity>1</quantity>\n" +
                "    </item>\n" +
                "  </items>\n" +
                "  <creditinfo>\n" +
                "     <cardnumber>0123456789</cardnumber>\n" +
                "     <expiration>01/06/2005</expiration>\n" +
                "     <lastname>Finn</lastname>\n" +
                "     <firstname>Huckleberry</firstname>\n" +
                "  </creditinfo>\n" +
                "</invoice>\n");

            //записать исходный XML-документ в файл
            StreamWriter file =
                new StreamWriter(originalFilename);
            file.Write(xmlDoc.OuterXml);
            file.Close();
```

```
            //сообщить пользователю
            Console.WriteLine(
                  "Original XML document written to:\n\t" +
                  originalFilename);
}

public byte [] CreateAndEncryptXmlSessionKey(
            String rsaExcludePrivateParamsFilename,
            String keyFilename)
{
            //получить сеансовый ключ для шифрования 3DES
            TripleDESCryptoServiceProvider tripleDES =
                  new TripleDESCryptoServiceProvider();

            //сохранить вектор инициализации и ключ
            IV = tripleDES.IV;
            Key = tripleDES.Key;

            //получить открытые параметры RSA из XML
            StreamReader fileRsaParams = new StreamReader(
                  rsaExcludePrivateParamsFilename);
            String rsaExcludePrivateParamsXML =
                  fileRsaParams.ReadToEnd();
            fileRsaParams.Close();

            //зашифровать сеансовый ключ ключом RSA
            RSACryptoServiceProvider rsa =
                  new RSACryptoServiceProvider();
            rsa.FromXmlString(rsaExcludePrivateParamsXML);
            byte[] keyEncryptedBytes =
                  rsa.Encrypt(tripleDES.Key, false);

            //сохранить зашифрованный сеансовый ключ 3DES
            //в строке Base64
            String keyEncryptedString = Convert.ToBase64String(
                  keyEncryptedBytes);

            //создать XML-документ для передачи сеансового
            //ключа 3DES
            XmlDocument xmlKeyDoc = new XmlDocument();
            xmlKeyDoc.PreserveWhitespace = true;

            //добавить в XML элемент EncryptedKey
            XmlElement xmlEncryptedKey =
                  xmlKeyDoc.CreateElement("EncryptedKey");
            xmlKeyDoc.AppendChild(xmlEncryptedKey);
            XmlAttribute xmlCarriedKeyName =
                  xmlKeyDoc.CreateAttribute("CarriedKeyName");
            xmlCarriedKeyName.Value = "My 3DES Session Key";
```

```
            xmlEncryptedKey.Attributes.Append(
                  xmlCarriedKeyName);

            //добавить в XML элемент EncryptionMethod
            XmlElement xmlEncryptionMethod =
                  xmlKeyDoc.CreateElement("EncryptionMethod");
            xmlEncryptedKey.AppendChild(xmlEncryptionMethod);
            XmlAttribute xmlAlgorithm =
                  xmlKeyDoc.CreateAttribute("Algorithm");
            xmlAlgorithm.Value =
                  "http://www.w3.org/2001/04/xmlenc#rsa-1_5";
            xmlEncryptionMethod.Attributes.Append(
                  xmlAlgorithm);

            //добавить в XML элемент KeyInfo
            XmlElement xmlKeyInfo =
                  xmlKeyDoc.CreateElement(
                  "ds",
                  "KeyInfo",
                  "http://www.w3.org/2000/09/xmldsig#");
            xmlEncryptedKey.AppendChild(xmlKeyInfo);

            //добавить в XML элемент KeyName
            XmlElement xmlKeyName =
                  xmlKeyDoc.CreateElement("ds", "KeyName", null);
            xmlKeyName.InnerText = "My Private Key";
            xmlKeyInfo.AppendChild(xmlKeyName);

            //добавить в XML элемент CipherData
            XmlElement xmlCipherData =
                  xmlKeyDoc.CreateElement("CipherData");
            xmlEncryptedKey.AppendChild(xmlCipherData);

            //добавить в XML элемент CipherValue
            XmlElement xmlCipherValue =
                  xmlKeyDoc.CreateElement("CipherValue");
            xmlCipherValue.InnerText = keyEncryptedString;
            xmlCipherData.AppendChild(xmlCipherValue);

            //сохранить XML-данные
            xmlKeyDoc.Save(keyFilename);

            //сообщить пользователю
            Console.WriteLine(
                  "Encrypted Session Key XML written to:\n\t" +
                  keyFilename);

            return IV; //требуется также получателю
      }
```

```
public void EncryptOriginalXmlDocument(
            String originalFilename,
            String rsaExcludePrivateParamsFilename,
            String keyFilename,
            String encryptedFilename)
{
            //загрузить XML-документ, подлежащий шифрованию
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.PreserveWhitespace = true;
            xmlDoc.Load(originalFilename);

            //получить текст узла creditinfo
            XmlElement xmlCreditinfo =
                   (XmlElement)xmlDoc.SelectSingleNode(
                   "invoice/creditinfo");
            byte[] creditinfoPlainbytes =
                   Encoding.UTF8.GetBytes(xmlCreditinfo.OuterXml);

            //загрузить XML-документ с ключом
            XmlDocument xmlKeyDoc = new XmlDocument();
            xmlKeyDoc.PreserveWhitespace = true;
            xmlKeyDoc.Load(keyFilename);

            //получить зашифрованный сеансовый ключ
            XmlElement xmlKeyCipherValue =
                   (XmlElement)xmlKeyDoc.SelectSingleNode(
                   "EncryptedKey/CipherData/CipherValue");
            byte[] xmlKeyCipherbytes =
                   Convert.FromBase64String(
                      xmlKeyCipherValue.InnerText);


            //создать объект 3DES
            TripleDESCryptoServiceProvider tripleDES =
                   new TripleDESCryptoServiceProvider();

            //задать поток для алгоритма 3DES
            MemoryStream ms = new MemoryStream();
            CryptoStream cs = new CryptoStream(
                   ms,
                   tripleDES.CreateEncryptor(Key, IV),
                   CryptoStreamMode.Write);

            //записать открытый текст в крипто-поток
            cs.Write(
                   creditinfoPlainbytes,
                   0,
                   creditinfoPlainbytes.Length);
            cs.Close();
```

```
//получить шифрованный текст из криптосистемы
byte[] creditinfoCipherbytes = ms.ToArray();
ms.Close();
String creditinfoCiphertext =
       Convert.ToBase64String(
              creditinfoCipherbytes);

//создать элемент EncryptedData в XML-файле
XmlElement xmlEncryptedData =
       xmlDoc.CreateElement("EncryptedData");
XmlAttribute xmlType =
       xmlDoc.CreateAttribute("Type");
xmlType.Value =
       "http://www.w3.org/2001/04/xmlenc#Element";
xmlEncryptedData.Attributes.Append(xmlType);

//добавить элемент KeyInfo
XmlElement xmlKeyInfo =
       xmlDoc.CreateElement(
       "ds",
       "KeyInfo",
       "http://www.w3.org/2000/09/xmldsig#");
xmlEncryptedData.AppendChild(xmlKeyInfo);

//добавить элемент KeyName
XmlElement xmlKeyName =
       xmlDoc.CreateElement("ds", "KeyName",null);
xmlKeyName.InnerText = "My 3DES Session Key";
xmlKeyInfo.AppendChild(xmlKeyName);


//добавить элемент CipherData
XmlElement xmlCipherData =
       xmlDoc.CreateElement("CipherData");
xmlEncryptedData.AppendChild(xmlCipherData);

//добавить элемент CipherValue с шифрованными данными
XmlElement xmlCipherValue =
       xmlDoc.CreateElement("CipherValue");
xmlCipherValue.InnerText = creditinfoCiphertext;
xmlCipherData.AppendChild(xmlCipherValue);

//заменить исходный узел зашифрованным узлом
xmlCreditinfo.ParentNode.ReplaceChild(
       xmlEncryptedData, xmlCreditinfo);

//сохранить XML в файл
xmlDoc.Save(encryptedFilename);
```

```
            //сообщить пользователю
            Console.WriteLine(
                  "Encrypted XML document written to:\n\t" +
                  encryptedFilename);
   }

   //отправитель нуждается в перекрестных вызовах метода
   static byte [] IV;
   static byte [] Key;
}

class Receiver
{
   public void EstablishXmlRsaParameters(
            String rsaIncludePrivateParamsFilename,
            String rsaExcludePrivateParamsFilename)
   {
            //создать объект RSA с новой парой ключей
            RSACryptoServiceProvider rsa =
                  new RSACryptoServiceProvider();

            //сохранить секретные и открытые параметры RSA в XML
            StreamWriter fileRsaIncludePrivateParams
                  = new StreamWriter(
                  rsaIncludePrivateParamsFilename);
            fileRsaIncludePrivateParams.Write(
                  rsa.ToXmlString(true));
            fileRsaIncludePrivateParams.Close();

            //сохранить только открытые параметры RSA в XML
            StreamWriter fileRsaExcludePrivateParams =
                  new StreamWriter(
                  rsaExcludePrivateParamsFilename);
            fileRsaExcludePrivateParams.Write(
                  rsa.ToXmlString(false));
            fileRsaExcludePrivateParams.Close();

            //сообщить пользователю
            Console.WriteLine(
                  "RSA parameters written to:\n\t" +
                  rsaIncludePrivateParamsFilename + "\n\t" +
                  rsaExcludePrivateParamsFilename);
   }
   public void DecryptXmlDocument(
            String encryptedFilename,
            String rsaIncludePrivateParamsFilename,
            String keyFilename,
            String decryptedFilename,
            byte [] IV)
```

```
{
        //загрузить шифрованный XML-документ
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.PreserveWhitespace = true;
        xmlDoc.Load(encryptedFilename);

        //получить зашифрованный узел creditinfo
        XmlElement xmlEncryptedData =
               (XmlElement)xmlDoc.SelectSingleNode(
               "invoice/EncryptedData");
        XmlElement xmlCipherValue =
               (XmlElement)xmlEncryptedData.SelectSingleNode(
               "CipherData/CipherValue");
        byte[] creditinfoCipherbytes =
               Convert.FromBase64String(
                  xmlCipherValue.InnerText);

        //загрузить XML-документ с ключом
        XmlDocument xmlKeyDoc = new XmlDocument();
        xmlKeyDoc.PreserveWhitespace = true;
        xmlKeyDoc.Load(keyFilename);

        //получить байты зашифрованного сеансового ключа
        XmlElement xmlKeyCipherValue =
               (XmlElement)xmlKeyDoc.SelectSingleNode(
               "EncryptedKey/CipherData/CipherValue");
        byte[] xmlKeyCipherbytes =
               Convert.FromBase64String(
                  xmlKeyCipherValue.InnerText);

        //получить открытый ключ RSA из XML
        StreamReader fileRsaParams = new StreamReader(
               rsaIncludePrivateParamsFilename);
        String rsaIncludePrivateParamsXML =
               fileRsaParams.ReadToEnd();
        fileRsaParams.Close();

        //дешифровать при помощи RSA сеансовый ключ 3DES
        RSACryptoServiceProvider rsa =
               new RSACryptoServiceProvider();
        rsa.FromXmlString(rsaIncludePrivateParamsXML);

        byte[] keyPlainBytes =
               rsa.Decrypt(xmlKeyCipherbytes, false);

        //создать объект 3DES
        TripleDESCryptoServiceProvider tripleDES =
               new TripleDESCryptoServiceProvider();

        //задать поток для объекта 3DES
        MemoryStream ms = new MemoryStream(
               creditinfoCipherbytes);
```

```
            CryptoStream cs = new CryptoStream(
                  ms,
                  tripleDES.CreateDecryptor(keyPlainBytes, IV),
                  CryptoStreamMode.Read);

            //прочитать расшифрованный текст creditinfo
            //из криптосистемы
            byte[] creditinfoPlainbytes =
                  new Byte[creditinfoCipherbytes.Length];
            cs.Read(
                  creditinfoPlainbytes,
                  0,
                  creditinfoPlainbytes.Length);
            cs.Close();
            ms.Close();

            String creditinfoPlaintext =
                  Encoding.UTF8.GetString(creditinfoPlainbytes);

            //записать дешифрованный XML в файл
            StreamWriter fileplaintext =
                  new StreamWriter(decryptedFilename);
            fileplaintext.Write(creditinfoPlaintext);
            fileplaintext.Close();

            //сообщить пользователю
            Console.WriteLine(
                  "Decrypted XML credit info written to:\n\t" +
                  decryptedFilename);
   }
}
```

Когда вы запустите программу **XmlEncryption**, то увидите на консольном выводе следующий текст. Из него ясно, какие файлы XML были созданы:

```
RSA parameters written to:
   RsaIncludePrivateParams.xml
   RsaExcludePrivateParams.xml
Original XML document written to:
   OriginalInvoice.xml
Encrypted Session Key XML written to:
   SessionKeyExchange.xml
Encrypted XML document written to:
   EncryptedInvoice.xml
Decrypted XML credit info written to:
   DecriptedCreditInfo.xml
```

Давайте посмотрим теперь на содержимое этих файлов. Вот листинг файла **OriginalInvoice.xml**. Как видите, он достаточно прост. Он представляет собой счет, состоящий из списка купленных товаров и информации о кредитной карточке. В этом примере мы полагаем, что список товаров не представляет собой секрета, но информацию о кредитной карточке необходимо защитить.

```
<invoice>
   <items>
      <item>
         <desc>Deluxe corncob pipe</desc>
         <unitprice>14.95</unitprice>
         <quantity>1</quantity>
      </item>
   </items>
   <creditinfo>
      <cardnumber>0123456789</cardnumber>
      <expiration>01/06/2005</expiration>
      <lastname>Finn</lastname>
      <firstname>Huckleberry</firstname>
   </creditinfo>
</invoice>
```

А вот как будет выглядеть файл **RsaExcludePrivateParams.xml** с учетом небольшого форматирования, имеющего целью показать его на книжной странице. Как видите, здесь доступны только открытые параметры RSA модуль и показатель.

```
<RSAKeyValue>
   <Modulus>1x6LG6hv3cf870n+3etxJAEZi...</Modulus>
   <Exponent>AQAB</Exponent>
</RSAKeyValue>
```

Далее следует файл **RsaIncludePrivateParams.xml**, также с учетом небольшого форматирования, в котором добавлена информация, относящаяся к секретному ключу, включая очень чувствительные параметры P, Q и D, понятно, что этот файл должен храниться в секрете.

```
<RSAKeyValue>
   <Modulus>1x6LG6hv3cf870n+3etxJAEZi...</Modulus>
   <Exponent>AQAB</Exponent>
   <P>7vMj9Ji9J4CR+obULD8qlsFgJwHiHiVLVJK4LKO9zA5KvFTV...</P>
   <Q>5ngYdhg+OfxvgJwHiH5KvFD8qlswHiH5wHH5PY19AROD8UuA...</Q>
   <DP>5mGwkfzIu4CRNEYCDLGfxvgJwwdhg+EwHiAIN3y0G96...</DP>
   <DQ>pmVtG86ThJ6YoGKMKXCBnKvrADQwBU6qYX7G1jCJf79...</DQ>
   <InverseQ>1KGPyuuUOa3A7LT5OO0csg4zwSZS4sb...</InverseQ>
   <D>kRNyMVKG2AVVmBweyL6TGYqxOa3A7LLGfw5vJPOdYAdo...</D>
</RSAKeyValue>
```

Далее приведено содержимое файла **SessionKeyExchange.xml**, отформатированное для книжной страницы. Здесь вы видите, что выбран метод шифрования, основанный на RSA, также приведено имя ключа. Наиболее важная информация, а именно зашифрованный сеансовый ключ, содержится в элементе CupherValue.

```
<EncryptedKey CarriedKeyName="My Session Key">
   <EncryptionMethod Algorithm="http://.../xmlenc#rsa-1_5">
   <ds:KeyInfo xmlns:ds="http://w3.org/2000/09/xmldsig#">
              <KeyName>My Private Key</KeyName>
   </ds:KeyInfo>
   <CipherData>
              <CipheValue>UgthOM4Cu6vcRSGIiI9rzg/Hk...
              </CipheValue>
   </CipherData>
<EncryptedKey>
```

А вот содержимое файла **EncryptedInvoice.xml**, отформатированное для книжной страницы. Мы видим, что оригинальный элемент creditinfo здесь заменен элементом EncryptedData. Собственно зашифрованный исходный элемент при этом содержится в элементе CipherValue.

```
<invoice>
   <items>
              <item>
                     <desc>Deluxe corncob pipe</desc>
                     <unitprice>14.95</unitprice>
                     <quantity>1</quantity>
              </item>
   </items>
   <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element">
   <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
              <KeyName>My 3DES Session Key</KeyName>
   </ds:KeyInfo>
   <CipherData>
              <CipheValue>Irdws1X+xx3Ej2BYvDd3gFfuKw...
              </CipheValue>
   </CipherData>
   </EncryptedData>
</invoice>
```

И вот, наконец, конечный результат. Дешифрованные данные с информацией о кредитной карте помещены в файл с именем **DecryptedCreditInfo.xml**. Как видно из листинга, его содержимое совпадает с исходными данными.

```
<creditinfo>
   <cardnumber>0123456789</cardnumber>
   <expiration>01/06/2005</expiration>
```

```
   <lastname>Finn</lastname>
   <firstname>Huckleberry</firstname>
</creditinfo>
```

XML Signatures – подпись XML

Спецификация XML Signatures определяет новую мощную технологию, которая используется для наложения цифровой подписи на данные в формате XML. Стандарт XML приобрел такое больше значение, что используется практически везде, где реализуется обмен данными между приложениями. Во многих случаях обмен данными нуждается в специальных мерах, гарантирующих целостность, аутентификацию и подтверждение обязательств.

Вспомним, в предыдущих главах мы рассматривали шифрование с открытым ключом, при котором для дешифрования используется секретный ключ. В технологии цифровой подписи только хеш или дайджест сообщения шифруется секретным ключом, при этом дешифровать его можно при помощи открытого ключа. Затем можно вычислить заново хеш (дайджест) сообщения и сравнить его с дешифрованным хешем, убедившись, как в целостности, так и в авторстве сообщения.

Для того чтобы воспользоваться новой технологией, вы должны изучить два различных аспекта подписей XML. Первый из них заключается в синтаксисе и правилах наложения подписи, определенных спецификацией. Второй аспект – это программная техника работы с подписью на платформе .NET с использованием классов, определенных в **System.Security.Cryptography.Xmlnamespace**.

Спецификация XML Signature (ее иногда называют XMLDSIG) издана концерном W3C. Эта спецификация определяет синтаксис и семантику подписи XML в форме XML-схемы[1] и определения типа документа (DTD-определения), также в ней описаны сопутствующие правила.

Спецификация XML Signature

Все практические детали вы можете узнать из официальной спецификации концерна W3C, которая называется XML Signature Syntax and Processing, доступной по ссылке *www.w3.org/TR/xmldsig-core/*. Эта технология описана также в документе *www.ietf.org/rfc/rfc3275*. Статус этой спецификации уже достиг уровня Рекомендации, а это означает, что этот документ стабилен и может широко применяться на практике.

[1] XML Schema, XML-схема – стандартизованный язык, определенный в терминах XML, который используется для определения синтаксиса XML-документов. Язык XML Schema развивает и усовершенствует традиционный язык – язык DTD-определений. Более подробные сведения о языке XML Schema вы найдете по ссылке *http://www.w3.org/TR/xmlschema-1/*.

Что предусматривает спецификация XML Signature

Подпись XML, согласно спецификации, можно применить к любым данным, находящимся внутри или вне XML-документа. Подпись обеспечивает следующие функции:

- целостность – подтверждение того, что данные не были каким-либо образом изменены после наложения подписи;
- аутентификация – подтверждение авторства данных, данные действительно принадлежат тому, кем они подписаны;
- подтверждение обязательств – владелец наложенной на данные подписи не может опровергнуть свое авторство.

Синтаксис XML Signature

Для объектов данных вычисляется хеш, который затем шифруется и помещается в элемент документа вместе с сопутствующей информацией. Цифровые подписи в XML представляются при помощи элемента **Signature**, вид которого приведен ниже. Символ ? соответствует нулю или одному вхождению, + соответствует одному или более вхождений, * соответствует нулю или более вхождений.

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

ЭЛЕМЕНТЫ XML SIGNATURE

Элемент **Signature** является главным компонентом в реализации подписи XML. Все остальные элементы, участвующие в подписи, должны быть его потомками. В этом разделе мы сделаем краткий обзор всех этих элементов.

- Элемент **Signature** является самым внешним (то есть, корневым) элементом подписи XML, заключающим в себе все остальное.
- Элемент **SignedInfo** содержит элементы канонизации и алгоритма наложения подписи, а также один или несколько элементов, описывающих ссылки. Элемент SignedInfo также может включать в себя необязательный атрибут ID, на который будут ссылаться другие элементы подписи.

- Элемент **CanonicalizationMethod** определяет алгоритм канонизации, который применяется к данным перед наложением подписи. Канонизация приводит данные к стандартному (каноническому) формату, благодаря чему подпись не зависит от используемой платформы, и на нее не влияют такие, например, отличия, как способ кодирования конца строки.
- Элемент **SignatureMethod** определяет алгоритм, которым канонизированные данные элемента SignedInfo преобразуются в элемент SignatureValue. В этом элементе определяется хеш-алгоритм (например, SHA-1), алгоритм шифрования (например, RSA) и, возможно, алгоритм дополнения данных.
- Элемент **SignatureValue** содержит вычисленное значение цифровой подписи, представленное строкой в базисе base64.
- Элемент **KeyInfo** необязателен, в нем приводится информация о ключе, необходимом для проверки подписи. Это может быть имя открытого ключа или информация о сертификате.
- Элемент **Object** необязателен и может встретиться один или несколько раз. В нем могут быть заключены произвольные данные.

Для того чтобы сделать описанный синтаксис более конкретным, рассмотрим пример, в котором приведен документ до и после наложения подписи. Вот документ, который мы уже использовали ранее в этой главе.

```
<invoice>
   <items>
            <item>
                  <desc>Deluxe corncob pipe</desc>
                  <unitprice>14.95</unitprice>
                  <quantity>1</quantity>
            </item>
   </items>
   <creditinfo>
            <cardnumber>0123456789</cardnumber>
            <expiration>01/06/2005</expiration>
            <lastname>Finn</lastname>
            <firstname>Huckleberry</firstname>
   </creditinfo>
</invoice>
```

Теперь мы наложим на этот документ подпись. Весь исходный документ invoice здесь заключен в элемент Signature, и здесь содержится все необходимое для верификации подписи RSA.

```
<Signature xmlns="http://.../xmldsig#">
   <SignedInfo>
            <CanonicalizationMethod Algorithm="..." />
            <SignatureMethod Algorithm="http://...
                                          /xmldsig#rsa-sha1"/>
```

```
            <Reference URI="#MyDataObjectID">
                  <DigestMethod Algorithm="http://...
                                          /xmldsig#sha1" />
                  <DigestValue>fFyEpmWrhIwMjnrBZOOGmAtvvG8=
                  </DigestValue>
            </Reference>
   </SignedInfo>
   <SignatureValue>V2tW6tmZnOnuvKi8cZBXp...
   </SignatureValue>
            <KeyInfo>
             <KeyValue xmlns="http://www.w3.org/2000/09/xmldsig#">
              <RSAKeyValue>
               <Modulus>rJzbWtkPyhg+eBMhRdimd...</Modulus>
               <Exponent>AQAB</Exponent>
              </RSAKeyValue>
             </KeyValue>
            </KeyInfo>
   <Object Id="MyDataObjectID">
            <invoice>
                  <items>
                        <item>
                              <desc>Deluxe corncob pipe</desc>
                              <unitprice>14.95</unitprice>
                              <quantity>1</quantity>
                        </item>
                  </items>
                  <creditinfo>
                        <cardnumber>0123456789</cardnumber>
                        <expiration>01/06/2005</expiration>
                        <lastname>Finn</lastname>
                        <firstname>Huckleberry</firstname>
                  </creditinfo>
            </invoice>
   </Object>
</Signature>
```

ВНЕШНЯЯ, ВКЛЮЧАЮЩАЯ И ВКЛЮЧЕННАЯ ПОДПИСЬ

Существует три способа определения подписи внутри XML-документа.

- **Внешняя** подпись вычисляется по данным, находящимся вне элемента XML Signature. В этом случае объект данных вычисляется при помощи URI или элемента transform. Элемент Signature и объект данных, на который накладывается подпись, могут располагаться в разных XML-документах или в разных участках одного XML-документа.
- **Включающая** подпись вычисляется по данным, определенных в элементе-объекте данных, входящего в состав самого элемента

подписи. Элемент Object или его содержимое определяется при помощи элемента Reference.

- **Включенная** подпись вычисляется по всему XML-содержимому, в которое входит собственно элемент подписи Signature.

Классы, используемые в XML Signatures

Давайте рассмотрим вкратце те классы, которые мы будем использовать в примере программы, посвященной цифровой подписи XML. Эти классы определены в пространстве имен **System.Security.Cryptography.Xml**. Нам здесь предстоит гораздо меньше работы, чем в предыдущем примере, поскольку спецификация XML Signatures поддержана в .NET более полно.

КЛАСС DATAOBJECT

Класс **DataObject** инкапсулирует XML-элемент, содержащий в себе данные, которые необходимо подписать. Мы воспользуемся свойством **Data**, которое возвращает или задает данные, и свойством **ID**, которое идентифицирует объект **DataObject**.

```
public XmlNodeList Data
   {get; set;}
public string Id
   {get; set;}
```

КЛАСС SIGNEDXML

Класс **SignedXml** инкапсулирует основной объект подписи XML для подписанного документа. У этого класса есть несколько свойств и методов, которые нам потребуются в следующем примере. Свойство **SigningKey** определяет асимметричный ключ, использованный в подписи.

```
public AsymmetricAlgorithm SigningKey
   {get; set;}
```

Метод **AddObject** добавляет новый объект **DataObject** в список объектов, которые необходимо подписать.

```
public void AddObject(
   DataObject dataObject
);
```

Метод **AddReference** добавляет ссылку – элемент Reference в список ссылок, которые необходимо хешировать и подписать.

```
public void AddReference(
   Reference reference
);
```

Свойство **KeyInfo** содержит элемент KeyInfo для объекта SignedXml.

```
public KeyInfo KeyInfo
   {get; set;}
```

Метод **ComputeSignature** собственно накладывает подпись на XML-элемент. Существует две перегруженные версии этого метода. Мы используем вариант без параметров.

```
public void ComputeSignature();

public void ComputeSignature(
   KeyedHashAlgorithm macAlg
);
```

Программа EnvelopingXmlSignature

Пример программы EnvelopingXmlSignature иллюстрирует наложение XML-подписи. Полный исходный код этой программы приведен далее.

Первое, что делает программа, – вызовом метода **CreateXMLDocument** создает XML-файл с простым документом по имени **OriginalInvoice.xml**. Затем программа вызывает метод **PerformXMLSignature**, который вычисляет подпись для документа и записывает подписанный документ в файл **SignedInvoice.xml**. Затем наступает черед метода **VerifyXMLSignature**, который проверяет действительность подписи на документе **SignedInvoice.xml**, и выводит результат успешной проверки на консоль. Затем, для того чтобы продемонстрировать действие подписи, метод **TamperSignedXMLDocument** вносит изменение в подписанный файл **SignedInvoice.xml**, меняя номер кредитной карты в счете, и записывает «подделанный» документ в файл **TamperedInvoice.xml**. В этом файле содержится тот же самый исходный документ (изменен только номер кредитной карты), а также подпись. Наконец, метод **VerifyXMLSignature** вызывается еще раз, но на этот раз он выводит на консоль сообщение о несоответствии подписи. Изучите следующий исходный код, чтобы понять суть выполняемых операций, и посмотрите также на листинги вывода на консоль и результирующих XML-файлов.

```
//EnvelopingXMLSignature.cs

//ПРИМЕЧАНИЕ: необходимо добавить ссылку на этот проект
//в System.Security

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

```
class EnvelopingXMLSignature
{
   static void Main(string[] args)
   {
            //создать участников
            Sender sender = new Sender();
            Receiver receiver = new Receiver();
            Tamperer tamperer = new Tamperer();

            //показать эффект подписи и ее подделки
            sender.CreateXmlDocument("OriginalInvoice.xml");
            sender.PerformXmlSignature(
                  "OriginalInvoice.xml", "SignedInvoice.xml");
            receiver.VerifyXmlSignature("SignedInvoice.xml");
            tamperer.TamperSignedXmlDocument(
                  "SignedInvoice.xml", "TamperedInvoice.xml");
            receiver.VerifyXmlSignature("TamperedInvoice.xml");
   }
}

class Sender
{
   public void CreateXmlDocument(String originalFilename)
   {
            //создать исходный XML-документ
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.PreserveWhitespace = true;
            xmlDoc.LoadXml(
                  "<invoice>\n" +
                  "  <items>\n" +
                  "    <item>\n" +
                  "      <desc>Deluxe corncob pipe</desc>\n" +
                  "      <unitprice>14.95</unitprice>\n" +
                  "      <quantity>1</quantity>\n" +
                  "    </item>\n" +
                  "  </items>\n" +
                  "  <creditinfo>\n" +
                  "    <cardnumber>0123456789</cardnumber>\n" +
                  "    <expiration>01/06/2005</expiration>\n" +
                  "    <lastname>Finn</lastname>\n" +
                  "    <firstname>Huckleberry</firstname>\n" +
                  "  </creditinfo>\n" +
                  "</invoice>\n");

            //записать исходный документ в файл
            StreamWriter file =
                  new StreamWriter(originalFilename);
            file.Write(xmlDoc.OuterXml);
            file.Close();
```

```
            //выдать сообщение пользователю
            Console.WriteLine(
                  "Original XML document written to\n\t" +
                  originalFilename);
}
public void PerformXmlSignature(
            String originalFilename, String signedFilename)
{
            //загрузить XML-документ
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.PreserveWhitespace = true;
            xmlDoc.Load(originalFilename);

            //создать оболочку подписи с RSA-ключом по умолчанию
            RSA key = RSA.Create();
            SignedXml signedXml = new SignedXml();
            signedXml.SigningKey = key;

            //создать объект данных для подписи
            DataObject dataObject = new DataObject();
            dataObject.Data = xmlDoc.ChildNodes;

            //создать идентификатор объекта
            dataObject.Id = "MyDataObjectID";

            //добавить объект в список объектов
            //для наложения подписи
            signedXml.AddObject(dataObject);

            //создать ссылку на объект
            Reference reference = new Reference();
            reference.Uri = "#MyDataObjectID";

            //добавить ссылку на объект
            signedXml.AddReference(reference);

            //добавить ключевую информацию
            KeyInfo keyInfo = new KeyInfo();
            keyInfo.AddClause(new RSAKeyValue(key));
            signedXml.KeyInfo = keyInfo;

            //сгенерировать подпись XML
            signedXml.ComputeSignature();

            //наложить подпись на документ
            XmlElement xmlSignature = signedXml.GetXml();
            xmlDoc = new XmlDocument();
            xmlDoc.PreserveWhitespace = true;
            XmlNode xmlNode = xmlDoc.ImportNode(xmlSignature,
```

```
                                                    true);
            xmlDoc.AppendChild(xmlNode);
            xmlDoc.Save(signedFilename);

            //выдать сообщение пользователю
            Console.WriteLine(
                  "Signed XML document written to\n\t" +
                  signedFilename);
   }
}

class Receiver
{
   public void VerifyXmlSignature(String signedFilename)
   {
            //загрузить подписанный документ
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.PreserveWhitespace = true;
            xmlDoc.Load(signedFilename);

            //создать оболочку подписи
            SignedXml signedXml = new SignedXml(xmlDoc);

            //получить узел <Signature>, если он существует
            XmlNodeList nodeList = xmlDoc.GetElementsByTagName(
                  "Signature",
                  "http://www.w3.org/2000/09/xmldsig#");
            signedXml.LoadXml((XmlElement)nodeList[0]);

            //выдать сообщение пользователю
            if (signedXml.CheckSignature())
                  Console.WriteLine(
                    signedFilename + " signature is VALID");
            else
                  Console.WriteLine(
                    signedFilename + " signature is NOT VALID");
   }
}
class Tamperer
{
   public void TamperSignedXmlDocument(
            String signedFilename, String tamperedFilename)
   {
            //загрузить подписанный документ
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.PreserveWhitespace = true;
            xmlDoc.Load(signedFilename);

            //подделать подписанный документ
```

```
            //и записать его в файл
            XmlNodeList nodeList =
                  xmlDoc.GetElementsByTagName("cardnumber");
            XmlNode xmlOldNode = (XmlElement)nodeList[0];
            XmlNode xmlNewNode = xmlOldNode.Clone();

            xmlNewNode.InnerText = "9876543210";
            XmlOldNode.ParentNode.ReplaceChild(
                  xmlNewNode, xmlOldNode);
            xmlDoc.Save(tamperedFilename);

            //выдать сообщение пользователю
            Console.WriteLine(
              "Tampered signed XML document written to\n\t" +
              tamperedFilename);
   }
}
```

Если вы запустите эту программу, то увидите вывод на консоль, листинг которого приведен ниже. Обратите внимание, файл **SignedInvoice.xml** содержит документ с действительной подписью, а в подделанном файле **TamperSignedXMLDocument** подпись более не соответствует документу. Эти два файла почти идентичны, не совпадает лишь номер кредитной карты, причем номер был изменен уже после наложения подписи. Поскольку секретный ключ, которым наложена подпись, известен только его владельцу, злоумышленнику невозможно модифицировать документ так, чтобы это не было обнаружено.

Конечно, в этом простом примере данные подделывает та же самая программа, что и подписывает. Вряд ли кому-то на самом деле придет в голову подделывать свои собственные данные. Тем не менее, в этом примере полностью реализована концепция подписи XML-документа, и из него видно, что цифровая подпись действительно защищает электронный документ от подделки.

```
Original XML document written to OriginalInvoice.xml
Signed XML document written to SignedInvoice.xml
SignedInvoice.xml signature is VALID
Tampered signed XML document written to TamperedInvoice.xml
TamperedInvoice.xml signature is VALID
Press any key to continue
```

Рассмотрим теперь три XML-файла, которые сгенерированы программой: **OriginalInvoice.xml**, **SignedInvoice.xml** и **TamperSignedXMLDocument**. Первым создан **OriginalInvoice.xml**, причем, обратите внимание, номер кредитной карты здесь равен 0123456789. Посмотрим, что с ним произойдет дальше.

```
<invoice>
   <items>
            <item>
                  <desc>Deluxe corncob pipe</desc>
                  <unitprice>14.95</unitprice>
                  <quantity>1</quantity>
            </item>
   </items>
   <creditinfo>
            <cardnumber>0123456789</cardnumber>
            <expiration>01/06/2005</expiration>
            <lastname>Finn</lastname>
            <firstname>Huckleberry</firstname>
   </creditinfo>
</Invoice>
```

Теперь давайте рассмотрим файл **SignedInvoice.xml**. Некоторые части сокращены при помощи многоточий, кроме того, файл был немного отформатирован для отображения на книжной странице. Как видите подписанная версия файла содержит как подпись, так и информацию об открытом ключе.

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
   <SignedInfo>
            <CanonicalizationMethod Algorithm="..." />
            <SignatureMethod Algorithm="..." />
            <Reference URI="#MyDataObjectID">
                  <DigestMethod Algorithm="..." />
                  <DigestValue>...</DigestValue>
            </Reference>
   </SignedInfo>
   <SignatureValue>...</SignatureValue>
   <KeyInfo>
            <KeyValue xmlns="...">
              <RSAKeyValue>
                 <Modulus>...</Modulus>
                 <Exponent>...</Exponent>
              </RSAKeyValue>
            </KeyValue>
   </KeyInfo>
   <Object Id="MyDataObjectID">
            <invoice xmlns="">
                  <items>
                        <item>
                              <desc>Deluxe corncob pipe</desc>
                              <unitprice>14.95</unitprice>
                              <quantity>1</quantity>
                        </item>
```

```
                    </items>
                    <creditinfo>
                          <cardnumber>0123456789</cardnumber>
                          <expiration>01/06/2005</expiration>
                          <lastname>Finn</lastname>
                          <firstname>Huckleberry</firstname>
                    </creditinfo>
              </invoice>
   </Object>
</Signature>
```

Мы не будем приводить содержимое файла **TamperSignedXMLDocument**, поскольку он почти полностью совпадает с файлом **SignedInvoice.xml**. Единственное отличие заключается в номере карты, он изменен с 0123456789 на 9876543210, как видно из следующего фрагмента. Вы можете сравнить файлы и убедиться, что это единственное различие.

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
   <SignedInfo>
              ...
                    <creditinfo>
                          <cardnumber>9876543210</cardnumber>
                          <expiration>01/06/2005</expiration>
                          <lastname>Finn</lastname>
                          <firstname>Huckleberry</firstname>
                    </creditinfo>
              </invoice>
   </Object>
</Signature>
```

Сочетание XML Signing и XML Encryption

Обе криптографические операции, шифрование XML и подпись XML, можно сочетать друг с другом. Если вы шифруете и подписываете XML-документ, то большое значение имеет последовательность, в которой выполняются эти операции. Приложение должно различать ситуации, когда шифруется уже подписанный документ, и когда подпись накладывается на уже шифрованный документ. Если шифрование выполнялось раньше подписи, то нельзя дешифровать документ до проверки подписи, поскольку после дешифрования верификация подписи станет невозможной. И наоборот, если шифрование выполнялось после наложения подписи, то дешифрование должно выполняться до проверки подписи. Подробней об этом см. *http://www.w3.org/TR/2002/PR-xmlenc-decrypt-20021003*.

Итоги главы

В этой главе мы рассмотрели две важнейших криптографических операции, приложенных к данным в формате XML. В первой части главы мы сосредоточились на шифровании XML, вторая же половина была посвящена вопросам цифровой подписи XML.

Глава 7

Концепция безопасности, основанной на идентификации пользователей в .NET

В своей простейшей форме назначение систем безопасности состоит в том, чтобы не позволить людям и программам выполнять действия, которые не предусмотрел программист или администратор. В этой главе мы рассмотрим первый из двух главных аспектов безопасности в программировании на .NET – безопасность, основанная на идентификации пользователей[1]. В прошлом системы защиты фокусировали свое внимание на правах пользователя и разрешали или запрещали действие, основываясь на идентификации текущего пользователя. Таким образом обеспечивался контроль над доступом любых конкретных пользователей к любым конкретным ресурсам – файлам, значениям в системном реестре и так далее. Если вы уже сталкивались с механизмами защиты в Windows или UNIX, то все это должно быть вам знакомо.

Прежде чем перейти к обсуждению вопросов безопасности при программировании в .NET, мы рассмотрим общую картину, общую модель безопасности на платформе .NET. Затем мы подробно рассмотрим те классы .NET Framework, что имеют отношение к концепции безопасности, основанной на идентификации пользователей. После этого мы познакомимся с несколькими примерами программ, которые иллюстрируют применение этих классов. Наконец, мы изучим несколько простых, но фундаментальных правил, которым необходимо следовать, создавая программы, так или иначе связанные с безопасностью.

В главе 8 мы познакомимся с совершенно новой и совершенно иной концепцией безопасности, которую называют безопасностью, основанной на доступе к коду – Code Access Security (CAS)[2]. В противоположность традиционной концепции, CAS позволяет контролировать выполняемые действия, исходя из идентификации самого кода программы, а не идентификации пользователя, который этот код выполняет.

---

[1] Ее также называют безопасностью, основанной на механизме ролей.

[2] CAS также называют безопасностью, основанной на свидетельствах.

Аутентификация и авторизация

Прежде чем обсуждать что либо во этой главе или далее, мы должны полностью прояснить для себя понятия *авторизации* и *аутентификации*. Вся система безопасности должна в конечном счете обеспечить такое положение дел, когда только аутентифицированные объекты выполняют только авторизованные действия. Таким образом, все последующее должно вытекать из ответов на два разных вопроса:

- Аутентификация: Кто вы?
- Авторизация: Что вам позволено делать?

Вопрос *«Кто вы?»*[1] может относиться, как к личности пользователя, выполняющего некий код, так и к идентичности тех сборок[2] (assemblies), в которых содержится выполняемый код. Собственно говоря, именно в этом и состоит различие между традиционной концепцией безопасности и CAS. В традиционной модели главным вопросом является личность пользователя, представляемая классами **Principal** и **Identity**. В случае CAS вопрос аутентификации сводится к идентификации сборок с кодом, представляемой классом **Evidence**. Этот класс «свидетельств» можно использовать для определения различных характеристик сборки таких, как ее цифровая подпись и происхождение. Программист может определить при помощи этого класса также собственные «свидетельства».

Вопрос *«Что вам позволено делать?»* проясняет, какие действия позволены текущему пользователю или выполняющимся в данный момент сборкам. Этот вопрос возникает уже после того, как идентичность пользователя или сборки установлена. Решение при этом принимается на основе выбранной на данный момент политике безопасности[3]. Как мы вскоре убедимся, политика безопасности основывается на разрешениях, заданных программно или при помощи прямого администрирования.

Существуют разрешения нескольких типов, представляемых множеством классов, реализованных в интерфейсе **IPermission**. Для реализации концепции безопасности, основанной на идентификации пользователей, используется класс **PrincipalPermission**. Для реализации концепции CAS используется множество классов, производных от **CodeAccessPermission**. Эти классы служат для проверки разрешений всех сборок, имеющих отношение к вызову того метода, который выполняется в данный момент.

---

[1] Этот вопрос стал знаменитым благодаря Питу Тауншенду (Pete Townshend) и Роджеру Долтри (Roger Daltrey) в 1978 году.

[2] Речь идет о многих сборках, а не о единственной сборке, поскольку в любой момент времени конкретный выполняемый метод работает «по заказу» всех тех методов, чьи адреса возврата содержатся в стеке.

[3] Политика безопасности – это настраиваемый набор правил, который среда выполнения CLR использует для принятия решений, касающихся вопросов безопасности. Обычно политику безопасности устанавливает администратор. Политику безопасности можно определить на уровне предприятия, компьютера, пользователя или приложения.

Модель безопасности .NET

Модель безопасности .NET строится поверх модели безопасности операционной системы, а также она может взаимодействовать с функциями безопасности различных серверных приложений таких, как SQL Server или IIS. Следовательно, характеристики безопасности приложения .NET зависят от нескольких факторов, и в том числе от конфигурации собственно функций безопасности .NET, от программирования компонентов приложения и от настроек Windows[1], сетевого окружения[2] и других приложений.

Из рисунка 7.1 видно, что модель обеспечения безопасности .NET строится поверх безопасности Windows. Администратор настраивает учетные записи пользователей и управляет политиками безопасности Windows при помощи консолей MMC (Microsoft Management Console – консоль управления Microsoft). Администратор также отвечает за управление конфигурацией безопасности .NET. Таким образом, когда пользователь регистрируется в системе и запускает приложение .NET, среда выполнения CLR аутентифицирует пользователя и авторизует действия программы, а затем передает эти операции монитору безопасности операционной системы.

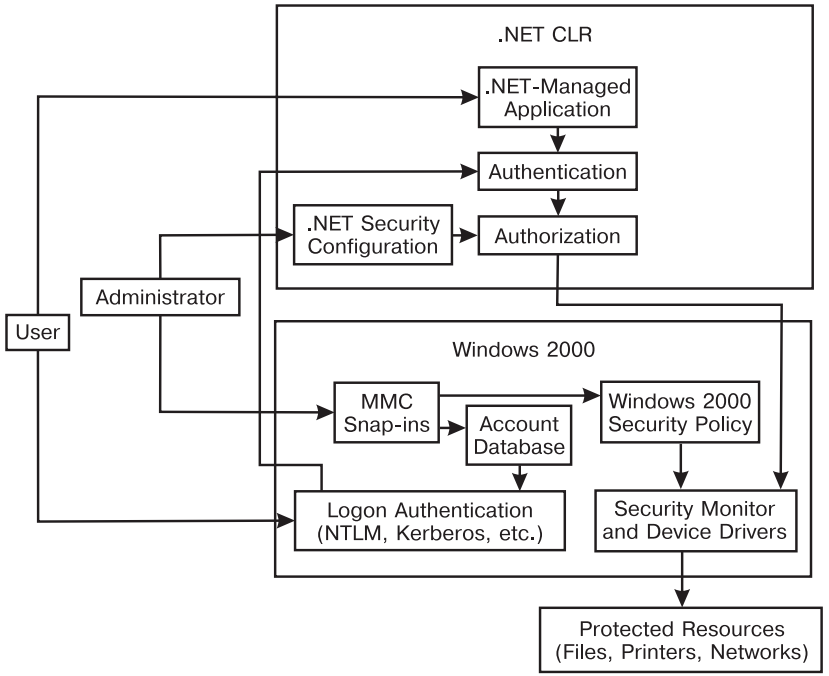


Рис. 7.1. Модель безопасности .NET

[1] Повсюду в этой главе под операционной системой Windows имеется в виду Windows 2000 или Windows XP.

[2] Например, вы можете запретить доступ к своей системе с определенных IP-адресов при помощи IPSec (Internet Protocol Security).

Для того чтобы понять, как среда .NET взаимодействует с операционной системой с точки зрения обеспечения безопасности, подумайте о том факте, что программный код выполняется всегда от имени некоторого пользователя. При регистрации в системе вы указываете свое пользовательское имя и пароль, доказывая, что являетесь тем, за кого себя выдаете. После того как ваша идентичность установлена, начинает действовать общее правило: любое действие, которое выполняет запущенная вами программа, совершается от вашего имени. Если администратор запретил вам выполнение некоторого действия (например, чтение или запись некоторого файла) при помощи списков контроля доступа ACL (Access Control List), то ваша программа не сможет выполнить это действие.

При рассмотрении примеров этой главы, описывающих разнообразные методы защиты объектов, важно помнить, что помимо контроля доступа средствами .NET, файл, например, может быть защищен на уровне операционной системы, и это выходит за пределы «юрисдикции» платформы .NET.

Администрирование безопасности на уровне Windows

Операционная система, под управлением которой работает платформа .NET, обладает своей собственной инфраструктурой безопасности. Системный администратор может настроить операционную систему тем или иным образом, и программист может использовать различные объекты этой инфраструктуры в своих целях. Администратор может сконфигурировать систему на уровне предприятия при помощи Active Directory, на уровне отдельной системы или даже на уровне доступа к отдельным файлам, используя средства файловой системы NTFS или EFS (Encrypted File System)[1]. На практике, зачастую, администратор также создает пользовательские учетные записи и задает для них пароли на локальной машине или в масштабах домена.

[1] Система NTFS поддерживает контроль доступа к каталогам и файлам в пределах зоны ответственности операционной системы, к которой она принадлежит. Однако при отсутствии мер физической безопасности (таких, как запертое помещение), NTFS сама по себе не обеспечивает должной защиты. Ведь злоумышленник может получить доступ к диску каким-нибудь нештатным способом, например, загрузив компьютер с дискеты. Даже если загрузка с дискеты защищена паролем, злоумышленник может извлечь жесткий диск и унести его с собой, выполнив затем его анализ на другом компьютере. Такая уязвимость NTFS объясняется тем обстоятельством, что существуют такие драйверы этой файловой системы (например, NTFSDOS), которые обеспечивают к ней доступ, однако игнорируют при этом ее систему прав доступа. Драйвер системы EFS решает эту проблему, поскольку он шифрует все данные при помощи комбинации алгоритмов RSA и DESF (усиленная версия DES). Таким образом, система EFS решает проблему безопасности, в том числе в ее физическом аспекте, что очень важно, например, для переносных компьютеров.

Определение пользователей и ролей в Windows

Вы можете без больших усилий определять новых пользователей и группы в Windows. Каждый пользователь или группа, определенные администратором, соотносится с пользователем или ролью с точки зрения программиста. Для того чтобы определить пользователя или группу с помощью значка Computer Management, выберите Start | Control Panel | Administrative Tools | Computer Management. Для того чтобы добавить пользователя, выберите узел Users в ветви Local users and groups, а затем выберите в меню команду Action | New user. Для того чтобы создать новую группу, выделите узел Groups и выберите команду Action | New group. Вид диалоговых окон New user и New group приведен на рисунках 7.2 и 7.3.

Определение прав доступа к общей папке

Администратор также задает права доступа определенных групп и пользователей к различным ресурсам операционной системы. Например, администратор может управлять доступом пользователей к общей (разделяемой) папке.

Предположим, вам требуется установить права доступа к общей папке, которые распространяются на все вложенные в нее папки и файлы. На рисунке 7.4 изображено, как сделать это для общей папки по имени **MyCodeExamples**. Чтобы выполнить эту операцию в Проводнике Windows, щелкните правой кнопкой мыши на значке нужной папки и выберите в контекстном меню команду Sharing. В открывшемся диалоговом окне выберите вкладку Sharing и выполните необходимые настройки. Щелкнув на кнопке Permissions, вы откроете одноименный диалог, где можно добавить и удалить пользователей и группы, определить разрешения для выполнения различных операций. Следует помнить, что действие этих ограничений распространяется только на доступ по сети. Если вы на этом же компьютере зарегистрируетесь под другим именем, ограничения прав доступа к этой папке вас не коснутся.

Средства безопасности в NTFS

Как уже упоминалось в предыдущем разделе, права доступа к разделяемой (общей) папке имеют силу только для доступа по сети с другого компьютера. Таким образом, эти ограничения доступа не обеспечивают никакой защиты в отношении пользователя, зарегистрировавшегося локально. Для обеспечения защиты на локальном уровне необходимо использовать механизм прав доступа файловой системы NTFS. При этом предполагается, конечно, что на вашем диске существует раздел или разделы NTFS. Есть несколько способов создать файловую систему NTFS. Вы можете выбрать форматирование раздела NTFS при установке Windows или воспользоваться утилитой **Convert.exe**, для того чтобы преобразовать в NTFS уже существующий раздел FAT32.

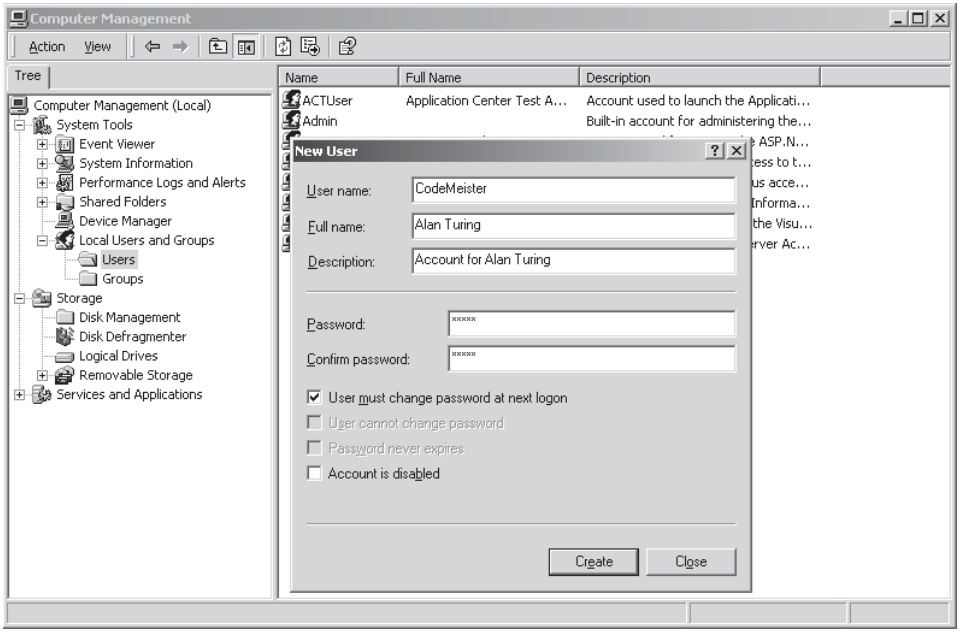


Рис. 7.2. Создание нового пользователя

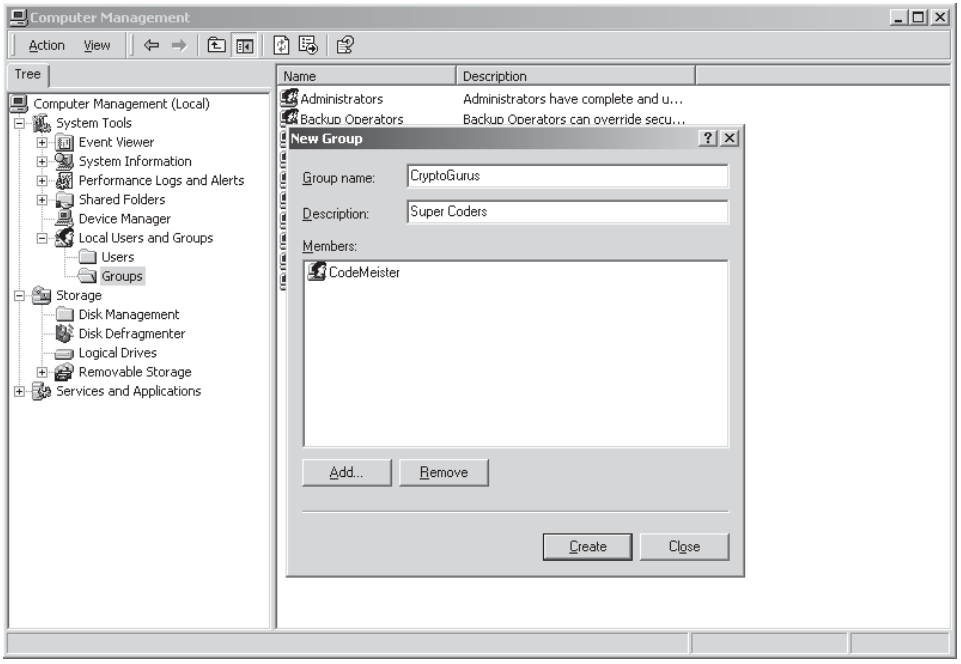


Рис. 7.3. Создание новой группы

Пожалуйста, прочтите внимательно документацию, чтобы выбрать наилучший для вашей ситуации вариант и отдавать себе отчет во всех возможных последствиях такой операции, особенно, если на вашем диске имеются важные данные.

ОПРЕДЕЛЕНИЕ ПРАВ ДОСТУПА К ПАПКЕ NTFS

Здесь описана процедура определения прав доступа к папке NTFS. Прежде всего, щелкните правой кнопкой на значке папки в Проводнике Windows и выберите в контекстном меню команду Properties. В диалоге свойств папки выберите вкладку Security. Если на вашем диске используется файловая система, отличная от NTFS, то вкладка Security в диалоге будет отсутствовать. На рисунке 7.5 приведены настройки для папки с именем **EncryptedFiles**. Это имя здесь выбрано для примера, оно может быть произвольным, и конечно папка вовсе не обязательно должна быть зашифрована, чтобы разграничивать права доступа к ней. Обратите внимание, вкладка Security в диалоге свойств папки позволяет задать различные права для разных пользователей и групп, присутствующих в списке имен. Кнопка Add позволяет добавлять пользователей и группы в список имен, а кнопка Delete удаляет их из списка. Список разрешений включает в себя такие элементы, как Read, Write, Modify и т. д. Кнопка Advanced обеспечивает доступ к более сложным настройкам прав доступа.

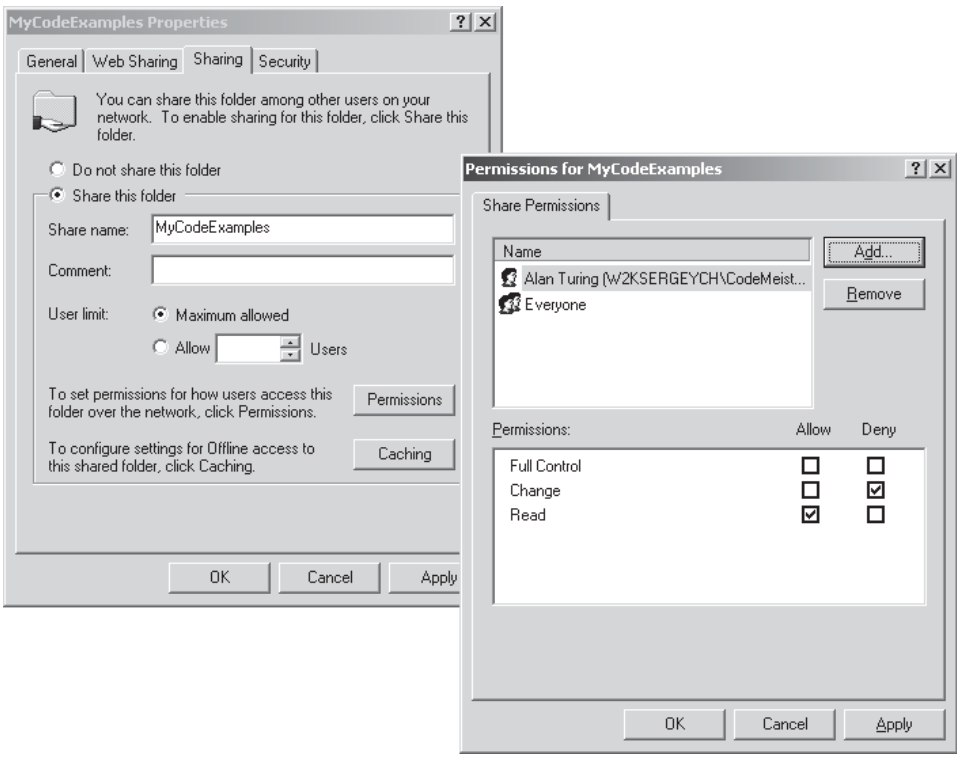


Рис. 7.4. Управление разрешениями для общей папки

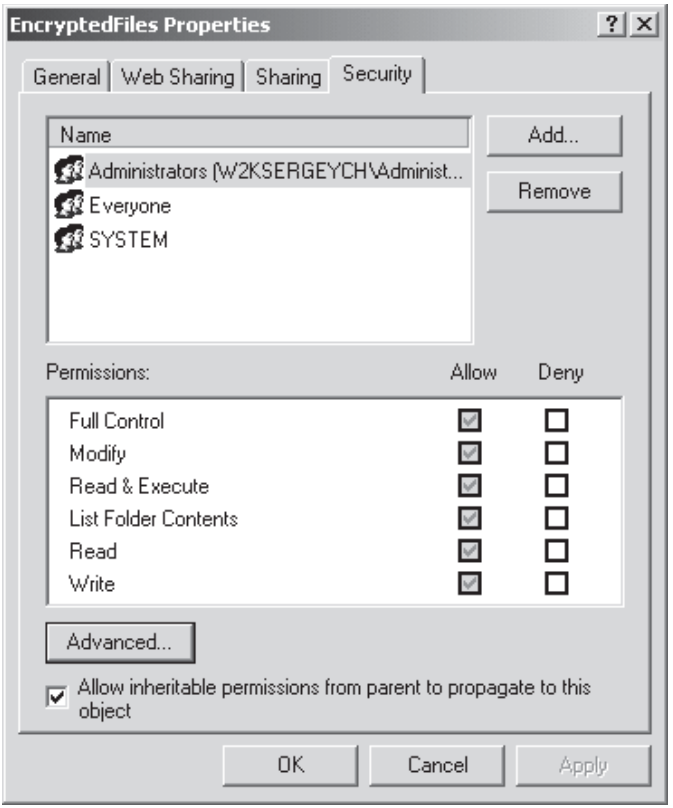


Рис. 7.5. Управление настройками безопасности файловой системы

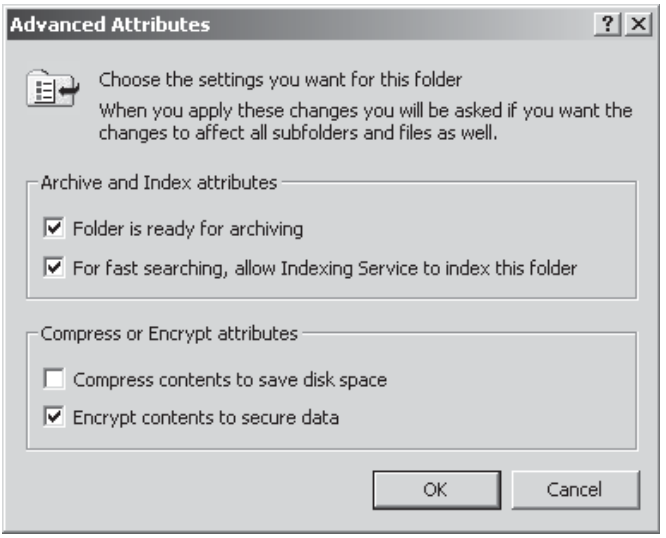


Рис. 7.6. Шифрование папки NTFS

ШИФРОВАНИЕ ПАПКИ NTFS

При использовании файловой системы NTFS шифрование папки необязательно, но вы можете это сделать при помощи кнопки Дополнительно на вкладке Общие в диалоге свойств папки. В результате откроется диалоговое окно, изображенное на рисунке 7.6. Этот диалог позволяет зашифровать папку вместе со всем ее содержимым при помощи драйвера шифрованной файловой системы. Все, что требуется сделать – установить флажок Encrypt contents to secure data и щелкнуть на OK. После этого вы можете не беспокоиться за содержимое папки – даже если кто-то получит ее физическую копию, без криптографического ключа информация останется недоступной.

Администрирование безопасности на уровне .NET

На рисунке 7.7[1] изображено средство администрирования .NET, при помощи которого можно настраивать различные функции .NET. Для того чтобы открыть это окно, откройте в панели управления папку Administrative Tools и выберите значок Microsoft .NET Framework Configuration. Затем выберите Runtime Security Policy. Подробней мы рассмотрим эту тему в главе 8.

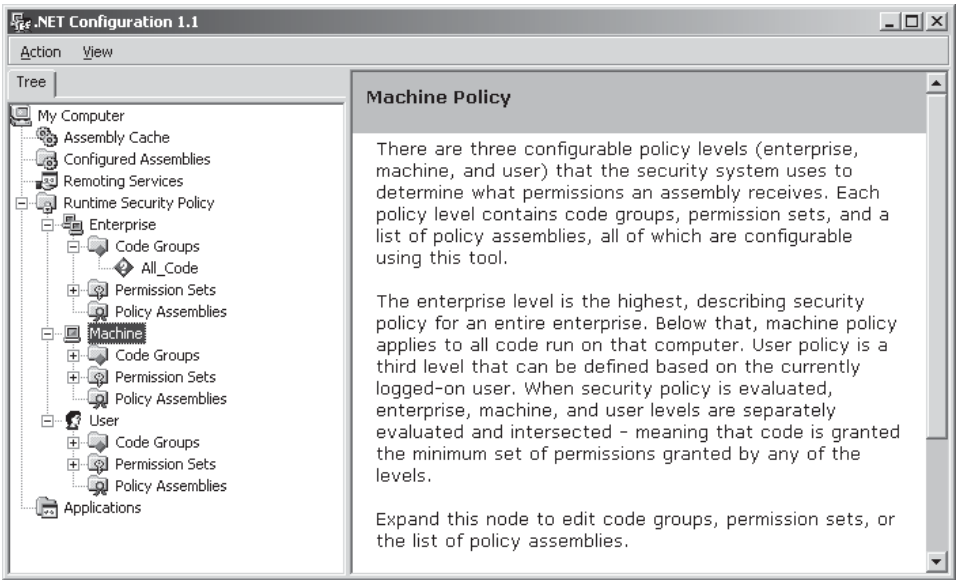


Рис. 7.7. Управление безопасностью .NET

[1] Для запуска выберите Start | Control Panel | Administrative Tools | Microsoft .NET Framework Configuration

Разрешения

Безопасность, основанная на идентификации пользователей, и концепция CAS являются двумя основными ветвями концепции безопасности .NET, и обе они требуют понимания общей концепции разрешений. В случае идентификации пользователей используется класс **PrincipalPermission**, при помощи которого идентичность текущего пользователя вызывающей нити. В случае CAS для проверки индивидуальных разрешений всех нитей, выполняющих текущий метод, используются различные классы, производные от класса **CodeAccessPermission**. Мы знакомимся с этой концепцией здесь потому, что в этой главе нам впервые придется ею воспользоваться. В следующей главе, посвященной CAS, мы будем использовать разрешения более активно, однако концепцию разрешений обсуждать более не будем, поэтому рекомендуем перечесть соответствующий материал, прежде чем двигаться вперед.

Разрешения представлены объектами, описывающими наборы операций, которые разрешены или запрещены в отношении защищаемых ресурсов в соответствии с некоторой политикой безопасности. Для разрешений CAS (но не для пользовательских разрешений) среда .NET CLR использует стековый механизм, для того чтобы определить, все ли элементы стека[1], вызвавшие методы, обладают соответствующими разрешениями. Объект-разрешение содержит в себе информацию о конкретном разрешении. Если ссылка на разрешение содержит значение Null, то она рассматривается как ссылка на объект-разрешение в состоянии **PermissionState.None**, что указывает на отсутствие разрешения. Разрешения используют, как правило, следующими способами:

- программный код может определять разрешения, которые ему необходимы для успешного выполнения;
- политика безопасности системы может предоставлять коду запрошенные разрешения или отказывать в них;
- код может требовать, чтобы вызывающий его код обладал некоторыми разрешениями при помощи метода **Demand**;
- код может преодолевать решения, принятые исходя из состояния стека, при помощи методов **Assert**, **Deny** и **PermitOnly**.

Возможна также работа с множествами (наборами) разрешений, представленными семействами (коллекциями) разнородных объектов-разрешений. Такая возможность реализуется классом **PermissionSet**, который позволяет манипулировать множеством объектов-разрешений, как группой разрешений.

---

[1] По соглашению стек вызовов представляется растущим вниз, поэтому методы, чьи вызовы располагаются в стеке выше, вызвали методы, чьи вызовы располагаются ниже. Иными словами, вызов текущего выполняемого метода всегда находится на дне стека.

Интерфейс IPermission

Интерфейс **IPermission** инкапсулирует в себе фундаментальную идею разрешения. Интерфейс **IPermission** поддерживает следующие публичные методы:

- метод **Copy** создает идентичную копию существующего объекта-разрешения;
- метод **Demand**[1] инициирует генерацию исключения **SecurityException**, если какой-либо из вызывающих методов, чьи вызовы расположены в стеке выше, не обладает разрешением, специфицированным данным объектом; этот метод используется для защиты в ситуации, когда посторонний код пытается воспользоваться текущим кодом для доступа к ресурсам, на которые у вызывающего кода нет разрешения;
- метод **Intersect** создает объект-разрешение, содержащий в себе пересечение (т. е. совпадающее подмножество) разрешений двух существующих объектов-разрешений;
- метод **IsSubsetOf** определяет, содержит ли объект-разрешение подмножество разрешений, заданных разрешениями другого объекта;
- метод **Union** создает объект-разрешение, чье множество разрешений является объединением множеств разрешений двух заданных объектов.

МЕТОД DEMAND

Вероятно наиболее важным членом интерфейса **IPermission** является метод **Demand**, который не требует никаких параметров и не возвращает никаких значений, но может инициировать генерацию исключения **SecurityException**.

```
void Demand()
```

Метод **Demand** проверяет, все ли вызывающие методы (т. е. методы, последовательность вызовов которых привела к вызову текущего метода) обладают разрешением на доступ к заданному ресурсу, и, если это не так, генерирует исключение. Конкретный вид ресурса, защищаемого таким образом, зависит от конкретной реализации интерфейса **IPermission**, т. е. от производного класса. Например, класс **FileIOPermission** защищает дисковые файлы, а класс **RegistryIPermission** защищает системный реестр.

Разумеется, до того как обратиться к методу **Demand**, необходимо вначале создать объект-разрешение, указать защищаемый ресурс и тип доступа, который подлежит контролю. В случае разрешения на файловый ввод-вывод, например, необходимо создать объект класса **FileIOPermission**, указать защищаемый файл и определить тип доступа, который

[1] С бытовой точки зрения слово demand (требование) наводит на мысль, что успех при его вызове гарантирован. Однако метод Demand может потерпеть неудачу и, наверное, его стоило бы назвать менее категоричным словом, как, например, Request (запрос) или Attempt (попытка). Но теперь уже поздно об этом сожалеть.

необходимо контролировать, например, операции чтения, записи и присоединения. Эту задачу можно решить при помощи соответствующего конструктора и методов **Copy**, **Intersect** и **Union**, которыми можно «выкроить» новый объект из объектов, уже существующих.

На интуитивном уровне это не вполне очевидно, но следует помнить: разрешения того кода, который вызвал метод **Demand**, не проверяются. Проверка начинается с вышестоящего вызова в стеке и продолжается до конца стека. Очевидно, что метод **Demand** успешно завершит свою работу только в том случае, если в ходе последовательных проверок не будет сгенерировано исключение.

МЕТОДЫ COPY, INTERSECT, UNION И ISSUBSETOF

Для манипуляций с объектами-разрешениями предназначены методы **Copy**, **Intersect** и **Union**. Метод **IsSubsetOf** предназначен для тестирования существующего объекта-разрешения. Вот синтаксические сигнатуры методов:

```
IPermission Copy();

IPermission Intersect(
   IPermission target
);

IPermission Union(
   IPermission target
);

bool IPermission IsSubsetOf (
   IPermission target
);
```

Иерархия наследования IPermission

Интерфейс **IPermission** обладает только двумя производными классами реализации: **PrincipalPermission** и **CodeAccessPermission**. **PrincipalPermission** обеспечивает проверку главного объекта, принадлежащего текущему принципалу, что соответствует концепции безопасности, основывающейся на идентификации пользователей. Принципал, связанный с текущей нитью или приложением, определяется в момент регистрации в системе, а конкретный объект этого принципала может быть задан (декларативно или императивно) в программном коде. Обсуждение класса **CodeAccessPermission** мы отложим до следующей главы, где будет описываться техника программирования с использованием CAS.

На рисунке 7.8 изображена иерархия наследования **IPermission**. Здесь не показаны индивидуальные пространства имен, но вам следует знать, что эти классы разрешений определены в нескольких отличающихся пространствах имен. Чаще всего для классов разрешений используется пространство имен **System.Security.Permissions**.

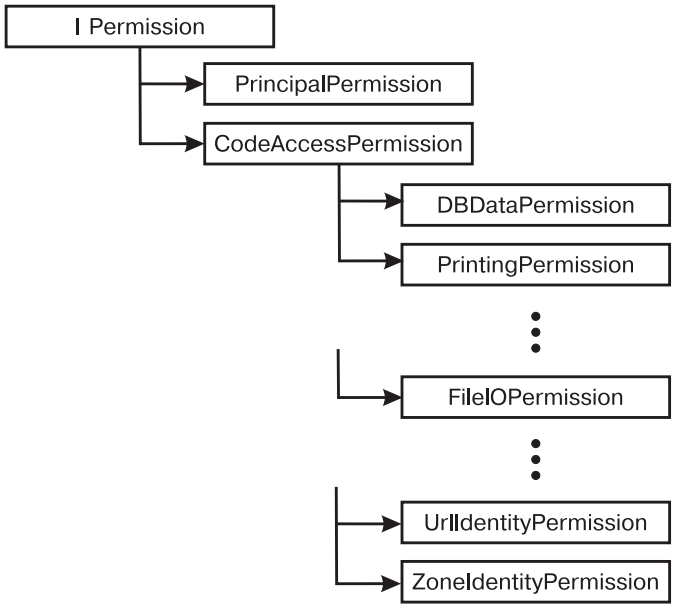


Рис. 7.8. Иерархия наследования IPermission

Класс PrincipalPermission

Как мы только что убедились, класс **PrincipalPermission** реализует интерфейс **IPermission**. Как мы увидим в программе **PrincipalPermission** далее в этой главе, класс **PrincipalPermission** используется для проверки объекта-принципала на предмет верификации пользователя. Для полного понимания того, что представляет класс **PrincipalPermission**, вам, вероятно, требуется понимать, что такое объект-принципал. Пока что вам достаточно знать, что принципал представляет зарегистрированного в данный момент пользователя, которого вы хотите распознать с точки зрения прав доступа. Далее мы подробней остановимся на понятии принципала, а сейчас, пока иерархия наследования интерфейса **IPermission** еще свежа в нашей памяти, давайте рассмотрим несколько важнейших членов класса **PrincipalPermission**.

Метод **Demand**, синтаксис которого аналогичен тому, что мы видели в наследовании **IPermission**, определяет, соответствует ли текущий принципал указанному объекту **PrincipalPermission**. Если проверка даст отрицательный результат, будет сгенерировано исключение **SecurityException**. Также этот метод используется для принудительного подтверждения аутентичности текущего принципала с тем, чтобы избежать генерации исключения.

```
public void Demand();
```

Другие методы **IPermission** работают для **PrincipalPermission** обычным образом – это относится к методам **Copy**, **Intersect** и **Union**. В качестве небольшого примера рассмотрим следующий отрывок кода. В этом коде два объекта-разрешения комбинируются при помощи метода **Union**. После вызова **Union** следует вызов **Demand**, который завершится успехом лишь в том случае, если объект-принципал pp соответствует пользователю Abbott, относящемуся к роли StraightMan, или пользователю Costello, относящемуся к роли FunnyMan.

```
String user1 = "Abbott";
String role1 = "StraightMan";
PrincipalPermission PrincipalPerm1 =
   new PrincipalPermission(user1, role1);
String user2 = "Costello";
String role2 = "FunnyMan";
PrincipalPermission pp =
   new PrincipalPermission(user2, role2);
PrincipalPermission.Union(pp) .Demand();
```

КОНСТРУКТОРЫ PRINCIPALPERMISSION

Для создания объекта **PrincipalPermission** с заданным перечислением **PermissionState** используется следующий конструктор. Перечисление **PermissionState** определяет разрешение со свободным доступом **(PermissionState.Unrestricted)** или разрешение с запретом доступа **(PermissionState.None)** к защищаемому ресурсу.

```
public PrincipalPermission(
   PermissionState state
);
```

Для создания объекта **PrincipalPermission** с заданными именем и ролью используют следующий конструктор.

```
public PrincipalPermission(
   string name,
   string role,
);
```

Для создания объекта **PrincipalPermission** с заданными именем, ролью и состоянием аутентификации используют следующий конструктор.

```
public PrincipalPermission(
   string name,
   string role,
   bool isAuthenticated
);
```

МЕТОДЫ FROMXML, TOXML И ISUNRESTRICTED

Помимо методов, определенных при помощи **IPermission** таких, как **Copy** и **Demand**, а также методов, определенных через **Object** таких, как **Equals** и **GetHashCode**, класс **PrincipalPermission** также определяет следующие дополнительные методы.

- **FromXml** реконструирует объект-разрешение из указанного XML-представления.
- **ToXml** создает XML-представление из текущего состояния объекта-разрешения.
- **IsUnrestricted** возвращает булево значение, указывающее, является ли разрешение разрешением со свободным доступом (unrestricted). Такой объект **PrincipalPermission** обеспечит доступ любому принципалу вне зависимости от его прав.

Ниже описан синтаксис этих трех методов. Обратите внимание, **SecurityElement** используется в качестве параметра метода **FromXml**, а также, как возвращаемое значение метода **ToXml**. Класс **SecurityElement** представляет модель простого XML-объекта, кодирующего объекты безопасности в форме тегов, атрибутов, узлов-потомков и текстовых элементов.

```
public void FromXml(
   SecurityElement elem
);

public SecurityElement ToXml();

public bool IsUnrestricted();
```

Безопасность, основанная на идентификации пользователей

Это традиционная концепция безопасности, используемая в программировании со времен первой версии Windows NT и Win32 Security API. Среда .NET Security Framework поддерживает эту концепцию в форме классов, интерфейсов, перечислений и т. д., которые инкапсулируют в себе средства управления доступом, основывающиеся на Win32 Security API. Такие объекты Win32 API совместно с объектами ACL, SA (Security Attribute – атрибут безопасности) и SIS (Security Identifier – идентификатор безопасности) по-прежнему остаются весьма эффективным средством, но в настоящее время они играют лишь роль дополнения к новым программным средствам .NET.

Объекты Principal и Identity

Объекты-принципалы (principal) и объекты идентификации (identity) используются для представления пользователей. Объект principal инкапсулирует текущего пользователя. Объект principal содержит в себе объект identity, который инкапсулирует идентификационную информацию о пользователе. По определению объект identity является экземпляром класса, который реализует интерфейс **IIdentity**, а объект principal является экземпляром класса, который реализует интерфейс **IPrincipal**.

Интерфейс IIdentity

Интерфейс **IIdentity** обладает только тремя публичными свойствами «только для чтения»: **AuthenticationType**, **IsAuthenticated** и **Name**.

- **AuthenticationType** – это строка, указывающая на тип аутентификации, который используется для текущего пользователя. Аутентификация может обеспечиваться операционной системой или другим поставщиком аутентификации таким, как IIS. Возможно использование даже пользовательской аутентификационной схемы. Примерами стандартных типов аутентификации являются Basic, NTLM, Kerberos и Passport.
- **IsAuthenticated** – это булево значение, показывающее, аутентифицирован текущий пользователь или нет.
- **Name** – это строка, содержащая регистрационное имя текущего пользователя. Это имя задается поставщиком аутентификации, например, операционной системой.

Вот синтаксическое определение этих трех свойств «только для чтения». Как видите, **AuthenticationType** возвращает строку, **IsAuthenticated** возвращает булево значение, а **Name** возвращает строку.

```
string AuthenticationType {get;}
bool IsAuthenticated {get;}
string Name  {get;}
```

Классы, реализующие интерфейс IIdentity

Существует четыре класса, которые реализуют интерфейс **IIdentity**: **FormsIdentity**, **GenericIdentity**, **PassportIdentity** и **WindowsIdentity**. Класс WindowsIdentity используется в случае, когда вы полагаетесь на стандартную аутентификацию Windows. Упрощенный класс **GenericIdentity** подходит в случае, когда вы используете собственный сценарий регистрации и аутентификации. Принципалы в этом классе представляют пользователей, совершенно независимых от пользователей Windows. Классы **FormsIdentity** и **PassportIdentity** используются в сценариях аутентификации ASP.NET и Passport, соответственно. Также вы можете произвести из

GenericIdentity свои собственные классы для хранения дополнительной информации, аутентифицирующей пользователей.

- **GenericIdentity** представляет упрощенный объект идентификации (объект identity) для пользователей, аутентификация которых осуществляется нестандартным сценарием регистрации.
- **WindowsIdentity** представляет объект идентификации для обычных пользователей Windows, которые зарегистрированы в системе.
- **FormsIdentity** обеспечивает класс объектов идентификации для приложений ASP.NET, которые используют формы аутентификации.
- **PassportIdentity** обеспечивает класс объектов идентификации для использования в приложениях с поддержкой технологии Passport. Для использования этого класса необходимо установить пакет Passport SDK.

Класс GenericIdentity

Класс **GenericIdentity** довольно прост и не связан с каким-либо конкретным протоколом аутентификации. Напротив, он предназначен для использования с нестандартными механизмами регистрации. Например, приложение может выполнять сверку с собственной базой данных, содержащей регистрационные имена пользователей и пароли. Пароли, разумеется, будут зашифрованы! Если имя пользователя и пароль действительны, приложение может создать соответствующий принципал-объект и связанный с ним объект идентификации, основываясь на совпадении введенных данных и данных в базе.

Класс **GenericIdentity** не содержит никаких дополнительных членов кроме трех свойств, определенных интерфейсом **IIdentity**, то есть **AuthenticationType**, **IsAuthenticated** и **Name**, с которыми мы познакомились ранее. Однако этот класс все же поддерживает два конструктора. Один конструктор принимает строковый параметр, содержащий имя пользователя. Другой конструктор требует двух параметров. Первым параметром должна быть строка имени пользователя, а вторым – произвольная строка типа аутентификации.

```
public GenericIdentity(
   string name
);

public GenericIdentity(
   string name,
   string type
);
```

Мы не будем вдаваться в детали использования **GenericIdentity** на данном этапе. Пока что просто приведем фрагмент кода, где оба доступных

конструктора используются для создания экземпляров этого класса. Позднее, познакомившись с классом **GenericPrincipal**, мы подробней познакомимся с использованием объектов класса **GenericIdentity** в реальных программах.

```
IIdentity genericIdentity1 =
   new GenericIdentity("JoeUser");

IIdentity genericIdentity2 =
   new GenericIdentity(
            "JoeUser", "MyAuthenticationProtocol");
```

Класс WindowsIdentity

Из классов, реализующих интерфейс **IIdentity**, нас в этой главе интересует класс **GenericIdentity**, который мы уже обсудили, а также класс **WindowsIdentity**. Классы **FormsIdentity** и **PassportIdentity** далее в этой главе рассматриваться не будут. Итак, класс **WindowsIdentity** представляет пользователя, зарегистрированного в системе Windows.

КОНСТРУКТОРЫ WINDOWSIDENTITY

Для создания объектов **WindowsIdentity**, представляющих заданного пользователя, предусмотрено несколько конструкторов. Простейший конструктор принимает единственный параметр типа **IntPtr**, ссылающийся на учетную запись пользователя Windows. Тип данных **IntPtr** вообще используется для ссылок на платформо-зависимые типы данных, представляющие либо указатель памяти, либо дескриптор. В последнем случае **IntPtr** ссылается на дескриптор Win32, представляющий 32-битовый указатель на учетную запись пользователя. Обычно такой указатель получается вызовом обычного API Win32 таким, как **LogonUser**. Сведения о вызовах обычного API из среды .NET смотрите в документации Microsoft Visual Studio .NET.

Все остальные конструкторы также принимают параметр типа **IntPtr**, сопровождаемый одним или несколькими дополнительными параметрами. Ниже приведено синтаксическое определение этих конструкторов. Обратите внимание, параметр **WindowsAccountType** должен содержать одно из значений перечисления **WindowsAccountType**, то есть **Anonymous**, **Guest**, **Normal** или **System**.

```
public WindowsIdentity(
   IntPtr userToken
);


public WindowsIdentity(
   IntPtr userToken,
```

```
   string authType
);

public WindowsIdentity(
   IntPtr userToken,
   string authType,
   WindowsAccountType accType
);

public WindowsIdentity(
   IntPtr userToken,
   string authType,
   WindowsAccountType accType,
   bool isAuthenticated
);
```

ПУБЛИЧНЫЕ СВОЙСТВА WINDOWSIDENTITY

Разумеется, класс **WindowsIdentity** предоставляет три публичных свойства «только для чтения», определенных интерфейсом **IIdentity**: **AuthenticationType**, **IsAuthenticated** и **Name**. Также этот класс предусматривает несколько дополнительных свойств. Далее приведен список свойств класса **WindowsIdentity**:

- **AuthenticationType** возвращает тип аутентификации пользователя;
- **IsAnonymous** возвращает булево значение, указывающее, является ли пользовательская учетная запись в операционной системе анонимной;
- **IsAuthenticated** возвращает булево значение, указывающее, был ли аутентифицирован пользователь операционной системой;
- **IsGuest** возвращает булево значение, указывающее, является ли пользовательская учетная запись в операционной системе гостевой;
- **IsSystem** возвращает булево значение, указывающее, является ли пользовательская учетная запись системной;
- **Name** возвращает регистрационное имя пользователя в системе в нотации Домен\Пользователь;
- **Token** возвращает указатель на учетную запись пользователя.

Три свойства – **AuthenticationType**, **IsAuthenticated** и **Name** – определены в интерфейсе **IIdentity**, который реализован в классе **WindowsIdentity**. В дополнение к этому в классе реализовано еще несколько свойств, как показывает предыдущий список. Вот синтаксическое определение всех этих свойств.

```
public virtual string AuthenticationType {get;}
public virtual bool IsAnonymous {get;}
```

```
public virtual bool IsAuthenticated {get;}
public virtual bool IsGuest {get;}
public virtual bool IsSystem {get;}
public virtual string Name {get;}
public virtual IntPtr Token {get;}
```

ПУБЛИЧНЫЕ МЕТОДЫ WINDOWSIDENTITY

Кроме обычных методов, определенных классом **Object**, класс **WindowsIdentity** включает в себя методы **GetAnonymous**, **GetCurrent** и **Impersonate**. Методы **GetAnonymous** и **GetCurrent** – статические, в то время как метод **Impersonate** может быть как статическим методом, так и методом экземпляра.

- **GetAnonymous** возвращает объект **WindowsIdentity**, представляющий анонимного пользователя.
- **GetCurrent** возвращает объект **WindowsIdentity**, представляющий текущего пользователя.
- **Impersonate** позволяет программному коду временно выполняться от имени другого пользователя.

Методы **GetAnonymous** и **GetCurrent** возвращают объект **WindowsIdentity**. Impersonate, как метод экземпляра, не принимает параметров и возвращает объект **WindowsImpersonationContext**, основанный на том **WindowsIdentity**, откуда он вызван. В своем статическом варианте этот метод получает параметр **IntPtr**, являющийся указателем на учетную запись Windows. В обоих случаях для представления пользователя Windows, от чьего имени должен выполняться код, используется объект **WindowsImpersonationContext**. Такой механизм полезен для серверных приложений, которым требуется сократить свои права доступа до более низкого уровня, соответствующему уровню пользователя, которого они обслуживают. Вот синтаксическое определение этих методов.

```
public static WindowsIdentity GetAnonymous();
public static WindowsIdentity GetCurrent();

public virtual WindowsImpersonationContext Impersonate();

public static WindowsImpersonationContext Impersonate(
   IntPtr userToken
);
```

Объекты-принципалы

Объект-принципал – это экземпляр одного из классов, реализующих интерфейс **IPrincipal**. Объект-принципал представляет пользователя в системах управления доступом, основанных на концепции пользователя. Пространство имен **System.Security.Principal** содержит классы объек-

тов-принципалов нескольких типов, которые инкапсулируют контекст безопасности, в котором может выполняться код приложений. Эти объекты содержат в себе информацию, которая используется для представления пользователя. Доступ к ресурсам таким образом можно контролировать на основе тех прав, которыми наделен конкретный пользователь. Вскоре мы познакомимся с примером того, как имя пользователя и его роль проверяются на предмет разрешения выполнения конкретных действий.

ПОЛУЧЕНИЕ ТЕКУЩЕГО ОБЪЕКТА-ПРИНЦИПАЛА

Каждая нить процесса ассоциирована с некоторым объектом-принципалом. Этот объект содержит в себе объект идентификации, представляющий пользователя, выполняющего текущую нить. Класс **Thread** обладает статическим свойством **CurrentPrincipal**, возвращающим соответствующий объект-принципал.

```
IPrincipal ip =
   Thread.CurrentPrincipal;
```

Интерфейс IPrincipal

Интерфейс **IPrincipal** содержит только одно публичное свойство с именем **Identity** и один публичный метод с именем **IsInRole**.

- Свойство **Identity** указывает на объект **IIdentity**, связанный с данным объектом-принципалом.
- Метод **IsInRole** получает в параметре строку с именем роли и возвращает булево значение, указывающее, принадлежит ли принципал к заданной роли.

Вот синтаксическое определение этих членов интерфейса **IPrincipal**.

```
IIdentity Identity {get;}

bool IsInRole(
   string role
);
```

Вы можете реализовать собственные классы принципалов, однако из предопределенных классов .NET Framework только два реализуют интерфейс **IPrincipal**. Нас в основном интересует класс **WindowsPrincipal**, и мы лишь вкратце коснемся класса **GenericPrincipal**.

- **GenericPrincipal** представляет общий объект-принципал для зарегистрированного пользователя.
- **WindowsPrincipal** представляет объект-принципал для зарегистрированного пользователя Windows.

Класс GenericPrincipal

Класс **GenericPrincipal** используется в сочетании с классом **GenericIdentity** для того, чтобы представлять аутентифицированного пользователя. После создания объекта **GenericIdentity**, как описывалось ранее, вы можете создать экземпляр класса **GenericPrincipal**. Конструктор **GenericPrincipal** позволяет инициализировать его с предварительно созданным объектом **GenericIdentity** и массивом строк, содержащим список ролей, к которым должен принадлежать создаваемый объект-принципал.

Чтобы понять, как может работать нестандартная аутентификация, посмотрите на следующий фрагмент кода, который взят из метода **CustomLogon** в примере программы **GenericPrincipal**. Он создает вначале объект **GenericIdentity**, а затем создает инкапсулирующий объект **GenericPrincipal**, используя также массив строк со списком ролей. Объект **GenericIdentity** создается с заданным регистрационным именем пользователя и нестандартным типом аутентификации, произвольно названным нами **MyAuthenticationType**. Этот тип аутентификации является, таким образом, нашей доморощенной альтернативой для Kerberos или NTLM. Для упрощения примера роли игнорируются, поэтому строковый массив пуст. После того как объект **GenericPrincipal** создан, он «прикрепляется» к текущей нити процесса так, чтобы его можно было впоследствии использовать для верификации текущего пользователя.

```
//создать объект идентификации
IIdentity myGenericIdentity =
   new GenericIdentity(
      strUserName, "MyAuthenticationType");

// создать объект-принципал
String[] roles = null; //в данном примере не используется
GenericPrincipal myGenericPrincipal =
   new GenericPrincipal(myGenericIdentity, roles);

//присоединить объект к текущей нити
Thread.CurrentPrincipal = myGenericPrincipal;
```

В начале программы, прежде создания объекта **GenericIdentity**, описанного выше[1], предлагается ввести имя пользователя. Для простоты в коде предусмотрено лишь два возможных имени – TrustedUser (пользователь, достойный доверия) и UntrustedUser (пользователь, недостойный доверия). Любые другие имена отвергаются, как недействительные. Также в целях упрощения не запрашивается пароль. Разумеется, в реалистическом сценарии необходимо запрашивать пароль и сравнивать его с

[1] Такой подход непригоден для приложений ASP.NET, поскольку пользователи там работают дистанционно. Приложения ASP.NET должны использовать свойство **HttpContext-User**.

зашифрованным паролем в базе данных, но это сделало бы наш пример слишком громоздким. Эти функции, дополнительно усложняющие код, легко реализовать, но они имеют слабое отношение к нашей теме и потому мы не будем в дальнейшем о них вспоминать. Еще одно упрощение в нашем примере состоит в том, что полностью игнорируется членство в роли. В реальной программе роли, связанные с пользователем, необходимо извлечь из базы данных и передать в форме строкового массива в качестве второго параметра для конструктора **GenericPrincipal**.

Затем программа дважды вызывает метод **AttemptCodeAsUser**, для того чтобы продемонстрировать оба возможных варианта. В первом случае пользователь регистрируется как TrustedUser и вызывает метод **AttemptCodeAsUser**. Затем пользователь регистрируется как UntrustedUser и снова вызывает метод **AttemptCodeAsUser**. Следующий фрагмент кода показывает, как имя пользователя извлекается из объекта-принципала и сравнивается с заданным прямо в коде именем TrustedUser. Если имена совпали, программа выполняется нормально, в противном случае будет сгенерировано исключение.

```
//получить текущий объект-принципал
IPrincipal principle = Thread.CurrentPrincipal;

if (!principle.Identity.Name.Equals("TrustedUser"))
{
   throw new SecurityException(
              strUserName + " - работа запрещена.\n");
}
Console.WriteLine(
   strUserName + " - работа разрешена.\n");
```

Ниже приведен консольный вывод программы. Как видите, программа позволяет зарегистрировать пользователя и принять решение на основании его личности.

```
Logon as TrustedUser
Enter username: TrustedUser

User name: TrustedUser
Authenticated: True

AuthenticationType:  MyAuthenticationType
TrustedUser - работа разрешена.

Logon as UntrustedUser
Enter username: UntrustedUser

User name: UntrustedUser
Authenticated: True

AuthenticationType:  MyAuthenticationType
UntrustedUser - работа запрещена.
```

Главное, что необходимо тут отметить: нестандартная функция регистрации никак не связана с этой программой. Несмотря на тот факт, что механизм регистрации здесь крайне упрощен, он демонстрирует все же ключевые аспекты программирования при помощи классов **GenericIdentity** и **GenericPrincipal**.

Класс WindowsPrincipal

Мы уже знаем, что класс **WindowsPrincipal** является одним из двух классов, реализующих интерфейс **IPrincipal**. Также мы убедились в простоте интерфейса **IPrincipal**, предоставляющего одно свойство по имени **Identity** и один публичный метод с именем **IsInRole**. Разумеется, **WindowsPrincipal** наследует стандартный метод **Object** и реализует стандартные элементы **IPrincipal**, однако мало что привносит своего. Он обеспечивает единственный конструктор, а его реализация метода **IsInRole** перегружена тремя способами, но это и все, что он привносит.

КОНСТРУКТОР WINDOWSPRINCIPAL

Класс **WindowsPrincipal** предусматривает лишь один конструктор, который создает объект **WindowsPrincipal** на основе существующего объекта **WindowsIdentity**. Класс **WindowsIdentity** мы обсуждали ранее в этой главе.

```
public WindowsPrincipal(
   WindowsIdentity ntIdentity
);
```

Вот пример использования этого конструктора для создания объекта **WindowsPrincipal** на основе существующего объекта **WindowsIdentity**, представляющего текущего пользователя. Удобный способ получения объекта **WindowsIdentity** (дальнейшей передачи конструктору **WindowsPrincipal**) состоит в вызове статического метода **WindowsIdentity.GetCurrent**.

```
WindowsIdentity wi = WindowsIdentity.GetCurrent();
WindowsPrincipal wp = new WindowsPrincipal(wi);
```

СВОЙСТВО IDENTITY

Вот синтаксическое определение свойства **Identity** класса **WindowsPrincipal**.

```
public virtual IIdentity Identity {get;}
```

МЕТОД WINDOWSPRINCIPAL.ISINROLE

Класс **WindowsPrincipal** обладает тремя вариантами перегруженного метода **IsInRole**. Первый вариант метода получает целое число, представляющее группу пользователя в форме RID (relative identifier – относительный

идентификатор)[1]. Значения RID определяются в заголовочном файле Winnt.h, располагающемся в папке ...\Microsoft Visual Studio .NET\Vc7\PlatformSDK\Include. Второй перегруженный метод получает строку с групповым именем пользователя в форме ИмяМашины\ИмяГруппы. Например, имя HPDESKTOP\CodeGurus представляет пользователей, принадлежащих к группе CodeGurus, определенной на машине HPDESKTOP. Если речь идет о встроенных (стандартных) группах, то правило формирования имени немного меняется. Например, для группы Administrators имя должно быть не HPDESKTOP\Administrators, а BUILTIN\Administrators. Это правило представляется не совсем логичным, и потому для стандартных встроенных групп лучше будет использовать третий вариант перегруженного метода, который предназначен специально для работы со встроенными типами. Этот метод получает перечисление **WindowsBuiltInRole**, которое может принимать такие значения, как **Administrator**, **Guest** и **User**. Далее приведено синтаксическое определение трех вариантов перегруженного метода **IsInRole**.

```
public virtual bool IsInRole(int);
public virtual bool IsInRole(string);
public virtual bool IsInRole(WindowsBuiltInRole);
```

Вот список всех значений, определенных в перечислении **WindowsBuiltInRole** для использования в параметрах перегруженного метода **IsInRole**:

- **AccountOperator** – управляет пользовательскими учетными записями компьютера или домена;
- **Administrator** – имеет неограниченный доступ к ресурсам компьютера или домена;
- **BackupOperator** – выполняет операции резервного копирования и восстановления в файловой системе;
- **Guest** – как пользователь, однако имеет больше ограничений;
- **PowerUser** – почти как администратор, однако некоторые ограничения все же имеются;
- **PrintOperator** – выполняет операции печати;
- **Replicator** – выполняет файловые репликации внутри домена;
- **SystemOperator** – управляет компьютером;
- **User** – наложены ограничения, предотвращающие выполнение опасных или критичных для системы действий.

[1] Идентификатор RID определяется, как хорошо известный доменный идентификатор. В файле Winnt.h определяются идентификаторы RID для нескольких пользователей, включая DOMAIN_USER_RID_ADMIN и DOMAIN_ USER_RID_GUEST. Здесь также определены RID для нескольких групп, включая DOMAIN_GROUP_RID_ADMINS, DOMAIN_GROUP_RID_USERS и DOMAIN_GROUP_RID_GUESTS.

Два подхода к безопасности, основанной на идентификации пользователей

В среде .NET возможно использование двух разных подходов к концепции безопасности, основанной на идентификации пользователей. Первый подход называют *императивным*, и заключается он в явных решениях, принимаемых кодом программы. Второй подход называют декларативным, и состоит он в использовании механизма атрибутов.

В применении императивного подхода имеется два слегка отличающихся стиля. Старый императивный стиль соответствует, в основном, традиционному программированию Win32, где вы выясняете, кто является текущим пользователем, и явным образом принимаете в программе решение при помощи конструкции if. Обычно речь идет о выборе между двумя ветвями кода, где либо продолжается нормальное выполнение, либо генерируется исключение **SecurityException**. Хотя эта техника знакома большинству программистов, она требует много дополнительного кода, делающего реализацию громоздкой и неуклюжей.

Новый стиль реализации императивного подхода заключается в том, что вы создаете объект **PrincipalPermission**, представляющий пользователя или роль, которую вы намерены проверить на предмет разрешений, а затем вызываете метод **Demand** этого объекта **PrincipalPermission** для проверки полномочий текущего пользователя. Метод **Demand** автоматически принимает решение и в нужном случае генерирует исключение **SecurityException**. Преимущество этого стиля состоит в том, что код получается гораздо меньшим по размеру и более ясным для понимания, поскольку явного принятия решения и явной генерации исключения в нем нет.

Альтернативой в случае реализации концепции безопасности, основанной на идентификации пользователей, является использование декларативного подхода. Если в «императивном» программировании вы пишете код, который явным образом выполняет какие-то действия, то при «декларативном» программировании вы описываете атрибуты, которые уже в период выполнения окажут влияние на поведение программы. В нашем случае реализация декларативного подхода заключается в применении атрибута **PrincipalPermission** к требуемому методу. Давайте рассмотрим два примера программ, реализующих эти два подхода к концепции безопасности, основанной на идентификации пользователей.

Императивный подход

Как вам вероятно известно, в системе Windows определено несколько «встроенных» пользователей таких, как Administrator или Guest, и групп таких, как Administrators, Users или Guests. Обычно администратор

определяет также нескольких своих пользователей. Код программы **ImperativeUserBasedSecurity** показывает, как просто можно реализовать принятие решений на основе распознавания пользователей и ролей, к которым они относятся. Почти в самом начале файла **ImperativeUserBasedSecurity.cs** вы обнаружите следующие операторы **using**, позволяющие использовать короткие имена классов.

```
using System.Security;
using System.Security.Principal;
using System.Threading;
```

В этой простой программе предусмотрено два элемента управления (кнопки) типа **Button**. Кнопка Test On User Name демонстрирует принятие решения по имени пользователя, введенном в текстовое поле (элемент управления **TextBox**). Кнопка Test On Role действует аналогичным образом с той только разницей, что решение принимается на основе роли, выбранной переключателем (элемент управления **RadioButton**) – доступны варианты Administrator, User или Guest. Давайте рассмотрим вначале метод **buttonTestOnUserName_Click**.

```
private void buttonTestOnUserName_Click(
   object sender, System.EventArgs e)
{
   //принципал по умолчанию для нитей домена приложения
   AppDomain appdomain = AppDomain.CurrentDomain;
   appdomain.SetPrincipalPolicy(
              PrincipalPolicy.WindowsPrincipal);

   //получить текущий объект-принципал
   WindowsPrincipal principle =
              (WindowsPrincipal)Thread.CurrentPrincipal;

   //получить заданное имя пользователя
   string userName = textUserName.Text;

   //выполнить код согласно полученному имени пользователя
   if (!principle.Identity.Name.Equals(userName))
   {
              throw new SecurityException(
                     "Задан пользователь " +
                     userName +
                     ".\n" +
                     "Поэтому текущему пользователю " +
                     principle.Identity.Name +
                     " работа запрещена.");
   }
   MessageBox.Show(
              "Задан пользователь " +
```

```
                userName +
                ".\n" +
                "Поэтому текущему пользователю " +
                principle.Identity.Name +
                " работа разрешена.");
}
```

SetPrincipalPolicy генерирует исключение SecurityException

Эта программа вызывает метод **SetPrincipalPolicy**, который генерирует исключение **SecurityException**, если у кода нет разрешения манипулировать политиками безопасности объекта **AppDomain**. Чтобы упростить пример, мы не проверяем, возможно ли это для нас, и просто вызываем метод **SetPrincipalPolicy** безо всякой подготовки. В реальной программе необходимо вначале проверить, имеется ли разрешение на выполнение этого действия. В следующем фрагменте кода такая проверка выполняется вызовом метода **Demand** специально созданного для этой цели объекта **SecurityPermission**. Класс **SecurityPermission** проверяет «метаразрешения», управляющие подсистемой безопасности среды выполнения CLR. Вскоре мы подробней рассмотрим объекты-разрешения и метод **Demand**.

```
SecurityPermission sp = new SecurityPermission(
            SecurityPermissionFlag.ControlPrincipal);
try
{
            Sp.Demand();
}
catch(SecurityException se)
{
            ...//невозможно вызвать SetPrincipalPolicy
}
//вызов SetPrincipalPolicy возможен
```

Первое, что делает наш пример программы – вызывает метод **SetPrincipalPolicy**, который задает тип принципала, который должен быть связан с доменом текущего приложения так, чтобы быть объектом типа **WindowsPrincipal**. Это необходимо по той причине, что только объект **WindowsPrincipal** заключает в себе необходимую информацию о текущем пользователе, аутентифицированном средствами Windows[1]. Без этой информации у нас не будет уверенности в личности пользователя. Далее мы извлекаем текущий объект-принципал, вызвав статический метод **Thread.CurrentPrincipal**.

[1] Если мы не вызовем **SetPrincipalPolicy** для указания того факта, что нам требуется именно **WindowsPrincipal**, то получим по умолчанию объект типа **GenericPrincipal**, в котором нет необходимой нам информации. Поскольку пользователь был аутентифицирован средствами Windows, то имя пользователя и его роль можно найти в **WindowsPrincipal**, но не в **GenericPrincipal**.

```
      //принципал по умолчанию для нитей домена приложения
   AppDomain appdomain = AppDomain.CurrentDomain;
   appdomain.SetPrincipalPolicy(
              PrincipalPolicy.WindowsPrincipal);

   //получить текущий объект-принципал
   WindowsPrincipal principle =
              (WindowsPrincipal)Thread.CurrentPrincipal;
```

Далее программа получает из пользовательского интерфейса заданное имя пользователя. Наконец, эта строка сравнивается с значением свойства **Name** объекта **Identity** для проверки, совпадает ли заданное имя с именем фактически зарегистрированного пользователя. Если да, действие разрешено. В противном случае генерируется исключение. Поскольку наш пример создан лишь для иллюстрации принципа, оба варианта поведения имитируются обычными текстовыми сообщениями.

```
      //получить заданное имя пользователя
   string userName = textUserName.Text;

   //выполнить код согласно полученному имени пользователя
   if (!principle.Identity.Name.Equals(userName))
...// генерируется исключение
```

Метод **buttonTestOnUserName_Click**, который мы рассмотрели, принимает решение на основе имени пользователя. Пользователь, однако, может принадлежать к одной или нескольким группам, что значительно упрощает дело, поскольку нужно принимать решения на основе обобщенных характеристик подмножеств пользователей вместо того, чтобы принимать решение по каждому пользователю в отдельности. Давайте рассмотрим метод **buttonTestOnRole_Click**, он принимает решение, основываясь на заданной группе.

```
private void buttonTestOnRole_Click(
   object sender, System.EventArgs e)
{
   // принципал по умолчанию для нитей домена приложения
   AppDomain appdomain = AppDomain.CurrentDomain;
   appdomain.SetPrincipalPolicy(
              PrincipalPolicy.WindowsPrincipal);

   // получить текущий объект-принципал
   WindowsPrincipal principle =
              (WindowsPrincipal)Thread.CurrentPrincipal;

   //получить заданную роль
   WindowsBuiltInRole role = 0;
   if (radioButtonGuest.Checked == true)
```

```
            role = WindowsBuiltInRole.Guest;
   if (radioButtonUser.Checked == true)
            role = WindowsBuiltInRole.User;
   if (radioButtonAdministrator.Checked == true)
            role = WindowsBuiltInRole.Administrator;

   //выполнить код согласно заданной роли
   if (!principle.IsInRole(role))
   {
            throw new SecurityException(
                  "Задана роль " +
                  role +
                  ".\n" +
                  "Поэтому текущему пользователю " +
                  principle.Identity.Name +
                  " работа запрещена.");
   }
   MessageBox.Show(
            "Задана роль " +
            role +
            ".\n" +
            "Поэтому текущему пользователю " +
            principle.Identity.Name +
            " работа разрешена.");
            }
   }
}
```

Метод стартует аналогично предыдущему, также задавая тип объект-принципала, с которым мы хотим иметь дело, вызовом метода **SetPrincipalPolicy**. Мы снова делаем это, чтобы получить затем объект типа **WindowsPrincipal**. Далее мы переходим к принятию решений, на этот раз основываясь на роли, к которой принадлежит пользователь, а не на самом пользователе. Заданную роль мы получаем из пользовательского интерфейса и для проверки ее используем метод **IsInRole**.

```
      //выполнить код согласно заданной роли
   if (!principle.IsInRole(role))
            ...// генерация исключения
```

На рисунках 7.9 и 7.10 изображены два возможных варианта работы программы. Здесь предполагается, что текущим зарегистрированным пользователем является Administrator, который принадлежит к ролям Administrators и Users, но не принадлежит к Guests. При выполнении примера в вашей системе вам следует заменить имя машины и имя домена на соответствующие им в вашей системе.

ИСПОЛЬЗОВАНИЕ PRINCIPALPERMISSION ПРИ РЕАЛИЗАЦИИ ИМПЕРАТИВНОГО ПОДХОДА

Пример программы **ImperativeUserBasedSecurity**, который мы рассмотрели, выглядит слегка громоздким и неуклюжим из-за того, что использует конструкции **if** и явным образом генерирует исключения. Программу можно упростить, применив класс **PrincipalPermission**. Проверку текущего пользователя на соответствие заданному можно заменить кодом, использующим класс **PrincipalPermission**. Например, посмотрите, как приведенный ниже код, взятый из программы **PrincipalPermission**, создает объект класса **PrincipalPermission** с заданными именем пользователя и/или ролью, а затем вызывает метод **Demand**.

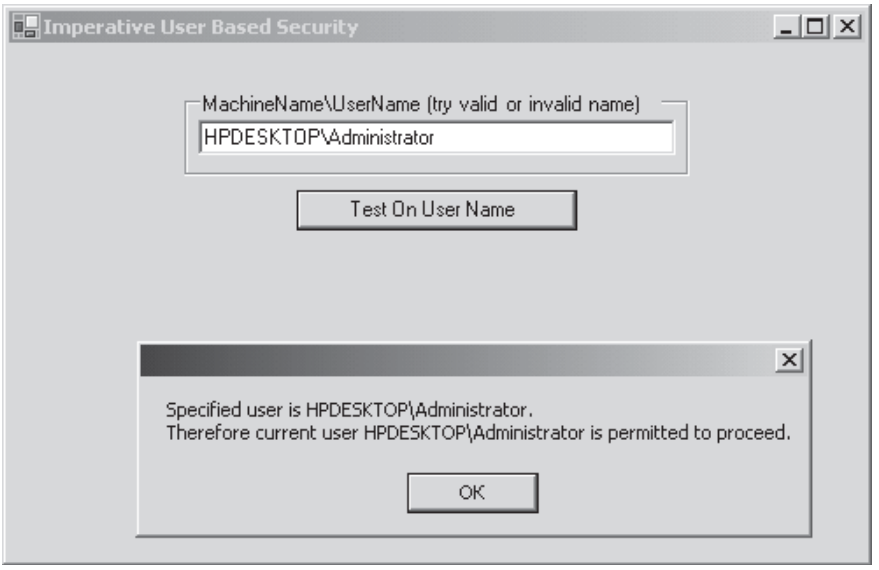


Рис. 7.9. Имя Administrator действительно

Так же, как и в предыдущей программе **ImperativeUserBasedSecurity**, здесь реализован скорее императивный подход, чем декларативный.

```
PrincipalPermission pp =
        new PrincipalPermission(
        strUserName, strUserRole);

//если пользователь недействителен, будет сгенерировано исключение
pp.Demand();

//если мы сюда попали, значит, пользователь действителен
MessageBox.Show(
        "Заданный пользователь соответствует текущему." +
        "Работа разрешена.");
```

Два строковых параметра конструктора **PrincipalPermission**, представляющих имя пользователя и роль, разумеется, необходимо подготовить заранее. Если один из параметров отсутствует, он будет проигнорирован. Теперь вместо условной конструкции **if**, которой мы пользовались в предыдущем примере, мы просто вызываем метод **Demand**, который автоматически выполняет проверки и принимает решение о генерации исключения **SecurityException**.

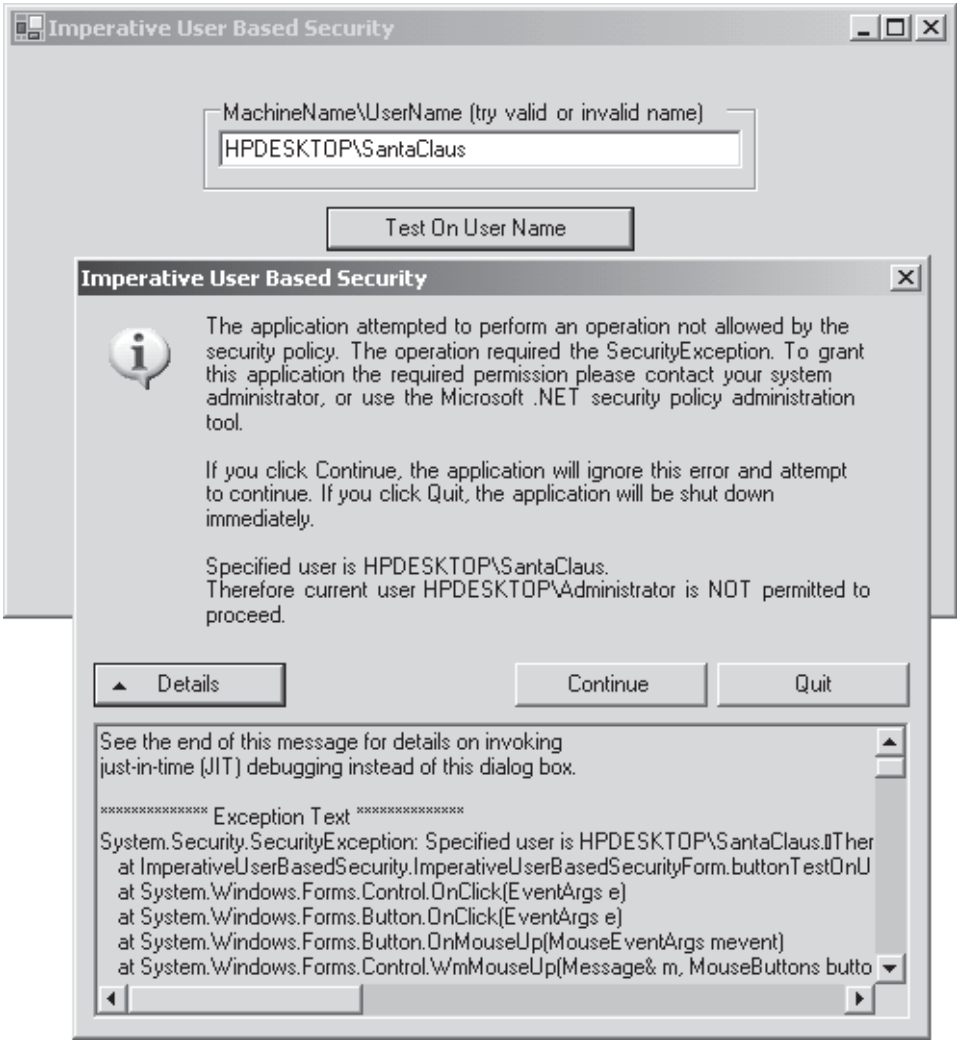


Рис. 7.10. Имя SantaClaus недействительно!

PrincipalPermission – не производный класс от CodeAccessPermission

Подобно другим объектам-разрешениям, **PrincipalPermission** реализует интерфейс **IPermission**, но в отличие от других, он не является производным классом от **CodeAccessPermission**. Причина в том, что **PrincipalPermission** основывается не на идентификации выполняемой сборки кода (т. е. не используется в CAS), а на идентификации текущего пользователя. Подробнее механизм разрешений мы рассмотрим в главе 8.

Декларативный подход

Пример программы **DeclarativeUserBasedSecurity** покажет нам, как реализовать концепцию безопасности, основывающуюся на идентификации пользователя, используя декларативный подход. Как видите, код программы очень прост, поскольку никакого кода, явным образом выполняющего проверки, вообще нет. Вместо этого мы применяем атрибут **PrincipalPermission** к методу, как к целому, и метод может заниматься своими делами, не беспокоясь о вопросах безопасности. Если текущий пользователь окажется отличным от заданного, то, благодаря атрибуту **PrincipalPermission**, исключение **SecurityException** будет сгенерировано автоматически.

```
[PrincipalPermission(
         SecurityAction.Demand,
         Name="HPDESKTOP\\Administrator")]
private void buttonTest_Click(
        object sender, System.EventArgs e)
{
        MessageBox.Show(
                "Данному пользователю работа запрещена.");
}
```

Квадратные скобки в этом коде объявляют атрибут **PrincipalPermission** для метода **buttonTest_Click**. Этот атрибут задает действие Demand и имя пользователя HPDESKTOP\\Administrator. Подробнее этот атрибут описан в документации к классу **PrincipalPermissionAttribute**, который его инкапсулирует. Если вы запустите эту программу, то, в зависимости от того, зарегистрированы вы как HPDESKTOP\\Administrator или нет, вы получите один из двух результатов, изображенных на рисунках 7.11 и 7.12.

Обратите внимание, что для проверки программы необязательно регистрироваться в системе под разными именами. Вместо этого вы можете воспользоваться командной строкой и утилитой **runas**, как показано ниже. После запуска команды вы получите приглашение ввести пароль соответствующего пользователя.

Рис. 7.11. PrincipalPermission: имя пользователя действительно

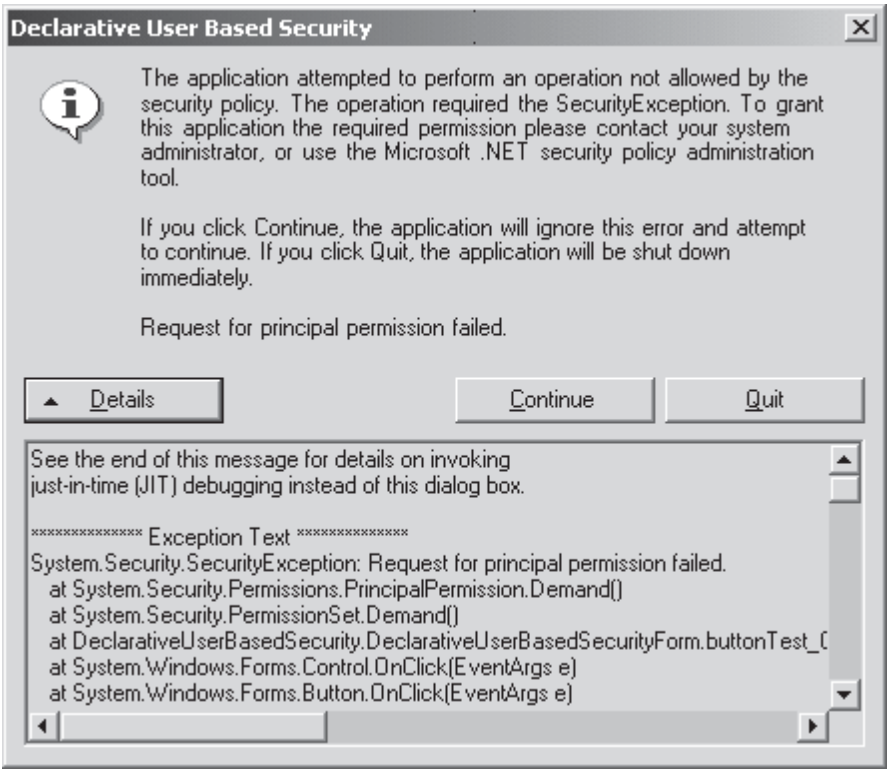


Рис. 7.12. PrincipalPermission: имя пользователя недействительно

```
c:\...>runas /user:HPDESKTOP\Administrator
   DeclarativeUserBasedSecurity.exe
c:\...>runas /user:HPDESKTOP\CodeMeister
   DeclarativeUserBasedSecurity.exe
```

При использовании декларативного подхода нам все равно требуется задать **WindowsPrincipale** в качестве объекта-принципала по умолчанию точно так же, как мы делали это при реализации императивного подхода. Однако, поскольку атрибут **PrincipalPermission** применяется к методу, как к целому, невозможно вызвать **SetPrincipalPolicy** внутри метода. По

этой причине программа **DeclarativeUserBasedSecurity** выполняет эту задачу заранее, при помощи кода, который приведен ниже (он помещается в методе **Main**). Заметим, что в данном случае нам не требуется получать объект-принципал из класса **Thread**, это делается автоматически, благодаря эффекту использования атрибута **PrincipalPermission**.

```
static void Main()
{
        // принципал по умолчанию для нитей домена приложения
        AppDomain appdomain = AppDomain.CurrentDomain;
        appdomain.SetPrincipalPolicy(
                PrincipalPolicy.WindowsPrincipal);

        Application.Run(new DeclarativeUserBasedSecurityForm());
}
```

Мандаты

Вспомним, что в концепции безопасности, основанной на идентификации пользователей, центральный вопрос заключается в установлении *личности* пользователя. *Мандатами* называют все то, что удостоверяет личность пользователя. Мандатом может быть пароль, электронная карточка или биометрическое устройство. Мандаты проверяются некоторой системой авторизации такой, как Windows или ASP.NET.

В некоторых случаях бывает необходимо предоставить неограниченный доступ к определенным ресурсам. Это называют анонимным доступом – такой тип доступа часто используется, например, для публичных ресурсов ASP.NET.

В концепции безопасности, основанной на идентификации пользователей, решается вопрос, разрешено или запрещено некоторому пользователю выполнить некоторое действие. Выполняется сверка со списком прав, чтобы выяснить, разрешен ли доступ. Например, на уровне файловой системы, когда вы обращаетесь к файлу, ваше регистрационное имя проверяется по списку ACL на предмет запрошенной операции, чтобы определить, разрешен ли вам запрошенный доступ к файлу.

В многопользовательской архитектуре зачастую возникает ситуация, когда пользователь, от имени которого выполняется серверное приложение, обладает очень широкими правами. При этом оказывается необходимым *ограничить права* серверного приложения, выполняющего запрос некоторого клиента, до уровня прав этого пользователя. В этой ситуации сервер может имитировать клиента, сокращая объем прав доступа до безопасного уровня. В случае же анонимного доступа сервер даже не знает, кто является его клиентом. При этом есть смысл использовать специальную учетную запись, созданную специально для анонимных клиентов, в которой предусмотрен необходимый минимум прав доступа.

Сетевые мандаты

Мандаты можно получать от службы аутентификации по сети. Интерфейс **ICredentials**, определенный в пространстве имен **System.Net**, обладает методом **GetCredentials**, который и выполняет эту функцию. Метод **GetCredentials** получает первый параметр, содержащий URI со ссылкой на службу аутентификации в сети. Вторым параметром является строка с требуемым типом аутентификации. Метод **GetCredentials** возвращает экземпляр **NetworkCredential**, который содержит мандаты, связанные с заданным URI, и схему авторизации. Если мандатов нет, метод **GetCredentials** возвращает значение Null.

```
NetworkCredentials GetCredentials(
   Uri uri,
   string authType
);
```

Имеется только два класса, реализующих интерфейс **ICredentials**: **CredentialCache** и **NetworkCredential**. Класс **CredentialCache** обеспечивает хранение множества мандатов[1]. Класс **NetworkCredential** обеспечивает мандаты для схем аутентификации, основанных на пароле таких, как NTLM и Kerberos. Следующий фрагмент кода дает общее представление об этой технике.

```
NetworkCredential nc = new  NetworkCredential(
   "JoeUser", "MyPassword", "SomeDomain");
CredentialCache cc = new CredentialCache();
cc.Add(new Url("www.xyz.com"), "Basic", nc);
WebRequest wr = WebRequest.Create("www.xyz.com");
wr. Credentials = cc;
```

Дисциплина безопасности

Существует пара простых правил, которым необходимо следовать, разрабатывая программы с точки зрения безопасности. Первое из них является принципом минимума полномочий, который помогает избежать ненужных рисков. Второе правило состоит в том, что вопросы безопасности необходимо продумывать на самой ранней стадии разработки проекта.

[1] Вы можете использовать **CredentialCache.DefaultCredentials** для получения контекста текущей нити для мандатов, основанных на NTLM и Kerberos.

Принцип минимума полномочий

Многие программисты рекомендуют не запускать инструментальные среды разработки, как например, Visual Studio .NET, с административными правами. В ситуации, когда вам потребуется временно расширить полномочия, вы можете воспользоваться утилитой **runas**, позволяющей выполнить команду от имени администратора.

Принцип минимума полномочий диктует, что для решения любой практической задачи следует использовать лишь необходимый минимум прав. При разработке вы должны отдавать себе отчет в ограничениях прав доступа и их влиянии на функционирование программы. К счастью, вам не требуются права администратора для запуска Visual Studio .NET.

К несчастью, это создает множество неудобств в процессе разработки, так что вы, возможно, предпочтете использовать административные права на этапе разработки. Затем на стадии тестирования и отладки вам придется гораздо подробней имитировать реальные ситуации с точки зрения прав доступа с тем, чтобы выяснить необходимый минимум полномочий. Разумеется, на этапе эксплуатации приложения, оно должно запускаться с необходимым минимумом полномочий.

Раннее формулирование политики безопасности

Важно, чтобы вопросы безопасности были продуманы, поняты и воплощены в решениях разработчика на самой ранней стадии жизненного цикла проекта. Встроить функции безопасности в действующий проект может оказаться очень трудной задачей.

Итоги главы

В этой главе мы рассмотрели основы программирования функций безопасности, основывающихся на концепции идентификации пользователя на платформе .NET. Мы изучили основные классы .NET, имеющие отношение к данной теме, и рассмотрели несколько примеров программ, иллюстрирующих эту тему. Другая концепция безопасности, а именно технология CAS, будет темой следующей главы.

Глава 8

Доступ к коду в .NET

Традиционно модели обеспечения безопасности строились на концепции регистрации в системе и контроля над процессами. Сам по себе, такой подход недостаточно гибок для современного, компонентно-ориентированного и мобильного программного кода. Технология CAS (Code Access Security – безопасность доступа к коду) решает эту проблему, накладывая гибкую, компонентно-ориентированную модель обеспечения безопасности на традиционную модель, обеспечиваемую операционной системой.

В главе 7 мы познакомились с первым вариантом обеспечения безопасности средствами .NET, сосредоточив свое внимание на методиках, связанных с идентификацией пользователей. В этой главе мы продолжим изучение техники программирования .NET с точки зрения безопасности, сосредоточившись на этот раз на ее главной технологии, именуемой CAS. В отличие от подхода, основанного на идентификации пользователей, CAS позволяет накладывать ограничения в зависимости от определенных характеристик выполняемых сборок, а не в зависимости от текущего пользователя. Как мы увидим, загруженная сборка обладает несколькими характеристиками, в совокупности называемыми свидетельством безопасности (security evidence), которые вместе с политикой безопасности среда выполнения CLR может использовать для принятия решений о доступе в ваших программах. Мы также рассмотрим тему управления политиками безопасности и применим на практике несколько классов разрешений доступа, а также реализуем императивный и декларативный варианты CAS.

Необходимость в контроле доступа

В те давние времена, когда сеть Internet не была еще по-настоящему глобальной, администраторы и пользователи устанавливали программное обеспечение в определенные каталоги на настольных компьютерах, серверах и сетевых дисках. Как правило, организации стремились обеспечить должное владение вопросами безопасности со стороны своих системных администраторов. Потенциальные риски уменьшались при помощи издания корпоративных стандартов безопасности, процедур

аудита, составления планов аварийного восстановления и соответствующего обучения конечных пользователей. Но даже в такой, относительно закрытой и контролируемой среде, все равно существовали реальные угрозы. Для настольных компьютеров основную опасность представляли бутовые вирусы и вирусы в исполняемых файлах, а также «троянские кони». Эти вредоносные программы распространялись относительно медленно, при помощи переноса на дискетах или в изолированной локальной сети. Серверы использовались в изолированных конфигурациях «клиент-сервер» и были слабо подвержены вирусным атакам, хотя и для них риск не отсутствовал полностью. Например, озлобленный на руководство сотрудник с широкими правами доступа мог оставить на сервере «бомбу с часовым механизмом» – вредоносную программу, срабатывающую через заданное время.

В наши дни, когда чуть ли не каждый компьютер связан со всеми остальными компьютерами посредством сети Internet, угрозы предстают перед нами во множестве новых форм, включая загрузку по сети исполняемых файлов, удаленное выполнение программ, вложения в сообщения электронной почты и атаки, основанные на переполнении буфера. К несчастью Internet предоставляет новые возможности для распространения вредоносного кода. Во-первых, сама скорость распространения выросла благодаря широкополосным линиям связи и огромной взаимосвязанности узлов сети. Во-вторых, благодаря большому числу служб и протоколов, функционирующих в сети, в огромной степени выросло число потенциально уязвимых мест. В третьих, стала легкодоступной обширная информация по взлому, накапливаемая на хакерских сайтах и форумах, благодаря чему расширяется круг потенциальных злоумышленников. Давно прошли те времена, когда для изобретения кибер-атаки нужно было быть компьютерным гением. Увы, но теперь опасным злоумышленником может стать вполне посредственный человек!

Затраты против риска

Как далеко следует зайти в деле обеспечения безопасности? Разумеется, это вопрос чисто экономического свойства. Деньги и силы, потраченные вами на защиту, должны как-то соотноситься с ценностью данных, которые вы защищаете. Выражаясь точнее, с тем ущербом, который причинит вашей организации потеря или компрометация этих данных. В сценариях высокого риска руководство соотносит стоимость аварийного восстановления системы с потерями от временной невозможности функционирования организации. В самых крайних случаях речь может идти о балансе между потерями от полного уничтожения системы и расходами на компьютер, спрятанный в противоатомном бункере. Но, как правило, программист имеет дело с менее радикальными сценариями, в которых решается вопрос, позволить ли некоторой сборке кода чтение некоторой переменной среды или файлов в некотором каталоге. Тем не менее, даже в таких скромных сценариях, необходимо принимать во внимание соображения безопасности. Каким образом специальные меры безопасности (т. е. затраты) соотнести с ожидаемым риском?

Может показаться, что затраты на реализацию защиты в программе пренебрежимо малы, однако это ни в коем случае не так. Разработка программного обеспечения требует значительных дополнительных затрат и усилий на каждой стадии – в процессе проектирования, кодирования, тестирования и поддержки для каждой дополнительной функции, которую вы реализуете. При этом функции, связанные с обеспечением безопасности, конкурируют со многими другими полезными для приложения функциями, которые вы могли бы реализовать. В этом смысле функции безопасности представляют собой реальные и значительные затраты при том, что многие разработчики имеют тенденцию оснащать свои приложения множеством красивых, но бесполезных функций в ущерб критическим функциям безопасности. В прошлом разработчиков Microsoft, Oracle и других некоторые специалисты по безопасности[1] часто критиковали за чрезмерное увлечение «бантиками» и «маленькими удобствами для пользователей» в ущерб безопасности. По моему мнению теперь положение улучшается. Выбор за вами. Если вы хотите, чтобы ваш код был чем-то большим, чем красивая игрушка, вы не можете полностью сосредоточиться на «бантиках», игнорируя вопросы безопасности.

Диапазон рисков

К сожалению, существует огромный диапазон потенциальных рисков, которые необходимо оценить для вашей системы и ваших данных, включая вредоносный код, взлом паролей, «кражу пакетов» и атаки типа «отказ в обслуживании». Даже такие физические угрозы, как кража или уничтожение носителя, шпионаж и мошенничество, также необходимо учитывать. В самых экстремальных случаях вы должны учесть возможность природной катастрофы или нападения террористов. Хотя реальные угрозы могут быть весьма разнообразными, и все их следует принимать во внимание, мы в этой главе рассмотрим только угрозу со стороны вредоносного программного кода, поскольку это единственная угроза, эффективно противостоять которой можно средствами .NET.

[1] Некоторые из крупнейших специалистов придерживаются этой точки зрения потому, что «бантики» на рынке более востребованы, чем безопасность, и производители программного обеспечения уделяют вопросам безопасности недостаточно внимания. Некоторые эксперты даже считают, что крупнейшие производители рассматривают серьезные уязвимости в программах, скорее, как проблему связей с общественностью. Часть экспертов являются сторонниками «полной открытости», то есть такого положения дел, когда обнаруженные уязвимости делаются достоянием общественности. Полная открытость – это обоюдоострое оружие, поскольку с одной стороны она побуждает производителей как можно быстрее устранять уязвимости, но с другой стороны публикация сведений об уязвимостях помогает злонамеренным элементам в их недобром деле. Трудно сказать, сыграла ли в этом роль политика полной открытости, однако несомненно, что корпорация Microsoft теперь значительно ответственней относится к проблемам безопасности. Свидетельством тому является огромное внимание, уделяемое безопасности в платформе .NET. Одну из блестящих статей Брюса Шнайера (Brice Schneier) на тему противопоставления полезных функциональных возможностей и соображений безопасности можно прочесть по адресу *http://www.counterpane.com/crypto-gram-0202.html*.

Давайте рассмотрим основные варианты угрозы со стороны вредоносного кода. Атаки типа «перекрытие стека»[1] уже доказали, что представляют собой серьезную опасность, в особенности для серверов. Если вам интересно посмотреть, как работают атаки типа «перекрытие стека», изучите пример программы Win32ProjectBufferOverflow в приложении А. Клиенты отправляют запросы на сервер, и эти запросы, должным образом сконструированные, позволяют использовать особенности небрежно написанного кода на сервере. Обычный код C/C++ опасно подвержен переполнениям буферов и ошибкам типизации, которыми могут воспользоваться клиентские программы. Самым знаменитым таким случаем стал так называемый «червь Морриса»[2].

Совсем нетрудно нечаянно внести уязвимость в традиционный код на C или C++, и для того чтобы этого избежать, необходимо прилагать серьезные усилия. Код, написанный для выполнения в управляемой среде, такой как C# или Java[3], по определению гораздо более безопасен[4], и не

---

[1] Примером атаки типа «перекрытие стека», основанной на ошибке работы со стеком в службе индексирования IIS, является червь Code Red II Worm. Уязвимость заключалась в отсутствии проверки буфера в коде, обрабатывающем URL, в библиотеке расширения ISAPI сервера индексации **(Idq.dll)**. Неконтролируемый буфер позволял переписать стек вызовов специально подготовленным кодом так, что атакуемое приложение в результате выполняло вредоносный код. Отправляя специально сконструированную строку URL на сервер IIS (версии 4.0 и 5.0 без «заплатки») с запущенной службой индексирования, атакующая сторона получала возможность выполнить на сервере произвольный код. Опасность еще более усугублялась тем обстоятельством, что служба индексации работает от имени учетной записи System, что дает враждебному коду весьма широкие полномочия.

[2] Червь – это программа, которая автоматически размножает и распространяет саму себя на доступных машинах в сети. Первым и самым знаменитым примером такой программы стал червь Морриса, написанный аспирантом Корнуэллского университета. Червь был запущен 2 ноября 1988 года и быстро «заразил» собой около 10% всех машин в сети Internet и затруднил или сделал невозможной их работу. Червь использовал некоторые уязвимости в некоторых программах системы BSD UNIX, включая ошибку переполнения буфера в демоне finger. Имеются свидетельства того, что автор червя интересовался лишь разработкой самой концепции червя и не намеревался причинять тот огромный ущерб, который был нанесен в результате. В коде червя содержались функции, ограничивающие его распространение, однако в этих функциях были ошибки, мешавшие им срабатывать должным образом. Как видите, даже лучшие умы ошибаются, что должно несколько утешить нас, простых смертных!

[3] Язык Java в основном обеспечивает те же возможности с точки зрения безопасности, что и C#, включая контроль типов и проверку границ. Среда выполнения Java, именуемая Java Virtual Machine (JVM) – виртуальная Java-машина – поддерживает в основном те же функции, что и среда CLR на платформе .NET, включая верификацию кода и контролируемое его выполнение. Библиотека классов Java также поддерживает набор классов, относящихся к системе безопасности, сходный по своей концепции с классами среды .NET Framework.

[4] Можно возразить, что избежать переполнения буфера в программах на C/C++ можно, не используя некоторые «рискованные» функции API, и что некоторые компиляторы C/C++ могут генерировать код, проверяющий границу стека при каждом вызове функции. Тем не менее, управляемая среда выполнения такая, как .NET или Java, обеспечивает подобную защиту автоматически, не требуя от программиста героической бдительности.

требует специальной бдительности со стороны программиста. Причина состоит в том, что управляемые языки генерируют код, который автоматически контролирует инициализацию данных, при этом переполнение буфера обнаруживается и предотвращается в период выполнения, а небезопасные преобразования типов исключаются на этапе компиляции. Конечно, для создания практически всего программного обеспечения, функционирующего ныне в сети Internet, был использован язык C. Чем больше серверного кода будет создаваться при помощи управляемых языков (C#, VB.NET, Java) в будущем, тем лучше будет становиться защита от атак существующих ныне типов.

Internet-мобильный код – например, вложения в сообщения электронной почты[1], сценарии или элементы управления ActiveX, – представляет собой основной фактор риска на стороне клиента. Эти угрозы выступают в разных формах, включая «троянские кони»[2], «логические бомбы»[3], традиционные вирусы[4] и даже просто старые добрые ошибки. К счастью, используя в разработке своих приложений технологию CAS, вы защититесь также и от этих угроз. Управляемый код помогает повысить надежность и безопасность также и клиентских приложений.

---

[1] Червь Nimda первоначально инфицирует вложения электронной почты, однако может также атаковать IIS (через «потайной ход», оставленный червем Code Red II) или любые незащищенные разделяемые ресурсы, которые сможет обнаружить. Nimda представляет собой сообщение электронной почты в формате HTML с вложенным выполняемым кодом. Благодаря хитрости, Internet Explorer выполняет вложенный выполняемый код при просмотре HTML. К сожалению, даже простой просмотр собственно письма инфицирует машину, даже если пользователь не открывал вложенный файл явным образом. С зараженной машины червь распространяется рассылкой копий самого себя по электронной почте. Для него уязвимы версии IE 5.01 и 5.5, IIS 4.0 и 5.0 без «заплаток».

[2] Троянским конем или «трояном» называют программу, которая объявлена полезной или нужной, однако на практике выполняет дополнительные, скрытые и, возможно, вредоносные действия. В отличие от вирусов и червей, «трояны», как правило, не воспроизводят сами себя. Известно много примеров «троянских коней», однако самым интересным примером, вероятно, является программа, описанная Кеном Томпсоном (Ken Thompson, «отец» системы UNIX) в его статье «Reflection on Trusting Trust» (http://www.acm.org/classics/sep95). Он описывает, как создать компилятор C, который установит встроенный «потайной ход» в систему UNIX при ее компиляции. Самое поразительное, он делает это таким образом, что нет никаких следов «трояна» в исходном коде как компилятора C, так и системы UNIX. Это означает, что вы не можете быть полностью уверены в программном обеспечении, даже если имеете его исходный код. Пугающая перспектива, не правда ли?

[3] Логическая бомба – это тайно разработанная программа, срабатывающая по наступлении заданного времени и либо наносящая серьезный ущерб системе, либо незаметно открывающая «потайной ход» для доступа в систему. Один из наиболее известных таких случаев связан с сотрудником оборонной компании в Нью-Джерси, который в качестве мести компании разработал программу, удалившую множество важных данных компании после его ухода.

[4] Вирус – это фрагмент кода, вставляющий себя в другие программы и изменяющих их таким образом, чтобы размножаться при их выполнении. Некоторые вирусы инфицируют обычные исполняемые файлы. Другие инфицируют системные сектора на диске, например, загрузочную запись. Примерами известных вирусов являются Brain, Stoned и Michelangelo.

Степень доверия к сборке

Фундаментальный вопрос, решаемый CAS, заключается в следующем: какому коду можно доверять и до какой степени? Проблема состоит в том, что код может происходить из многих источников, представляя собой различные степени риска. Этот вопрос возникает везде, где имеется конкретный фрагмент кода и существует возможность того, что он по злому умыслу или ненамеренно может причинить вред системе или данным, или привести к утечке конфиденциальной информации. Например, вы можете доверить коду, который написали сами, такие права доступа, которые не доверили бы коду, написанному другим разработчиком. Или вы можете разрешить некоторой сборке выполнение определенных операций, только если эта сборка создана определенной компанией. Вероятно, вам потребуются разные степени доверия, в зависимости от физического происхождения кода, например, для сборок, установленных локально администратором, и сборок, установленных простым пользователем или же загруженным автоматически через Web.

Проблема еще более усложняется, если речь идет о приложении, состоящем из сочетания сборок, которые попадают в разные категории с точки зрения доверия. При совместной работе многих сборок в составе одного приложения может случиться, что более доверенный компонент будет «обманут» менее доверенным компонентом и выполнит злонамеренные действия[1].

Технология CAS позволяет делать оценку подобных рисков на основе многих факторов, связанных со степенями доверия к конкретным сборкам. CAS также позволяет более тонко «настроить» уровень доверия к коду, чем это возможно в традиционных системах. Например, вы можете выбрать степень доверия на уровне сборки, на уровне класса и даже на уровне отдельного метода.

Необходимость в CAS становится особенно ясной, если учесть, что код может использоваться для выполнения множества разных задач, и пользователю, администратору и даже программисту не всегда очевидно, какие именно операции может попытаться выполнить конкретная сборка. Совершенно ясно, что модель безопасности, основанная на идентификации пользователей и описанная в предыдущей главе, не в состоянии решать эти новые проблемы. Тем более в эпоху мобильного кода и дистанционно выполняющихся методов и Web-служб.

Риски, связанные с обращением к традиционному коду

Важно заметить, что необходимое условие выполнения этих функций CAS состоит в том, что выполняемый код должен быть управляемым. Это означает, что среда выполнения CLR должна иметь возможность верифицировать

[1] Эту разновидность атаки называют «атакой с приманкой» (luring attack). Технология CAS отлично подходит для противостояния таким атакам.

безопасность типов в сборке при ее загрузке в память. Использование PInvoke для обращения к традиционному коду библиотек Win32 представляет собой опасность, поскольку среда выполнения никак этот код контролировать не может. Очевидно, что разрешать использование PInvoke следует только для кода с высокой степенью доверия, при этом если вызываемая DLL использует Win32 Security API, или вы вызываете совершенно безобидную функцию Win32 API такую, как **GetSystemTime**, то такое обращение вы можете разрешить. Но вы должны постоянно иметь в виду, что, используя обращение к традиционному коду, вы открываете потенциальную «дыру» в системе безопасности. По этой причине для обращения к внешнему коду требуется специальное разрешение. Если у вас есть опыт программирования на C/C++, вы вероятно знаете, как легко совершить такие ошибки, – неинициализированная переменная, неверный указатель, выход индекса за границы массива или неверное преобразование типов, – как легко можно получить «утечку» памяти и как опасно пользоваться такими традиционными функциями как **strcopy**, **gets**, **strcat** и **sprintf**.

Необходимость обращения к традиционному коду, тем не менее, в обозримом будущем не исчезнет, поэтому важно так строить приложение, чтобы ограничить обращения к традиционному коду лишь несколькими полностью доверенными сборками. Затем вы можете при помощи настройки конфигурации или вызовом метода **Deny** запретить PInvoke в большей части своего кода и методом **Assert** разрешить PInvoke в тех методах, где это необходимо. Позднее мы еще поговорим о методах **Deny** и **Assert**. В следующем примере программы **PInvoke** мы увидим, как конфигурированием политики безопасности разрешать или запрещать обращение к традиционному коду.

Безопасность, управляемый код и среда CLR

На самом фундаментальном уровне безопасность всей платформы .NET основывается на двух вещах: безопасность управляемого (с контролем типов) кода и бдительность среды выполнения CLR. Управляемый код с контролем типов гарантирует, что для всех типов данных, включая массивы и строки, выполняется проверка границ. Также он не допускает опасных преобразований типа и прямых манипуляций с данными в памяти через указатели. Практически для управляемого код с контролем типов доступны только области памяти, которые для него являются «законными», и доступны они только «законными» способами. Например, управляемый код с контролем типов не может получить доступ или изменить область памяти, принадлежащую к полям объекта или исполняемому коду класса. Доступ к полям и методам для него возможен только нормальным, «штатным» способом.

Результатом всего этого является такое положение дел, что управляемый код с контролем типов эффективно предотвращает ситуации типа «переполнение буфера» и вставку произвольного кода. Слово «эффективно» означает здесь, что заставить управляемую программу выполнить посторонний код крайне трудно. Однако, говоря о вопросах безопасности, никогда не следует говорить «никогда», поскольку контроль типов практически исключает возможность вставки вирусного кода или атаку типа «перекрытие стека». Но даже если злоумышленнику удастся каким-то образом вставить свой код в загруженную сборку, вступит в действие защита, обеспечиваемая средой выполнения, которая не позволит сборке выполнить действия, не предусмотренные политикой безопасности, заданной для данной сборки. И, наконец, последним рубежом обороны выступит цифровая подпись сборки, благодаря которой программы, использующие данную сборку, могут удостовериться в целостности ее содержимого. Кроме того, цифровые подписи и сертификаты удостоверяют источник происхождения сборки, что усиливает вашу позицию в случае судебной тяжбы. Благодаря этому уволенный и затаивший злобу сотрудник должен будет дважды подумать, прежде чем оставить после себя «логическую бомбу»!

Промежуточный язык Microsoft

Управляемый код .NET первоначально компилируется в язык промежуточного уровня, именуемый MSIL (Microsoft Intermediate Language – Промежуточный язык Microsoft). Для краткости этот язык иногда называют IL. Язык MSIL содержит в себе управляемый код совместно с метаданными, описывающими содержимое и атрибуты сборок. Это означает, что каждая сборка, существующая обычно в форме EXE- или DLL-файла, является самодостаточным объектом, не требующим никаких дополнительных данных для помещения в системный реестр. Код MSIL затем компилируется в исполняемый код уже в период выполнения, «на лету», компилятором JIT.

Компилятор JIT выполняет верификацию кода[1], проверяя безопасность преобразования типов и пытаясь обнаружить в коде запрещенные операции такие, как обращение к памяти напрямую. Верификация выполняется путем проверки кода на промежуточном языке и метаданных на предмет рискованных операций. Это означает, что даже в таких случаях, когда для генерации кода использовался специальный («хакерский») компилятор, или когда посторонний фрагмент кода был неким образом вставлен в сборку до наложения электронной подписи, опасный код все равно будет обнаружен и блокирован уже в период выполнения[2]. Управляемый код с контролем типов безопасен потому, что он не допускает прямого обращения к памяти или вызова традиционного кода, если при его создании не были предусмотрены соответствующие разрешения.

---

[1] Этап верификации может быть пропущен, если код имеет разрешение SkipVerification.

[2] По причинам, связанным с техническими ограничениями и соображениями производительности, нет твердой гарантии, что любое опасное преобразование типов будет обнаружено компилятором JIT во время компиляции (то есть для приложения в период выполнения). Все же, компилятор JIT прилагает колоссальные усилия для обнаружения наиболее типичных «нарушителей».

Верифицируемый код с контролем типов

Политика безопасности, которую мы более подробно рассмотрим позднее, может применяться только к верифицируемому коду с контролем типов. Важно помнить, что не все языки .NET генерируют только управляемый код с контролем типов, и что не любой управляемый код с контролем типов может быть верифицируемым. Visual Basic .NET всегда генерирует верифицируемый управляемый код с контролем типов. Visual C# генерирует верифицируемый управляемый код с контролем типов, если вы не использовали директиву **unsafe**. Код C#, сгенерированный с директивой **unsafe**, удобен для использования, но потенциально опасен, если вы передаете массивы и указатели в качестве параметров традиционному неуправляемому коду C/C++ посредством PInvoke. Компилятор Visual C++ может генерировать управляемый код с контролем или без контроля типов, и даже неуправляемый традиционный код, однако, он не может генерировать код, являющийся верифицируемым с точки зрения компилятора JIT. Что же до всех прочих языков, рекомендуем ознакомиться с соответствующей документацией, чтобы выяснить, при каких условиях в них генерируется верифицируемый управляемый код с контролем типов.

При использовании таких языков, как C#, которые генерируют верифицируемый управляемый код с контролем типов только при определенных условиях, вы можете воспользоваться утилитой **PEVerify.exe**, которой можно проверить, является ли некий код верифицируемым. Сборкам, не являющимся верифицируемыми, разрешается выполнение, только если к ним имеется полное доверие. Таковыми, согласно политике безопасности по умолчанию, являются сборки, созданные локально. Сборками «с полным доверием» можно дать разрешение **SkipVerification**, позволяющее пропустить этап верификации, который необходим для управляемого кода C/C++ и для кода C#, скомпилированного с директивой unsafe.

Запросы разрешений

Еще одна возможность с точки зрения обеспечения безопасности состоит в том, что код может запрашивать разрешения, которые ему необходимы для решения своих задач. В этом случае среда выполнения CLR принимает решение на основе текущей политики безопасности и выдает разрешение или отказывает в нем. Благодаря этому сборка может заранее выяснить, имеются ли у нее все необходимые разрешения, вместо того чтобы натолкнуться на неожиданное препятствие уже в процессе решения своих задач. Это гораздо более корректный способ выйти из ситуации недостаточности полномочий, поскольку сгенерировать исключение можно еще до выполнения части действий, рискуя внезапно натолкнуться на препятствие – при этом может быть трудно чисто и корректно «откатить» назад неоконченные операции. Мы познакомимся с императивным запросом разрешения, реализуемым методом **Demand**, а также с запросом разрешения декларативным, реализуемым при помощи атрибута **PermissionAttribute** и перечисления **SecurityAction**, что применимо к сборкам, классам и методам.

Также, код может специфицировать для среды CLR те разрешения, в которых нет нужды, и от которых он явным образом отказывается. Такое поведение может быть эффективной оборонительной тактикой, минимизирующей риски благодаря тому обстоятельству, что данный код впоследствии не удастся вовлечь в выполнение злонамеренных действий. Мы увидим, как разрешения задаются императивно при помощи метода Deny, и как они задаются декларативно, через посредство атрибута PermissionAttribute и перечисления SecurityAction.

Использование CAS

Технология CAS обеспечивает весьма полезную модель безопасности, совершенно отличную от (но при этом с ней совместимую) традиционной модели безопасности, основанной на идентификации пользователей[1]. Давайте рассмотрим способы, которыми CAS можно использовать для решения проблем безопасности, которые невозможно решить при помощи традиционных подходов.

Гибкий подход к обеспечению безопасности

Технология CAS обеспечивает гибкую надстройку над традиционной системой обеспечения безопасности, которая позволяет определять различные степени доверия. В чем-то это похоже на концепцию зон безопасности в Internet Explorer, но отличается более широкими возможностями в плане конфигурирования и расширения. Технология CAS минимизирует количество кода, который нуждается в полном доверии, и принимает во внимание тот факт, что вопрос доверия вовсе не обязательно решается в стиле «полностью доверять или полностью не доверять».

Как описывалось в предыдущих главах, система обеспечения безопасности, основанная на идентификации пользователей, хорошо подходит для ситуаций, в которых у вас имеется обозримое множество пользователей и групп, с которыми вы в состоянии «разобраться» с точки зрения вопросов доверия-недоверия. Но что, если числу пользователей нет никакого предела? Например, вы создаете публичную Web-службу с обеспечением некоторых аспектов безопасности. Другой вариант – вы создаете программную библиотеку, к которой могут обращаться сотни программ, написанных различными программистами, и функции из которой будут выполняться неведомо кем и неведомо где. Здесь бессмысленно создавать пользовательскую учетную запись или роль и назначать права доступа для произвольного, ничем не ограниченного числа пользователей. Традиционное решение состоит в использовании специальной анонимной учетной записи для представления неизвестных пользователей. Затем Web-сервер для

[1] В отличие от CAS, традиционная модель безопасности в литературе не обозначается какой-либо специальной аббревиатурой.

обслуживания поступившего запроса выступает от имени этой учетной записи. Во многих случаях такого решения оказывается достаточно, но зачастую требуется более сложное управление правами доступа для анонимных пользователей, которые должны меняться в зависимости от характера исполняемого кода. В подобных ситуациях CAS также может обеспечить большую гибкость решений.

Атака «с приманкой» и проход по стеку

Традиционная концепция безопасности также может подвести в ситуациях, когда злоумышленник «заманивает» ничего не подозревающего пользователя, побуждая его запустить на выполнение враждебный код. Если у вас есть дети-подростки, то вероятно вам знакома необходимость некоторых ограничений, вызванная скорее вашим недоверием к его окружению, чем недоверием к самому ребенку. Сходным образом при выполнении доверенного кода, который взаимодействует с внешним кодом неизвестного происхождения, вам для уменьшения риска потребуется контролировать полномочия вашего доверенного кода. Всегда следует ограничивать права доступа минимально необходимым набором, который достаточен для решения штатных задач.

Поскольку менее доверенная сборка обладает меньшим набором разрешений, она, вероятно, не сможет причинить много вреда сама по себе. Более доверенному коду обеспечиваются более широкие полномочия, поэтому он должен каким-то образом проверять, кто его вызывает, прежде чем выполнить какие-либо действия, представляющие потенциальную опасность. Среда CLR снабжена отличным встроенным механизмом, именуемым *«проход по стеку»* (stack walking), который обеспечивает такую возможность. Каким бы образом вы ни вызвали метод, который требует наличия определенных разрешений, среда выполнения CLR выполнит проход по стеку, проверяя вначале тот код, который непосредственно вызвал метод, и, продолжая затем проверки, восходя по стеку до тех пор, пока не убедится, что весь код, непосредственно или опосредованно имеющий отношение к вызову данного метода, обладает необходимыми разрешениями. Если политика безопасности не позволяет выполнить данное действие или у одного из вызывающих методов нет необходимого разрешения, будет сгенерировано исключение, а действие выполнено не будет.

Управление политиками безопасности при помощи групп кода

Вопросы безопасности становятся настоящим «вызовом», когда речь заходит о мобильном коде или о компонентах, поставляемых третьими сторонами. Например, вы вероятно захотите ограничить возможности компонентов-макросов и компонентов-сборок таким образом, чтобы они не могли получить доступ к чему-либо вне пределов того документа или

приложения, в котором они содержатся. Не исключено, что вы даже захотите защитить свою систему от ошибок в приложениях, полученных от доверенных поставщиков. Для решения подобных задач технология CAS предусматривает возможность назначить политику безопасности для сборок кода на основе правил членства сборок в группах кода. Это означает, что решения в отношении конкретных сборок, принадлежащих приложению, принимаются на основе того уровня доверия, который установлен для группы кода, к которой конкретная сборка относится.

Было бы трудно управлять политиками безопасности для каждой сборки индивидуально. Поскольку на любой заданной машине может быть установлено потенциально тысячи сборок, определить политику для каждой из них было бы трудной задачей. Вместо этого вы можете манипулировать группами кода, определяя критерии членства сборок в группе для целых категорий сборок сразу. О группе кода можно думать, как о подмножестве сборок, для которых определена одна и та же степень доверия. Таким образом, политика безопасности определяется, как набор разрешений для определенной группы кода, вместо того чтобы рассматривать каждую сборку индивидуально.

Основные концепции управления политиками безопасности

Имеется три уровня политики безопасности, доступных для конфигурирования: предприятие, машина и пользователь. Уровень предприятия – самый высокий, а уровень пользователя, понятно, самый низкий. Существует также четвертый, самый низкий на деле уровень политики безопасности, именуемый доменом приложения[1] (AppDomain – Application Domain), но этот уровень доступен только программно, причем в пределах конкретного приложения, и сконфигурировать его административно нельзя. Каждый из уровней используется для определения политики безопасности, но нижние уровни не могу предоставлять более широкие разрешения, чем определено верхними уровнями. Каждый уровень политики безопасности содержит иерархию групп кода. Каждая такая группа специфицирует условия членства, согласно которым сборки могут принадлежать к группе. Странно, но условия членства можно определить таким образом, что сборка будет принадлежать нескольким группам кода.

[1] Домен приложения, часто обозначаемый, как AppDomain, обеспечивает контролируемую средой CLR изоляцию памяти внутри программы. По своей концепции, это напоминает процесс в операционной системе, где тоже безопасность обеспечивается благодаря изоляции, однако представляет собой много более легкую и масштабируемую реализацию данной концепции. Внутри одного процесса в операционной системе может существовать множество доменов приложения, и, как легко догадаться, в пределах домена может выполняться только верифицируемый, безопасный по типам код. Уровень домена приложения недоступен для конфигурирования, вместо этого он задается вызовом метода **AppDomain.SetAppDomainPolicy**, который действителен только при наличии разрешения **SecurityPermission**.

Каждая группа обладает именованным набором разрешений, с ней ассоциированных. Набор разрешений определяет те действия, которые среда CLR позволит выполнить членам этой группы. Каждое из разрешений в наборе точно задает некое действие программы или определенный доступ к ресурсу. Разрешения для сборок определяются на каждом из уровней политики безопасности, и результирующие разрешения получаются в результате пересечения разрешений на разных уровнях. В конечном счете разрешения определяются согласно членству в группах, которое, в свою очередь, основывается на свидетельствах машины и сборки. Следовательно, политику безопасности можно представить себе как некое соответствие между свидетельствами и набором разрешений. Управляя наборами разрешений и критериями членства в группах, администратор может в результате определить, какие разрешения, в конечном счете, получит каждая конкретная сборка. Для управления политикой безопасности платформа .NET предусматривает два средства:

- средство конфигурирования .NET Framework Configuration, Mscorcfg.msc;
- утилита политики безопасности CAS, Caspol.exe.

Средство конфигурирования .NET Framework Configuration можно использовать для администрирования политики безопасности на локальной машине, а также, с его помощью можно создавать распределяемые пакеты политики безопасности для использования в масштабах предприятия при помощи сервера SMS (System Management Server – сервер управления системой) и групповой политики Active Directory. Утилита командной строки **Caspol** менее удобна для использования, но позволяет создавать сценарии автоматизации для администрирования политики.

Использование средства конфигурирования .NET Framework Configuration

Это средство запускается, как оснастка консоли управления MMC. Его можно использовать для решения различных конфигурационных задач, но в этой главе мы сосредоточимся на конфигурировании политики безопасности .NET. Вы можете запустить это средство двойным щелчком на значке файла **Mscorcfg.msc**, или же, открыв панель управления, выбрать папку Администрирование и значок Microsoft .NET Framework Configuration.

В окне этой программы разверните узел Runtime Security Policy, как показано на рисунке 8.1. Как видите, здесь имеется три вложенных узла с именами Enterprise, Machine и User, отображающим те три конфигурируемых уровня политики безопасности, о которых мы говорили выше. В каждой такой ветви имеется три вложенных узла с именами Code Groups, Permission Sets и Policy Assemblies. Каждый из них, в свою очередь, содержит вложенные узлы, которые мы вскоре обсудим. Пока что нам достаточно понимать, что узел Code Groups содержит критерии для отнесения сборок к тем или иным группам, узел Permission Sets содержит именованные наборы разрешений, а узел Policy Assemblies содержит полностью доверенные сборки, реализующие классы, которые определяют объекты, представляющие условия членства в группах и разрешения доступа.

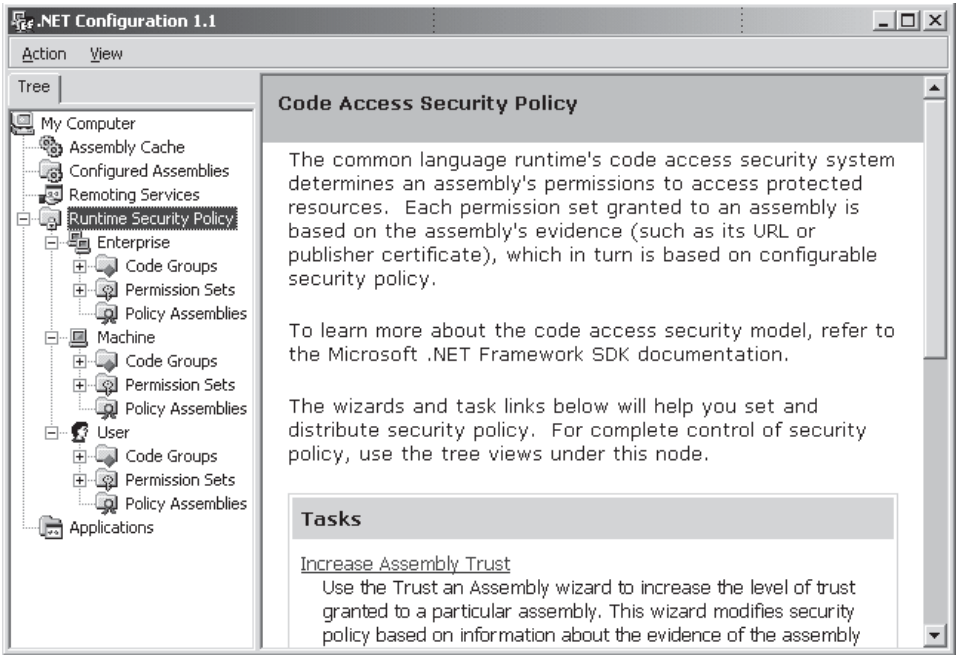


Рис. 8.1. Узел Runtime Security Policy

Фактически, сборки политик безопасности (policy assemblies) весьма специфичны. Эти сборки определяют разрешения и условия членства, используемые системой при определении прав доступа. Если вы создаете собственное, пользовательское разрешение или пользовательское условие членства, то вначале вы должны создать сборку, реализующую объект «security object» и все сборки, от которых он зависит. Затем система политик предоставит этим сборкам «полное доверие», для того чтобы использовать их в вычислении политик безопасности, определяемых пользовательскими объектами. Сборки политик должны обладать полным доверием (fully trusted), снабжены цифровой подписью и размещены в глобальном кеше сборок, прежде чем их можно будет внести в список сборок политик. Поскольку сборки политик являются святая святых всей системы безопасности в целом, ясно, что вы должны быть полностью уверены в их «добропорядочности», иначе вся ваша структура обеспечения безопасности будет скомпрометирована!

Верхнюю позицию в уровнях политики безопасности занимает уровень предприятия (Enterprise), который отвечает за политику безопасности для организации в целом. Второй уровень – это уровень конкретного компьютера (Machine), он определяет правила для любого кода, выполняемого на данной машине. Третий уровень, уровень пользователя (User), отвечает за политики, применяемые к текущему зарегистрированному пользователю. Назначение уровней политики безопасности описывается таблицей 8.1.

Таблица 8.1. Конфигурируемые уровни политики безопасности

Конфигурируемый уровень политики безопасности	Назначение
Enterprise (уровень предприятия)	Определяет политику безопасности для предприятия в целом
Machine (уровень компьютера)	Определяет политику безопасности для любого кода, выполняемого на конкретном компьютере
User (уровень пользователя)	Определяет политику безопасности для текущего зарегистрированного пользователя

В период выполнения, когда среда CLR загружает сборку, вначале она определяет, к какой группе (или группам) кода эта сборка принадлежит согласно свидетельствами, которыми она располагает, затем среда выполнения определяет соответствующую политику безопасности, основываясь на членстве сборки в группах кода. Политика безопасности «вычисляется» с учетом всех трех уровней, и результирующая политика получается пересечением множеств всех разрешений, имеющихся на трех уровнях. Иными словами, для получения некоторого разрешения требуется «единогласное голосование» всех трех уровней политики безопасности, и если на одном из уровней разрешение отсутствует, то итоговое разрешение не будет выдано.

Иерархия групп кода, определенных на уровне компьютера, изображена на рисунке 8.2. В каждом узле дерева групп вы можете редактировать критерии членства в группах, а также можете создавать дочерние группы кода (вложенные узлы дерева). Иерархическая структура позволяет задавать нестрогие критерии вблизи корня дерева и ужесточать их по мере углубления в ветви. Каждая группа кода обладает единственным набором разрешений, ассоциированным с ней, а также набором критериев, позволяющих относить сборки к этой группе.

Если вы выделите в дереве некоторую группу кода и щелкнете на гиперссылке Edit Code Group Properties (Редактировать свойства группы кода), то откроется диалоговое окно Code Group Properties (Свойства группы кода). При помощи этого диалога вы можете изменить имя группы, ее описание, условие членства в ней и набор разрешений, который она предоставляет сборкам. Все это для существующей группы My_Computer_Zone изображено на рисунках 8.3, 8.4 и 8.5.

Как мы уже знаем, каждая группа кода обладает единственным набором разрешений. Однако набор может включать в себя множество разрешений. Каждое такое разрешение представляет собой право на выполнение некоторого действия или на определенный вид доступа к некоторому ресурсу. Имеется несколько предустановленных наборов разрешений, но средство конфигурирования .NET Framework Configuration позволяет конструировать также и новые наборы.

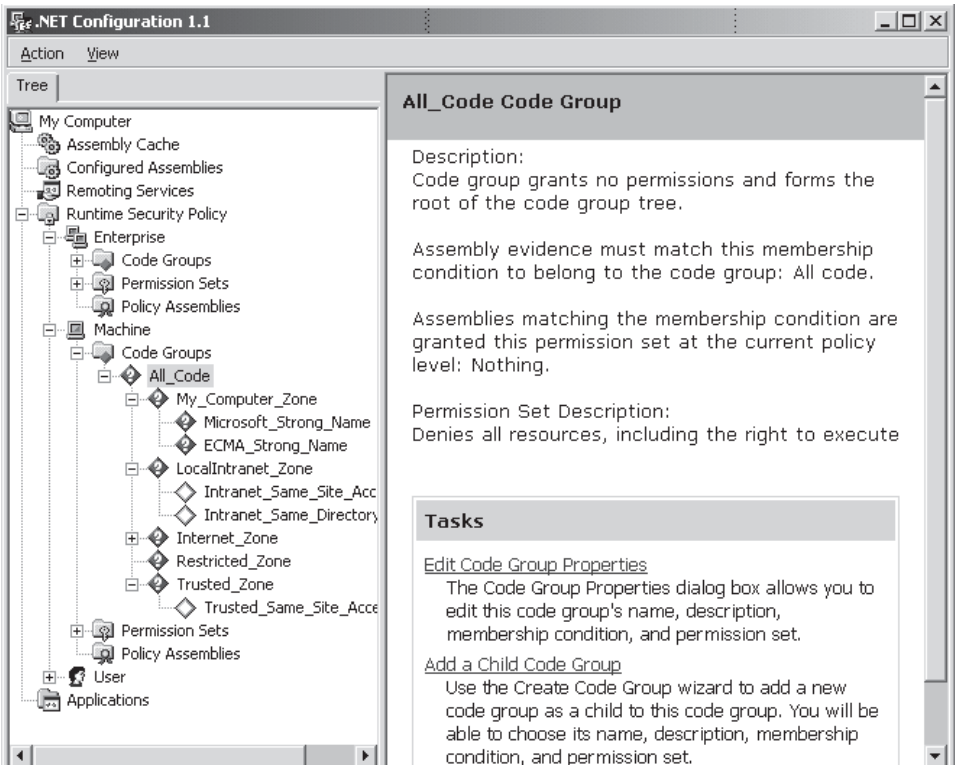


Рис. 8.2. Группы кода уровня Machnine

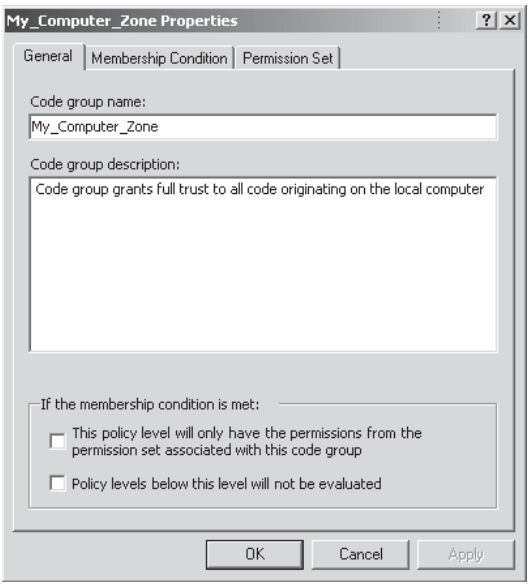


Рис. 8.3. Вкладка General диалога свойств группы кода

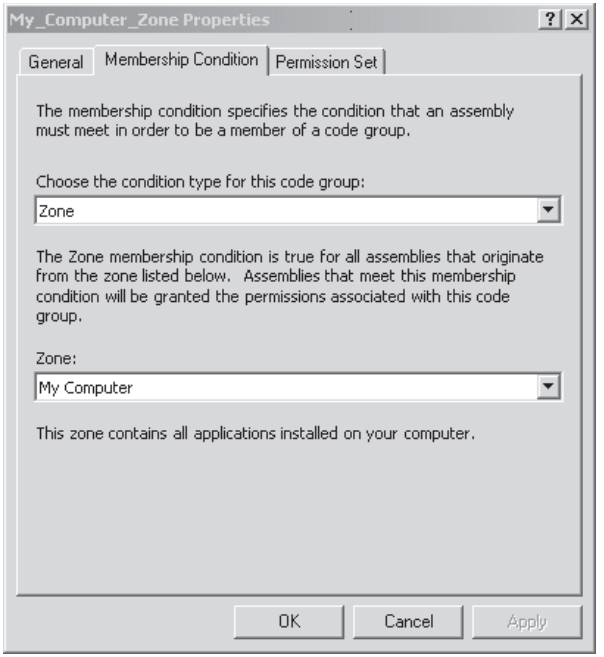


Рис. 8.4. Вкладка условий членства (Membership Condition) диалога свойств группы кода

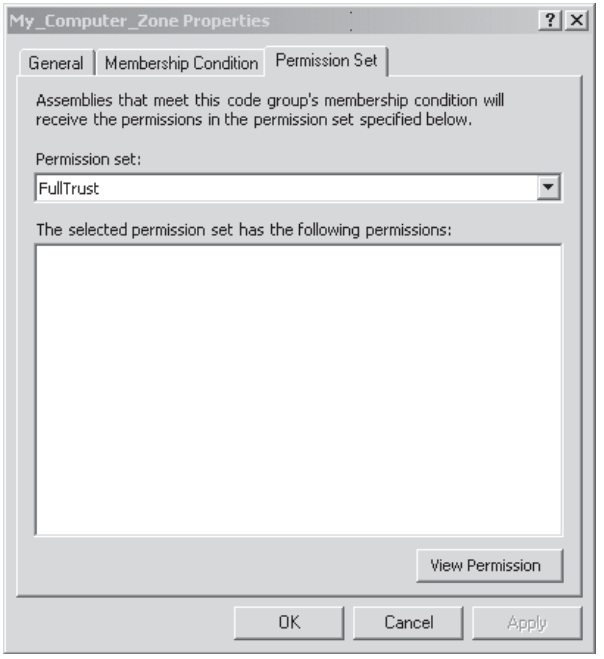


Рис. 8.5. Вкладка набора разрешений (Permission Set) диалога свойств группы кода

Имеется несколько предустановленных наборов разрешений, поставляемых в составе .NET Framework. Так же, как и разрешения на доступ к коду, условия членства в группах реализованы в виде классов, определенных в сборках политик. И подобно тому, как вы можете создавать собственные классы разрешений, возможно и создание классов условий членства в группах. В сущности, объект-условие членства в группе кода просто определяет, удовлетворяют ли свидетельства, которыми располагает предлагаемая сборка, критериям членства. Имена и описания предустановленных условий членства приведены в таблице 8.2.

Таблица 8.2. Критерии членства в группах кода

Группа кода	Свидетельство, используемое для принятия решения о членстве в группе
All code	Любой код вне зависимости от имеющихся свидетельств
Application directory	Каталог установки приложения
Hash	Хеш MD5 или SHA1
Publisher	Открытый ключ цифровой подписи (сертификат X.509)
Site	Имя сайта Web или FTP
Strong name	Цифровая подпись
URL	Полный создаваемый URL
Zone	Зона создания (MyComputer, LocalIntranet, Internet, Restricted или Trusted)

ПРИМЕР ПРОГРАММЫ PINVOKE: ОПРЕДЕЛЕНИЕ НОВОЙ ГРУППЫ КОДА

Имеется несколько предопределенных групп кода общего назначения таких, как My_Computer_Zone или LocalIntranet_Zone, с которым ассоциированы соответствующие наборы разрешений. Бывают ситуации, когда необходимо модифицировать существующую группу кода или создать новую. Давайте разберем по шагам типичный пример решения такой задачи при помощи средства конфигурации .NET Framework Configuration.

Пример **PInvoke** – это простая программа, которая пытается вызвать две API-функции Win32, а именно **GetComputerName** и **GetLastError**. Если вы запустите эту программу под стандартной политикой безопасности, то обнаружите, что она работает правильно, хотя и в ее выполнении имеется риск обращения к неконтролируемому коду. Поскольку программа выполняется локально, согласно стандартной политики безопасности, зона MyComputer обладает полным доверием, как видно из рисунка 8.6.

Вскоре мы увидим, как данное положение дел можно изменить таким образом, что программе PInvoke будет запрещено обращаться к неконтролируемому коду, но вначале давайте посмотрим на исходный код самой программы.

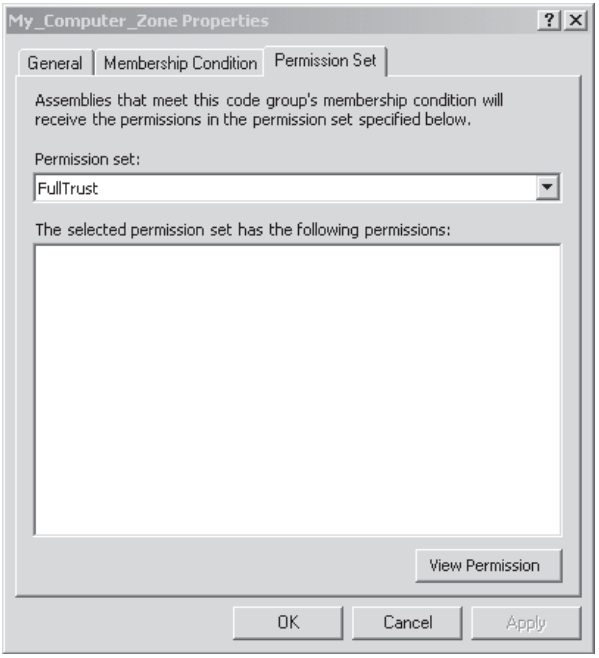


Рис. 8.6. Зона My_Computer_Zone обладает полным доверием (fully trusted)

Как легко увидеть, в ней нет ничего, имеющего отношение к системе обеспечения безопасности. На данном этапе мы хотим сосредоточиться на вопросах администрирования политик безопасности. Когда вы запустите программу в условиях стандартной политики безопасности, она просто сообщит вам имя компьютера.

```
//PInvoke.cs

using System;
using System.Text;
using System.Runtime.InteropServices;
using System.Security;

public class Test
{
   [DllImport("kernel32.dll", CharSet=CharSet.Ansi)]
   public static extern bool GetComputerName(
             StringBuilder name, out uint buffer);
   [DllImport("kernel32.dll")]
   public static extern uint GetLastError();
   public static int Main(string[] args)
   {
             bool result = true;
             uint error = 0;
```

```
            StringBuilder name = new StringBuilder(128);
            uint length = 128;
            Console.WriteLine(
                  "Пытаюсь вызвать GetComputerName");
            result = GetComputerName(name, out length);

            if (result == true)
                  Console.WriteLine(
                        "GetComputerName вернула: " +
                        name);
            else
            {
                  error = GetLastError();
                  Console.WriteLine(
                        "Ошибка! GetComputerName  вернула: " +
                        error);
            }

            Console.Write("Нажмите Enter для выхода...");
            Console.Read();
            return 0;
   }
}
```

Ниже приведен консольный вывод этой программы. Обратите внимание, мы еще не вносили никаких изменений в стандартную настройку политик безопасности. Понятно, что на вашей машине полученное имя будет другим.

```
Пытаюсь вызвать GetComputerName
GetComputerName вернула: HPDESKTOP
```

Разумеется, программа **PInvoke** не причиняет никакого вреда, иначе бы вы не захотели ее выполнять в качестве демонстрационного примера. Но давайте вообразим, что вы администратор и что исходного кода этой программы у вас нет. Поскольку вы не знаете, что может попытаться сделать эта программа, вы должны сконфигурировать политику безопасности таким образом, чтобы она не смогла причинить никакого вреда.

Для этого вы вначале определяете новую группу кода с критерием членства таким, чтобы группа включала в себя сборку **PInvoke.exe**. Запустите средство конфигурирования .NET Framework Configuration и раскройте узел Code Groups в ветви уровня Machine policy. С таким же успехом вы можете работать с уровнями Enterprise и User, но в данном примере мы работаем с уровнем Machine. Щелкните на узле All_Code (ветвь Code Groups), а затем щелкните на гиперссылке Add a Child Code Group[1]. В стартовавшем в результате мастере Create Code Group введите имя My_Own_Zone и описание новой зоны, как показано на рисунке 8.7.

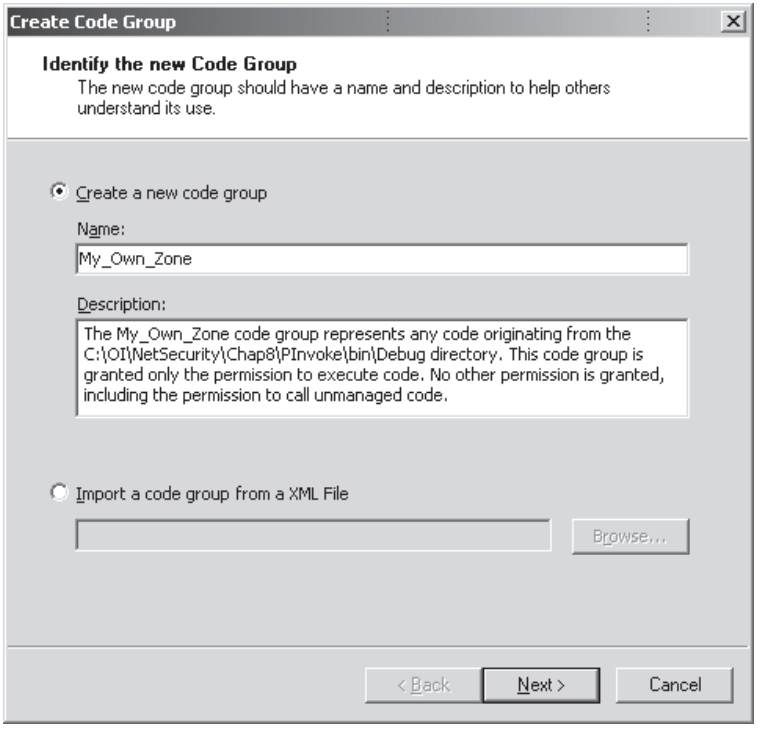


Рис. 8.7. Первый шаг создания группы My_Own_Zone

Щелкните на Next (Далее), для того чтобы перейти к следующему шагу мастера, и выберите условие членства. В поле со списком выберите вариант URL, а в поле URL введите **file://C:/OI/NetSecurity/Chap8/PInvoke/bin/Debug/***, как показано на рисунке 8.8. Благодаря этому любая сборка, помещенная в указанную папку, станет членом группы кода My_Own_Zone.

Щелкните на Next, для того чтобы перейти к третьему шагу мастера, где вы должны присвоить своей новой группе набор разрешений. В поле со списком Use existing permission set выберите разрешение Execution (Выполнение), как показано на рисунке 8.9.

Завершите работу мастера щелчками на Next и Finish (Готово). Новая группа кода теперь появится в окне .NET Framework Configuration, как показано на рисунке 8.10. Обратите внимание, что узел My_Own_Zone появился в иерархическом дереве, а его описание и условие членства отображаются в правой панели.

Щелкните теперь на гиперссылке Edit Code Group Properties (Редактировать свойства группы кода) и посмотрите на диалоговое окно свойств группы My_Own_Zone. На вкладке General установите флажок This policy..., который указывает, что данный уровень политики обладает лишь теми разрешениями, которые имеются в том наборе, и что он ассоциирован с данной группой (см. рисунок 8.11).

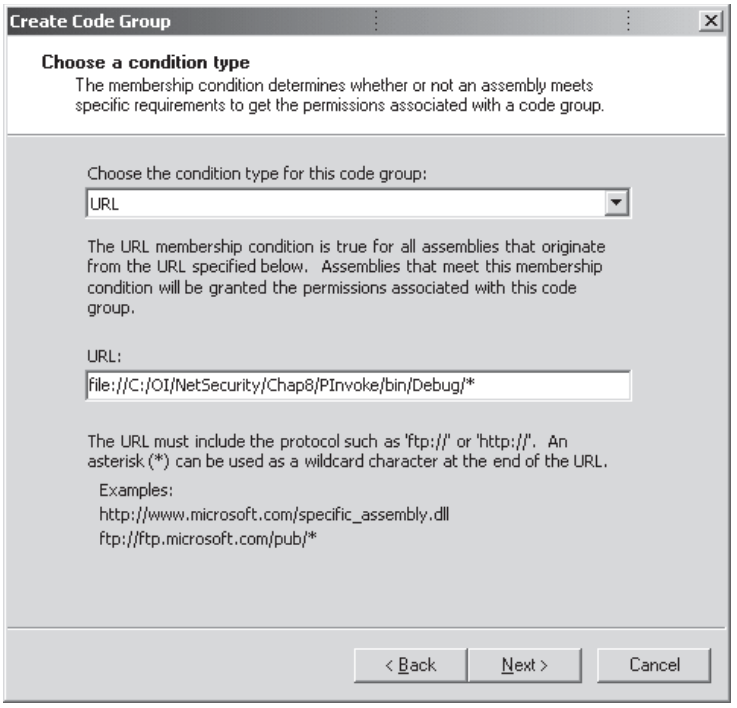


Рис. 8.8. Второй шаг создания группы My_Own_Zone

На вкладке Permission Set (Набор разрешений) вы увидите те разрешения, которые предоставляются набором Execute. Как видно из рисунка 8.12, набор Execute включает в себя единственное разрешение с именем Security.

Если вы теперь выделите в списке разрешение Security и щелкнете на кнопке View Permission (Просмотр разрешений), то в окне просмотра разрешений увидите более подробные сведения (см. рисунок 8.13).

Здесь видно, что это разрешение допускает выполнение кода (Enable Code Execution = Yes), но запрещает все остальные аспекты[1]. Особенное внимание обратите на тот факт, что здесь выключено разрешение на вызовы неконтролируемого кода (Allow Calls to Unmanaged Code = No).

Поскольку в наборе разрешений нет никаких других разрешений кроме Security, ясно, что права группы кода My_Own_Zone весьма ограничены. Закройте окно просмотра разрешений и щелкните на OK в диалоговом окне свойств группы, чтобы закрыть и его тоже. Теперь, когда вы завершили создание собственной группы кода, установили для нее критерий членства и задали набор разрешений, вы можете запустить программу PInvoke снова.

[1] Если вы посмотрите на документацию класса **SecurityPermission**, то увидите там все перечисленные в этом окне аспекты в форме свойств типа **Flag**. Определяется этот набор перечислением **SecurityPermissionFlag**.

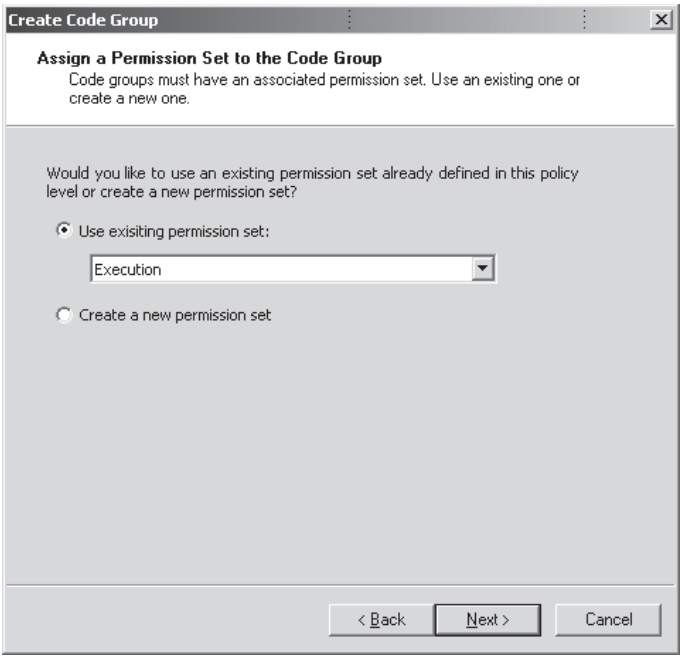


Рис. 8.9. Третий шаг создания группы My_Own_Zone

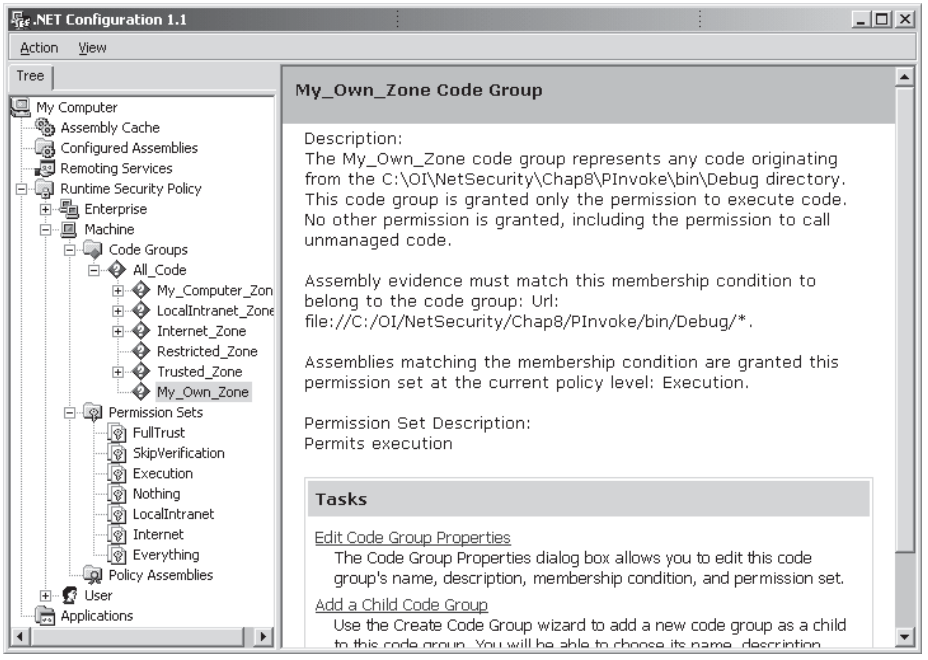


Рис. 8.10. Группа My_Own_Zone создана

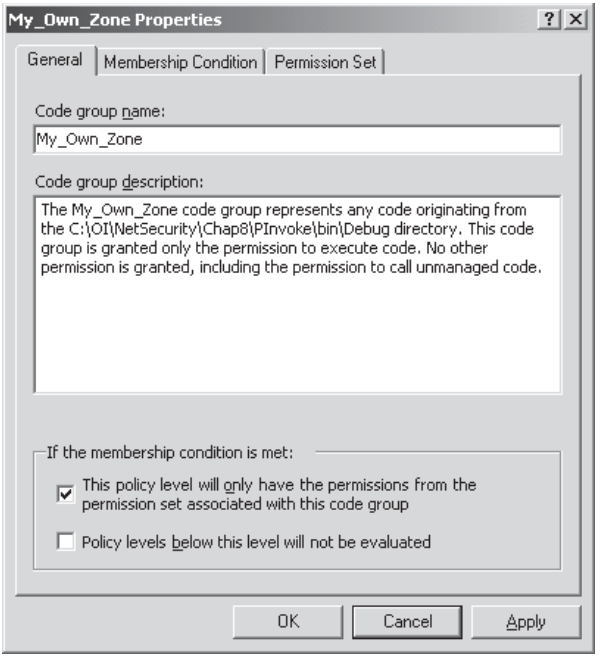


Рис. 8.11. Вкладка General диалога свойств группы My_Own_Zone

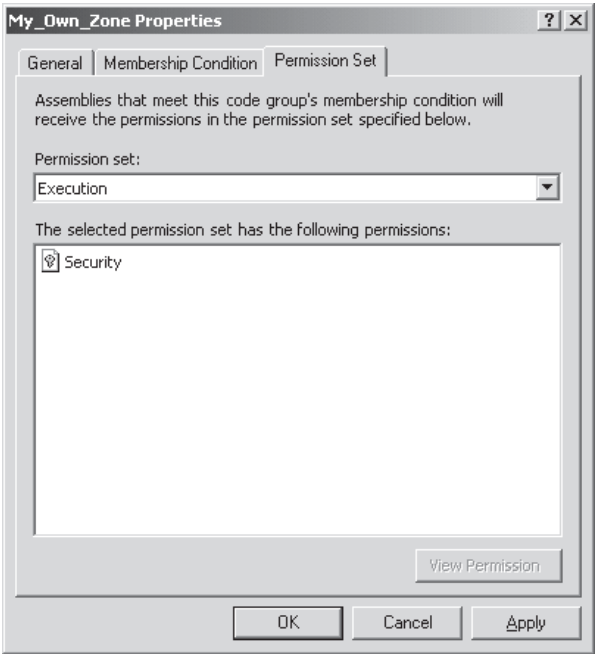


Рис. 8.12. Вкладка Permission Set диалога свойств группы My_Own_Zone

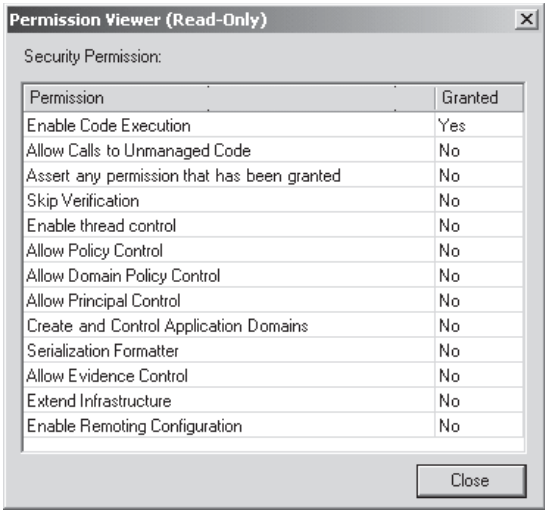


Рис. 8.13. Просмотр разрешений группы My_Own_Zone

Ранее программа просто вывела имя машины. Но теперь она сгенерирует исключение, как видно из ее консольного вывода, который приведен ниже. Бессмысленными будут попытки перехватить это исключение в методе **Main** этой программы, поскольку оно фактически генерируется при попытке обращения к этому методу. Причина заключается в атрибуте **DllImport**, примененном к методу **Main**.

```
Unhandled Exception: System.Security.SecurityException:
   System.Security.Permissions.SecurityPermission
    at Test.Main(String[] args)

The state of the failed permission was:
<IPermission class=
   "System.Security.Permissions.SecurityPermission,
   mscorlib, Version=1.0.3300.0, Culture=neutral,
   PublicKeyToken=b77a5c561934e089"
              version="1"
              Flags="UnmanagedCode"/>
```

ОПРЕДЕЛЕНИЕ НОВОГО НАБОРА РАЗРЕШЕНИЙ

Мы только что создали новую группу кода и ассоциировали ее с набором разрешений. Однако мы использовали при этом существующий набор разрешений по имени Execution, который весьма сурово ограничивает права кода. Вспомните, набор разрешений Execution включает в себя единственное разрешение по имени Security, которое допускает всего лишь только выполнение кода. Что если нам потребуется набор, разрешающий выполнение кода и управление нитями процесса, но запрещающий все остальное? Возможно, вам потребуется определить новый набор разрешений, представляющий собой комбинацию из нескольких разрешений та-

ких, как Security, User Interface и разрешений файлового ввода/вывода. В таких ситуациях проблема решается созданием нового набора разрешений[1].

Для того чтобы определить новый набор разрешений, запустите средство конфигурирования .NET Framework Configuration и раскройте ветвь нужного уровня политики безопасности. В нашем примере мы выберем уровень Machine и раскроем затем узел Permission Sets. Щелкните на гиперссылке Create New Permission Set (Создать новый набор разрешений) в правой панели и после старта соответствующего мастера введите имя нового набора – MyPermissionSet и описание нового набора, как показано на рисунке 8.14.

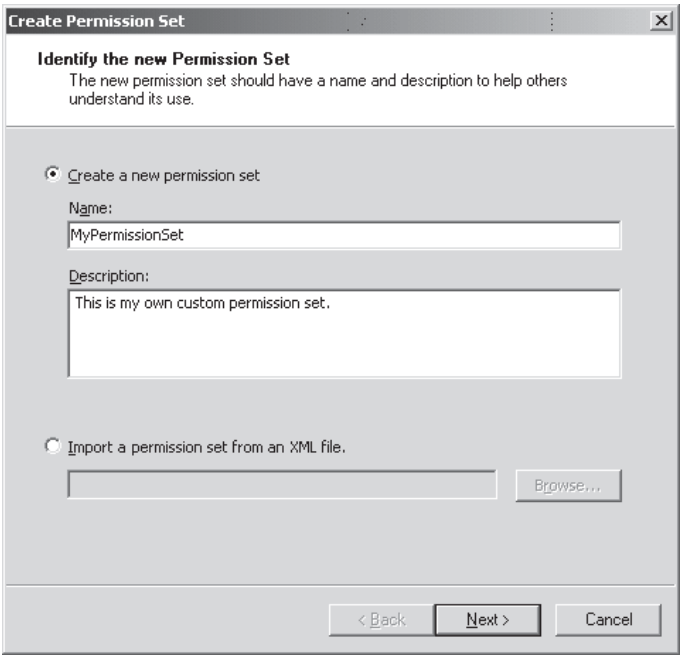


Рис. 8.14. Создание набора разрешений MyPermissionSet: первый шаг

Щелкните на Next для перехода к следующему шагу мастера, где вы должны сконструировать содержимое нового набора. Выбирайте эле-

[1] Фактически вы можете пойти еще на шаг дальше, программно определив целый класс разрешений, который управляет правами доступа, специфичными для вашего приложения. Решение такой задачи включает в себя несколько шагов, в том числе определение класса, производного от **CodeAccessPermission** в сборке с цифровой подписью, которую необходимо поместить в глобальный кеш и которая должна обладать полным доверием (fully trusted). Потребуется также XML-представление этого разрешения, а результирующая сборка затем добавляется в ветвь Policy Assemblies в средстве конфигурирования .NET Framework Configuration. После того как все это будет сделано, новое разрешение можно включить в набор и использовать при конфигурировании политики для нужной группы кода.

менты разрешений в левом списке и перемещайте их в правый список при помощи кнопки Add. На рисунке 8.15 это диалоговое окно изображено в состоянии, когда еще ни один элемент не выбран. Для каждого элемента, помещаемого вами в правый список, открывается диалоговое окно, вид которого зависит от характера разрешения. Здесь вы должны сделать детальные настройки добавляемого разрешения. На рисунке 8.16 приведен диалог для настройки разрешения на файловый ввод/вывод, а диалоговое окно для разрешения типа Security изображено на рисунке 8.17.

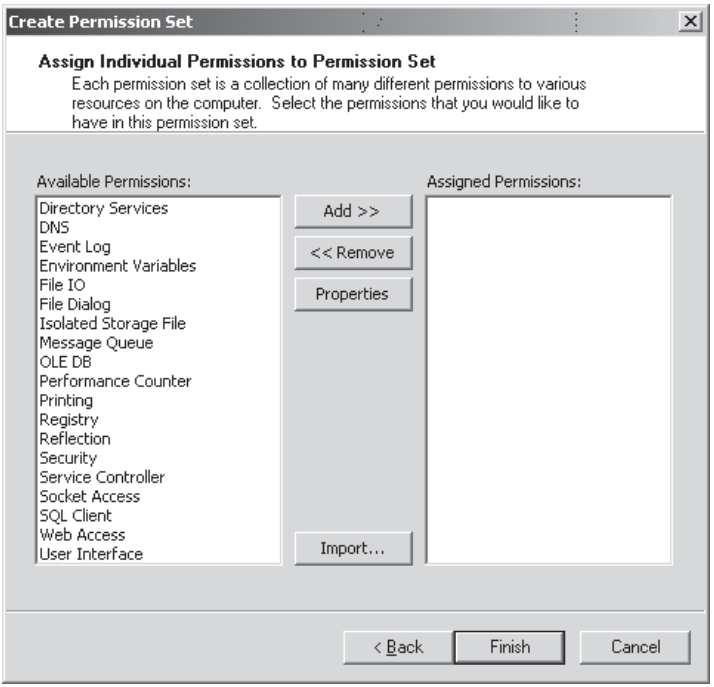


Рис. 8.15. Создание набора разрешений MyPermissionSet: второй шаг

Добавив в свой набор все нужные разрешения, щелкните на Finish, чтобы завершить работу мастера. Окончательный результат изображен на рисунке 8.18.

Теперь, когда набор разрешений создан, вы можете ассоциировать его с группой кода точно так же, как вы это уже проделывали.

Обратите внимание на тот факт, что вновь созданный набор разрешений теперь автоматически появится в тех окнах мастера создания групп кода и диалога редактирования свойств группы, с которыми мы уже знакомы.

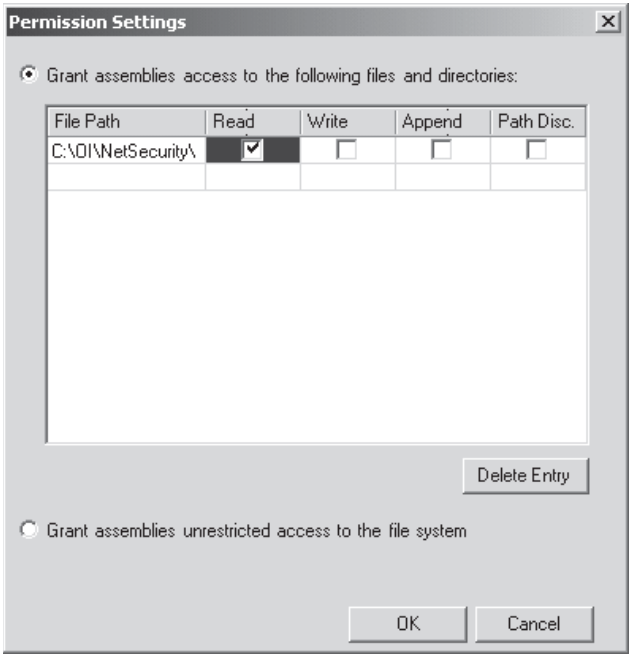


Рис. 8.16. Настройка разрешения файлового ввода/вывода

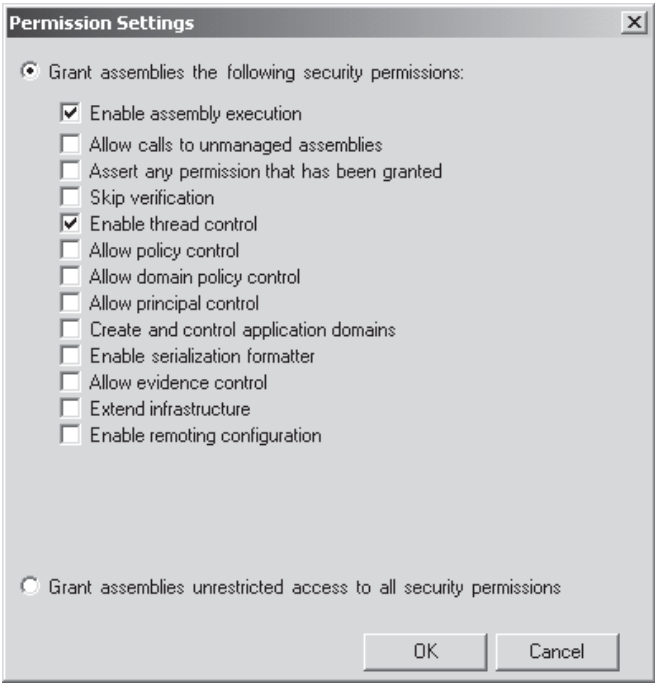


Рис. 8.17. Настройка разрешения типа Security

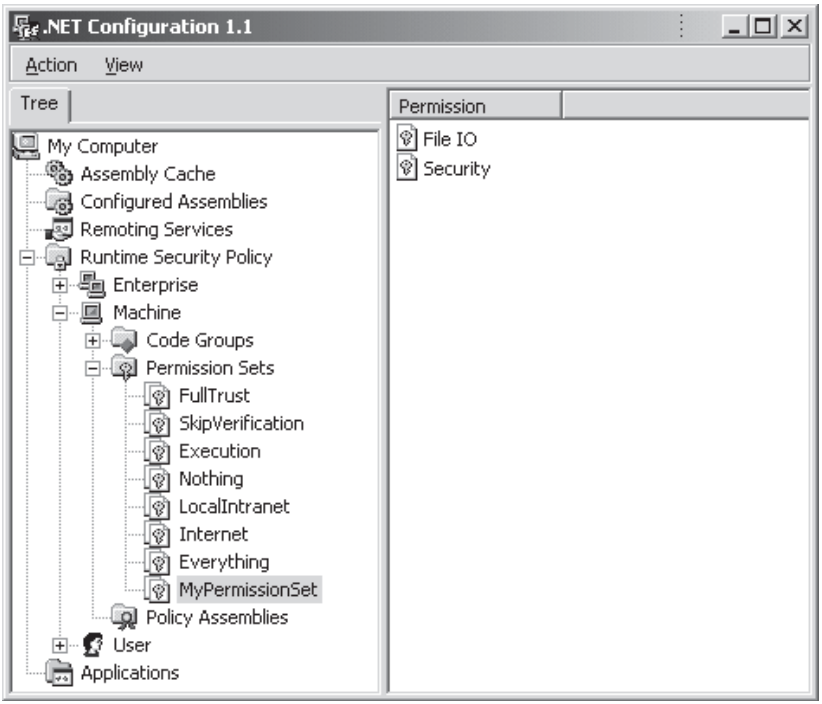


Рис. 8.18. Набор разрешений MyPermissionSet создан

Использование утилиты Caspol.exe

Для средства конфигурирования .NET Configuration существует альтернатива в виде утилиты командной строки **Caspol.exe**[1]. Обычно люди предпочитают использовать более удобный графический интерфейс, в особенности, если речь идет о первоначальном знакомстве с концепцией. Однако в сценариях и пакетных файлах, используемых при автоматизации процесса конфигурирования, более уместны утилиты командной строки.

Аргументы командной строки, поддерживаемые **Caspol**, многочисленны, и мы не будем рассматривать их здесь подробно. Вместо этого мы рассмотрим несколько примеров тех операций, которые можно выполнить при помощи этой утилиты. Примеры наиболее употребительных вариантов командной строки с использованием **Caspol** приведены в таблице 8.3. Сокращения для аргументов и операнды здесь не показаны, подробней об этом смотрите документацию по утилите.

[1] «Caspol» расшифровывается, как «code access security policy» («политика безопасности доступа»).

Таблица 8.3. Аргументы командной строки утилиты Caspol

Командная строка Caspol	Назначение
caspol –help	Отобразить справку по командной строке Caspol
caspol –machine	Последующие команды должны действовать на уровне компьютера
caspol –user	Последующие команды должны действовать на уровне пользователя
caspol –enterprise	Последующие команды должны действовать на уровне предприятия
caspol –addgroup ...	Добавить группу кода на ранее заданном уровне политики
caspol –remgroup ...	Удалить группу кода на ранее заданном уровне политики
caspol –listgroups	Вывести список групп кода
caspol –addpset ...	Добавить именованный набор разрешений на ранее заданном уровне политики
caspol –rempset ...	Удалить именованный набор разрешений на ранее заданном уровне политики
caspol –listpset	Вывести список разрешений
caspol –addfulltrust ...	Добавить сборку, обладающую полным доверием, на ранее заданном уровне политики
caspol –remfulltrust ...	Удалить сборку, обладающую полным доверием, на ранее заданном уровне политики
caspol –listfulltrust	Вывести список сборок с полным доверием
caspol –resolvegroup ...	Вывести список групп кода, к которым принадлежит сборка
caspol –security ...	Включить или выключить режим безопасности (on или off)
caspol –reset	Сбросить выбор уровня политики безопасности в стандартное состояние

Императивный и декларативный подходы в CAS

Как это было в случае обеспечения безопасности на основе идентификации пользователей, при реализации CAS тоже можно использовать императивный или декларативный подход. Как это было и в прошлой главе, мы увидим вскоре, что императивный подход можно реализовать с использованием двух слегка отличающихся стилей.

В первом варианте императивный подход подразумевает явное использование класса **Evidence**. Этот стиль называют иногда явным использованием свидетельств, поскольку вы при принятии программных решений должны явным образом определить свидетельства безопасности машины и сборки. Другой императивный стиль основывается на использовании различных классов разрешений, производных от класса **CodeAccessPermission**, которые автоматически генерируют исключение **SecurityException** в нужных случаях. Затем мы ознакомимся с реализацией CAS при помощи декларативного подхода, применяя атрибуты к методам, классам и сборкам.

Концепция безопасности, основанная на свидетельствах

Для того чтобы изучить императивный подход в реализации CAS и, в особенности, явное использование свидетельств, мы вначале рассмотрим сам класс этих объектов, то есть класс **Evidence**. Затем мы узнаем, как получить объект этого класса, относящийся к домену выполняющегося приложения. Наконец, мы проанализируем его содержимое с тем, чтобы принять программные решения на основе обнаруженных свидетельств.

Класс Evidence

Прежде чем перейти к примеру программы **ImperativeCASComponent**, нам необходимо хорошо понимать устройство класса **Evidence**, поскольку это важнейший элемент в программе. Класс **Evidence**, помещающийся в пространстве имен **System.Security.Policy**, инкапсулирует в себе информацию о свидетельствах, которую можно использовать для принятия программных решений.

Терминология: свидетельства и политика безопасности

Свидетельство – это набор характеристик с точки зрения безопасности, ассоциированных с выполняющимся кодом таких, как, например, цифровая подпись сборки, зона и сайт ее происхождения. Свидетельства используют, для того чтобы отнести сборку к той или иной группе кода, где полученные ею разрешения будут зависеть от действующей политики безопасности.

Политика безопасности – это набор правил, установленных администратором, которые задают разрешения для управляемого кода, определяя, какие операции может выполнить любая конкретная сборка.

По соображениям безопасности класс **Evidence** закрыт, то есть его нельзя использовать в качестве суперкласса для любых производных классов.

Представьте, какие были бы возможны хакерские трюки, если бы вы могли подменить класс **Evidence** на ваш производный класс!

Типы свидетельств, получаемых посредством объекта **Evidence**, могут включать в себя цифровую подпись, место происхождения или даже вашу собственную информацию о свидетельствах, которую вы сочтете полезной.

Класс **Evidence** реализует два интерфейса: **ICollection** и **IEnumerable**. Как мы вскоре увидим, интерфейс **ICollection** определяет элементы семейства объектов, а интерфейс **IEnumerable** обеспечивает доступ к этим объектам посредством интерфейса **IEnumerator**. В случае класса **Evidence** объекты семейства представляют отдельные части свидетельства машины и свидетельства сборки. Вот объявление класса **Evidence:**

```
Public sealed class Evidence :
   ICollection, IEnumerable
```

Терминология: свидетельство машины и свидетельство сборки

При определении разрешений доступа политика безопасности может использовать два источника свидетельств, которые называют свидетельствами машины и свидетельствами сборки.

Свидетельство машины предоставляется компьютером, являющимся источником происхождения сборки – это может быть URL, сайт и зона, а также идентификатор сборки – хеш, цифровая подпись или сертификат.

Свидетельство сборки представляет собой дополнительное свидетельство, заключенное в самой сборке, его может ввести в политику безопасности администратор или программист. Свидетельство сборки расширяет набор свидетельств, доступных для принятия решений политикой безопасности. По умолчанию, политика безопасности игнорирует свидетельство сборки, но ее можно сконфигурировать так, чтобы свидетельство сборки учитывалось.

КОНСТРУКТОРЫ КЛАССА EVIDENCE

В классе **Evidence** предусмотрено три конструктора. Конструктор **Evidence** без параметров создает пустой экземпляр класса **Evidence.** Разумеется, пользы от пустого объекта **Evidence** немного. Однако класс **Evidence** располагает методами, позволяющими добавлять в объект свидетельства. Конструктор **Evidence** с одним параметром создает пустую копию. Конструктор **Evidence** с двумя параметрами – массивами **Object** – создает экземпляр класса **Evidence** с двумя массивами свидетельств сборки и машины.

```
public Evidence(); //инициализация нового пустого экземпляра

public Evidence(
   Evidence evidence //пустая копия
);
```

```
public Evidence(
   object[] hostEvidence,    //массив свидетельств машины
   object[] assemblyEvidence // массив свидетельств сборки
);
```

СВОЙСТВА КЛАССА EVIDENCE

В классе **Evidence** имеется пять публичных свойств с именами **Count**, **Locked**, **IsSynchronized**, **SyncRoot** и **IsREadOnly**. Как ни странно, только два из них, **Count** и **Locked**, имеют практический смысл. Как мы уже упоминали, класс **Evidence** реализует интерфейс **ICollection**, который обладает публичными свойствами **Count**, **IsSynchronized** и **SyncRoot**, а также публичным методом **CopyTo**. Поскольку класс **Evidence** инкапсулирует набор объектов с информацией о свидетельствах, логично, что свойство **Count**, представляющее число элементов в этом наборе, доступно только для чтения.

Метод **IsSynchronized** обычно используется в семействах, для того чтобы определить, пригодно ли семейство для многонитевого доступа[1]. В случае класса **Evidence** метод **IsSynchronized** всегда возвращает значение False, поскольку многопоточная работа с объектами-свидетельствами не поддерживается. В этом есть практический смысл, поскольку трудно представить себе множество нитей, имеющих одновременно дело со столь деликатной материей, как информация, необходимая для обеспечения безопасности. Так или иначе, но свойство **IsSynchronized** практически не используется ввиду своей бесполезности.

Свойство **SyncRoot** обычно содержит в себе объект, служащий для синхронизированного доступа к элементам семейства. Поскольку синхронизация здесь не поддерживается, свойство **SyncRoot** возвращает просто **this**. Следовательно, свойство **SyncRoot** также не имеет практического смысла и не используется. Несмотря на свою бесполезность в нашем случае, свойства **IsSynchronized** и **SyncRoot** определенны в **ICollection**, и потому класс **Evidence** должен их содержать.

Свойство **IsReadOnly** всегда возвращает **false**, поскольку объекты-свидетельства «только для чтения» не поддерживаются. Поскольку значение этого свойства предопределено, оно точно так же бесполезно и потому не используется.

Свойство **Locked** возвращает или принимает булево значение, указывающее, заблокировано ли в данный момент свидетельство. Если в этом свойстве содержится значение false, значит, свидетельство можно модифицировать вызовами методов **AddHost** и **MergeHost**, с которыми мы вскоре познакомимся. Если же в свойстве **Locked** содержится значение true, то при попытке обращения к методу **AddHost** или **MergeHost**[2] будет

---

[1] Многонитевый доступ подразумевает, что многие нити процесса одновременно добавляют, удаляют и модифицируют элементы семейства, при этом семейство ведет себя полностью корректно. Если семейство непригодно для многонитевого доступа, то одновременная работа с ним нескольких нитей может привести к повреждению данных.

[2] Как ни странно, сказанное распространяется на методы **MergeHost** и **AddHost**, однако не распространяется на метод **AddAssembly**.

сгенерировано исключение **SecurityException**, если только вызывающий код не обладает разрешением **ControlEvidence**. Фактически, уже для того чтобы изменить значение свойства **Locked**, вам требуется иметь разрешение **ControlEvidence**. По умолчанию свойство **Locked** содержит значение false, и если вы не планируете изменять содержимое объекта **Evidence**, вы можете попросту игнорировать это свойство.

В приведенном ниже списке описываются все перечисленные свойства объектов класса **Evidence**. Учтите, что три из них бесполезны, поскольку всегда возвращают фиксированное значение, что свойство **Count** доступно только для чтения, а свойство **Locked** доступно для чтения и записи.

- **Count** возвращает число элементов в семействе. Только чтение.
- **Locked** возвращает и позволяет задать значение, указывающее на состояние семейства, заблокировано оно или нет (чтение и запись).
- **IsSynchronized** указывает, пригодно ли семейство для многонитевого доступа (всегда возвращает false).
- **SyncRoot** обычно используется для синхронизации доступа к семейству (всегда возвращает this).
- **IsReadOnly** указывает, является ли семейство доступным только для чтения (всегда возвращает false).

МЕТОДЫ КЛАССА EVIDENCE

Класс **Evidence** располагает семью[1] публичными методами: **GetEnumerator**, **CopyTo**, **AddAssembly**, **AddHost**, **GetAssemblyEnumerator**, **GetHostEnumerator** и **Merge**. Базовые операции, для выполнения которых предназначены эти методы, заключаются в перечислении и копировании, и также в добавлении и слиянии свидетельств.

Один из этих методов, **GetEnumerator**, происходит из интерфейса **IEnumerable**. Метод **GetEnumerator** просто возвращает интерфейс **IEnumerator**, который можно использовать для последовательного доступа поочередно ко всем элементам семейства (то есть для их «перечисления»). При этом в интерфейсе используется его свойство **Current** и методы **MoveNext** и **Reset**. Другой метод, **CopyTo**, происходит из интерфейса **ICollection**. Метод **CopyTo** просто копирует элементы семейства свидетельств в массив типа **Object**, начиная с некоторой заданной позиции. Остальные методы специфичны для класса **Evidence**, как такового.

- **GetEnumerator** обеспечивает доступ ко всем свидетельствам, как свидетельствам машины, так и свидетельствам сборки.
- **CopyTo** копирует свидетельства в массив типа **Object**.
- **AddAssembly** добавляет заданное свидетельство сборки в семейство.
- **AddHost** добавляет заданное свидетельство машины в семейство.

[1] Класс **Evidence** обладает семью публичными методами, не считая тех, что происходят из суперкласса **Object**. Все классы наследуют и зачастую переопределяют эти публичные методы класса **Object**: **Equals**, **GetHashCode**, **GetType** и **ToString**, а также защищенные методы **Finalize** и **MemberwiseClone**.

- **GetHostEnumerator** обеспечивает доступ к свидетельствам машины.
- **GetAssemblyEnumerator** обеспечивает доступ к свидетельствам сборки.
- **Merge** выполняет сияние двух семейств свидетельств в одно семейство.

Получение свидетельства текущего домена приложения

Хотя класс **Evidence** и предусматривает конструкторы, вы чаще будете иметь дело с уже готовыми объектами-свидетельствами, которые отражают положение дел в конкретной среде выполнения. Способ получения такого объекта **Evidence** заключается в использовании свойства **Evidence** объекта – домена текущего приложения. Статическое свойство **AppDomain.CurrentDomain** содержит текущий домен приложения для текущей нити. В программном коде это выглядит следующим образом:

```
Evidence evidence =
            AppDomain.CurrentDomain.Evidence
```

Перечисление объектов Evidence

Вы можете последовательно получить доступ ко всем доступным свидетельствам, используя перечисление объекта **Evidence** домена текущего приложения. Если вы это сделаете, то обнаружите информацию об URL и зоне безопасности, которая определяется физическим происхождением сборки. Также вы обнаружите хеш, идентифицирующий код сборки. Если же сборка содержит цифровую подпись (только в этом случае) вы также обнаружите криптографическое «сильное» имя. Наличие «сильного» имени позволяет математически строго идентифицировать того, кто наложил подпись, и увериться в том, что сборка не была подделана или изменена каким-либо образом с момента наложения подписи. Надежность подобных гарантий, впрочем, зависит от надежности сертификационного центра, который удостоверяет подпись.

Все эти типы свидетельств представлены следующими классами (все они определены в пространстве имен **System.Security.Policy**):

- **Zone** указывает зону безопасности такую, как **MyComputer**, **Internet** и т. д.
- **Url** специфицирует протокол и адрес доступа к источнику, из которого происходит сборка.
- **Hash** содержит хеш кода сборки по протоколу SHA1 или MD5.
- **StrongName** содержит, если на сборку наложена цифровая подпись, имя, версию и открытый ключ сборки.
- **Site** указывает (в случае, когда сборка загружена через Internet), адрес сайта, откуда происходит сборка.

Если вы получили объект **Evidence**, принадлежащий домену текущего приложения, то вы можете, используя интерфейс **IEnumerator**, просмотреть информацию о свидетельствах, принимая соответствующие решения в смысле безопасности. Для того чтобы проделать это, просто вызовите метод **GetEnumerator** нужного вам объекта **Evidence**, и войдите в цикл типа while, вызывая на каждом проходе метод **MoveNext**.

```
//просмотр свидетельств
IEnumerator enumerator = evidence.GetEnumerator();
while (enumerator.MoveNext())
{
   object item = enumerator.Current;
   //принять решение на основе полученного свидетельства
   if (...)
   {
            ...
   }
}
```

Пример программы WalkingThruEvidence

Давайте рассмотрим на примере программы **WalkingThruEvidence**, как все это можно проделать на практике. Эта программа получает текущий объект **Evidence** и отображает его содержимое. В этом примере мы используем сборку с цифровой подписью, поэтому будем иметь дело также и со свидетельством **StrongName**.

Как подписать сборку

Пример программы **WalkingThruEvidence** представляет собой сборку с цифровой подписью. Существует два способа наложить на сборку подпись. Первый заключается в использовании **Al.exe** (Assembly Linker – компоновщик сборок) для добавления подписи к уже существующей сборке. Второй путь состоит в том, чтобы через атрибут кода **AssemblyKeyFileAttribute** или **AssemblyKeyNameAttributeassembly** задать подпись непосредственно в исходном коде сборки.

В любом случае вам потребуется предварительно сгенерировать пару ключей. Поскольку мы здесь заинтересованы скорее в аутентификации, чем в секретности, мы должны подписать сборку секретным ключом, а затем сделать общедоступными саму сборку и ее открытый ключ. Пару ключей вы можете создать при помощи утилиты **Sn.exe** (Strong Name – «сильное» имя). Например, следующая командная строка создаст файл с парой ключей под именем **MyKeyPair.snk**. Утилита **Sn.exe** может также записать полученную пару ключей в именованный контейнер для ключей, управляемый CSP (cryptographic service provider – поставщик услуг криптографии). Подробней об этой утилите можно узнать из ее документации.

```
sn -k MyKeyPair.snk
```

В результирующем файле содержатся оба ключа и его необходимо хранить в секрете, поскольку в нем имеется секретный ключ. Для того чтобы отделить от него открытый ключ, необходимо извлечь его и сохранить в другом файле. Следующая командная строка извлекает открытый ключ из файла **MyKeyPair.snk** и помещает его в файл **MyPublicKey.snk**.

```
sn –p MyKeyPair.snk MyPublicKey.snk
```

Для того чтобы подписать сборку в форме DLL или EXE, используете утилиту **Al.exe**. Следующая командная строка накладывает на сборку **WalkingThruEvidence.exe** подпись, используя секретный ключ из файла **MyKeyPair.snk**. Компоновщик сборок **Al.exe** ищет ключевую пару относительно текущего и выходного каталогов.

```
al /out:WalkingThruEvidence.exe /keyfile:MyKeyPair.snk
```

Другая техника наложения подписи состоит в использовании атрибутов кода. Вы можете добавить к своему исходному коду атрибут AssemblyKeyFileAttribute или AssemblyKeyNameAttributeassembly. Атрибут AssemblyKeyFileAttribute задает имя файла, в котором содержится секретный ключ, а атрибут AssemblyKeyNameAttributeassembly задает имя контейнера CSP, где в этом случае должен содержаться ключ. В следующем коде использован атрибут AssemblyKeyFileAttribute и файл MyKeyPair.snk. Необходимо указывать полный путь к файлу, но в нашем примере имя файла сокращено, для того чтобы текст помещался на книжной странице. Этот атрибут обычно добавляется к файлу исходного кода с именем AssemblyInfo.cs, который генерируется автоматически при создании проекта типа C# в среде Visual Studio.NET.

```
[assembly:AssemblyKeyFileAttribute(@"...\MyKeyPair.snk ")]
```

Программа WalkingThruEvidence вначале получает объект Evidence, просматривает все свидетельства, которые в нем содержатся. Затем она отображает сведения по каждому из свидетельств. Вот ее исходный код.

```
    //получить свидетельство домена приложения
Evidence evidence =
          AppDomain.CurrentDomain.Evidence;

//получить перечисление
IEnumerator enumerator = evidence.GetEnumerator();

//просмотреть все свидетельства
while (enumerator.MoveNext())
{
          object item = enumerator.Current;

          //отобразить
          Type type = item.GetType();
          Console.WriteLine(type.Name + ": ");
```

```
            if (type == typeof(Url))
            {
              Console.WriteLine(
                   "   Value: " +
                   ((Url)item).Value);
            }
            if (type == typeof(Zone))
            {
              Console.WriteLine(
                   "   SecurityZone: " +
                   ((Zone)item).SecurityZone);
            }
            if (type == typeof(Hash))
            {
              Console.WriteLine(
                   "   MD5: " +
                   BitConverter.ToString(((Hash)item).MD5));
              Console.WriteLine(
                   "   " + "SHA1: " +
                   BitConverter.ToString(((Hash)item).SHA1));
            }
            if (type == typeof(StrongName))
            {
              Console.WriteLine(
                   "   Name: " +
                   ((StrongName)item).Name);
              Console.WriteLine(
                   "   Version: " +
                   ((StrongName)item).Version);
              Console.WriteLine(
                   "   PublicKey: " +
                   ((StrongName)item).PublicKey);
            }
            if (type == typeof(Site))
            {
              Console.WriteLine(
                   "   Name: " +
                   ((Site)item).Name);
            }
  }
```

Ниже приведен консольный вывод программы **WalkingThruEvidence** при запуске ее непосредственно в локальной файловой системе. Строки вывода местами слишком длинны для печатной страницы, и здесь некоторые из них сокращены.

Как видите, зоной безопасности является локальный компьютер, а URL описывает файловый протокол с адресом местоположения сборки. Поскольку на сборку наложена цифровая подпись, мы видим ее среди свидетельств. Наконец, среди прочего, представлена информация о хешах

MD5 или SHA-1. Если бы мы создали сборку без цифровой подписи, то информация о «сильном имени» здесь бы отсутствовала. Поскольку сборка запускалась локально, а не при помощи Internet Explorer, информация о сайте в Internet отсутствует.

```
Zone:
   SecurityZone: MyComputer
Url:
   Value: file://C:/.../WalkingThruEvidence.exe
StrongName:
   Name: WalkingThruEvidence
   Version: 1.0.1010.20177
   PublicKey:
00240000048000009400000006020000002400005253...
   5C5703B8AEEA06C1CFD72327CD0F35FD650345ACA6806E7
Hash:
   MD5: A6-AB-D6-AD-42-42-38-67-BF-57-32-4C-55-A4-6C-A4
   SHA-1: F6-E1-17-1A-4B-6C-BE-DB-4B-ED-...-E4-E2-C3-37
```

Доступ к WalkingThruEvidence через IIS

Давайте теперь попробуем пойти по другому пути, опубликовав программу **WalkingThruEvidence** на локальном Web-сервере IIS и выполнив ее через Internet Explorer по протоколу http. Для того чтобы опубликовать файл на сайте, просто скопируйте его в каталог **\inetpub\wwwroot**. Затем для осуществления доступа через Internet Explorer укажите в качестве сервера локальную машину – **localhost**. В строке адреса Internet Explorer, соответственно, необходимо ввести **http://localhost/WalkingThruEvidence.exe**.

Если вы предварительно поместили сборку в корневой каталог IIS, то при такой попытке доступа Internet Explorer загрузит и выполнит сборку, но в результате будет сгенерировано исключение **SecurityException**. Причина в том, что сборка загружена теперь не из зоны MyComputer, которая обладает полным доверием, а из зоны Local Intranet, которой, по умолчанию, полное доверие не предоставлено.

Для того чтобы программа смогла работать, необходимо изменить уровень доверия для зоны Local Internet на полное (full trust). **Предупреждение**: это всего лишь эксперимент. В реальной промышленной среде вам никогда не следует предоставлять полное доверие зоне Internet! Такие эксперименты следует проводить только на несетевой машине, предназначенной для разработок и экспериментов. После завершения эксперимента необходимо вернуть уровень доверия к начальному значению, чтобы избежать очевидных опасностей. Для того чтобы изменить уровень доверия для зоны, откройте Панель управления и в папке Администрирование выберите значок Microsoft .NET Framework Wizards, затем выберите Adjust .NET Security, в результате стартует мастер Security Adjustment Wizard, как показано на рисунке 8.19. На первом шаге мастера выберите позицию переключателя Make changes to this computer и щелкните на Next. Щелкните затем на значке Local Intranet и измените уровень доверия на Full Trust, как показано на рисунке 8.20.

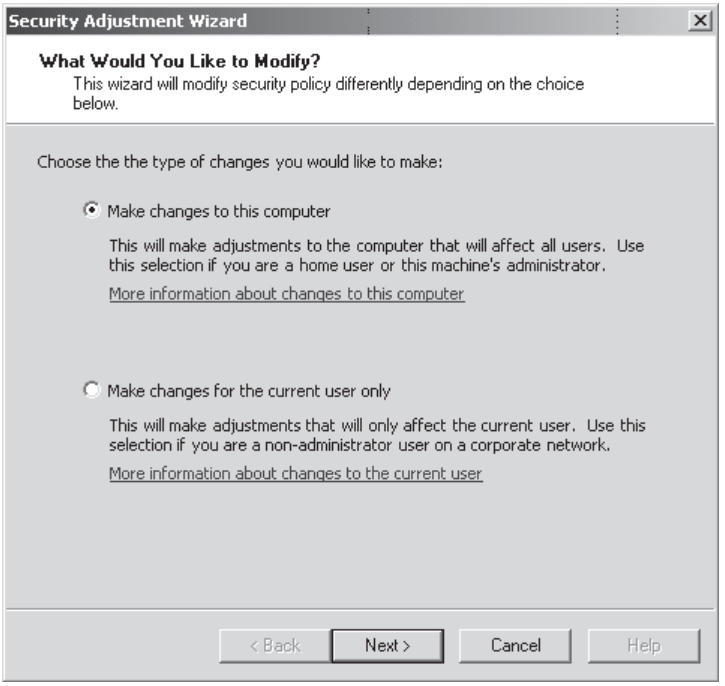


Рис. 8.19. Мастер Security Adjustment Wizard

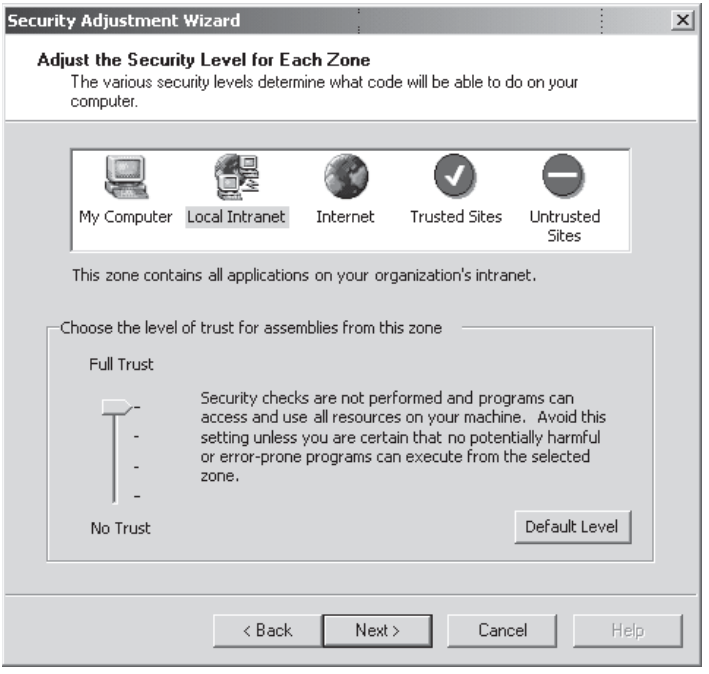


Рис. 8.20. Зона Local Intranet получает полное доверие (Full Trust)

Щелкните затем на Next и, чтобы завершить работу мастера, на Finish. Проделав все это, вы можете снова попытаться выполнить программу через Internet Explorer. Результат приведен в следующем консольном выводе. Зона безопасности изменилась с MyComputer на Intranet. Предыдущий URL, **file://C:/.../WalkingThruEvidence.exe**, сменился на **http://localhost/WalkingThruEvidence.exe**. Здесь вы видите разницу в протоколах (http вместо file).

Свидетельство Site, которое ранее полностью отсутствовало, теперь указывает на имя машины, а свидетельства «сильного имени» и хеша теперь отсутствуют. Вполне очевидно, что теперь мы имеем дело с совершенно другой группой кода.

```
Zone:
   SecurityZone: Intranet
Site:
   Name: localhost
Url:
   Value: http://localhost/WalkingThruEvidence.exe
```

CAS в императивном стиле

Теперь мы обратим внимание на императивный подход в реализации CAS. Вначале мы рассмотрим, каким образом это делается при просмотре доступных свидетельств, а затем увидим, как ту же самую задачу можно решить посредством производных классов **CodeAccessPermission**.

ПРИМЕР ПРОГРАММЫ IMPERATIVECAS

Пример программы **ImperativeCAS**, размещенный в каталоге **ImperativeCAS** наряду со связанными с ним по смыслу примерами **TrustedClient** и **EvilClient**, иллюстрирует явно выраженный императивный подход в реализации CAS, заключающийся в защите компонента путем запрета его вызовов некоторыми клиентскими приложениями, не обладающими достаточным уровнем доверия. Программы **TrustedClient** и **EvilClient** попытаются вызвать метод **DoSomethingForClient**, предоставляемый сборкой **ImperativeCASComponent**, но только один из них сделает это успешно.

В этом примере явным образом используется класс **Evidence**, здесь делается выбор между двумя действиями (нормальное выполнение или генерация исключения) на основе проверки определенной детали в свидетельствах сборки. Затем мы познакомимся с другим вариантом императивного подхода, где вместо изучения содержимого объектов-свидетельств используются объекты-разрешения.

Программа **EvilClient** крайне проста. Она всего лишь вызывает статический метод по имени **DoSomethingForClient**, определенный классом **ImperativeCASComponent**, в отдельной сборке с именем **ImperativeCASComponent.dll**.

```
//EvilClient.cs
using System;
using System.Security;
class EvilClient
{
   static void Main(string[] args)
   {
            //необходима ссылка на компонент
            //ImperativeCASComponent.dll
            //попытка вызова компонента
            try
            {
                  ImperativeCASComponent.DoSomethingForClient();
            }
            catch (SecurityException se)
            {
                  Console.WriteLine(
                        "SecurityException: " + se.Message);
            }
   }
}
```

Обратите внимание, из приведенного ниже консольного вывода ясно, что запуск программы **EvilClient** завершился неудачей, что указывает на недостаточный уровень доверия к вызывающему клиенту. Мы поймем, почему так получилось, изучив код метода **DoSomethingForClient**.

```
DoSomethingForClient called
SecurityException: Client is not trustworthy
```

Прежде чем изучить метод **DoSomethingForClient**, давайте попробуем запустить другую программу, которая практически идентична программе **EvilClient**. Как легко убедиться из ее листинга, приведенного ниже, программа **TrustedClient** выглядит в точности так же, как и программа **EvilClient**, однако при ее запуске вы получите совершенно иной результат.

```
//TrustedClient.cs

using System;
using System.Security;

class TrustedClient
{
   static void Main(string[] args)
   {
            // необходима ссылка на компонент
            // ImperativeCASComponent.dll
```

```
            // попытка вызова компонента
            try
            {
                  ImperativeCASComponent.DoSomethingForClient();
            }
            catch (SecurityException se)
            {
                  Console.WriteLine(
                        "SecurityException: " + se.Message);
            }
   }

}
```

В приведенном ниже листинге содержится консольный вывод этой программы. Как видно из него, на этот раз вызов завершился успешно, а это означает, что вызывающий клиент оказался достойным доверия. Мы поймем, почему так получилось, изучив код метода **DoSomethingForClient**.

```
DoSomethingForClient called
Permitted: Client is trustworthy
```

Давайте теперь рассмотрим код, реализующий метод **DoSomethingForClient**. Этот статический метод реализован в классе с именем **ImperativeCASComponent**. Обе программы, **EvilClient** и **TrustedClient**, пытаются вызвать этот метод совершенно одинаковым образом.

Метод **DoSomethingForClient** начинается с получения объекта **Evidence**, принадлежащего домену текущего приложения. Затем он извлекает интерфейс **IEnumerator** и с помощью цикла while просматривает все свидетельства в этом объекте. Как мы уже знаем, в объекте **Evidence** могут содержаться свидетельства нескольких различных типов, но мы в данном примере заинтересованы в свидетельстве, представленном классом **Url**, который определен в пространстве имен **System.Security.Policy**. Мы проверяем при помощи оператора **if** тип свидетельства и, если тип оказывается равным **Url**, проверяем, заканчивается ли значение этого свидетельства строкой **TrustedClient.exe**. Только в этом случае мы признаем вызывающего клиента достойным доверия. Если свидетельство типа **Url** заканчивается иначе, чем **TrustedClient.exe**, вызов отвергается генерацией исключения **SecurityException**. Конечно, это лишь упрощенный пример, иллюстрирующий концепцию. В более реалистичном сценарии вы, вероятно, будете принимать более сложные решения на основе нескольких свидетельств.

```
//ImperativeCASComponent.cs

using System;
using System.Security;
using System.Collections;
```

```
using System.Security.Policy;

using System.Windows.Forms;

public class ImperativeCASComponent
{
   //этот метод работает только для TrustedClient.exe
   public static void DoSomethingForClient()
   {
            Console.WriteLine(
                  "DoSomethingForClient called");

            //получить свидетельства текущего домена приложения
            Evidence evidence =
                  AppDomain.CurrentDomain.Evidence;

            //получить интерфейс
            IEnumerator enumerator = evidence.GetEnumerator();

            bool trustworthy = false; //по умолчанию доверия нет
            while (enumerator.MoveNext()) //просмотрим свидетельства
            {
                  object item = enumerator.Current;

                  //проверим Url
                  Type type = item.GetType();
                  if (type == typeof(System.Security.Policy.Url))
                  {
                        String strUrl =
                              ((Url)item).Value.ToString();
                        if (strUrl.EndsWith("TrustedClient.exe"))
                        {
                              trustworthy = true; //есть доверие
                              break;
                        }
                  }
            }

            //если доверия нет, генерируем исключение
            if (!trustworthy)
                  throw new SecurityException(
                        "Client is not trustworthy");

            //если мы здесь, значит, доверие есть
            Console.WriteLine(
                  "Permitted: Client is trustworthy");
   }
}
```

Разрешения доступа кода

Мы познакомились с несколькими примерами использования класса **Evidence** явным образом, напрямую, когда решения принимаются на основе изучения содержащихся в нем свидетельств. Другой возможный путь реализации CAS состоит в том, чтобы позволить классам разрешений доступа автоматически принимать неявные решения. Они обнаруживают отличие между разрешениями текущей политики безопасности и разрешениями, которыми располагает выполняемый код, и генерируют исключение **Security.Exception** автоматически.

Производные классы CodeAccessPermission

Программирование при помощи CAS обычно подразумевает использование классов, производных от класса **CodeAccessPermission**. Список этих классов приведен ниже. Поскольку они не принадлежат к одному общему для всех пространству имен, в списке приведены их полные имена. Назначение большинства этих производных классов явствует из их имен. Например, **DBDataPermission** отвечает за доступ к базе данных, **PrintingPermission** управляет доступом к принтерам, а **SocketPermission** представляет разрешения на создание или прием запросов на создание соединений TCP/IP.

Некоторое подмножество этих классов разрешений доступа называют разрешениями идентификации (identity permissions), поскольку они не контролируют доступ к каким-либо ресурсам, а, скорее, специализируются на свидетельствах машины, обеспечивающих идентификацию сборок. Вы легко опознаете эти классы по слову Identity, содержащемуся в их именах, например, **SiteIdentityPermission** или **ZoneIdentityPermission**.

Обратите внимание, в списке производных от **CodeAccessPermission** классов нет класса **PrincipalPermission**, который мы обсуждали в предыдущей главе. Причина в том, что **PrincipalPermission** это равный по уровню класс, производный непосредственно от класса **Object**, и он представляет не разрешения доступа кода, а разрешение, инкапсулирующее в себе систему безопасности, основанную на идентификации пользователей.

- System.Data.Common.**DBDataPermission**
- System.Drawing.**PrintingPermission**
- System.Messaging.**MessageQueuePermission**
- System.Net.**DnsPermission**
- System.Net.**SocketPermission**
- System.Net.**WebPermission**
- System.Security.**Permissions.EnviromentPermission**
- System.Security.**Permissions.FileDialogPermission**
- System.Security.**Permissions.FileIOPermission**
- System.Security.**Permissions.IsolatedStoragePermission**
- System.Security.**Permissions.PublisherIdentityPermission**

- System.Security.**Permissions.ReflectionPermission**
- System.Security.**Permissions.RegistryPermission**
- System.Security.**Permissions.ResourcePermissionBase**
- System.Security.**Permissions.SecurityPermission**
- System.Security.**Permissions.SiteIdentityPermission**
- System.Security.**Permissions.StrongNameIdentityPermission**
- System.Security.**Permissions.UIPermission**
- System.Security.**Permissions.UrlIdentityPermission**
- System.Security.**Permissions.ZoneIdentityPermission**

Класс CodeAccessPermission

Класс **CodeAccessPermission** обладает несколькими методами, работу которых необходимо понимать, поскольку они присутствуют во всех производных классах, с которыми вы будете иметь дело. Позднее, некоторые из этих методов мы изучим детально, и увидим, как они работают, на демонстрационном примере программы. Пока же давайте познакомимся с кратким описанием этих методов. Методы, просто унаследованные от класса **Object**, здесь не показаны.

- **Assert** выдает заданное разрешение текущему методу и методам, чьи вызовы располагаются в стеке ниже, даже если код, чьи вызовы расположены выше[1], такого разрешения не имеет. Этот метод, в сущности, блокирует контроль стека, начиная с заданной точки, предотвращая генерацию исключения **SecurityException**. Этот метод успешно отработает, только если вызывающий код сам пройдет все проверки, необходимые для предоставления того разрешения, которое необходимо выдать, а также, если вызывающий код обладает разрешением на саму операцию assert, то есть на выдачу такого «расширенного» разрешения. Выданное разрешение остается в силе с момента вызова метода **Assert**, и до завершения работы вызвавшего его метода или до вызова метода **RevertAssert** в текущем сегменте стека. В каждом сегменте стека может действовать только одно разрешение, выданное таким способом, и повторный вызов метода **Assert** в одном и том же сегменте стека приведет к генерации исключения **SecurityException**. Этим методом следует пользоваться с осторожностью, поскольку он делает режим контроля разрешений более «либеральным» и повышает потенциальные риски, но зачастую его использование может быть очень удобным. Например, вы можете агрессивно отказывать в разрешениях выше по стеку и в то же время выборочно выдать специфическое разрешение выбранным методам.

[1] Традиционно, стек вызовов представляется растущим вниз. Иными словами, методы, чьи вызовы расположены в стеке ниже, были вызваны методами, чьи вызовы располагаются в стеке выше. Каждый вызванный метод получает в стеке некую область, называемую *сегментом стека (stackframe)*, в которой размещаются параметры и переменные текущего метода.

- **Copy** – это абстрактный метод, который должен быть реализован в каждом производном классе и который создает копию объекта-разрешения.
- **Demand** используется для того, чтобы убедиться, что у вызывающих методов имеется указанное разрешение. Он выполняет эту проверку, делая проход по стеку и контролируя разрешения у всех методов, чьи вызовы расположены выше по стеку. Этот метод генерирует исключение **SecurityException**, если у одного из вызывающих методов отсутствует заданное разрешение. Данный метод используют для предварительной проверки разрешения, выполняемой перед проведением операции, для которой это разрешение необходимо. Многие классы .NET Framework активно используют метод **Demand** для того, чтобы проверить свои полномочия в соответствии с действующей политикой безопасности. Например, многие **FileStream** методы создают объект **FileIOPermission** для требуемого файла и требуемой файловой операции, а затем вызывают **Demand**, чтобы убедится, что необходимые права для этой операции с этим файлом у них есть. Вы можете определить свой класс разрешений и использовать его метод **Demand** для реализации собственных механизмов защиты.
- **Deny** отказывает в заданном разрешении текущему методу и методам ниже него по стеку, даже если код выше по стеку этим разрешением обладает. При проверке проходом по стеку, он вызывает отрицательный результат по данному разрешению для текущего сегмента стека. Эффект отказа в разрешении действует с момента вызова метода **Deny** и до момента, когда метод **Deny** вернет управление вызвавшему его коду или до момента, когда в текущем сегменте стека будет вызван метод **RevertDeny**. В одном сегменте стека можно использовать только один отказ в разрешении и повторный вызов метода **Deny** в том же сегменте приведет к генерации исключения **SecurityException**. Этот метод используется для «оборонительного» программирования, когда вы хотите предотвратить некоторые действия, про которые знаете, что в текущем и нижележащих методах для этих действий нет разумных поводов.
- **FromXml** – абстрактный метод, который должен быть реализован в любом производном классе для того, чтобы можно было реконструировать объект из его XML-представления.
- **Intersect** – абстрактный метод, который должен быть реализован в любом производном классе для того, чтобы можно было создать объект-разрешение, являющийся пересечением двух существующих объектов-разрешений.
- **IsSubsetOf** абстрактный метод, который должен быть реализован в любом производном классе для того, чтобы можно было определить, является ли некоторый объект-разрешение логическим подмножеством другого объекта-разрешения.
- **PermitOnly** выдает только одно, заданное разрешение, текущем методу и методам, чьи вызовы лежат в стеке ниже, даже если вышележащие вызовы обладают другими разрешениями. Этот метод

похож на **Deny** с той разницей, что **Deny** отказывает в некотором разрешении, а **PermitOnly** выдает его. Метод работает по тому же принципу блокирования прохода по стеку.

- **RevertAll** является статическим методом, который аннулирует все вызовы **Assert**, **Deny** и **PermitOnly** в текущем сегменте стека.
- **RevertAssert** является статическим методом, который аннулирует все вызовы метода **Assert** в текущем сегменте стека.
- **RevertDeny** является статическим методом, который аннулирует все вызовы метода **Deny** в текущем сегменте стека.
- **RevertPermitOnly** является статическим методом, который аннулирует все вызовы метода **PermitOnly** в текущем сегменте стека.
- **ToXml** – абстрактный метод, который должен быть реализован в любом производном классе для того, чтобы можно было создать XML-представление заданного объекта-разрешения.
- **Union** – абстрактный метод, который должен быть реализован в любом производном классе для того, чтобы можно было создать объект-разрешение, являющийся логическим объединением двух объектов-разрешений.

Обратите внимание, некоторые методы такие, как **Copy** и **IsSubsetOf**, являются абстрактными методами. В коде приложения невозможно вызвать такой метод обычным образом. Эти методы вызываются функциями CAS среды .NET Framework. Если вы реализуете собственный класс разрешений, то должны также реализовать и эти методы для нормального взаимодействия с функциями CAS.

Для того чтобы вы получили представление, что представляют собой эти абстрактные методы, рассмотрим следующий исходный код, находящийся в файле **codeaccesspermission.cs**, который входит в состав документации Rotor BCL[1].

```
[DynamicSecurityMethodAttribute[2]()]
public void Deny()
{
   CodeAccessSecurityEngine icase =
SecurityManager.GetCodeAccessSecurityEngine();
   if (icase != null)
   {
            StackCrawlMark stackMark =
                  StackCrawlMark.LookForMyCaller;
            icase.Deny(this, ref stackMark);
   }
```

[1] Copyright 2002 Microsoft Corporation. Документация Rotor BCL была опубликована корпорацией Microsoft и может использоваться оговоренным образом, но она не является открытым кодом в том смысле, как это трактуется публичной лицензией GNU. Пожалуйста, по вопросу авторских прав на этот код обращайтесь к документации Rotor BCL.

[2] Этот атрибут указывает, что пространство необходимо распределено в стеке вызывающего метода с тем, чтобы использовать объект для проверок методом прохода по стеку. Это дает некоторый намек на представление об устройстве таких специальных методов, как Deny или Demand.

```
}
...
public static void RevertDeny()
{
   SecurityRuntime isr =
              SecurityManager.GetSecurityRuntime();

   if (isr != null)
   {
              StackCrawlMark stackMark =
                     StackCrawlMark.LookForMyCaller;
              isr.RevertDeny(ref stackMark);
   }
}
...
[DynamicSecurityMethodAttribute[1]()]
public void Demand()
{
   CodeAccessSecurityEngine icase =
              SecurityManager.GetCodeAccessSecurityEngine();
   if (icase != null && !this.IsSubsetOf(null))
   {
              StackCrawlMark stackMark =
                     StackCrawlMark.LookForMyCaller;
              icase.Check(this, ref stackMark);
   }
}
```

Класс UrlIdentityPermission

В этом разделе мы сосредоточимся на одном из производных классов **CodeAccessPermission**, а именно на классе **UrlIdentityPermission**. Этот класс инкапсулирует разрешения, основанные на URL происхождения сборки. Мы также разберем пример программы **UrlIdentityPermission**, чтобы понять, как действует этот класс, но вначале нам необходимо хотя бы немного понимать устройство самого класса **UrlIdentityPermission**. Этот класс включает в себя реализации всех тех абстрактных методов **CodeAccessPermission**, которые мы рассматривали выше. Также он переопределяет некоторые неабстрактные методы **CodeAccessPermission**. Для того чтобы понять назначение этих методов, достаточно просмотреть описания, приведенные в предыдущем разделе. Кроме всего этого, класс **UrlIdentityPermission** добавляет очень немногое в смысле функциональности. Единственные добавочные члены класса – это два конструктора и одно строковое свойство с именем **Url**.

[1] Этот атрибут указывает, что пространство необходимо распределить в стеке вызывающего метода с тем, чтобы использовать объект для проверок методом прохода по стеку. Это дает некоторый намек на представление об устройстве таких специальных методов, как Deny или Demand.

Отличный способ изучения: документация Rotor BCL

Если вам когда-нибудь захочется понять, как на самом деле устроена и работает среда .NET Framework или вам потребуются примеры хорошего программирования на языке C#, вы не найдете лучшего источника знания, чем документация Rotor BCL (Base Class Library – библиотека базовых классов). Здесь содержится весь исходный код библиотеки .NET! Документация доступна для просмотра по ссылке *http://dotnet.di.unipi.it/Content/sscli/docs/doxygen/fx/bcl/index.html*.

Фактически, вы можете загрузить все дерево исходного кода Rotor и создать собственную среду выполнения, которая позволит вам пошагово выполнять код .NET Framework при помощи отладчика Cordbg.exe в системе Windows. Вы даже сможете запустить эту среду в системе FreeBSD! См. *http://msdn.microsoft.com/downloads/default.asp?URL=/downloads/sample.asp?url=/msdn-files/027/001/901/msdncompositedoc.xml*.

Еще одна альтернатива исходит от Ximian, спонсора проекта Mono, который представляет собой усилия сообщества Open Source реализовать .NET Framework, CLR и компилятор C#. Подробней об этом см. *www.go-mono.com/*.

Удачных исследований!

КОНСТРУКТОРЫ КЛАССА URLIDENTITYPERMISSION

Один из конструкторов класса **UrlIdentityPermission** инициализирует новый объект с параметром **PermissionState.** Этот параметр определяет два значения: **None** и **Unrestricted.** Однако конструктор **UrlIdentityPermission** не принимает значение **Unrestricted,** и потому, чтобы избежать генерации исключения **ArgumentException,** необходимо указывать значение **None.**

```
public UrlIdentityPermission(
   PermissionState state //использовать только None
   permission state
);
```

Второй конструктор инициализирует новый объект, основываясь на строке, представляющей некоторый URL. Строка может содержать в конце символы шаблона. Однако строка должна удовлетворять правилам синтаксиса URL и не должна отсутствовать, иначе будет сгенерировано исключение **ArgumentNullException**, **FormatException** или **ArgumentException.**

```
public UrlIdentityPermission(
   string site //URL может содержать символы шаблона
);
```

СВОЙСТВО URL КЛАССА URLIDENTITYPERMISSION

Доступное для чтения и записи свойство **Url** представляет собой строку, состоящую из имени протокола (например, http или ftp), двоеточия, двух прямых косых черт, за которыми следует адрес и имя файла с разделяющими прямыми косыми чертами. Значение URL может быть точным или включать в себя символы шаблона в конце. Вот примеры корректных строк URL:

```
http://www.SomeWebSite.com/SomePath/TrustedClient.exe
http://www.SomeWebSite.com/SomePath/*
file://C:/SomePath/TrustedClient.exe
```

Работа с разрешениями CAS

Мы уже упоминали, что, аналогично концепции безопасности, основанной на идентификации пользователей, здесь тоже имеется два слегка отличающихся стиля в реализации императивного подхода. Как мы видели уже в примере программы **ImperativeCAS**, можно явным образом принимать решения, выбирая варианты действия при помощи конструкции **if**, анализирующей разрешения текущего домена приложения. Решение состояло в выборе между двумя ветвями, причем в одной из них генерировалось исключение **SecurityException**. Такая техника хорошо знакома многим традиционным программистам, но излишний дополнительный код делает ее несколько громоздкой.

При использовании нового стиля вы создаете объект, производный от **CodeAccessPermission**, а затем вызываете метод **Demand** этого объекта. Затем любой метод, который вы вызовете, автоматически приведет к генерации **SecurityException**, если заданное разрешение отсутствует. Иными словами, в пределах оставшейся части текущего сегмента стека так же, как и сегментах стека вызванных методов, исключение **SecurityException** будет генерироваться при необходимости автоматически. Преимущество данного способа состоит в том, что код несколько упрощается и выглядит более аккуратным, поскольку здесь нет явного просмотра свидетельств и условных конструкций, как нет и явной генерации исключения.

Как и в примере **ImperativeCAS**, в программе **UrlIdentityPermission** используется скорее императивный подход к реализации CAS. В каталоге **UrlIdentityPermission** содержится проект **UrlIdentityPermissionComponent**, а также проекты **EvilClient** и **TrustedClient**, в которых используется сборка **ImperativeCASComponent**.

ПРИМЕР ПРОГРАММЫ URLIDENTITYPERMISSION

Пример программы **UrlIdentityPermission** сходен с примером **ImperativeCAS** в том, что здесь распознавание так же производится на основе свидетельства URL, однако, вместо трудоемкого просмотра информации о свидетельствах и явных решений, здесь использован класс **UrlIdentity-**

Permission. Класс **UrlIdentityPermission** позволяет гарантировать, что успешно вызвать метод **DoSomethingForClient** сможет только клиентский код, имеющий источником своего происхождения заданный URL. Одновременно демонстрируется техника ограничения доверия к клиентскому коду на базе заданного свидетельства. Для проверки компонента мы предусмотрели две программы: **TrustedClient** и **EvilClient.** Обе программы практически идентичны с точки зрения их исходного кода. Разница между ними заключается в URL их происхождения. Мы увидим, как URL используется для представления протокола, пути и имени файла для каждой их этих двух программ. Вот исходный код программы **EvilClient.**

```
//EvilClient.cs

using System;
using System.Security;

class EvilClient
{
   static void Main(string[] args)
   {
              //необходима ссылка на компонент
              //ImperativeCASComponent.dll

              //попытка вызова компонента
              try
              {
                     UrlIdentityPermissionComponent.
                            DoSomethingForClient();
              }
              catch (SecurityException se)
              {
                     Console.WriteLine("Error: " + se.Message);
              }
   }
}
```

Когда вы запустите программу **EvilClient**, то получите консольный вывод, приведенный ниже. Обратите внимание, код фактически не вызвал генерации исключения, поскольку источником исключения является URL, отличающийся от того, который является доверенным для данной сборки. Как видите, сгенерировано исключение **SecurityException**, а его сообщение свидетельствует о том, что запрос к **UrlIdentityPermission** потерпел неудачу. Когда мы рассмотрим код **UrlIdentityPermissionComponent**, то поймем, почему так получилось.

```
Error: Request for the permission of type
System.Security.Permissions.UrlIdentityPermission,
mscorlib, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089 failed.
```

Вот исходный код программы **TrustedClient**. Как легко увидеть, он практически идентичен коду программы **EvilClient**. Однако результат выполнения здесь будет на удивление другим.

```
//TrustedClient.cs

using System;
using System.Security;

class TrustedClient
{
   static void Main(string[] args)
   {
              //необходима ссылка на компонент
              //ImperativeCASComponent.dll

              //попытка вызова компонента
              try
              {
                     UrlIdentityPermissionComponent.
                            DoSomethingForClient();
              }
              catch (SecurityException se)
              {
                     Console.WriteLine("Error: " + se.Message);
              }
   }
}
```

Ниже приведен вывод программы. На этот раз исключение не было сгенерировано и сообщение об ошибке отсутствует. Единственная строка вывода принадлежит методу **Console.WriteLine** в сборке **UrlIdentityPermissionComponent**, и свидетельствует она об успешном вызове со стороны клиентской программы.

```
Client call permitted
```

Почему эти две совершенно одинаковые программы так по-разному себя ведут? Чтобы это понять, мы должны рассмотреть код сборки **UrlIdentityPermissionComponent**. Как видите, мы создаем объект **UrlIdentityPermission**, указывая для него доверенное клиентское приложение, а затем вызываем метод **Demand**. Далее этот метод просто делает свое дело, и, если вызывающий клиент не соответствует тому URL, которому должен соответствовать доверенный клиент, исключение генерируется автоматически. Никакой нужды в манипуляциях объектами **Evidence** и просмотре свидетельств. Вы предоставляете все сделать CAS.

```
//UrlIdentityPermissionComponent.cs

using System;
using System.Windows.Forms;
using System.Security.Permissions;

public class UrlIdentityPermissionComponent
{
   public static void DoSomethingForClient()
   {
            UrlIdentityPermission urlidperm =
                  new UrlIdentityPermission(
                  "file://C:/.../TrustedClient.exe");
            urlidperm.Demand();

            //если мы оказались здесь - все в порядке
            Console.WriteLine(
                  "Client call permitted");
   }
}
```

В реальном исходном коде строка URL-файла довольно длинна и не поместилась бы на странице, потому здесь она сокращена. В этом листинге вызов метода **Demand** гарантирует, что в период выполнения среда CLR обнаружит любое несоответствие между заданным разрешением и теми разрешениями, которыми фактически обладает выполняющийся код.

Вспомним, что при загрузке сборки среда CLR проверяет все доступные свидетельства машины и назначает сборке все доступные для нее разрешения, вытекающие из этих свидетельств. Пример, который мы только что рассматривали, включает использование класса **UrlIdentityPermission**, который является единственным доступным классом идентификации. Но мы могли бы точно также использовать и другие классы разрешений, основанных на идентификации. Вспомним, такие разрешения относятся к источнику происхождения сборки (сайт, URL и зона) или тому, кто наложил на сборку цифровую подпись («сильное имя» или издатель).

- **PublisherIdentityPermission** – сертификат X.509
- **SiteIdentityPermission** – имя машины, как часть URL.
- **StrongNameIdentityPermission** – Криптографическая подпись.
- **UrlIdentityPermission** – полный URL в неформатированной форме.
- **ZoneIdentityPermission** – зона MyComputer, LocalIntranet, Internet, Restricted или Trusted.

ПРИМЕР ПРОГРАММЫ FILEIOPERMISSION

Обратим теперь наше внимание на класс разрешений, не имеющий отношения к идентификации. Пример программы **FileIOPermission** показывает, как использовать класс **FileIOPermission** для контроля над операциями

файлового ввода-вывода. Каталог **FileIOPermission** содержит три проекта: **FileIOPermission**, представляющий собой проект исполняемого файла типа EXE, и каталоги **AttemptIO** и **AvoidIO**, представляющие проекты библиотек типа DLL. Программа **FileIOPermission** создает объект **FileIOPermission**, предоставляющий неограниченный доступ, но затем следует вызов метода Deny, в результате чего все операции файлового ввода-вывода становятся запрещенными. Затем следует вызовы методов **DoNoFileIO** и **DoFileIO**, находящихся в двух сборках типа DLL.

Несмотря на то, что метод **Deny** вызван не из той сборки, в которой происходит фактическая попытка файлового ввода-вывода, будет выполнен проход по стеку вплоть до метода **Main**, где обнаружится, что файловый ввод-вывод запрещен, и в результате будет сгенерировано исключение. Исходный код всех трех проектов приведен ниже.

```
//FileIOPermission.cs

//необходимо добавить ссылку на AvoidIO.dll
//необходимо добавить ссылку на AttemptIO.dll

using System;
using System.IO;
using System.Security.Permissions;
using System.Security;

class FileIOPermissionExample
{
   public static void Main()
   {
            FileIOPermission fiop = new FileIOPermission(
                  PermissionState.Unrestricted);
            fiop.Deny();
            try
            {
                  AvoidIO avoidio = new AvoidIO();
                  avoidio.DoNoFileIO();

                  AttemptIO attemptio = new AttemptIO();
                  attemptio.DoFileIO();
            }
            catch(SecurityException se)
            {
                  Console.WriteLine(se.Message);
            }

   }
}

//AvoidIO.cs

using System;
```

```
public class AvoidIO
{
   public void DoNoFileIO()
   {
            Console.WriteLine("Вызов DoNoFileIO...");
            Console.WriteLine("Ничего не пишем.");
   }
}
//AttemptIO.cs

using System;
using System.IO;

public class AttemptIO
{
   public void DoFileIO()
   {
            Console.WriteLine("Вызов DoFileIO...");
            String text = "Данные для записи";
            FileStream fs = new FileStream(
                  "outputdata.txt",
                  FileMode.Create, FileAccess.Write);
            StreamWriter sw = new StreamWriter(fs);
            sw.Write(text);
            sw.Close();
            fs.Close();
            Console.WriteLine(
                  "Записано в файл outputdata.txt: " + text);
   }
}
```

В предыдущем примере, где использовался класс **UrlIdentityPermission**, мы явным образом вызывали метод **Demand** для того, чтобы определить, имеется ли у нас заданное разрешение. В противоположность этому в коде, приведенном выше, нет явного вызова метода **Demand**. Причина в том, что класс **FileStream**, который мы здесь используем, сам вызывает метод **Demand** при необходимости. В общем случае предопределенные классы разрешений, не использующие свидетельство идентификации, не требуют явного вызова метода **Demand**. Среда выполнения .NET Framework, как правило, обеспечивает это в нужных случаях сама. Вы можете явно обращаться к методу **Demand** в целях тестирования или оптимизации. Если вы реализуете собственный класс разрешений, то должны взять на себя ответственность за обеспечение вызовов метода **Demand** в нужных случаях.

Ниже приведен вывод программы **FileIOPermission**. Как видите, метод, не пытавшийся воспользоваться файловым вводом-выводом, благополучно завершил свою работу, а метод, обратившийся к файлу, сгенерировал исключение **FileIOPermission**.

```
Вызов DoNoFileIO...
Ничего не пишем.
Вызов DoFileIO...
Request for permission of type
   System.Secutity.Permissions.FileIOPermission,
   mscorlib, Version=1.0.3300.0, Culture=neutral,
   PublicKeyToken=b77a5c571932e089 failed.
```

Декларативное разрешение доступа

Ранее мы видели, как работает императивный подход в CAS. Мы создали экземпляр класса, производного от **CodeAccessPermission**, а именно **UrlIdentityPermission**, а затем вызвали его метод **Deny**, для того чтобы запретить нужное действие. Ту же задачу можно решить в декларативном стиле, используя классы, производные от **CodeAccessSecurityAttribute**. Вскоре мы познакомимся с этим на примере **UrlIdentityPermissionAttribute**.

Главное отличие состоит в том, что мы не можем нормальным образом создать экземпляр атрибута, используя императивный подход, при помощи оператора **new**. Вместо этого мы применяем атрибут к сборке, классу или методу, используя специальный синтаксис с квадратными скобками. Другое отличие заключается в том, что информация, декларируемая атрибутом, хранится в метаданных сборки и доступна для среды выполнения в момент загрузки сборки. Это позволяет утилите **PermView.exe** выводить сведения об атрибутах разрешений сборки на консоль.

Синтаксис объявления атрибутов с квадратными скобками

Синтаксис объявления атрибутов в этом случае отличается от синтаксиса, используемого при императивном подходе. Дело в том, что атрибуты применяются к таким программным единицам, как сборка, класс или метод, и фигурируют в объявлении этих единиц. Таким образом, атрибут оказывает свое воздействие в момент компиляции, а результат этого воздействия хранится в метаданных сборки. Специальный синтаксис с использованием квадратных скобок выглядит следующим образом:

```
[<имя_класса_атрибута>(
   <значение_свойства_по_умолчанию>
   <имя_свойства>=<значение>...)]
```

Эти квадратные скобки заключают имя класса атрибута с последующими круглыми скобками. В круглых скобках может содержаться несколько значений свойств, причем первое из них может быть значением

по умолчанию и не нуждается в имени, а далее следует несколько пар «имя=значение». Чтобы говорить более конкретно, рассмотрим фрагмент кода из примера программы **DeclarativeCAS**, который ждет нас впереди. Имя атрибута – **UrlIdentityPermission**, и он инициализируется при помощи **SecurityAction.LinkDemand** своим свойством по умолчанию **Action**, а свойство с именем **Url** инициализируется значением **"file://C:/... TrustedClient.exe"**.

```
[UrlIdentityPermission(
   SecurityAction.LinkDemand,
   Url="file://C:/.../TrustedClient.exe")]
```

Вы можете задаться вопросом – как определить, к каким программным единицам можно применить этот атрибут, и откуда берется свойство по умолчанию, а также именованные свойства. Ответы на эти вопросы содержатся в документации на класс атрибута. Вначале необходимо найти документацию для класса атрибута, затем в списке **AttributeUsage** найти те программные единицы, к которым можно применить атрибут. Это может быть сборка, класс или метод. Затем нужно посмотреть на конструктор класса. В аргументах конструктора видно, какое свойство является свойством по умолчанию[1]. Это свойство, которое не требует явного указания своего имени в синтаксисе с квадратными скобками. Для того чтобы узнать, какие именованные свойства поддерживает данный атрибут, просто посмотрите на свойства, определяемые классом атрибута. Ответы на все вопросы, как видите, можно найти в документации класса. Давайте теперь рассмотрим класс **UrlIdentityPermissionAttribute** на конкретном примере.

Атрибут Url Identity Permission

Прежде чем перейти к изучению примера программы **DeclarativeCAS**, мы вкратце рассмотрим класс **UrlIdentityPermissionAttribute**. Все детали этого класса нас не интересуют, но для наших целей необходимо знать, как работает этот атрибут при использовании синтаксиса с квадратными скобками.

КЛАСС URLIDENTITYPERMISSIONATTRIBUTE

Ниже приведено определение класса **UrlIdentityPermissionAttribute**. Как видите, **AttributeUsage** указывает, что данный атрибут можно применять к сборке, классу, структуре, конструктору и методу.

```
[AttributeUsage(AttributeTargets.Assembly | AttributeTargets.Class
| AttributeTargets.Struct | AttributeTargets.Constructor |
AttributeTargets.Method)]
```

[1] Фактически каждый атрибут разрешения обладает свойством по умолчанию с именем **Action** типа **SecurityAction**. В противоположность этому, именованные свойства обладают «узкой специализацией» и разнятся от атрибута к атрибуту.

```
[Serializable]
public sealed class UrlIdentityPermissionAttribute :
   CodeAccessSecurityAttribute
```

КОНСТРУКТОР URLIDENTITYPERMISSIONATTRIBUTE

Вот конструктор **UrlIdentityPermissionAttribute**. Легко видеть, что он получает один параметр типа **SecurityAction**. Это означает, что у атрибута есть свойство по умолчанию для указания одного из значений типа **SecurityAction**, то есть **LinkDemand**, **InheritanceDemand**, **DemandDeny**, **RequestMinimum** и т. д.

```
[AttributeUsage(AttributeTargets.Assembly | AttributeTargets.Class
| AttributeTargets.Struct | AttributeTargets.Constructor |
AttributeTargets.Method)]
[Serializable]
public sealed class UrlIdentityPermissionAttribute :
   SecurityAction action
```

СВОЙСТВО URL

Из документации также видно, что класс **UrlIdentityPermissionAttribute** обладает свойством **Url** строкового типа. Это означает, что данное свойство можно использовать в синтаксисе определения атрибута с квадратными скобками в качестве именованного свойства. Вот объявление свойства **Url**.

```
[AttributeUsage(AttributeTargets.Assembly | AttributeTargets.Class
| AttributeTargets.Struct | AttributeTargets.Constructor |
AttributeTargets.Method)]
[Serializable]
public string Url {get; set;}
```

Класс SecurityAction

Мы только что видели, что конструктор **UrlIdentityPermissionAttribute** принимает параметр типа **SecurityAction**. Эти значения определены перечислением **SecurityAction** и могут использоваться в синтаксисе определения атрибута с квадратными скобками. Ниже приведен список допустимых значений **SecurityAction** с необходимыми пояснениями.

- **Assert** – запрашивает заданное разрешение, даже если вызывающая сторона этим разрешением не обладает.
- **Demand** – проверяет, обладает ли вызывающая сторона заданным разрешением.
- **Deny** – отказывает в разрешении, даже если у вызывающей стороны такое разрешение имеется.
- **InheritanceDemand** – проверяет, имеют ли заданное разрешение производные классы.

- **LinkDemand** – проверяет, имеет ли заданное разрешение непосредственно вызывающая сторона.
- **PermitOnly** – проверяет, является ли запрошенное разрешение единственным имеющимся разрешением.
- **RequestMinimum** – запрашивает минимум разрешений, запрошенных сборкой.
- **RequestOptional** – запрашивает дополнительные разрешения, полезные, но не необходимые для сборки.
- **RequestRefuse** – запрашивает отказ в разрешениях, в которых сборка не нуждается.

ПРИМЕР ПРОГРАММЫ DECLARATIVECAS

Программа **DeclarativeCAS** во всем похожа на программу **UrlIdentityPermission** с той разницей, что вместо императивного подхода в ней используется подход декларативный. Исходные тексты в файлах **EvilClient** и **TrustedClient** идентичны тем, что использовались в программе **UrlIdentityPermission**, поэтому здесь мы их рассматривать не будем. Давайте лучше посмотрим, как декларативный подход к CAS реализуется в исходном тексте **DeclarativeCASComponent.cs**. Снова, по причине большой длины строки полного пути к файлу, описываемого атрибутом **UrlIdentityPermission**, мы сокращаем строки на книжной странице при помощи многоточия. Как видите, вместо того чтобы создавать экземпляр **UrlIdentityPermission** и обращаться к его методу **Demand**, мы просто объявляем, что к методу **DoSomethingForClient** применен атрибут **UrlIdentityPermission**.

```
//DeclarativeCASComponent.cs

using System;

using System.Security;
using System.Security.Permissions;

public class UrlIdentityPermissionComponent
{
   [UrlIdentityPermission(
               SecurityAction.LinkDemand,
               Url="file://C:/.../TrustedClient.exe")]
   public static void DoSomethingForClient()
   {
            //если мы здесь, значит, все в порядке
            Console.WriteLine(
                  "Клиентский вызов разрешен");
   }
}
```

Когда вы запустите **TrustedClient**, он отработает до конца без генерации исключений. При запуске же **EvilClient** будет сгенерировано исключение **SecurityException**.

В сущности, данный пример ничем не отличается от предыдущего, и все дело здесь в том, что в одном использовался императивный подход и оператор **new**, а в другом – декларативный подход и атрибут разрешения.

Запросы разрешений

Сборка может декларативно задать определенные запросы разрешений. Например, она может указать минимальный набор разрешений, которые ей абсолютно необходимы. Если среда выполнения обнаружит, что требуемых разрешений не имеется, то сборка вообще не будет загружена. Также сборка может указать желательный набор разрешений, отсутствие которых не помешает ей загрузиться. Еще сборки могут указывать разрешения, которые им не нужны и от которых им необходимо безусловно отказаться, чтобы избежать потенциально опасных ситуаций.

Пример программы PermissionRequest

Программа **PermissionRequest** показывает, как в сборке выполняются запросы разрешений. При помощи атрибута **UIPermission** устанавливаются три атрибута, управляющие способностью к использованию системного буфера и функций пользовательского интерфейса. Первый атрибут задает **RequestMinimum** для всех операций с буфером, **RequestOptional** – для незапрещенных операций с пользовательским интерфейсом, и **RequestRefuse** – для всех операций с окнами.

Результатом первого запроса разрешения будет то, что при запуске программы, если политика безопасности не предоставляет доступа к операциям с буфером, при загрузке сборки будет сгенерировано исключение.

Второй запрос разрешения не производит особенных эффектов, он лишь определяет, что разрешения на доступ к операциям с пользовательским интерфейсом не обязательны. Третий запрос указывает, что сборка хочет предотвратить любую возможность взаимодействия с окнами пользовательского интерфейса. Когда вы запустите программу, метод **TryWindowsAccess** попытается вывести на экран окно сообщения, что приведет к генерации исключения.

```
//PermissionRequest.cs

using System;
using System.Security;
using System.Security.Permissions;
using System.Windows.Forms;
[assembly:UIPermission(
   SecurityAction.RequestMinimum,
   Clipboard=UIPermissionClipboard.AllClipboard)]
[assembly:UIPermission(
   SecurityAction.RequestOptional,
   Unrestricted=true)]
[assembly:UIPermission(
   SecurityAction.RequestRefuse,
   Window=UIPermissionWindow.AllWindows)]
public class PermissionRequest
{
   public static void Main()
   {
            Console.WriteLine("Вызываю TryClipboardAccess");
            TryClipboardAccess();
            Console.WriteLine("Вызываю TryWindowAccess");
            TryWindowAccess();
            Console.Write("Нажмите любую клавишу");
            Console.Read();
   }
   private static void TryClipboardAccess()
   {
            try
            {
                  Clipboard.SetDataObject(
                        "TryClipboardAccess", true);
            }
            catch (SecurityException se)
            {
                  Console.WriteLine(se.Message);
            }
   }
   private static void TryWindowAccess()
   {
            try
            {
                  MessageBox.Show("Попытка вывести сообщение");
            }
            catch (SecurityException se)
            {
                  Console.WriteLine(se.Message);
            }
   }
}
```

Наборы разрешений

Наборы разрешений позволяют объединить объекты **IPermission** в группу и манипулировать группой, как целым при помощи уже знакомых вам методов, например, как **Deny**, **Demand** и **Assert**. Если помните, уже говорилось о том, что такие методы, как **Deny** и **PermitOnly** невозможно вызвать дважды в одном сегменте стека, если только не имеет место промежуточная реверсия. Практически это означает, что вам придется использовать набор разрешений, если вы хотите работать с двумя и более разрешениями одновременно и в одном сегменте стека. Сейчас мы рассмотрим пример программы **PermissionSet**, в которой класс **PermissionSet** используется для манипуляций несколькими разрешениями одновременно.

Класс PermissionSet

Класс **PermissionSet** обладает таким же базовым набором методов, как и производные классы **IPermission**, включая методы **Deny**, **Demand** и **Assert**. В добавление к этому здесь имеется метод **AddPermission**, позволяющий добавлять разрешения в набор. Интерфейс **IPermission** включает в себя все такие классы разрешений, как **FileIOPermission**, **UrlIdentityPermission** и т. д.

```
public virtual IPermission AddPermission(
   IPermission perm
);
```

Давайте рассмотрим теперь на примере, как наборы разрешений используются для объединения разрешений в группы и манипуляций с этими группами.

ПРИМЕР ПРОГРАММЫ PERMISSIONSET

Программа **PermissionSet** комбинирует три разрешения, а именно **EnvironmentPermission**, **FileIOPermission** и **UIPermission**, в один набор разрешений. Также она обеспечивает простой пользовательский интерфейс, позволяющий управлять некоторыми аспектами этого объединения. Пользовательский интерфейс также дает возможность вызвать методы **Deny** и **PermitOnly** для того, чтобы продемонстрировать их действие. Наконец, пользовательский интерфейс позволяет вызвать один из двух методов, проверяющих действие набора разрешений на файловые операции и операции с переменными среды окружения.

Тот факт, что программа пытается сообщить результат своей работы при помощи окна сообщения, также проверяет действие **UIPermission**. Запустив эту программу и поэкспериментировав с ней, вы наглядно поймете механизм действия разрешений. Изучив исходный текст методов **EstablishPermissionSet**, **buttonAttemptFileAccess_Click** и **buttonAttempt-**

EnvVarAccess_Click, вы узнаете, как достигать подобных эффектов в собственных программах. На рисунке 8.21 показано окно программы **PermissionSet** (ситуация, когда в разрешении доступа к файлу отказано) а на рисунке 8.22 изображена ситуация, когда программа пытается получить доступ к файлу, несмотря на отказ.

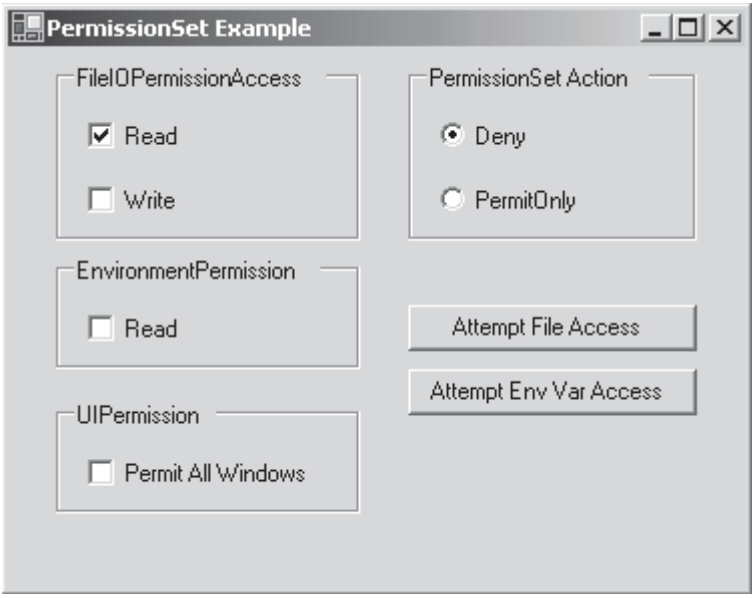


Рис. 8.21. Программа PermissionSet: отказ в доступе на чтение

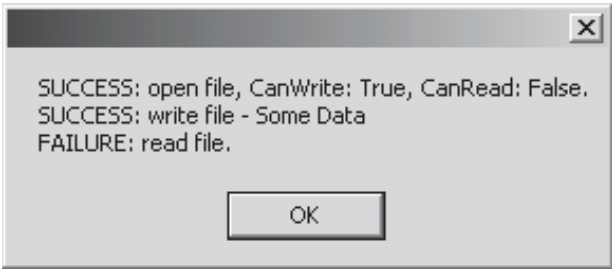


Рис. 8.22. Программа PermissionSet: попытка доступа несмотря на отказ

Вот исходный код трех наиболее важных методов этой программы.

```
//PermissionSetForm.cs
...
namespace PermissionSetForm
{
...
   public class PermissionSetForm : System.Windows.Forms.Form
```

```
{
...
        private void buttonAttemptFileAccess_Click(
                object sender, System.EventArgs e)
        {
                //построить набор согласно положению
                //переключателей
                EstablishPermissionSet();

                //Deny, или PermitOnly
                if (radioButtonDeny.Checked)
                        ps.Deny();
                if (radioButtonPermitOnly.Checked)
                        ps.PermitOnly();

                //попытка создания файла
                FileStream fs = null; //для файла TestFile.txt
                try
                {
                        fs = new FileStream(
                                "TestFile.txt",
                                FileMode.OpenOrCreate,
                                FileAccess.ReadWrite);
                }
                catch (Exception)
                {
                }
                if (fs == null)
                {
                        try
                        {
                                fs = new FileStream(
                                        "TestFile.txt",
                                        FileMode.OpenOrCreate,
                                        FileAccess.Read);
                        }
                        catch (Exception)
                        {
                        }
                }
                if (fs == null)
                {
                        try
                        {
                                fs = new FileStream(
                                        "TestFile.txt",
                                        FileMode.OpenOrCreate,
                                        FileAccess.Write);
                        }
                        catch (Exception)
                        {
                        }
```

```
}
if (fs == null)
{
       MessageBox.Show(
              "НЕУДАЧА: открытие файла");
       return;
}

String strMessageBox =
       "УСПЕХ: открытие файла" +
       ", CanWrite: " + fs.CanWrite +
       ", CanRead: " + fs.CanRead +
       ".\n";

//попытка записи в файл
String strDataOut = "Какой-нибудь текст";
byte [] bytes=
       Encoding.UTF8.GetBytes(strDataOut);
try
{
       fs.Write(bytes, 0, bytes.Length);

       strMessageBox +=
          "УСПЕХ: запись в файл - "+
          strDataOut + "\n";
}
catch (Exception)
{
       strMessageBox +=
              "НЕУДАЧА: запись в файл.\n";
}

//попытка чтения из файла
bytes = new byte[256];
try
{
       fs.Seek(0, SeekOrigin.Begin);
       fs.Read(bytes, 0, bytes.Length);
       String strDataIn =
         Encoding.UTF8.GetString(
              bytes, 0, bytes.Length);
       strMessageBox +=
         "УСПЕХ: чтение файла - " + strDataIn;
}
catch (Exception)
{
  strMessageBox += "НЕУДАЧА: чтение файла.\n";
}

//показать результат
MessageBox.Show(strMessageBox);
```

```
        fs.Close();

        //RevertDeny или RevertPermitOnly
        if (radioButtonDeny.Checked)
              CodeAccessPermission.RevertDeny();
        if (radioButtonPermitOnly.Checked)
              CodeAccessPermission.RevertPermitOnly();

        //очистить набор
        DestroyPermissionSet();
}

private void buttonAttemptEnvVarAccess_Click(
        object sender, System.EventArgs e)
{
        //построить набор согласно положению
        // переключателей
        EstablishPermissionSet();

        //Deny или PermitOnly
        if (radioButtonDeny.Checked)
              ps.Deny();
        if (radioButtonPermitOnly.Checked)
              ps.PermitOnly();

        //попытка чтения переменной среды TEMP
        String ev = null;
        try
        {
              ev = Environment.GetEnvironmentVariable(
                    "TEMP");

              //показать результат
              MessageBox.Show(
               "УСПЕХ: read environment variable - "+
               ev);
        }
        catch (Exception)
        {
              MessageBox.Show(
                "НЕУДАЧА:: read environment variable");
        }

        //RevertDeny или RevertPermitOnly
        if (radioButtonDeny.Checked)
              CodeAccessPermission.RevertDeny();
        if (radioButtonPermitOnly.Checked)
              CodeAccessPermission.RevertPermitOnly();

        //очистить набор
        DestroyPermissionSet();
}
```

```
//построить набор согласно положению переключателей
private void EstablishPermissionSet()
{
       ps = new PermissionSet(PermissionState.None);

       //задать EnvironmentPermission
       EnvironmentPermission ep =
              new EnvironmentPermission(
                EnvironmentPermissionAccess.NoAccess,
                "TEMP");
       if (checkBoxEnvironmentPermissionRead.Checked)
       {
              ep.AddPathList(
                 EnvironmentPermissionAccess.Read,
                 "TEMP");
       }
       ps.AddPermission(ep);

       //задать FileIOPermission
       FileIOPermission fp =
              new FileIOPermission(
                     FileIOPermissionAccess.NoAccess,
                     Path.GetFullPath("TestFile.txt"));
       if (checkBoxFileIOPermissionAccessRead.Checked)
       {
              fp.AddPathList(
                     FileIOPermissionAccess.Read,
                     Path.GetFullPath("TestFile.txt"));
       }
       if (checkBoxFileIOPermissionAccessWrite.Checked)
       {
              fp.AddPathList(
                     FileIOPermissionAccess.Write,
                     Path.GetFullPath("TestFile.txt"));
       }
       ps.AddPermission(fp);

       //задать UIPermission
       UIPermission  uip = new UIPermission(
              UIPermissionWindow.NoWindows);
       if (checkBoxAllWindows.Checked)
       {
              uip.Window =
                     UIPermissionWindow.AllWindows;
       }
       ps.AddPermission(uip);
}

void DestroyPermissionSet()
{
       ps.RemovePermission(
              typeof(EnvironmentPermission));
```

```
                ps.RemovePermission(
                        typeof(FileIOPermission));
                ps.RemovePermission(
                        typeof(UIPermission));
                ps = null;
            }

            PermissionSet ps;
    }
}
```

Определение набора разрешений при помощи конфигурационного файла

Давайте посмотрим теперь, как можно определить набор разрешений при помощи конфигурационного файла, который затем может использоваться приложением для управления разрешениями. Если характеристики безопасности записаны во внешнем файле, программисту гораздо проще делать свою работу, а пользователям и администраторам легче определять политику безопасности для приложения. Возможность переносить и распределять готовую политику безопасности – это очень сильная возможность, учитывая тот факт, что приложение может работать в различном окружении с различными требованиями к безопасности.

ПРИМЕР ПРОГРАММЫ CONFIGUREDFILEIOPERMISSION

Программа **ConfiguredFileIOPermission** очень похожа на **FileIOPermission**, с которой мы познакомились ранее в этой главе с той разницей, что вместо программного создания объекта-разрешения здесь разрешение[1] определяется в наборе разрешений, который определен конфигурационным файлом в формате XML. Этот пример вновь состоит из трех отдельных проектов. Два из них являются библиотеками DLL, а один – проект исполняемого EXE-файла. Первая из библиотек, **AttemptIO**, пытается выполнить операцию ввода-вывода с заданным файлом, а вторая, **AvoidIO**, не делает никаких попыток обращения к файлу. Исполняемая программа, **ConfiguredFileIOPermission**, обращается к публичным методам, **DoFileIO** и **DoNoFileIO**, содержащимся в этих библиотеках. Вот исходный код этих двух DLL-проектов, который не отличается от кода, знакомого нам по примеру **FileIOPermission**.

---

[1] Фактически этот пример слегка отличается от **FileIOPermission** еще в некоторых небольших деталях. Вместо работы с одиночным разрешением мы манипулируем здесь набором разрешений, содержащим в себе нужное нам разрешение. Эту программу возможно стоит сравнить с программой **PermissionSet**, где манипуляции с набором разрешений выполняются программно вместо использования конфигурационного файла. Кроме того, программа **FileIOPermission** отменяет все разрешения файлового доступа, здесь же мы имеем дело с выборочными разрешениями.

```
//AttemptIO.cs

using System;
using System.IO;

public class AttemptIO
{
   public void DoFileIO()
   {
            Console.WriteLine("Вызов DoFileIO ...");
            String text = "Данные для записи";
            FileStream fs = new FileStream(
                  "outputdata.txt",
                  FileMode.Create, FileAccess.Write);
            StreamWriter sw = new StreamWriter(fs);
            sw.Write(text);
            sw.Close();
            fs.Close();
            Console.WriteLine(
                  "Записано в файл outputdata.txt: " + text);
   }
}

//AvoidIO.CS

using System;

public class AvoidIO
{
   public void DoNoFileIO()
   {
            Console.WriteLine("Вызов DoNoFileIO ...");
            Console.WriteLine("Ничего не пишем.");
   }
}
```

Далее приведен исходный код EXE-проекта. Как видите, код почти идентичен коду программы **FileIOPermission**; однако, имеется небольшое, но существенное отличие. В этом исходном коде объект **FileIOPermission** не создается напрямую. Вместо этого мы видим атрибут, указывающий на набор разрешений, записанный в XML-файле. Вскоре мы увидим, как этот набор было определен и стал фигурировать в исходном коде программы.

```
//ConfiguredFileIOPermission.cs

//необходимо добавить ссылку на AvoidIO.dll
//необходимо добавить ссылку на AttemptIO.dll
```

```
using System;
using System.IO;
using System.Security.Permissions;
using System.Security;

class FileIOPermissionExample
{
   public static void Main()
   {

            try
            {
                  AvoidIO avoidio = new AvoidIO();
                  avoidio.DoNoFileIO();

                  AttemptIO attemptio = new AttemptIO();
                  attemptio.DoFileIO();
            }
            catch(SecurityException se)
            {
                  Console.WriteLine(se.Message);
            }
   }
}
```

Прежде чем заняться конфигурационным файлом, давайте посмотрим, как работает программа без набора разрешений. Из приведенного ниже листинга видно, что программа действительно выполнила файловый ввод-вывод.

```
Вызов DoNoFileIO ...
Ничего не пишем.
Вызов DoFileIO ...
Записано в файл outputdata.txt: Данные для записи
```

Теперь мы создадим XML-файл **MyPermissionSet.xml** в каталоге EXE-проекта, этот файл станет конфигурационным файлом для нашего приложения. Содержимое файла приведено ниже. Мы используем тег **PermissionSet**, для того чтобы определить набор разрешений, который будет содержать разрешение **IPermission**, управляющее доступом к файлу с именем **outputdata.txt**. Опять мы сокращаем здесь полный путь к файлу, чтобы уместить листинг на книжной странице. Разрешение допускает запись в файл и чтение из него.

```
<PermissionSet class="System.Security.NamedPermissionSet"
   version="1"
   Name="MyPermissionSet"
   Description="Набор разрешений для доступа к файлу outputdata.txt">
   <IPermission class=
```

```
                "System.Security.Permissions.FileIOPermission ..."
                Read="C\OI\...\outputdata.txt"
                Write="C\OI\...\outputdata.txt"/>
</PermissionSet>
```

Теперь, для того чтобы использовать набор разрешений, определенный в этом файле, в нашей программе, мы должны обратиться к атрибуту **PermissionSetAttribute** метода **Main**, что приведет к отказу во всех разрешениях, определенных в этом наборе. Полный путь к файлу **MyPermissionSet.xml** здесь снова сокращен. Вам нет нужды отказывать в этом разрешении при помощи атрибута, как это сделано ниже. С таким же успехом вы можете сделать это программно. Причина, почему мы здесь поступаем так, а не иначе, заключается в том обстоятельстве, что разрешение определено в отдельном XML-файле и не фигурирует в исходном коде программы. Это дает нам значительную гибкость и позволяет отложить вопросы прав доступа на время после завершения разработки программы.

```
class FileIOPermissionExample
{
   //PermissionSetAttribute ссылается на разрешения,
   //определенные в конфигурационном файле MyPermissionSet.xml
   [PermissionSetAttribute(
               SecurityAction.Deny,
               File="C:\\OI\...\\MyPermissionSet.xml")]
   public static void Main()
   {
               try
               {
                      AvoidIO avoidio = new AvoidIO();
                      avoidio.DoNoFileIO();

                      AttemptIO attemptio = new AttemptIO();
                      attemptio.DoFileIO();
               }
               catch(SecurityException se)
               {
                      Console.WriteLine(se.Message);
               }
   }
}
```

Когда вы запустите эту программу, то увидите следующий консольный вывод. Очевидно, что программа сгенерировала исключение при попытке выполнить операцию файлового ввода-вывода. Отрицание набора разрешений все еще фигурирует в нашем коде, но сам набор разрешений теперь определяется во внешнем файле **MyPermissionSet.xml** и независим от программы. Благодаря этому набор можно изменить в любое время, независимо от самой программы.

```
Вызов DoNoFileIO ...
Ничего не пишем.
Вызов DoFileIO ...
Request for the permission of type
System.Security.Permissions.FileIOPermission,
mscorlib, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089 failed.
```

Итоги главы

В этой главе мы объяснили основные концепции, стоящие за CAS, а также диапазон рисков, которые можно адресовать CAS. Мы рассмотрели, каким образом повышает безопасность среда выполнения CLR и сама природа управляемого кода .NET, в котором типы надежно контролируются. Мы познакомились с управлением политиками безопасности и использованием классов разрешений. Мы также узнали, как реализуются императивный и декларативный подходы в CAS, и как можно управлять ими при помощи средства конфигурирования .NET Framework и утилиты Caspol.exe. Наконец, мы увидели, как определить набор разрешений во внешнем конфигурационном XML-файле, для того чтобы повысить гибкость управления разрешениями.

Глава 9

ASP.NET

Обеспечение безопасности является важным элементом в разработке Web-приложений. Вне сомнений, проблемы безопасности в Web-приложениях – вопрос не второстепенный. В настоящее время в мире Internet все больше становится Web-приложений реального времени, работающих в пространстве Internet или частных сетей. Хотя такие коммуникационные возможности обеспечивают множество преимуществ, они также создают множество потенциальных рисков. Web-приложения зачастую имеют дело с конфиденциальной информацией, которую необходимо защищать от злоумышленников. Именно по этой причине корпорация Microsoft, осознав растущее значение вопросов безопасности в современной сетевой среде, выдвинула стратегию Trustworthy Computing (достойные доверия вычисления)[1]. Технология ASP.NET обеспечивает многоуровневую и полнофункциональную систему обеспечения безопасности, расширяющую возможности встроенных средств .NET Framework. ASP.NET обеспечивает авторизацию, аутентификацию, заимствование прав (impersonation) и технику делегирования прав, что позволит вам усовершенствовать средства защиты Web-приложений и серверов.

Технология ASP.NET согласованно взаимодействует с IIS[2], обеспечивая поддержку авторизации и аутентификации для Web-приложений. В сочетании с .NET Framework и IIS, технология ASP.NET значительно повышает безопасность Web-приложений. Разработчик приложений ASP.NET имеет в своем распоряжении все встроенные средства защиты .NET Framework, такие как CAS, или средства обеспечения безопасности на основе ролей и прав пользователей, поскольку ASP.NET является компонентом .NET Framework.

В этой главе мы вначале рассмотрим механизмы обеспечения безопасности ASP.NET такие, как аутентификация ASP.NET, авторизация[3] ASP.NET и заимствование прав (impersonation) ASP.NET. Мы детально изучим раздел конфигурационного файла, посвященный параметрам безопасности. Аутентификация ASP.NET реализуется на практике при

---

[1] Подробней о концепции Trustworthy Computing см. *http://www.microsoft.com/security/default.mspx и http://www.microsoft.com/mscorp/twc/twc_whitepaper.mspx*.

[2] IIS (Internet Information Server) – Web-сервер корпорации Microsoft. Это программное обеспечение играет важную роль в программных решениях, связанных с обеспечением безопасности. Средства обеспечения безопасности IIS доступны даже тогда, когда средства ASP.NET отсутствуют.

[3] В ASP.NET авторизация осуществляется двумя основными способами: *на основе файлов и на основе URL*.

помощи поставщиков (провайдеров) аутентификации таких, как аутентификация на основе форм, на основе «паспорта» и аутентификация Windows. Мы углубимся в некоторые важные классы, необходимые для реализации функций безопасности в ASP.NET. Также мы рассмотрим примеры программ, иллюстрирующие использование всех этих техник в приложениях ASP.NET.

Базовые механизмы безопасности

При разработке Web-приложений на основе технологии ASP.NET вам необходимо иметь дело с тремя фундаментальными механизмами:

- аутентификация;
- авторизация;
- заимствование прав (impersonation)[1].

Аутентификация: Кто вы?

Аутентификация – это процесс, заключающийся в проверке идентичности клиентского приложения, выполняемой перед тем, как предоставить клиенту доступ к некоторому ресурсу. Например, пользователю или приложению-клиенту необходимо удостоверить свою идентичность при помощи указания своего регистрационного имени и пароля.

Авторизация: Дозволен ли вам доступ к этому ресурсу?

Авторизация определяет, какие именно права доступа к данному ресурсу вам предоставляются. После того как пользователь аутентифицирован, авторизация представляет собой процесс предоставления прав доступа на основе идентификации пользователя. Это следующий после аутентификации этап, на котором определяется, какими правами в отношении ресурса располагает уже идентифицированный пользователь или приложение. Например, авторизация может заключаться в проверке: полными или ограниченными правами обладает клиент.

Заимствование прав: Приложение действует от чьего-то имени

Заимствование прав (impersonation) заключается в том, что некоторые действия выполняются от чьего-то имени (например, клиенту назначается общая учетная запись анонимного пользователя). Web-приложения

[1] Заимствование прав обычно оказывается полезным для реализации добавочной аутентификации и идентификации в отношении некоторых дополнительных ресурсов.

ASP.NET могут обеспечивать анонимный доступ к ресурсам на стороне сервера при помощи механизма заимствования прав, когда анонимные пользователи Web-сайта аутентифицируются при помощи стандартной учетной записи (**IIS_ИмяСервера**). Заимствование прав сильно зависит от характера ресурса. Локальные ресурсы в той же нити выполнения не представляют проблемы. Удаленные ресурсы требуют делегирования прав, что представляет собой расширение механизма заимствования прав и требует использования аутентификационного протокола с поддержкой делегирования, сконфигурированного должным образом (например, Kerberos).

Реализация механизма аутентификации в ASP.NET

Аутентификация является одним из базовых средств обеспечения безопасности в Web-приложениях. Имеется три способа реализации этого механизма в Web-приложениях ASP.NET, отличающихся поставщиком (провайдером) аутентификации[1]: аутентификация при помощи *форм (Forms)*, *паспорта (Passport)* и *аутентификация Windows*. Для выбора поставщика услуг аутентификации для приложения ASP.NET необходимо сконфигурировать **атрибут <mode> элемента <authentication>** *в конфигурационном файле* приложения следующим образом:

```
//конфигурационный файл
<authentication mode = "[Windows/Forms/Passport/None]">
</authentication>
```

Атрибуту mode можно задать одно из следующих значений: *Windows*, *Forms*, *Passport* или *None*. По умолчанию выбирается значение Windows.

- **Forms (Аутентификация при помощи форм).** В этом варианте неаутентифицированные запросы перенаправляются на Web-форму регистрации, где пользователь должен ввести имя и пароль, а затем отправить данные формы на сервер. Если приложение аутентифицирует запрос на основе своего списка или базы данных пользователей, то ASP.NET выпустит файл «cookie» с маркером или ключом для клиента. Затем, с каждым последующим запросом, файл «cookie» передается в заголовке запроса, благодаря чему дальнейшая аутентификация не требуется. Этот метод подходит для коммерческих Web-приложений.
- **Passport (Аутентификация при помощи «паспорта»).** Централизованная аутентификация на основе регистрации в службе Microsoft Passport. В этом варианте неаутентифицированные запросы (например,

[1] Поставщик аутентификации есть не что иное, как программный модуль, содержащий необходимый для выполнения аутентификации код.

запросы новых пользователей) перенаправляются на сайт Microsoft так, что пользователь может единственный раз ввести свое имя и указать пароль, и эти имя и пароль будут идентифицировать его при доступе на многие Web-сайты. Этот метод удобен при работе со многими коммерческими приложениями.

- **Windows (Аутентификация средствами Windows).** В этом случае ASP.NET работает во взаимодействии со схемой аутентификации IIS. В качестве первого шага IIS производит аутентификацию одним из следующих методов: Basic, Digest, Integrated Windows Authentication (прежние названия последнего метода – Windows NT Challenge/Response, NT/CR, NT Lan Manager, NTLM или Certificates. После того как аутентификация средствами IIS будет завершена, на втором шаге ASP.NET использует результат для авторизации доступа. Этот метод хорошо подходит для приложений корпоративной сети типа intranet и корпоративных Web-приложений.
- **None (Нет).** Полное отсутствие аутентификации. Все пользователи рассматриваются, как анонимы. Поскольку ASP.NET не выполняет аутентификацию, службы аутентификации неактивны. Однако следует помнить, что службы аутентификации IIS все еще могут присутствовать. Этот вариант можно использовать в случаях, когда аутентификация вообще не нужна, или же когда вы реализуете собственную схему аутентификации.

Конфигурация ASP.NET

Прежде чем двигаться дальше, давайте посмотрим, как настраиваются параметры ASP.NET. Конфигурационные файлы ASP.NET представляют собой текстовые файлы в формате XML, заключающие в себе иерархию тегов и атрибутов XML. Конфигурационную информацию в XML-файлах легко просматривать и редактировать при помощи текстового редактора или XML-анализатора. Для ASP.NET предусмотрено два конфигурационных файла – конфигурация сервера **(Machine.config)** и конфигурация приложения **(Web.config)**.

- Корневой конфигурационный файл, **Machine.config**, описывает стандартные настройки для Web-сервера в целом. Любая система, где установлена среда .NET Framework, содержит файл **Machine.config**, расположенный в каталоге **%windir%\Microsoft.NET\Framework\<версия>\CONFIG\Machine.config**. Для каждой версии .NET Framework имеется отдельный файл **Machine.config**. Благодаря этому обстоятельству разные версии .NET Framework могут работать, не мешая друг другу.
- Конфигурационный файл **Web.config** определяет параметры конкретного приложения или виртуального каталога. Если файл **Web.config** недоступен, то используются стандартные настройки из

файла **Machine.config**. При наличии файла **Web.config** его настройки имеют приоритет над стандартными настройками **Machine.config**. Настройки файла **Web.config** имеют силу для каталога, в котором он расположен, а также для всех его дочерних каталогов.

Теги, субтеги и атрибуты, которые присутствуют в файлах **Machine.config** и **Web.config**, чувствительны к регистру, и синтаксис тегов должен удовлетворять требованиям «формально-правильного» XML. В конфигурационных файлах могут использоваться кодировки ANSI, UTF-8 и Unicode. Тип кодировки система распознает автоматически. Все конфигурационные настройки в этих файлах заключены между тегами **<configuration>** и **</configuration>**.

Как устроена система конфигурирования ASP.NET и чем она хороша

- Поскольку конфигурационные файлы основаны на синтаксисе XML[1], администраторам и разработчикам легко просматривать и редактировать их при помощи стандартного текстового редактора или XML-анализатора. Это упрощает как локальное, так и удаленное конфигурирование.
- Для того чтобы избежать прямого доступа браузера к конфигурационным файлам, ASP.NET защищает конфигурационные файлы при помощи настроек IIS[2].
- Конфигурационные файлы можно расширять, и, таким образом, можно вводить дополнительные параметры и настройки.
- В конфигурационный файл можно включить новые параметры, при этом для их обработки добавляются соответствующие обработчики.
- Если вы примените новые значения конфигурационных параметров для своих Web-ресурсов, ASP.NET автоматически обнаружит изменения в конфигурационных файлах, обработает их и внесет необходимые коррективы. Нет никакой необходимости в перезагрузке сервера для того, чтобы новые параметры вступили в силу.
- Конфигурационные параметры можно блокировать при помощи тега **<location>** и атрибута **allowOverride**.
- В сервере Web-приложений ASP.NET конфигурационные данные для ресурсов ASP.NET содержатся в нескольких конфигурационных файлах. Эти конфигурационные файлы могут располагаться в нескольких каталогах, но все они называются **Web.config**.
- Набор конфигурационных параметров в файле **Web.config** применяется к его собственному каталогу, а также ко всем его субкаталогам.

---

[1] Подобно тому как настройки приложения в ASP хранятся в метабазе IIS, конфигурационные настройки в ASP.NET хранятся в XML-файлах.

[2] На попытку запросить конфигурационный файл ASP.NET вернет ошибку HTTP 403 (доступ запрещен).

Иерархия конфигурационных параметров

Когда .aspx получает запрос, ASP.NET иерархически вычисляет значения конфигурационных параметров:

- Конфигурационные параметры в файле **Web.config** в подкаталоге приложения имеют более высокий приоритет, чем параметры в файле **Web.config**, находящемся в каталоге приложения.
- Конфигурационные параметры в файле **Web.config** в каталоге приложения имеют более высокий приоритет, чем параметры в файле **Web.config**, находящемся в корневом каталоге Web-сайта.
- Конфигурационные параметры в файле **Web.config** в корневом каталоге имеют более высокий приоритет, чем параметры в файле **Machine.config**.

В таблице 9.1 приведено примерное расположение конфигурационных файлов для некоторого заданного URL – **http://localhost/Myapplication/Mydir/Page.aspx**.

Таблица 9.1. Размещение конфигурационніх файлов

Уровень	Путь
Конфигурационные параметры заданной <**версии**> .NET Framework	%windir%\Microsoft.NET\Framework\<версия>\CONFIG\Machine.config
Параметры Web-сайта	Inetpub\wwwroot\Web.config
Параметры приложения	Inetpub\wwwroot\Myapplication\Web.config
Параметры подкаталога	Inetpub\wwwroot\Myapplication\Mydir\Web.config

Файл **Web.config** на любом уровне может отсутствовать, но файл **Machine.config** обязателен. Вначале система конфигурирования считывает файл Machine.config. ASP.NET определяет параметры для ресурсов иерархическим образом. ASP.NET последовательно, по нисходящей, применяет значения параметров в соответствии со всем конфигурационными файлами, обнаруженными в виртуальном каталоге. Параметры из файлов, находящихся в подкаталогах добавляются к параметрам, полученным в родительских каталогах. Самый последний конфигурационный параметр, найденный в самом «низколежащем» подкаталоге, переопределяет значение этого же параметра, которое было ранее получено в каталогах более высокого уровня.

Давайте рассмотрим следующий URL: **http://localhost/Myapplication/Dir1/Dir2/Myresource.aspx**.

Здесь Myapplication – виртуальный каталог приложения. Если сервер получит запрос по вышеуказанному URL, ASP.NET будет вычислять значения конфигурационных параметров, спускаясь вниз по иерархии каталогов, как показано на рисунке 9.1.

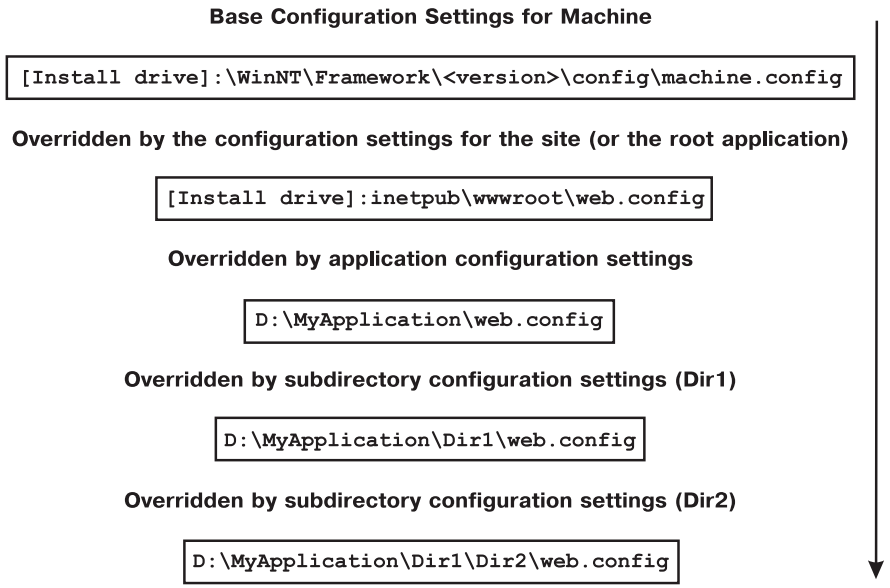


Рис. 9.1. Иерархическое вычисление значений конфигурационных параметров

Система конфигурирования[1] ASP.NET отличается гибкой и расширяемой инфраструктурой, что позволяет настраивать конфигурационные параметры, минимально влияя на Web-приложения и на сервер.

Если вы изменяете значения параметров для некоторых ресурсов, ASP.NET автоматически обнаруживает изменения в конфигурационных файлах и вносит необходимые коррективы. То есть ASP.NET получает оповещения об изменениях в конфигурационных файлах и перезапускает с новыми значениями параметров те приложения, которых данные изменения касаются. Таким образом, нет никакой необходимости перезагружать сервер, для того чтобы изменения в конфигурации вступили в силу. ASP.NET защищает файлы определенных типов (например, конфигурационные файлы и файлы исходного кода) от попыток доступа извне. Эти типы файлов определяются в **Machine.config** при помощи ссылки на обработчик **HttpForbiddenHandler** следующим образом:

```
<add verb="*" path="*.asax"
   type="System.Web.HttpForbiddenHandler" />
<add verb="*" path="*.ascx"
   type="System.Web.HttpForbiddenHandler" />
<add verb="*" path="*.config"
```

[1] Система конфигурирования ASP.NET влияет только на ресурсы ASP.NET, которые зарегистрированы для обработки библиотекой Aspnet_isapi.dll. Например, файлы ASP, HTML, TXT, GIF и JPEG не защищаются параметрами Web.config. Для защиты файлов этих типов их необходимо явным образом зарегистрировать средствами администрирования IIS.

```
   type="System.Web.HttpForbiddenHandler" />
<add verb="*" path="*.cs"
   type="System.Web.HttpForbiddenHandler" />
<add verb="*" path="*.csproj"
   type="System.Web.HttpForbiddenHandler" />
<add verb="*" path="*.vb"
   type="System.Web.HttpForbiddenHandler" />
<add verb="*" path="*.vbproj"
   type="System.Web.HttpForbiddenHandler" />
<add verb="*" path="*.webinfo"
   type="System.Web.HttpForbiddenHandler" />
   ...
```

XML-формат файлов **Machine.config** и **Web.config** идентичен. Давайте рассмотрим общий формат раздела, посвященного безопасности, в файле **Web.config**. Здесь имеется три главных подраздела посвященных аутентификации, авторизации и идентификации.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.web>

   //Раздел аутентификации:
   <authentication mode= "[Windows/Forms/Passport/None]">

      //Атрибуты формы:
      <forms name="[имя]" loginUrl="[Url]"
         protection="[All, None, Encryption, Validation]"
         timeout="[время в минутах]" path="[путь]" >

         //Атрибуты мандата (регистрационных данных
         //пользователя):
         <credentials passwordFormat="[Clear, SHA1, MD5]">
           <user name="[ИмяПользователя]" password="[пароль]"/>
         </credentials>
      </forms>
   </authentication>


            //Атрибуты паспорта:
            <passport redirectUrl="Internal" />


   //Раздел авторизации:
   <authorization>
   <allow users="*" /> <!—Разрешить всем -->
   <!-- <allow    users="[список пользователей через запятую]"
                          roles="[список ролей через запятую]"/>
```

```
   <deny     users=" [список пользователей через запятую]"
                          roles="[список ролей через запятую]"/>
   -->
   </authorization>


   //Раздел атрибутов идентификации:
   <identity impersonate="[true/false]" />
</system.web>
</configuration>
```

Описание

Вся конфигурационная информация размещается между корневыми XML-тегами <**configuration**> и </**configuration**>. Тег <**system.web**> представляет параметры класса ASP.NET.

АУТЕНТИФИКАЦИЯ

- Этот раздел задает политику аутентификации для приложения. Имена допустимых режимов: "**Windows**", "**Forms**", "**Passport**" и "**None**".
- По умолчанию выбран режим <**authentication mode="Windows**">.
- Режим аутентификации нельзя задать на уровне ниже уровня корневого каталога приложения.

АТРИБУТЫ ФОРМЫ

При использовании аутентификации на основе формы вы можете задать имя файла «cookie», тип защиты, URL страницы с формой регистрации, время действительности файла «cookie» и путь для выпущенного файла «cookie».

- **name="[имя файла cookie]":** Имя файла «cookie», используемого для аутентификации при помощи формы. По умолчанию принято <**forms name=".ASPXAUTH**">. Если аутентификацию при помощи формы на одной и той же машине одновременно используют несколько приложений, то следует использовать уникальные имена файлов «cookie».
- **loginUrl="[url]":** Страница с формой регистрации, куда перенаправляется запрос пользователя, если он еще не был аутентифицирован. По умолчанию принято <**forms loginUrl="default.aspx**">. Перенаправление может осуществляться как в пределах одной машины, так и на другой сервер. Если речь идет о перенаправлении на другой компьютер, то на обеих машинах должны использоваться одинаковые значения атрибута **decryptionkey**.
- **protection="[All|None|Encryption|Validation]":** Режим защиты (например, тип шифрования или верификации для файла «cookie»).

По умолчанию принято **<forms protection="All">**, что подразумевает как шифрование, так и верификацию. Возможные значения этого атрибута описаны в таблице 9.2.

Таблица 9.2. Варианты значения атрибута Protection

Значение	Описание
All	Указывает, что данные в файле «cookie» должны быть защищены как шифрованием, так и верификацией. При этом используется алгоритм верификации, определенный в конфигурации системы (элемент **<machineKey>**). Если доступен алгоритм Tripple DES, и длина ключа достигает 48 двоичных разрядов, его можно использовать и для шифрования. Это рекомендованное значение по умолчанию
None	В этом варианте достигается наилучшая производительность .NET Framework. Используется на сайтах, где файлы «cookie» применяются исключительно для учета общей персональной информации. При использовании этого метода помните, что для файлов «cookie» не выполняется ни шифрование, ни верификация
Encryption	Файлы «cookie» можно шифровать при помощи алгоритмов DES или Tripple DES, которые мы обсуждали в предыдущих главах. Однако верификация не выполняется, и, следовательно, файлы «cookie» остаются подверженными атакам на уровне открытого текста
Validation	Данные в файлах «cookie» не шифруются, однако выполняется их верификация

- **timeout="[число минут]":** Продолжительность времени (целое число минут), в течение которого файл «cookie» остается действительным. Эта величина сбрасывается при каждом запросе. По умолчанию принято **<forms timeout="30">**.
- **path="/":** Задает путь к месту хранения файла «cookie» на машине пользователя. По умолчанию используется значение "/". Рекомендуется использовать именно это значение, чтобы избежать трудностей, связанных несовпадением регистра символов, поскольку браузеры учитывают регистр символов в пути, и несовпадение регистра может привести к ошибке при отправке файла «cookie».

АТРИБУТЫ МАНДАТА

Элемент **<credentials>** позволяет сохранить список пользователей в файле **Web.config**. Вы можете также реализовать собственную схему проверки пароля с использованием внешнего источника (например, базы данных) для выполнения верификации по собственной схеме.

- **passwordFormat="[Clear|SHA1|MD5]":** Формат пользовательского пароля, хранимого в субтеге **<user>** (см. табл. 9.3).
- Субтег **<credentials>** поддерживает один атрибут и один субтег (см. табл. 9.4).

- Субтег **<user>** поддерживает два атрибута (см. табл. 9.5).
- Значение **<credentials passwordFormat= "SHA1">** выбрано по умолчанию.

Таблица 9.3. Значения атрибута passwordFormat

Атрибут	Значение	Описание
password Format		Определяет формат шифрования для хранения паролей
	Clear	Пароли не шифруются
	MD5	Пароли шифруются при помощи хеш-алгоритма MD5
	SHA1	Пароли шифруются при помощи хеш-алгоритма SHA-1

Таблица 9.4. Субтег субтега <credentials>

Субтег	Описание
<user>	Позволяет размещать имена и пароли пользователей в конфигурационном файле

Таблица 9.5. Два атрибута субтега <user>

Атрибут	Описание
Name	Регистрационное имя пользователя
Password	Пароль пользователя

АТРИБУТЫ ПАСПОРТА

Атрибут паспорта **redirectUrl=[Url]** задает URL страницы, куда следует перенаправить запрос, если запрошенная страница требует аутентификации или если пользователь еще не зарегистрировал свой «паспорт».

АВТОРИЗАЦИЯ

В этом разделе определяется политика авторизации для приложения. Вы можете запретить или разрешить доступ к определенным ресурсам приложения для пользователя или для роли. Возможно использование символов шаблона, например, **«*»** означает «все пользователи», а **«?»** означает «анонимный» (не аутентифицированный пользователь).

АТРИБУТЫ ИДЕНТИФИКАЦИИ

Атрибут **impersonate="[true|false]"** разрешает или запрещает заимствование прав для источника запроса (например, использование прав локального пользователя Windows).

Аутентификация при помощи формы

Аутентификация при помощи формы является одной из служб аутентификации ASP.NET. Web-приложения могут пользоваться ею для индивидуальной идентификации регистрирующихся в приложении пользователей и верификации их паролей. Она широко используется на Web-сайтах для реализации такой аутентификации пользователей, когда при регистрации пользователя создается файл «cookie», который сопровождает его при перемещении по всему сайту.

В этой схеме аутентификации пользователи автоматически перенаправляются на страницу с формой регистрации, где он должны предъявить регистрационные данные, например, ввести свое имя и пароль. Код приложения верифицирует введенные данные. Если все в порядке, ASP.NET выпускает файл «cookie» или маркер и перенаправляет пользователя на ту страницу, к которой он первоначально обратился. В противном случае пользователь перенаправляется вновь на страницу регистрации и получает сообщение, что его имя и пароль недействительны. Такая техника аутентификации весьма популярна и используется на многих Web-сайтах. Аутентификацию при помощи форм часто применяют для «персонализации» ресурсов, то есть для настройки вида страниц по желанию для каждого зарегистрированного пользователя.

Первоначально сервер выпускает файл «cookie»[1], содержащий небольшую порцию данных, и передает его клиенту. При последующих запросах клиент каждый раз отправляет этот файл на сервер, доказывая тем самым, что он уже аутентифицирован. В следующем примере мы увидим, как создается простое ASP.NET-приложение, реализующее аутентификацию при помощи форм.

Имеется три файла: **default.aspx**, **login.aspx** и **Web.config**. Первоначально пользователь запрашивает защищенную страницу **default.aspx**. Поскольку пользователь еще не аутентифицирован, его запрос перенаправляется на страницу **login.aspx**, где пользователь должен ввести действительные регистрационное имя и пароль. Если пользователь будет успешно аутентифицирован, то его запрос перенаправляется на запрошенную страницу, то есть на **default.aspx**. Аутентификация с помощью формы использует классы, размещенные в пространстве имен **System.Web.Security**. Следовательно, вы должны включить в свой код это пространство имен.

Для того чтобы реализовать аутентификацию при помощи формы, выполните следующие шаги.

1. В конфигурационном файле **Web.config** задайте режим аутентификации.
2. Создайте Web-форму для регистрации пользователей.

[1] В ASP.NET применяются изощренные алгоритмы хеширования и шифрования, которые предотвращают подделку файлов «cookie».

3. Сохраните регистрационные данные пользователей в базе данных или в файле.
4. Аутентифицируйте пользователя при помощи данных из файла или базы.

В механизме аутентификации при помощи форм вы можете хранить регистрационные данные одним из следующих способов:

- в файле Web.config;
- в специальном XML-файле;
- в базе данных.

Метод 1: хранение регистрационных данных в файле Web.config

При использовании данного метода вся информация, относящаяся к пользователю, хранится в разделе <**credentials**> файла **Web.config**, который располагается в каталоге приложения. Хранение этих данных в файле **Web.config** приемлемо только в случае простой аутентификации. Этот метод непригоден для случаев, когда ваши пользователи будут сами создавать свои учетные записи. Тогда необходимо будет хранить имена и шифрованные пароли в отдельном XML-файле или в базе данных. Создайте Web-приложение ASP.NET при помощи пакета Visual Studio.NET, переименуйте веб-страницу в **default.aspx** и добавьте еще одну страницу с именем **login.aspx**. Затем сконфигурируйте файл **Web.config** так, как показано в следующем разделе.

КОНФИГУРИРОВАНИЕ РАЗДЕЛОВ <AUTHENTICATION> И <AUTHORIZATION> В ФАЙЛЕ WEB.CONFIG

Вам необходимо сконфигурировать файл **Web.config** так, как показано ниже, и поместить его в корневой каталог приложения, то есть каталог, где находится файл **default.aspx**.

```
<configuration>
   <system.web>

//Выбрать режим аутентификации: формы
   <authentication mode="Forms">
//Задать атрибуты формы
     <forms name=".ASPXFormAuth" loginUrl="login.aspx"
             protection="All" timeout="15" path="/">
//Записать идентификаторы пользователей и пароли
//в раздел <credentials>
             <credentials passwordFormat="Clear">
                    <user name="Arun" password="Spiritual" />
                    <user name="Ganesh" password="Divine" />
             </credentials>
```

```
      </forms>
    </authentication>
    <authorization>
//Запретить вход неаутентифицированных пользователей
              <deny users="?" />
    </authorization>
    <globalization requestEncoding="UTF-8"
              responseEncoding="UTF-8" />
    </system.web>
</configuration>
```

В раздел <**credentials**> мы записали действительные имена пользователей и пароли. Через 15 минут файл «cookie» устареет, поскольку мы задали таймаут равным 15. Таким образом, файл «cookie» будет генерироваться заново каждые 15 минут, что уменьшит вероятность его «похищения» другим пользователем.

ФАЙЛ LOGIN.ASPX

Если пользователь еще не аутентифицирован, его запрос будет переадресован файлу **login.aspx**. Это Web-форма, чье имя определено в элементе <**forms**> файла **Web.config**. Импортируйте пространство имен **System.Web.Security**. В режиме конструктора добавьте два текстовых поля **(Userid** и **Passid)**, а также кнопку **Submit**, как показано на рисунке 9.2. Исходный код файла **login.aspx** приведен ниже.

```
private void Button1_Click(object sender, System.EventArgs e)
{
/* public static bool Authenticate(string name, string password);
              Аутентификация Userid и Passid */
     if (FormsAuthentication.Authenticate(Userid.Text, Passid.Text))
```

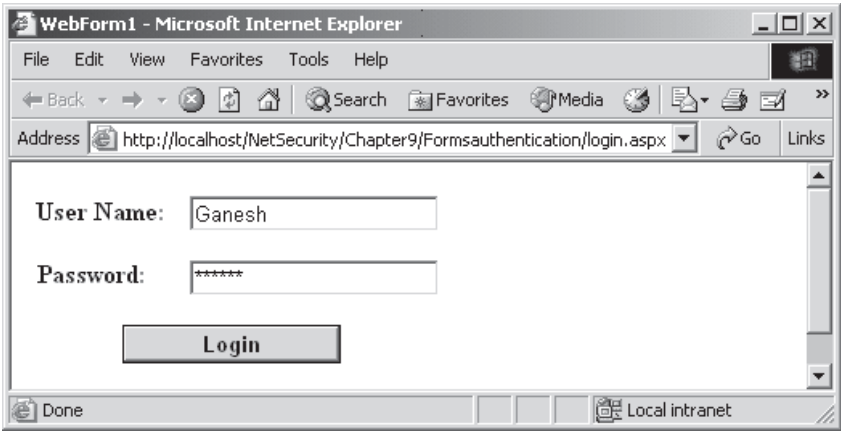


Рис. 9.2. Страница Login.aspx, где пользователь должен ввести свои регистрационные данные

```
    {
            FormsAuthentication.RedirectFromLoginPage(Userid.Text,
                  false);
            this.Label3.Text = "You are authenticated!";
    }
    else
    {
     // Очистить поле пароля
            Passid.text="";
            Label3.Text = "Недействительный пароль и/или имя!";
    }
}
```

Здесь код **FormsAuthentication.Authenticate(Userid.Text, Passid.Text)** получает имя пользователя и пароль, а возвращает булево значение true, если эти регистрационные данные действительны. В противном случае вернется значение false. Затем этот код создает файл «cookie» с регистрационными данными, присоединяет его к исходному запросу и направляет его на первоначально запрошенную страницу при помощи кода **FormsAuthentication.RedirectFromLoginPage(Userid.Text, false).** Здесь второй параметр указывает, должна ли аутентификация осуществляться при помощи сеансового файла «cookie» (false) или при помощи постоянного файла «cookie» (true). В данном случае мы используем сеансовый файл.

ФАЙЛ DEFAULT.ASPX

Файл **default.aspx** представляет собой тот ресурс, который первоначально запрашивает пользователь. На странице **default.aspx** мы просто отображаем приветственное сообщение и кнопку **Sign out**, как показано на рисунке 9.3. При помощи метода **FormsAuthentication.SignOut** вы можете легко отвергнуть полученный файл «cookie». Исходный код страницы **default.aspx** приведен ниже.

```
private void Page_Load(object sender, System.EventArgs e)
{
/* Обратите внимание, приведенный ниже код допускает возможность
кросс-сайтовой скриптовой атаки, если данные, введенные
пользователем, не проверить предварительно. Таким образом,
необходимо обеспечить еще верификацию введенных данных.*/
   Label1.Text=  "Welcome, "+ User.Identity.Name+"!"+
                 "You have successfully logged in.";
}
private void SignOut_Click(object sender, System.EventArgs e)
{
   FormsAuthentication.SignOut();
   Response.Redirect("login.aspx");

}
```

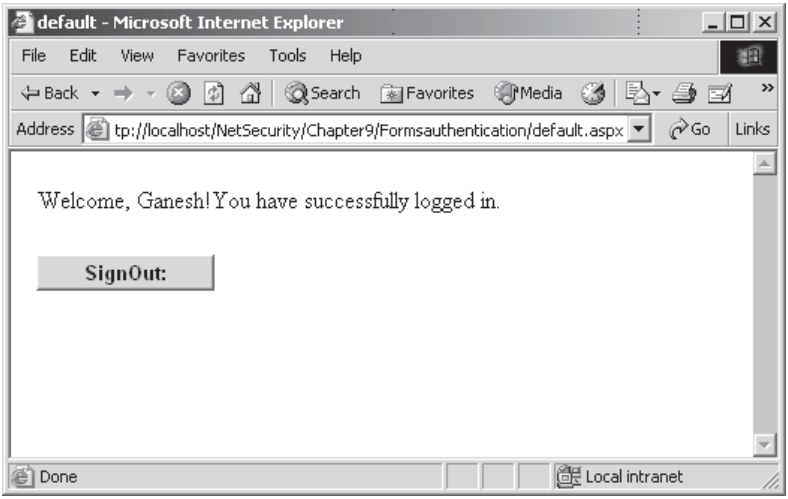


Рис. 9.3. Приветствие, отображаемое на странице default.aspx после аутентификации пользователя

ЗАЩИТА ПАРОЛЕЙ ШИФРОВАНИЕМ

Вообще говоря, не рекомендуется хранить пароли в виде открытого текста. Хотя пользователь и не имеет прямого доступа к файлу Web.config, в ситуации, когда сервер доступен по локальной сети, всегда есть риск несанкционированного доступа к этому файлу. По этой причине для хранения имен и паролей в базе данных или файле Web.config вам следует зашифровать их при помощи метода HashPasswordForStoringInConfigFile класса FormsAuthentication. Этот метод использует алгоритмы хеширования SHA-1 или MD5:

```
Passid =
FormsAuthentication.HashPasswordForStoringInConfigFile
(Passid.text, "SHA1");
```

Например, хеш для пароля "Divine" по алгоритму SHA-1 будет равен C1FF7FB589DDC7CECD412F515709D8DC2B2B2C22.

Также вы можете защитить конфиденциальную информацию в файле **Web.config** при помощи программного интерфейса Data Protection API (DPAPI). Подробней использование DPAPI в приложениях ASP.NET для шифрования чувствительных данных см. *http://msdn2.microsoft.com/en-us/aa302403.aspx*.

Метод 2: хранение регистрационных данных в XML-файле

При использовании этого метода вся пользовательская информация хранится в специальном XML-файле. Большое число имен и паролей хранить файле **Web.config** непрактично. В этом случае лучше разместить эту информацию в отдельном файле или в базе данных. При этом раздела <**credentials**> в файле **Web.config** не будет. Файл **Web.config** будет выглядеть следующим образом:

```
//Выбрать режим аутентификации: формы
   <authentication mode="Forms">
//Задать атрибуты формы
     <forms name=".ASPXFormAuthxml" loginUrl="login.aspx"
              protection="All" timeout="60" />
   </authentication>
   <authorization>
//Запретить вход неаутентифицированных пользователей
              <deny users="?" />
   </authorization>
```

Как мы уже говорили ранее, ASP.NET верифицирует каждый запрос на предмет аутентификации. Если запрос не аутентифицирован, он перенаправляется на страницу **login.aspx**. Там клиент должен будет ввести надлежащие регистрационные данные. Страница **login.aspx** проверяет введенные данные на соответствие списку, хранящемуся в файле Users.xml. Если введенные имя и пароль имеются в файле, страница **login.aspx** перенаправит запрос на первоначально запрошенную страницу **default.aspx**. В противном случае запрос будет перенаправлен на другую страницу, где введенные имя и пароль будут записаны в файл Users.xml.

Файл Users.xml

Файл **Users.xml**, заключающий в себе регистрационные данные пользователей, приведен ниже. Поле **UserPassword** зашифровано при помощи метода **HashPasswordForStoringInConfigFile** класса **FormsAuthentication** (использован алгоритм SHA-1).

```
<Users>
   <Users>
<UserID>G.A.Gnanavel</UserID>
   <UserPassword>
44DD5C3AA9E4E693A9E394DAA5D3F3A449DC25CA</UserPassword>
   </Users>
   <Users>
<UserID>G.N.Vadivambal</UserID>
   <UserPassword>
```

```
FE71C5444F9327DAA2F1D69B7510ABF59B80672F</UserPassword>
   </Users>
   <Users>
<UserID>G.N.Vadivambal</UserID>
   <UserPassword>
CF77B1B76919D5D2B8B79D63FCA5E70383A3D7E5</UserPassword>
<UserID>G.N.Vadivambal</UserID>
   <UserPassword>
FE71C5444F9327DAA2F1D69B7510ABF59B80672F</UserPassword>
   </Users>
</Users>
```

Файл login.aspx

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.Security;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.IO;

namespace Formsauthenticationxml
{
   public class login : System.Web.UI.Page
   {
            protected System.Web.UI.WebControls.TextBox Passid;
            protected System.Web.UI.WebControls.Button Button1;
            protected System.Web.UI.WebControls.Label Label1;
            protected System.Web.UI.WebControls.Label Label2;
            protected System.Web.UI.WebControls.Label Label3;
            protected System.Web.UI.WebControls.TextBox Userid;


   public login()
            {
              Page.Init += new System.EventHandler(Page_Init);
            }

   private void Page_Load(object sender, System.EventArgs e)
            {
            }

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

```
            InitializeComponent();
            }

            #region
            //Код, сгенерированный Web Form Designer
            #endregion


  private void Button1_Click(object sender, System.EventArgs e)
    {
     /* Зашифровать (хешировать) пароль, введенный пользователем,
        для сравнения с хешем, находящемся в XML-файле.*/
       string Passidvalue =
          FormsAuthentication.HashPasswordForStoringInConfigFile(
            Passid.Text,"SHA1");
                  String str = "UserID='" + Userid.Text+ "'";
                  DataSet ds = new DataSet();
                  FileStream fs =
                    new FileStream(Server.MapPath("Users.xml"),
                              FileMode.Open,FileAccess.Read);
                  StreamReader reader = new StreamReader(fs);
                  ds.ReadXml(reader);
                  fs.Close();
                  DataTable clients = ds.Tables[0];
                  DataRow[] items = clients.Select(str);
                  if( items != null && items.Length > 0 )
                  {
                    DataRow row = items[0];
                    String pass = (String)row["UserPassword"];
                    if (pass == Passidvalue)
                     FormsAuthentication.RedirectFromLoginPage(
                         Userid.Text, false);
                    else
                     Label3.Text =
                       "Пожалуйста, введите правильный пароль!";
                  }
                  else
                  {
                    Label3.Text = "Неверный пароль и/или имя!";
                         /* Запрос перенаправляется на другую
                         Страницу, где введенные имя и пароль
                         Будут добавлены в XML-файл.*/
                    Response.Redirect(
                         "adduser/adduser.aspx?UserID ="+
                         Userid.Text);
                  }
            }
  }
}
```

Мы прочитали XML-файл, в котором хранятся регистрационные данные пользователей и поместили эти данные в объект **DataSet ds**. Затем мы проверяем введенное имя на соответствие элементам списка. Если в XML-файле имеются какие-либо имена, мы сохраняем их в именованных элементах **DataRow**. Затем мы проверяем, соответствует ли введенный пароль шифрованному паролю из XML-файла. Если пароли совпали, мы перенаправляем запрос на первоначально запрошенную страницу **default.aspx**. В противном случае запрос направляется на другую страницу, где введенные имя и пароль будут занесены в XML-файл. Однако в приложениях реального времени рекомендуется хранить пользовательские данные в реляционной базе данных.

Метод 3: хранение регистрационных данных в базе данных

В этом случае все регистрационные данные хранятся в базе данных. Файлы **default.aspx** и **Web.config** здесь аналогичны тем, что мы видели выше. Единственная разница заключается в странице **login.aspx**. Исходный код метода **LoginBtn_Click** из этого файла выглядит следующим образом:

```
// Приведенный ниже код предназначен только для учебных целей
   private void LoginBtn_Click(object sender,
               System.EventArgs e)
   {
     /* SqlConnection conn =
              new SqlConnection ("Server=(local);" +
              "Integrated Security = SSPI;" +
              "Database=login");
     */
     OleDbConnection conn = new OleDbConnection(
       "Provider=Microsoft.Jet.OLEDB.4.0;Data Source="+
       "C:\\NetSecurity\\Chapter9\\Formsauthdatabase\\login.mdb");
     try
     {
              conn.Open();
                    /* В приведенном ниже коде, возможно,
                    потребуется защита на уровне SQL, на
                    случай когда введенные пользователем
                    данные преднамеренно некорректны.*/

              String str = "select count (*) from login where
UserID= '" +Userid.Text+ "' and Password= '" + Passid.Text + "' ";

     /* SqlCommand command = new SqlCommand(str, conn)*/

     OleDbCommand command = new OleDbCommand(str, conn);
     int count = (int)command.ExecuteScalar();
```

```
    if (count!=0)
      FormsAuthentication.RedirectFromLoginPage(Userid.Text,
              false);
    else
      Label3.Text = "Пожалуйста, введите действительный пароль!";
    }
    finally
    {
                    conn.Close();
    }
}
```

Проверяя регистрационные данные при помощи базы данных, учитывайте два обстоятельства:

- храните в базе односторонние хеши (со случайным числом «salt»);
- принимайте меры против несанкционированного ввода пользователем SQL-кода.

В приложениях реального времени не рекомендуется хранить в базе данных пароли, будь то пароли в виде открытого текста или зашифрованные. Проблема с шифрованием паролей заключается в том, что трудно сохранить в секрете ключ шифрования. А если атакующая сторона завладеет ключом, то сможет расшифровать все пароли, имеющиеся в базе. Поэтому наилучшим методом хранения паролей является одностороннее хеширование (сочетание хеша пароля со случайным числом salt)[1]. Такое число salt вы можете создать следующим образом.

```
public static string GenerateSalt(int size)
{
/* Класс RNGCryptoServiceProvider в пространстве имен
    System.Security.Cryptography обеспечивает
    генерацию случайных чисел. */

  RNGCryptoServiceProvider crypto = new RNGCryptoServiceProvider;
  byte[] buff = new byte[size];
  Crypto.GetBytes(buff);
  return Convert.ToBase64String(buff);
}
```

Создать хеш из заданного пароля и полученного числа salt вы можете следующим образом:

```
public static string GeneratePasswordHash(string passid,
             string salt)
   {
     string saltpassid = string.Concat(passid, salt);
```

[1] Под числом salt имеется в виду криптографически сильное случайное число.

```
    string password =
    FormsAuthentication.HashPasswordForStoringInConfigFile(
             saltpassid. "SHA1");
    return password;
}
```

НЕСАНКЦИОНИРОВАННАЯ ВСТАВКА ТЕКСТА В SQL-КОД

Вставка в SQL подразумевает, что пользователь имеет возможность передать произвольный добавочный (возможно, злонамеренный) SQL-код, который окажется присоединенным к нормальном у SQL-коду без вашего ведома. Например, рассмотрим следующий фрагмент.

```
String str = "select count (*) from login where UserID= '"
+Userid.Text+ "' and Password= '" + Passid.Text + "' ";
```

Если злоумышленник введет следующий текст пароля, то есть значение Passid.Text,

```
' ; Любой злонамеренный SQL-код
```

то вышеприведенный фрагмент программы выполнит злонамеренный SQL-код, поскольку символ ' (одинарная кавычка) указывает, что строчный литерал окончен, а символ ; (точка с запятой) задает начало нового оператора.

Для предотвращения подобных атак учитывайте следующие обстоятельства:

- введенный пользователем текст необходимо обязательно верифицировать, например, вы можете ограничивать размер строки и тип символов;
- выполняйте SQL-код от имени учетной записи с минимальными правами;
- при построении SQL-операторов используйте семейство **Parameters** следующим образом:

```
SqlDataAdapter myCommand = new SqlDataAdapter (
  "Select * From login WHERE UserID = @userid", myConn);
SqlParameter parm =
  myCommand.SelectCommand.Parameters.Add("@userid",
  SqlDbType.VarChar, 15);
parm.Value= Userid.Text;
```

Классы для аутентификации при помощи форм

Пространство имен **System.Web.Security** содержит все классы, используемые защитными функциями ASP.NET в серверных Web-приложениях. В таблице 9.6 приведены важные классы .NET Framework, необходимые для аутентификации при помощи форм.

Таблица 9.6. Классы .NET Framework для авторизации при помощи форм

Класс	Описание
FormsAuthenticationModule	Позволяет приложению ASP.NET использовать аутентификацию при помощи форм
FormsAuthentication	Содержит статические методы, позволяющие манипулировать «билетами» аутентификации
FormsAuthenticationTicket	Содержит информацию для аутентификационного файла «cookie»
FormsIdentity	Класс, производный от **Identity**. Используется классом **FormsAuthenticationModule**, позволяет приложению получить информацию из «cookie»
FormsAuthenticationEventArgs	Содержит данные для события **FormsAuthentication_OnAuthenticate**

Публичные статические (разделяемые) свойства, определенные классом **FormsAuthentication**, приведены в таблице 9.7.

Таблица 9.7. Публичные статические (разделяемые) свойства, определенные классом FormsAuthentication

Свойство	Описание
FormsCookieName	Возвращает имя файла «cookie», используемого для текущего приложения
FormsCookiePath	Возвращает путь к файлу «cookie», используемому для текущего приложения

Публичные статические (разделяемые) методы, определенные классом **FormsAuthentication**, приведены в таблице 9.8.

Публичные свойства экземпляра, определенные классом **FormsAuthenticationTicket**, приведены в таблице 9.9.

Публичные свойства экземпляра, определенные классом FormsIdentity, приведены в таблице 9.10.

Таблица 9.8. Публичные статические (разделяемые) методы, определенные классом FormsAuthentication

Метод	Свойство
Authenticate	Верифицирует представленные пользователем регистрационные данные при помощи данных, хранящихся в файле или базе данных. Возвращает true, если имя и пароль действительны, и false – в противном случае
Decrypt	Возвращает объект FormsAuthenticationTicket, содержащий зашифрованный аутентификационный «билет» из полученного по HTTP файла «cookie»
Encrypt	Возвращает строку, содержащую зашифрованный аутентификационный «билет», пригодный для помещения в файл «cookie» и передачи по HTTP
GetAuthCookie	Перегруженный метод. Создает файл «cookie» с заданным именем
GetRedirectUrl	Возвращает URL для перенаправления исходного запроса, который был перенаправлен на страницу logon. Если такового нет, использует default.aspx
HashPasswordFor-StoringInConfigFile	На основе заданного пароля создает хеш, пригодный для хранения в конфигурационном файле. Тип хеша также задается. (Хеш пароля не содержит числа «salt», его необходимо добавить самостоятельно). Поддерживаются алгоритмы хеширования SHA-1 и MD5
Initialize	Инициализирует объект FormsAuthentication чтением конфигурации и получением значений «cookie» и ключей шифрования для заданного приложения
RedirectFromLoginPage	Перегруженный метод. Перенаправляет запрос аутентифицированного пользователя назад, на первоначально запрошенный URL
SetAuthCookie	Перегруженный метод. Не выполняет перенаправление, но создает аутентификационный «билет» для заданного имени пользователя и присоединяет его к семейству «cookie» исходящего ответа сервера
SignOut	Удаляет аутентификационный «билет», выполняя SetForms с пустым значением для заданного аутентифицированного пользователя. Этим удаляются, как сеансовые, так и постоянные файлы «cookie»

Таблица 9.9. Публичные свойства экземпляра, определенные классом FormsAuthenticationTicket

Свойство	Описание
CookiePath	Возвращает путь, по которому был создан файл «cookie»
Expiration	Возвращает дату и время, до которых файл «cookie» действителен
Expired	Возвращает true, если срок действия файла «cookie» истек
IsPersistent	Возвращает true, если был выпущен постоянный файл «cookie». В противном случае срок действия файла определяет браузер
IssueDate	Возвращает дату и время выпуска файла «cookie». Эта возможность используется для создания собственных схем управления сроком действительности
Name	Возвращает имя пользователя, ассоциированное с файлом «cookie». Сохраняется максимум 32 байта
UserData	Возвращает определенную приложением строку, которую можно поместить в файл «cookie»
Version	Возвращает байт номера версии (для использования в будущем)

Таблица 9.10. Публичные свойства экземпляра, определенные классом FormsIdentity

Свойство	Описание
AuthenticationType	Тип аутентификации (в данном случае – «Forms»)
IsAuthenticated	Указывает, имела ли место аутентификация
Name	Имя (в данном случае – имя пользователя)
Ticket	Возвращает объект FormsAuthenticationTicket, связанный с текущим запросом

Аутентификация при помощи паспорта

Аутентификация при помощи паспорта представляет собой централизованную аутентификационную службу с однократной регистрацией от компании Microsoft. В настоящее время большое число Internet-пользователей прибегает к услугам таких служб Microsoft, как MSN или HotMail. В процессе регистрации они передают свои профили в эти службы Microsoft. Реальный выигрыш состоит в том, что вы можете использовать пользовательские профили в своем Web-сайте, если реализуете в нем

аутентификацию при помощи паспорта. При этом информация о пользователе доступна вашему приложению через профиль, который хранится в службе Microsoft. Многие компании такие, как McAfee или eBay, используют такую аутентификацию на своих сайтах. Удобство для пользователя состоит в том, что ему не нужно помнить имена и пароли для многих сайтов, и все свои регистрационные данные он может хранить в единственном месте. Служба .NET Passport[1] обеспечивает пользователям принцип однократной регистрации и защищает всю информацию при помощи таких крипто-технологий, как Secure Sockets Layer (SSL) и алгоритм 3DES. На рисунке 9.4 приведен вид стартовой страницы службы Microsoft .NET Passport.

Рис. 9.4. Стартовая страница службы Microsoft .NET Passport

Служба Passport является аутентификационной службой, основанной на формах. Когда пользователь присылает запрос (речь идет о запросе GET по протоколу HTTP) к защищаемому ресурсу, ASP.NET проверяет, есть ли у пользователя действительный «паспорт». Если нет, пользователь перенаправляется для регистрации на сайт службы Passport Logon Service, где он должен будет ввести свои регистрационные данные (адрес e-mail и пароль), как показано на рисунке 9.5. Если веденные данные корректны, служба Passport Logon Service перенаправляет пользователя к запрошенному им ресурсу. В противном случае пользователь должен будет зарегистрироваться.

Объект **PassportAuthenticationModule** обеспечивает оболочку для инструментального программного обеспечения Passport для ASP.NET-приложений. Он обеспечивает службу аутентификации и информацию о профиле, полученную из класса **PassportIdentity**, производного от **Identity**. Вам нужно будет зарегистрировать свой сайт в службе Passport, принять лицензионное соглашение и установить у себя инструментальный пакет для реализации «паспортной» аутентификации.

[1] Компания Microsoft запустила Web-службы Microsoft .NET Passport и Service-to-Consumer (S2C) в 1999 году.

Рис. 9.5. Регистрационная форма .NET Passport Sign, где пользователь должен указать свой адрес e-mail и пароль

Общая процедура реализации «паспортной» аутентификации в приложении ASP.NET выглядит следующим образом:

- Зарегистрируйте свой сайт в службе Passport.
- Загрузите, установите и сконфигурируйте инструментальный пакет Passport SDK с сайта загрузки Microsoft или с сайта *http://www.passport.com*. Для использования службы на Web-сайтах в реальном времени вам потребуется зарегистрироваться и получить ключ, за который компания Microsoft взимает лицензионный сбор[1]. Для аутентификации пользователь перенаправляется на сайт *http://login.passport.com*.
- Во время тестирования сайта вы не можете использовать рабочие учетные записи Passport для регистрации. Вместо этого, в целях тестирования, вы должны завести специальную запись типа PREP Passport[2]. Тестовая среда PREP позволяет сайтам подтверждать тестовые запросы к .NET Passport без реального доступа к учетным записям и регистрационным данным пользователей .NET Passport.

[1] Имеется два варианта оплаты лицензионного сбора .NET Passport: периодическая договорная плата за тестирование в размере 1500 USD и ежегодная плата в размере 10000 USD, начисляемая на каждую компанию.

[2] Не используйте ваш реальный пароль для тестирования с помощью PREP. Система PREP представляет собой отдельную тестовую среду, в которой содержатся только тестовые данные, поэтому реальные производственные пароли хранить в ней нельзя.

Регистрацию с целью тестирования можно осуществить по адресу *http://current-login.passporttest.com*.

- Сконфигурируйте свой файл **Web.config** следующим образом:

```
<authentication mode="Passport">
<passport redirectURL="login.aspx" />
</authentication>
```

После установки пакета .NET Passport SDK и конфигурирования **Web.config** вы сможете обратиться к классу **PassportIdentity** через посредство интерфейса **Identity**, который он реализует. Следующий код показывает, каким образом можно получить экземпляр объекта **PassportIdentity**.

```
using System.Web.Security;
...
public class WebForm1 : System.Web.UI.Page
{
   public PassportIdentity pass;

   private void Page_Load(object sender, System.EventArgs e)
   {
     pass = (PassportIdentity)User.Identity
   }
}
```

Располагая экземпляром объекта **PassportIdentity**, вы можете получить специфичную для .NET Passport функциональность через посредство класса **PassportIdentity**.

Давайте обсудим теперь, как отображать кнопки входа в регистрацию и выхода из нее. В общем случае вы видите кнопки **.NET Passport Sign In** или **.NET Passport Sign Out** в правом верхнем углу страницы, и кнопки эти выглядят подобно изображенным на рисунке 9.6.

Рис. 9.6. Кнопки Sing In и Sing Out в Microsoft .NET Passport

Метод **LogoTag2** объекта **PassportIdentity** возвращает фрагмент HTML, заключающий в себе тег **<img>** со ссылкой на Microsoft .NET Passport. Эта ссылка отображает либо **Sign Out** (выйти, если обнаружены действительные регистрационные данные), либо **Sign In** (войти, если таковых не нашлось). Следующий код иллюстрирует использование метода **LogoTag2** на страницах ASP.NET.

```
using System.Web.Security;
...
public class WebForm1 : System.Web.UI.Page
{
   public PassportIdentity pass;

   private void Page_Load(object sender, System.EventArgs e)
   {
     pass = (PassportIdentity)UserIdentity;
     string returnURL =
              "http://phptr/default.aspx";
     string logo = pass.LogoTag2(
              returnURL, 10000, true, null, 1033, false,
              Context.Request.ServerVariables("SERVER_NAME"),
              0, false);
     Response.Write(logo);
   }
...
}
```

Вот синтаксис обращения к методу **PassportIdentity.LogoTag2:**

```
public string LogoTag2(string strReturn, int iTimeWindow,
  bool fForceLogin, string strCoBrandedArgs, int iLangID,
  bool fSecure, string strNameSpace, int iKPP,
  bool bUseSecureAuth);
```

- Первый параметр метода **LogoTag2**, **[returnURL]**, необязателен. Он задает URL, по которому сервер должен перенаправить запрос пользователя после успешной регистрации в .NET Passport. Если этот параметр отсутствует, то используется значение по умолчанию.
- Второй параметр, **[TimeWindow]**, также необязателен. Он определяет интервал времени в секундах, в течение которого пользователь должен зарегистрироваться. Это должно быть число в диапазоне от 100 до 1 000 000.
- Остальные параметры метода **LogoTag2 – [ForceLogin], [coBrandeArgs], [lang_id], [bSecure], [NameSpace], [KPP]** и **[SecureLevel]** определяют, должен ли пользователь быть принудительно зарегистрирован или нет, URL для ассоциированного изображения, язык страницы регистрации (по умолчанию EN/US), должен ли доступ осуществляться по защищенному соединению SSL, доменное имя запроса, разрешение доступа детям и уровень безопасности.

Три уровня безопасности в .NET Passport описываются в таблице 9.11.

В качестве альтернативы механизму однократной регистрации вы можете использовать такой механизм, в котором диалоговое окно регистрации встроено непосредственно в страницу вашего сайта. Такая встроенная регистрация обладает большой гибкостью. Компактный и стандартный варианты модуля встроенной регистрации показаны на рисунке 9.7.

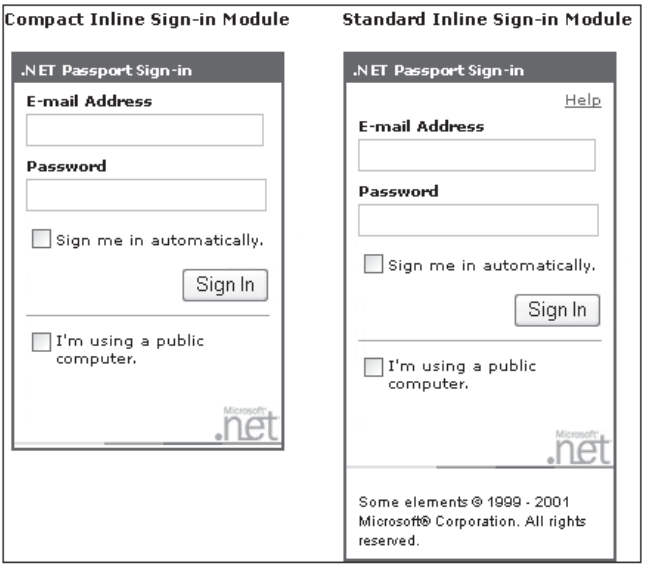


Рис. 9.7. Компактный и стандартный модули встроенной регистрации

Таблица 9.11. Три уровня безопасности для регистрации .NET Passport

Значение SecureLevel	Описание
0 (или не указано)	Регистрируемый идентификатор обрабатывается протоколом HTTP в домене .NET Passport (по умолчанию)
10	Регистрируемый идентификатор обрабатывается протоколом HTTPS в домене .NET Passport. Требуется, чтобы ответный URL был URL по протоколу HTTPS, иначе аутентификация потерпит неудачу
100	Регистрируемый идентификатор обрабатывается протоколом HTTPS в домене .NET Passport, а процесс регистрации требует, кроме пароля, ввода секретного PIN-кода

Также вы можете использовать мобильный механизм регистрации[1] для мобильных устройств.

Значительная выгода от использования аутентификации .NET Passport состоит в том, что, как уже упоминалось, вы можете получить через «паспорт» данные пользовательского профиля такие, как **FirstName** (имя) или **Country** (страна), если пользователь разрешил использование своего профиля при регистрации в .NET Passport. Но идентификатор

[1] Мобильная регистрация обеспечивает многие функциональные возможности, необходимые для поддержки .NET Passport на мобильных устройствах. Пользователи могут создавать себе паспорта на основе телефонного номера или PIN-кода для использования паспорта в мобильных телефонах или «наладонниках».

паспорта пользователя (PUID – Passport User ID) доступен всегда через посредство свойств **Name** и **HexPUID** класса **PassportIdentity.** Вы можете использовать значение PUID в качестве индекса для хранения информации о пользователях на вашем сайте. Из данных профиля можно извлечь различные полезные атрибуты: **FirstName (имя), LastName (фамилия), NickName (прозвище), Gender (пол), Birthdate (дата рождения), PreferredEmail (предпочитаемый адрес e-mail), TimeZone (часовой пояс), Occupation (род занятий)** и **Country (страна).** Как извлекать данные профиля, показывает следующий код.

```
private void Page_Load(object sender, System.EventArgs e)
   {
   if (pi.IsAuthenticated)
              {
                string name = pi["FirstName"];
                  if (name == "")
                  {
                    Label1.Text = "Добро пожаловать!";
                  }
                  else
                  {
                    Label1.Text = "Здравствуйте, " + name + "!";
                  }
   }
```

Вначале мы проверяем, зарегистрирован ли пользователь в .NET Passport, при помощи метода **IsAuthenticated**, который возвращает true, если пользователь зарегистрирован. Поскольку для управления состоянием .NET Passport использует файлы «cookie», вам необходимо создать отдельную страницу для удаления файлов «cookie», которые содержат аутентификационную информацию. Ваша страница регистрации обязательно должна вернуть изображение, оповещающее об успешной регистрации. В приведенном ниже коде удаляются файлы «cookie» и возвращается подходящее GIF-изображение.

```
<%@ Page language="c#" %>
<%
  Response.ContentType = "image/gif";
  Response.Expires = -1
  Response.AddHeader("P3P", "CP=TST");

  HttpCookie Cookie1 =
   new HttpCookie("MSPProf", "");
  Cookie1.Expires = Now();
  Response.Cookies.Add(Cookie1);

  HttpCookie Cookie2 =
   new HttpCookie("MSPAuth", "");
  Cookie2.Expires = Now();
```

```
  Response.Cookies.Add(Cookie2);
  HttpCookie Cookie3 =
   new HttpCookie("MSPSecAuth", "");
  Cookie3.Expires = Now();
  Response.Cookies.Add(Cookie3);
  HttpCookie Cookie4 =
   new HttpCookie("MSPProfC", "");
  Cookie4.Expires = Now();
  Response.Cookies.Add(Cookie4);
  HttpCookie Cookie5 =
   new HttpCookie("MSPConsent", "");
  Cookie5.Expires = Now();
  Response.Cookies.Add(Cookie5);
  Response.WriteFile(
   "/images/signout_Clear.gif");
%>
```

Мы очищаем здесь файлы «cookie» **MSPProf**, **MSPAuth**, **MSPSecAuth**, **MSPProfC** и **MSPConsent**. Вам необходимо использовать тег P3P, для того чтобы очистить файлы «cookie» в Web-браузерах поколения 6.0.

Публичные свойства экземпляра, определенные классом **PassportIdentity**, описаны в таблице 9.12.

Аутентификация Windows

Такие методы аутентификации, предлагаемые IIS, как Basic, Digest, Integrated Windows Authentication или их другие формы (NTLM/Kerberos или Certificates), используются в аутентификационном механизме Windows. Этот метод очень прост в реализации, поскольку вам потребуется очень мало кода ASP.NET, и IIS сам будет верифицировать регистрационные данные пользователей. Аутентификация средствами Windows хорошо подходит для приложений корпоративной сети intranet. Более того, этот метод хорошо работает с приложениями разных типов, а не только с приложениями ASP.NET.

Если пользователь запросил защищаемый ресурс, IIS первоначально аутентифицирует пользователя и прикрепляет к запросу идентификационный маркер. Затем ASP.NET использует этот маркер, для того чтобы определить, действителен ли запрос. Для управления доступом к защищаемым ресурсам вы можете использовать заимствование прав. Если заимствование прав (impersonation) разрешено, ASP.NET переопределяет права пользователя на основе идентификационного маркера, включенного в запрос. Затем производится проверка, уполномочен ли данный пользователь на доступ к ресурсу. Если права доступа имеются, ASP.NET пересылает запрошенный ресурс через IIS, в противном случае будет отослано сообщение об ошибке.

Таблица 9.12. Публичные свойства экземпляра, определенные классом PassportIdentity

Свойство	Описание
Error	Возвращает код ошибки, связанной с текущим паспортом
GetFromNetwork-Server	Возвращает true, если имеющееся соединение является соединением с Passport-сервером, и если данные в строке запроса действительны. Это важная проверка, поскольку в этом случае аутентификационный «билет» содержится в строке запроса и приложение должно перенаправить запрос без строки запроса, чтобы убедиться, что аутентификационный «билет» не получен из сохраненной браузером «истории» (особенно важно для электронных магазинов)
HasSavedPassword	Возвращает true, если пароль владельца аутентификационного «билета» был сохранен на Passport-сервере при создании «билета»
HasTicket	Возвращает true, если имеется аутентификационный «билет» в форме файла «cookie» в составе строки запроса
IsAuthenticated	Возвращает true, если пользователь аутентифицирован
Item	Представляет семейство объектов по умолчанию. Обращение к этому свойству эквивалентно обращению к **GetProfileObject** и **SetProfileObject**
Name	Представляет имя идентифицированного объекта. В данном случае это будет значение идентификатора паспорта PUID
TicketAge	Время в секундах с момента выпуска или обновления аутентификационного «билета»
TimeSinceSignIn	Время в секундах с момента регистрации пользователя на Passport-сервере

Для того чтобы использовать аутентификацию Windows, сконфигурируйте файл **Web.config**[1] следующим образом:

```
<configuration>
  <system.web>
    <authentication mode= "Windows"/>

     <authorization>
              <allow
```

[1] Конфигурирование приложения ASP.NET никак не влияет на настройки IIS Directory Security. Эти системы не зависят друг от друга и отрабатываются последовательно, друг за другом.

```
                    users="ИмяДомена\ИмяГруппы"/>
            <deny users="*"/>
    </authorization>
    <identity impersonate="true" />
  </system.web>
</configuration>
```

Публичные свойства экземпляра, определенные классом **WindowsIdentity**, описаны в таблице 9.13.

Таблица 9.13. Публичные свойства экземпляра, определенные классом WindowsIdentity

Свойство	Описание
IsAnonymous	Указывает, является ли учетная запись пользователя анонимной
IsAuthenticated	Указывает, аутентифицирован ли пользователь системой Windows
IsGuest	Указывает, является ли учетная запись пользователя гостевой
IsSystem	Указывает, является ли учетная запись пользователя системной
Name	Регистрационное имя пользователя в Windows
Token	Маркер учетной записи Windows для данного пользователя

Публичные статические (разделяемые) методы, определенные классом **WindowsIdentity**, описаны в таблице 9.14.

Таблица 9.14. Публичные статические (разделяемые) методы, определенные классом WindowsIdentity

Метод	Описание
GetAnonymous	Возвращает объект **WindowsIdentity**, представляющий анонимного пользователя Windows
GetCurrent	Возвращает объект **WindowsIdentity**, представляющий текущего пользователя Windows
Impersonate	Позволяет коду заимствовать права другого пользователя Windows

Для того чтобы реализовать аутентификацию средствами Windows, сконфигурируйте режим аутентификации в файле **Web.config** так, как показано ниже, и отключите в системе анонимный доступ. Затем сконфигурируйте учетные записи пользователя Windows на вашем Web-сервере, если они там отсутствуют.

```
<authentication mode= "Windows"/>
<authorization>
<deny users="?"/>
</authorization>
```

При использовании аутентификации Windows вы можете получать информацию непосредственно из объекта **User**. То есть, если пользователь аутентифицирован и авторизован, ваше приложение может считывать сведения о нем из свойств объекта **User**. Например, результат выполнения следующего кода приведен на рисунке 9.8.

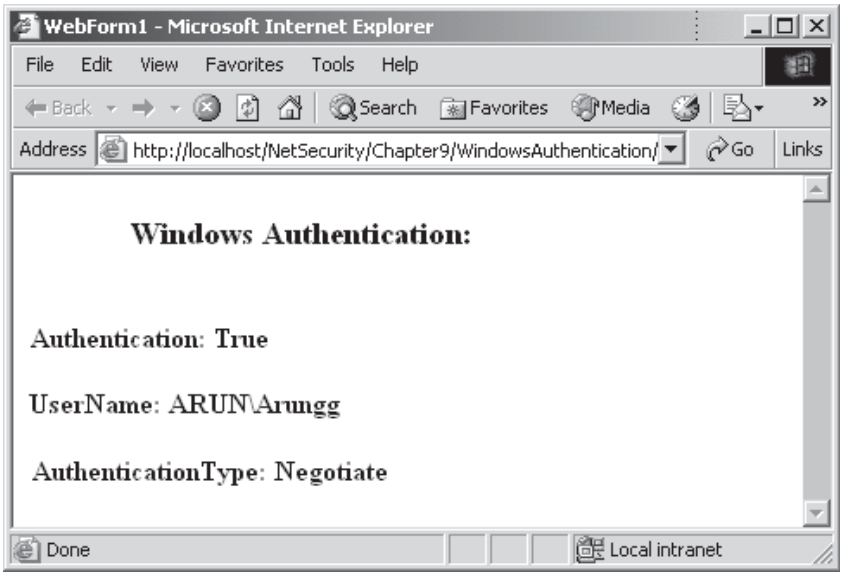


Рис. 9.8. Получение аутентификационной информации Windows (выполняется локально)

```
private void Page_Load(object sender, System.EventArgs e)
{
   AuthLabel.Text = "Authentication: " +
               User.Identity.IsAuthenticated.ToString();
   UserLabel.Text = "UserName: " + User.Identity.Name;
   AuthtypeLabel.Text = "AuthenticationType: " +
               User.Identity.AuthenticationType;

}
```

Если вы запустите код с удаленной машины, ASP.NET откроет диалоговое окно для получения имени и пароля, как показано на рисунке 9.9. Если введенные имя и пароль действительны в данном домене сети, ASP.NET аутентифицирует вас для запрошенного приложения.

Рис. 9.9. Диалоговое окно для регистрации

Реализация авторизации ASP.NET

Авторизация – это процесс, в котором вы определяете, обладает ли уже аутентифицированный пользователь правами доступа к запрошенному им ресурсу или странице. В ASP.NET предусмотрено два основных вида авторизации: авторизация на доступ к файлу и авторизация на доступ к URL. Давайте рассмотрим теперь эти два типа авторизации.

Авторизация на доступ к файлу

Авторизация на доступ к файлу осуществляется на основе аутентифицированной учетной записи, имеющейся у IIS. Выполняется она при помощи объекта **FileAuthorizationModule**. Он проверяет список контроля доступа (ACL[1]) и определяет, обладает ли данный пользователь правом доступа к файлу. **FileAuthorizationModule** обеспечивает авторизацию на основе списков ACL, хранящихся как структуры файловой системы. Вы можете конфигурировать список ACL для любого конкретного файла или каталога при помощи вкладки Security (Безопасность) диалога свойств файла. Заметьте, что метод **AccessCheck** вызывается только тогда, когда с запросом ассоциирован объект **WindowsIdentity**, поэтому он бесполезен при использовании аутентификации с помощью форм или паспорта, где, как правило, используется только одна (анонимная) учетная запись Windows.

[1] ACL (Access Control List – список контроля доступа) представляет собой список всех групп и пользователей, обладающих различными правам доступа в отношении данного конкретного файла. Такие структуры Windows, как DACL или SACL, являются экземплярами этого списка.

Авторизация на доступ к URL

Авторизация на доступ к URL выполняется при помощи объекта **URLAuthorizationModule**. Для этого типа авторизации анонимный пользователь проверяется на предмет прав доступа согласно конфигурационным данным. Если для данного URL доступ разрешен, то запрос будет авторизован. Применяя **URLAuthorizationModule**, вы можете использовать как положительный, так и отрицательный принцип авторизации. То есть вы можете разрешать доступ или отказывать в нем для ролей и групп пользователей. Для того чтобы реализовать авторизацию на доступ к URL, поместите список пользователей и/или ролей в элементы <**allow**> и <**deny**> раздела <**authorization**> конфигурационного файла.

Общий синтаксис раздела <**authorization**> выглядит следующим образом:

```
<[элемент]> [пользователи] [роли] [методы HTTP] />
```

Имеется два элемента – <**allow**> и <**deny**>:

- элемент <**allow**> разрешает пользователям доступ к ресурсу;
- элемент <**deny**> запрещает доступ.

Атрибуты, поддерживаемые этими элементами, описаны в таблице 9.15.

Например, следующий фрагмент конфигурационного файла иллюстрирует предоставление прав доступа пользователю с именем Arun и роли Administrator. Всем остальным пользователям в доступе отказано.

Таблица 9.15. Атрибуты элементов <allow> и <deny>

Атрибут	Описание
Roles	Определяет роли, включенные в элемент. Ассоциированный объект **Iprincipal** определяет членство пользователя в ролях. Вы можете присоединить к контексту текущего запроса произвольный объект **Iprincipal** и таким способом определить любую необходимую роль. Например, стандартный класс **WindowsPrincipal** использует для определения членства в ролях группы Windows NT
Users	Идентификаторы пользователей для данного элемента
Verbs	Определяет методы HTTP (например, GET, HEAD или POST), к которым относится разрешение или запрет

```
<configuration>
  <system.web>
    <authorization>
            <allow users="Arun" />
            <allow roles="Administrator" />
            <deny users = "*" />
    </authorization>
```

```
    <identity impersonate="true" />
  </system.web>
</configuration>
```

Вы можете задать список пользователей, разделяя идентификаторы запятыми:

```
<allow users="Arun, Saru, admin\Gnv" />
```

Доменное имя (admin\Gnv) включает в себя как имя домена, так и имя пользователя.

Вы можете указать конкретный метод HTTP, как показано в следующем фрагменте. Здесь пользователям Arun и Saru разрешено использовать также и метод POST, а всем остальным будет доступен только метод GET.

```
<allow VERB="POST" users="Arun, Saru" />
<deny VERB="POST" users = "*" />
<allow VERB="GET" users = "*" />
```

Имеется два специальных шаблона:

- * — все пользователи;
- ? — неаутентифицированные (анонимные) пользователи.

Например, следующий фрагмент запрещает доступ анонимным пользователям.

```
<authorization>
   <deny users = "?" />
</authorization>
```

Имеется также специальный тег <**location**>, которым можно указать конкретный файл или каталог.

```
<location path="путь к файлу или каталогу">
```

Приведенные ниже два раздела <**authorization**> отличаются друг от друга. В первом случае доступ запрещен всем пользователям, поскольку запрещающая строка идет первой и «перекрывает» действие остальных строк. Во втором случае доступ разрешен только пользователю Arun.

```
<authorization>
   <deny users = "*" />
   <allow users="Arun" />
</authorization>
<authorization>
   <allow users="Arun" />
   <deny users = "*" />
</authorization>
```

Реализация заимствования прав ASP.NET

Заимствование прав позволяет вместо написания кода ASP.NET для аутентификации и авторизации просто использовать для этих целей средства IIS. Если пользователь аутентифицирован, в приложение ASP.NET передается свидетельствующий это маркер аутентификации, в противном случае будет передан маркер, указывающий на отсутствие аутентификации. В приложении ASP.NET, если разрешен режим заимствования прав, ASP.NET исходит из заданной идентичности клиента и руководствуется настройками прав доступа на файлы и каталоги файловой системы NTFS. Если заимствование прав отключено, ASP.NET работает от имени локальной машины. Для совместимости с ASP режим заимствования прав по умолчанию отключен.

Для включения этого режима измените конфигурационный файл следующим образом:

```
<identity impersonate="true" name="ИмяПользователя"
  password="Пароль" />[1]
```

Итоги главы

В этой главе мы детально рассмотрели средства обеспечения безопасности в среде ASP.NET. Мы познакомились с наиболее важными классами .NET, необходимыми для этих целей, и увидели несколько программных примеров, иллюстрирующих реализацию механизмов аутентификации, авторизации и заимствования прав в приложениях ASP.NET. В следующей главе нам предстоит знакомство с самыми яркими «звездами» мира безопасности Web-приложений.

[1] В .NET Framework версии 1.1 вы можете поместить пароль в шифрованный ключ регистра. Имеется также соответствующая «заплатка» для версии 1.0. Подробней об этом см. *http://support.microsoft.com/default.aspx?scid=kb;en-us; 329250*.

Глава 10

Защита Web-служб

В наше время выражение «Web-сервисы» стало модным, и это результат революционных преобразований в мире программного обеспечения. Однако именно вопросы безопасности пока что делают невозможным глобальное распространение Web-сервисов (или Web-служб – оба термина одинаково употребительны). Согласно исследованиям Forrester Research, соображения безопасности остаются главным барьером на пути глобального распространения Web-служб промышленного характера. Разумеется, любая Web-служба, предоставляющая для доступа извне конфиденциальную информацию, обеспечивает защиту от критических и прямых угроз. В приложениях реального времени важнейшую роль играют надежные коммуникации между бизнес-партнерами. Соответственно, корпорация Microsoft и другие гиганты программного бизнеса придают огромное значение надежности и безопасности Web-сервисов. В столь разнородной среде, какую представляют собой Web-службы с их многодоменными и многоуровневыми приложениями, к основным проблемам безопасности относятся проблемы безопасности коммуникаций, аутентификация, авторизация, защита данных, контроль целостности и конфиденциальности передаваемых данных, а также подтверждение обязательств.

В деловых приложениях реального времени Web-службы могут использовать сложную иерархию уровней доступа, и этот доступ должен быть ограничен только авторизованными клиентами. Для своей аутентификации (то есть удостоверения идентичности) пользователь или клиент должен представить, например, регистрационное имя и пароль. Еще один аспект, на который вам придется обратить внимание, заключается в безопасности коммуникаций между клиентом и службой. Для обеспечения безопасности коммуникаций необходимо будет использовать либо такую защиту на транспортном уровне, как протокол SSL (Secure Sockets Layer – протокол защищенных сокетов) и IPSec (Internet Protocol Security – защищенный протокол IP), либо защиту на уровне сообщений такую, как спецификация WS-Security. Защита транспортного уровня («точка-точка») лучше всего подходит для ситуаций, тесно связанных сред Microsoft Windows, как корпоративные сети типа intranet, а защита на уровне сообщений более уместна в разнородной среде Web-служб. Вы можете также реализовать собственные механизмы защиты, используя, например, заголовки SOAP.

Итак, в зависимости от требований, защиту Web-служб можно реализовать на трех уровнях:

- защита на транспортном уровне («точка-точка»), такая, как протоколы SSL/IPSec, аутентификация и авторизация ASP.NET и аутентификация IIS;
- защита на уровне приложения, реализуемая разработчиком с использованием, например, заголовков SOAP;
- защита на уровне сообщения такая, как глобальная инициатива по архитектуре XML (GXA – Global XML Architecture), спецификация WS-Security.

В этой главе мы вначале рассмотрим основные принципы защиты такие, как брандмауэры, виртуальные частные сети VPN и аутентификацию IIS. Затем мы познакомимся с аутентификацией Web-служб при помощи заголовков SOAP и криптотехнологии XML: XML Signature – подпись XML, XML Encryption – шифрование XML, спецификация управления ключами XML (XKMS – XML Key Management Specification) и язык разметки утверждений безопасности (или «язык разметки допуска к информации») SAML (Security Assertion Markup Language). Наконец, мы поговорим о спецификациях GXA и WS-Security, а также о том, как это реализуется при помощи расширений Web-служб WSE (Web Services Enhancements).

Основные техники защиты Web-служб

Основные процедуры защиты, которые необходимо рассмотреть, приведены в таблице 10.1.

Защищенное соединение

Существует три технологии, широко используемые для реализации безопасного (защищенного) соединения:

- брандмауэры;
- шифрование SSL;
- виртуальные частные сети VPN.

БРАНДМАУЭРЫ

Брандмауэр защищает локальную сеть от вторжений извне. Брандмауэр действует подобно барьеру, ограничивающему движение информации в обоих направлениях, то есть внутрь сети и наружу. Можно задать различные ограничения для разных клиентов внутри сети, основываясь на их происхождении или идентичности, используя сервер ISA (Microsoft Internet

Security And Acceleration). Использование брандмауэра имеет смысл, если вы в состоянии идентифицировать все те компьютеры, которые должны иметь доступ к вашему Web-сервису. В этом случае вы можете реализовать защиту при помощи протокола IPSec или брандмауэра, чтобы ограничить доступ компьютерами с известными IP-адресами. Однако, в реальности IP-адреса всех компьютеров могут быть неизвестны, и к тому же злонамеренный клиент может использовать технику подмены IP-адреса. Техника подмены IP-адреса ("IP spoofing") заключается в том, злоумышленник получает доступ к ресурсу, подменяя в своих пакетах IP-адрес на адрес авторизованного пользователя. Впрочем, правильно сконфигурированный брандмауэр для таких атак неуязвим.

Таблица 10.1. Основные процедуры защиты Web-служб

Процедуры	Описание
Безопасное соединение	«Безопасное соединение» означает, что вообще соединение между клиентом Web-службы и самой Web-службой должно быть безопасным
Аутентификация	Аутентификация – это проверка идентичности отправителя или получателя (Кто вы?)
Авторизация	Авторизация определяет, разрешен ли вам доступ к службе, и если да, то с какими полномочиями
Защита данных и конфиденциальность	Нам необходимо позаботиться о безопасности данных не только в процессе их передачи, но и при хранении в локальной системе. Защита данных – это процесс, обеспечивающий конфиденциальность данных. Шифрование SSL (HTTPS) обеспечивает конфиденциальность на уровне «точка-точка» между запрашивающими и отвечающими участниками Web-обмена. Шифрование XML можно использовать для сквозной защиты на уровне сообщения
Целостность	Целостность данных – это уверенность в том, что данные не были подвергнуты несанкционированному изменению. Для реализации защиты целостности можно использовать XML Signature, при этом вы «подписываете» части XML-документов так, что в месте получения данных обеспечивается их целостность, вне зависимости от числа систем, через которые эти данные пройдут

Web-служба представляет собой программную сущность, которая обеспечивает определенные элементы функциональности такие, как логика приложения, и предоставляет их любому числу разнородных систем посредством использования общих форматов и протоколов таких, как XML и HTTP. Потоки данных в Web-службах направляются через порты 80 и 443 аналогично обычным потокам Web-сайта, при помощи протокола

HTTP. Одно из ограничений Web-служб с точки зрения безопасности заключается в использовании протокола HTTP. Конечно, протокол HTTP прост, и при его использовании совместно с XML мы можем легко сделать функциональность приложения доступной для множества разнородных систем и приложений. Однако проблема с HTTP состоит в том, что протокол HTTP свободно переносит данные через брандмауэры и делает возможным вторжение в сеть предприятия извне. Хотя брандмауэры и играют важную роль в защите от злонамеренных вторжений, они не могут контролировать содержимое SOAP, для того чтобы полностью исключить атаки извне. Таким образом, брандмауэры пригодны для обеспечения безопасности только на сетевом уровне, но не на уровне приложения.

SLL и HTTPS

SLL стал общепринятым в WWW протоколом для аутентификации и шифрования обмена между клиентами и серверами. В настоящее время протокол SSL наряду с TLS[1] (Transport Level Security – защита транспортного уровня) широко используется для защиты сетевого обмена в реальном времени. SSL обеспечивает конфиденциальность, целостность данных и аутентификацию (аутентификацию сервера и аутентификацию клиента). Этот протокол уже используется множеством предприятий для защиты соединений, устанавливаемых через Internet. Протокол SSL был разработан компанией Netscape и стал общепринятой технологией защиты TCP/IP-коммуникаций между источниками запросов по протоколу HTTP (Web-браузеры) и приемниками этих запросов (Web-серверы). SSL, в сущности, реализует криптографию с открытым ключом[2]. Протокол HTTPS защищает свои соединения, просто передавая запросы и ответы HTTP по каналу, защищенному посредством SSL.

Протокол SSL обеспечивает определенный уровень безопасности для Web-служб. Он защищает канал, по которому производится обмен между клиентом и сервером. Согласно SSL, данные клиента перед передачей серверу шифруются. Полученные сервером данные дешифруются, и тем самым подтверждается идентичность их отправителя. SSL обеспечивает безопасное и надежное соединение между двумя корреспондентами в сети. Создание защищенного канала инициируется специальным обменом сообщениями, который называется «рукопожатие» (handshake) SSL/

[1] TLS представляет собой SSL версии 3.0 с исключенной поддержкой частных технологий и поддерживается организацией IETF (Internet Engineering Task Force – проблемная группа проектирования Internet). Подробней об IETF см. *http://www.ietf.org/rfc/rfc2246.txt*.

[2] Криптография с открытым ключом – это технология, в которой используется несимметричная пара ключей, для шифрования и дешифрования раздельно. Каждая такая пара состоит из открытого (публичного) и закрытого (секретного) ключей. Открытый ключ публикуется так, чтобы быть общедоступным. Секретный ключ хранится у участника обмена и ни по каким каналам связи не передается. Данные, зашифрованные открытым ключом, можно расшифровать только при помощи секретного ключа. И наоборот, зашифрованные секретным ключом данные можно расшифровать только при помощи открытого ключа. Криптография с открытым ключом была разработана в 1976 году исследователями из Стэндфордского университета.

TLS. Это «рукопожатие» позволяет серверу аутентифицировать себя для клиента, и при необходимости клиент тоже может аутентифицировать себя для сервера. На практике, как правило, аутентифицирует себя только сервер. Затем все сообщения, которыми обмениваются обе стороны, шифруются и подписываются, что обеспечивает конфиденциальность и целостность данных. Схема обмена данными по защищенному каналу между клиентом Web-службы и самой Web-службой изображена на рисунке 10.1.

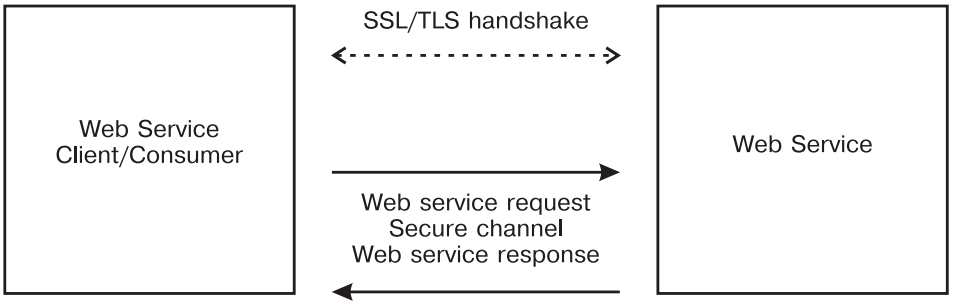


Рис.10.1. Передача данных между клиентом Web-службы и самой Web-службой по защищенному каналу

Ограничение протокола SSL состоит в его низкой производительности. SSL использует сертификаты и свидетельства, которые радикально снижают общую производительность в сравнении с протоколом HTTP. Иногда проблему отчасти удается решить при помощи специальных «ускорителей» SSL.

Еще одна трудность с использованием протокола SSL заключается в том, что он обеспечивает защиту только на транспортном уровне («точка-точка»). Это означает, что защита распространяется только на две аутентифицированные обменивающиеся стороны. Если участников обмена становится больше двух, и при этом каждый участник может просматривать и модифицировать данные, SSL не может обеспечить защиту данных.

Протокол SSL защищает коммуникации только на транспортном уровне, но не на уровне сообщений. Следовательно, данные находятся в безопасности только тогда, когда они путешествуют по проводам или эфиру. Когда данные достигают пункта назначения и становятся сообщением, они уже не защищены, однако в Web-службах данные могут проходить через многие пункты, прежде чем достигнуть своего конечного назначения. Предположим к примеру, что пункт A желает установить защищенное соединение с пунктом B. Если данные при этом должны проходить через некоторый промежуточный пункт C, например, межсетевой шлюз, то в пункте C данные будут известны. Иными словами, сценарий SSL не обеспечивает сквозную защиту от начального до конечного пункта. Следовательно, остается возможность атаки на данные в пункте C на уровне сообщения, пока данные путешествуют между пунктами A и B.

Таким образом, протокол SSL можно использовать для защиты Web-служб только в такой среде, где данные не проходят через промежуточные приложения. Иными словами, мы можем использовать SSL в среде тесно связанных систем Microsoft Windows.

НЕОБХОДИМОСТЬ В СКВОЗНОЙ ЗАЩИТЕ

SSL/TLS обеспечивает такие функции, как конфиденциальность и целостность данных, действуя на транспортном уровне «точка-точка». Сходным образом для защиты сеансов связи широко используют протокол IPSec[1].

Современные топологии Web-приложений и служб могут включать в себя множество таких разнообразных систем, как мобильные устройства, шлюзы, прокси-серверы, балансировщики нагрузки, демилитаризованные зоны DMZ и центры аутсорсинговых данных. Web-службы теперь представляют собой глобально распределенную, сложную, мультидоменную и гетерогенную среду, поэтому сообщения SOAP могут маршрутизироваться по многим промежуточным пунктам, прежде чем достичь пункта назначения. Проблема состоит в том, что если передача данных между двумя пунктами происходит не на транспортном уровне, то теряется как целостность, так и конфиденциальность данных. Таким образом, нам требуется сквозная защита, действующая от начального до конечного пункта, то есть защита на уровне сообщения, а не только защита транспортного уровня, обеспечиваемая SSL/TLS. Следовательно, для реализации полной архитектуры безопасности Web-сервисов, требуется одновременно реализовать защиту на транспортном уровне и на уровне сообщения. Эта конфигурация иллюстрируется рисунком 10.2.

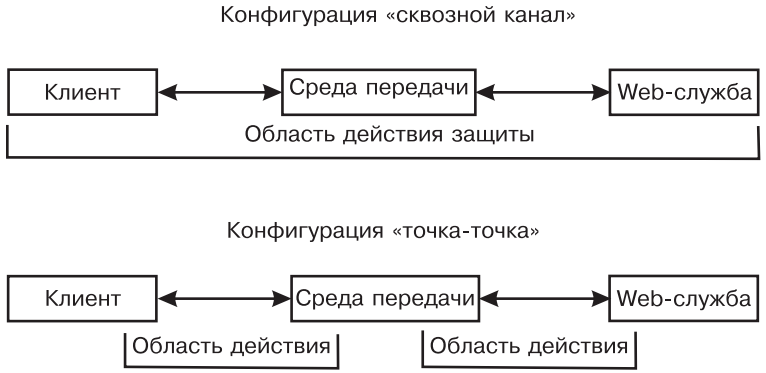


Рис.10.2. Конфигурации «точка-точка» и «сквозной канал»

[1] Протокол IPSec был разработан для обеспечения универсальной, высококачественной криптографической защиты протоколов IPv4 и IPv6. Он предусматривает такие функции безопасности, как контроль доступа, целостность на уровне пакетов, аутентификация источника данных, защита от повторения пакетов, конфиденциальность (шифрование) и ограниченная конфиденциальность трафика. Эти функции реализованы на уровне IP и обеспечивают защиту для протокола IP и для протоколов более высокого уровня.

КОНФИГУРИРОВАНИЕ SSL В WINDOWS 2000

Для настройки защиты Web-служб при помощи SSL в системе Windows 2000 необходимо выполнить следующие действия.

- **Сконфигурируйте для использования SSL ваш Web-сервер:** для этого установите на ваш Web-сервер «серверный» сертификат SSL. Если вы намерены запрашивать сертификат у третьей стороны (сертификационного центра), то переходите к следующему шагу.
- **Установите на клиентской системе сертификат CA:** если вы используете собственную службу сертификации, то вам необходимо установить на клиентской машине сертификат CA, как доверенный корневой сертификат.
- **Измените настройку WSDL с HTTP на HTTPS:** адрес URL вашей Web-службы должен начинаться с имени протокола https, а не http. Откорректируйте соответственно файлы WSDL (Web Services Description Language – язык описания Web-служб). Укажите https, расположение URL вашей Web-службы при добавлении соответствующей ссылки в конфигурацию Visual Studio.NET. После этого мы сможем использовать Web-службу по каналу, защищенному SSL.
- **Разрешение доступа только по SSL:** если предполагается, что Web-служба будет принимать запросы только по SSL, то сконфигурируйте виртуальный каталог следующим образом.

1. Щелкните правой кнопкой мыши на виртуальном каталоге, где предполагается разместить Web-службу, затем выберите в контекстном меню команду Properties (Свойства).
2. Щелкните на вкладке Directory Security (Защита каталога) и затем щелкните Edit (Правка) под Security Communications (Защищенные коммуникации).
3. Щелкните на Require secure channel (SSL) (Требовать защищенный канал), а затем дважды щелкните на OK.

ВИРТУАЛЬНАЯ ЧАСТНАЯ СЕТЬ VPN

1. Виртуальная частная сеть VPN (Virtual Private Network) соединяет множественные сети, пользователей с радиодоступом и других удаленных пользователей. В сущности сеть VPN является расширением частных сетей типа WAN, но обладает большей эффективностью, поскольку использует уже существующую инфраструктуру Internet.
2. Сеть VPN обеспечивает защищенные соединения, через которые могут циркулировать данные из различных сетей, причем в роли среды передачи выступает обычный Internet.

Аутентификация и авторизация

Аутентификация – это процесс подтверждения идентичности клиента, пользователя или приложения, необходимый для предоставления доступа к некоторому ресурсу. Например, пользователи, желающие получить доступ к ресурсу, должны предоставить некоторые свидетельства своей подлинности такие, как регистрационное имя и пароль. В ответ пользователь получает от сервера некий маркер безопасности. В мире Web-служб такой маркер может выступать в форме файла cookie, загружаемого браузером клиента, номера сеанса, который запоминает сервер и тому подобного. Простейший способ реализовать механизм аутентификации в Web-службе заключается в использовании аутентификационных функций протокола HTTP.

Механизмы аутентификации в протоколе HTTP

Microsoft IIS версии 5.0 поддерживает несколько механизмов аутентификации для протокола HTTP такие, как Basic (Базовый), Digest (Дайджест сообщения), Windows Integrated (Интегрированный в Windows) и Client certificate (Сертификат клиента).

BASIC (БАЗОВЫЙ)

В аутентификации по методу Basic пользователю предлагается ввести регистрационное имя и пароль. Если пользователь ввел действительные данные, то соединение будет установлено. Главный недостаток этого метода состоит в том, что Web-браузеры передают пользовательские данные Web-серверу в открытом, не зашифрованном виде. Просто отслеживая сетевой обмен между ними, злоумышленник может легко перехватить имя и пароль, для этого достаточно будет общедоступных в сети средств. Таким образом, рекомендуется использовать этот метод с протоколом HTTPS, то есть через канал SSL.

DIGEST (ДАЙДЖЕСТ СООБЩЕНИЯ)

Аутентификация по методу Digest аналогична аутентификации по методу Basic. Единственное отличие состоит в том, что используется другой способ передачи свидетельств. Здесь для защиты имени и пароля используется хеширование. Злоумышленник не может расшифровать пароль на основе перехваченного хеша или дайджеста сообщения. Дайджест сообщения – это уникальное значение, полученное из текста исходного сообщения хешированием. Этот метод, однако, не поддерживается на многих платформах, отличных от Microsoft Windows. Браузер Internet Explorer 5.0 и старше поддерживает аутентификацию по методу Digest, однако другие браузеры, например, Netscape Navigator, в своей стандартной конфигурации этот метод не поддерживают.

WINDOWS INTEGRATED (ИНТЕГРИРОВАННЫЙ В WINDOWS)

Аутентификация по методу Windows Integrated подходит для использования в корпоративных сетях типа intranet. Свидетельства, предоставленные пользователем, передаются на сервер по протоколу NT LAN Manager (NTLM), NT Challenge/Response (NTCR) или Kerberos[1]. Проблема с использованием данного метода аутентификации состоит в том, что он не работает через прокси-серверы HTTP и брандмауэры. Сервер IIS авторизует доступ пользователя к Web-службе, если представленные пользователем свидетельства соответствуют имеющейся действительной учетной записи. При аутентификации по методу Windows Integrated у пользователя первоначально не запрашиваются имя и пароль. Для аутентификации используется учетная запись текущего пользователя Windows на клиентской машине. Браузер запрашивает у пользователя имя и пароль в том и только в том случае, если аутентифицировать текущего пользователя при обмене с сервером не удалось. На рисунке 10.3 изображены настройки Internet Services Manager (Диспетчер Internet-служб). Чтобы настроить данный метод аутентификации, установите флажок Integrated Windows authentication в диалоговом окне Internet Services Manager, как показано на рисунке 10.4.

CLIENT CERTIFICATE (СЕРТИФИКАТ КЛИЕНТА)

При этом методе аутентификации для получения доступа к службе клиент должен получить клиентский сертификат от некоторой третьей организации, доверенной для обеих сторон. Затем сертификатам ставятся в соответствие пользовательские учетные записи, которые IIS использует для авторизации доступа к службам. Клиентский сертификат представляет собой электронный документ, содержащий идентификационную информацию такую, как данные о пользователе и об организации, выпустившей сертификат.

Альтернативой методам аутентификации IIS является возможность реализовать собственные методы, например, передачу данных пользователя в заголовках SOAP.

[1] Протокол Kerberos является свободно распространяемым протоколом сетевой аутентификации, разработанным в Массачусетском Технологическом Институте. Это протокол «с открытым кодом», в котором для идентификации пользователей используется криптография с разделяемым секретным ключом. Он обеспечивает надежную идентификацию участников обмена в открытой сети.

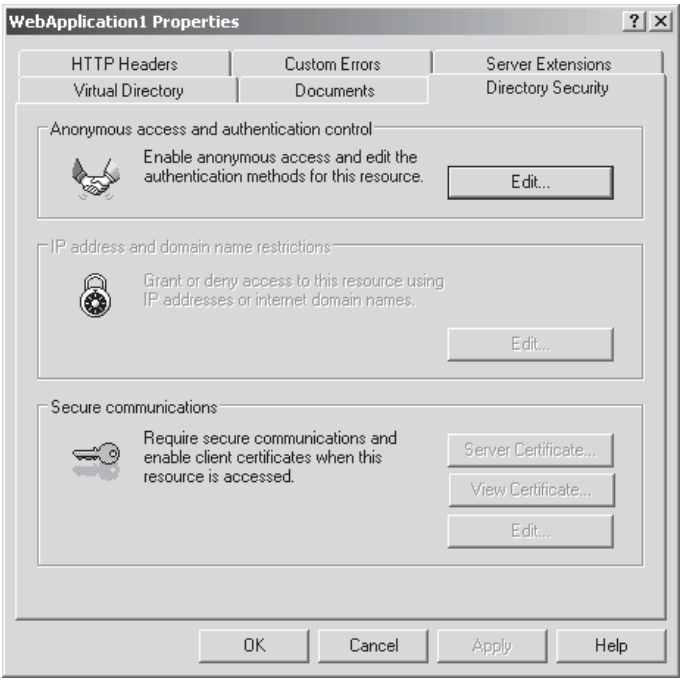


Рис.10.3. Настройки Internet Services Manager (Диспетчер Internet-служб)

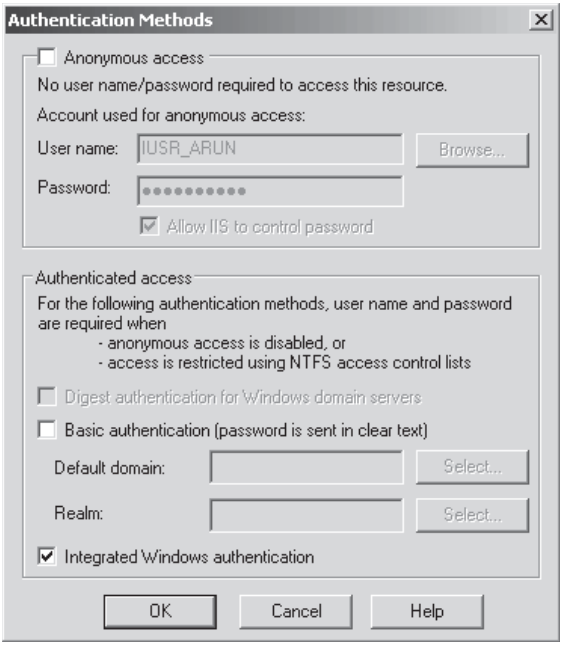


Рис.10.4. Настройки Internet Services Manager (Диспетчер Internet-служб) для аутентификации методом Integrated Windows authentication

Аутентификация Web-служб при помощи заголовков SOAP

Архитектура сообщения SOAP

Сообщение SOAP состоит из Конверта (Envelope), необязательного заголовка (элемент Header) и обязательного элемента Body, то есть тела сообщения, как изображено на рисунке 10.5. Элемент Body содержит в себе собственно сообщение. Необязательный элемент Header может содержать некоторые дополнительные данные, относящиеся к сообщению. Каждый дочерний элемент, содержащийся в элементе Header, называется «заголовком SOAP». Эти «заголовки» вы можете использовать в Web-службах ASP.NET для своих нужд, включая в них некоторую дополнительную информацию. Поскольку спецификация SOAP не определяет строго содержимое заголовков SOAP, они могут нести в себе данные, обрабатываемые на уровне инфрастуктуры. Элемент **<soap:Header>** можно использовать для обмена такой информацией, как аутентификационные данные, номер сеанса, идентификатор транзакции и т. п. Таким образом, заголовки SOAP представляют собой средство расширения функциональности сообщений SOAP.

Рис.10.5. Архитектура сообщения SOAP

Давайте создадим Web-службу, которая аутентифицирует пользователей на основе представленных ими имен и паролей, используя базовый класс **SOAPHEADER**, и создадим затем клиентское приложение для этой службы.

```
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
namespace AuthenService
{
   public class Header : SoapHeader
   {
     public string Username;
     public string Password;
   }

   [WebService(Namespace="http://www.phptr.com")]
   public class AuthenticateWebService :
   System.Web.Services.WebService
   {

     public AuthenticateWebService()
     {
              InitializeComponent();
     }

   //Код, сгенерированный Web Service Designer

     [WebMethod(Description =
       "Использование заголовков SOAP в Web-службах ASP.NET")]
     [SoapHeader("HeaderMemberVariable")]
       public string Authenticate()
       {

        // Обработка SoapHeader.
        if (HeaderMemberVariable.Username == "Adminarun" &&
            HeaderMemberVariable.Password == "Anystrongpassword")
        {
          return "Добро пожаловать! Вы успешно зарегистрированы.";
        }

        return "Пароль и/или имя пользователя недействительны!";
       }
   }
}
```

- Создайте класс с именем **Header**, производный от класса **SoapHeader**, и представляющий данные, передаваемые в заголовке SOAP.
- Добавьте две публичных строки **(Username; Password)** в класс **WebService**.
- Добавьте переменную **(HeaderMemberVariable)** в тип, производный от **SoapHeader**.

- Примените атрибут **SoapHeaderAttribute** к методу Web-службы, такому, как **[SoapHeader("HeaderMemberVariable")]**.
- В методе **WebService** используйте свойство **MemberName** для обработки данных, переданных в заголовке SOAP.
- Проверьте условие: **HeaderMemberVariable.Username == "Adminaram" && HeaderMemberVariable.Password ==** "Anystrongpassword"[1].
- Если условие выполнено, верните строку **«Добро пожаловать! Вы успешно зарегистрированы.»**.
- Если условие не выполнено, верните строку «Пароль и/или имя пользователя недействительны!».
- На тестовой экранной форме вы заметите сообщение «No test form is available, as this service or method does not support the HTTP GET protocol (тестовая форма недоступна, поскольку данная служба или метод не поддерживает метод GET протокола HTTP)».

На тестовой странице в Internet Explorer вы увидите образец запроса SOAP, включающего в себя элементы заголовка, как показано на листинге ниже.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
  <soap:Header>
    <MyHeader xmlns="http://www.phptr.com/">
            <Username>строка</Username>
            <Password>строка</Password>
    </MyHeader>
  </soap:Header>
  <soap:Body>
    <Authenticate xmlns="http://www.phptr.com/" />
  </soap:Body>
</soap:Envelope>
```

Создание прокси при помощи Visual Studio .NET

Создайте Web-форму ASP.NET и переименуйте ее в **Authenticateclient.aspx**. Добавьте в нее два текстовых поля для строк Username и Password, а также этикетку Label для отображения результата и кнопку. Завершив работу с пользовательским интерфейсом, добавьте ссылку на проект, щелкнув на Project | Add Web Reference. Затем щелкните на ссылке Web References на Local Web Server или введите с клавиатуры требуемое имя службы. Вид тестовой страницы приведен на рисунке 10.6.

[1] Под строкой "Anystrongpassword" здесь имеется в виду любой «сильный» пароль. – *прим. пер.*

Щелкните на кнопке Add Reference. Вы увидите файлы Discovery (DISCO) и WSDL в узле Web References.

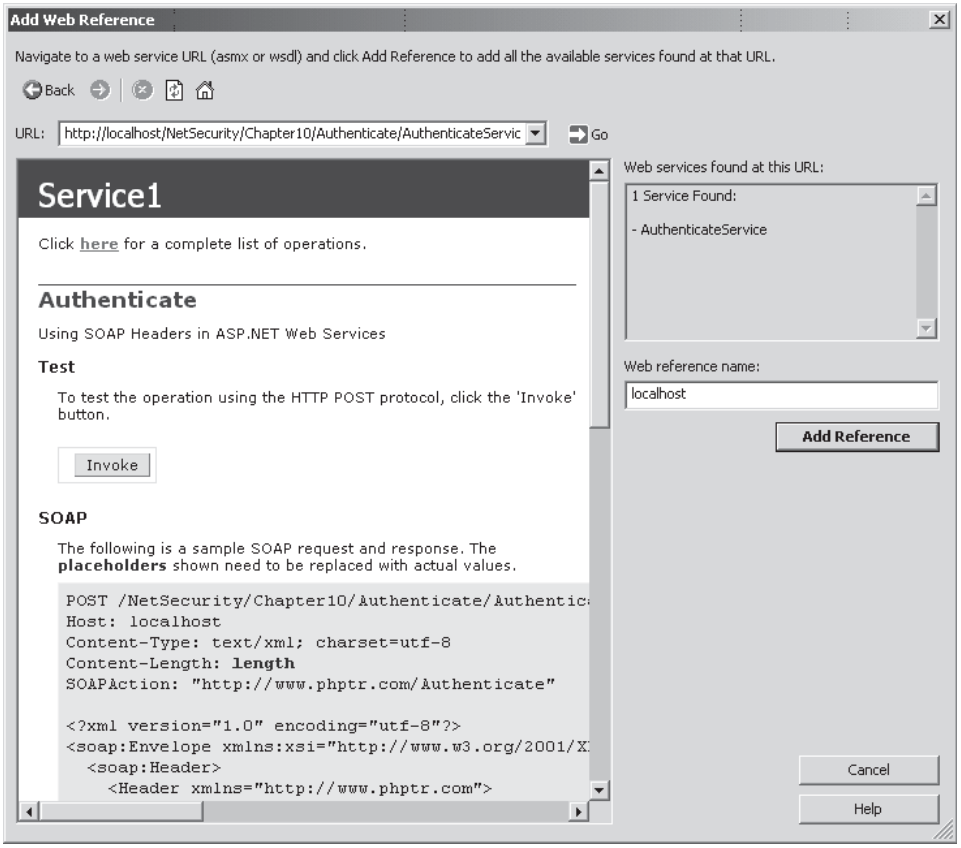


Рис.10.6. Диалоговое окно добавления Web-ссылок Add Web reference

СОЗДАНИЕ КЛИЕНТСКОЙ WEB-ФОРМЫ

Для обработки заголовков SOAP в клиентской части необходимо выполнить следующую процедуру. Создайте новый экземпляр класса, представляющего заголовок SOAP.

```
localhost.Header aa = new localhost.Header();
```

Назначьте для заголовка SOAP значения текстовых полей.

```
aa.Username = TextBox1.Text;
aa.Password = TextBox2.Text;
```

Назначьте объект – заголовок SOAP в качестве значения переменной класса proxy.

```
proxy.HeaderValue = aa
```

Вызовите метод класса proxy, который сообщается с методом Web-службы и отображает результат в элементе управления – этикетке Label.

```
string results = proxy.Authenticate();
Label1.Text =results;
```

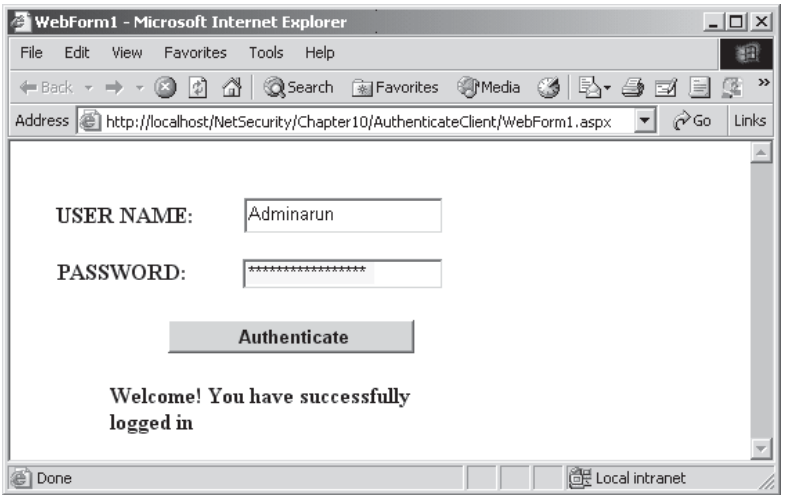


Рис.10.7. Вид формы Authenticateclient.aspx с корректно введенными именем и паролем

Результат работы **Authenticateclient.aspx** приведен на рисунке 10.7. Опять же, если вы передаете сообщения SOAP через HTTP вместо HTTPS, то клиентские данные будут передаваться в сети в открытом виде.

Технологии безопасности XML

Технологии безопасности XML работают на уровне сообщения, поэтому обеспечивают сквозную защиту. Различают следующие технологии безопасности XML:

- XML Signature – подпись XML;
- XML Encryption – шифрование XML;
- XKMS – спецификации управления XML-ключами;
- SAML – язык разметки утверждений безопасности (или «язык разметки допуска к информации»).

Целостность

Под *целостностью* имеется в виду уверенность в том, что данные не были несанкционированно изменены. Целостность обеспечивается при помощи технологии XML Signature и алгоритмов хеширования.

При использовании хеш-алгоритмов для защиты пересылаемых данных вычисленный хеш данных может пересылаться вместе с самим данными. Сервер затем может сравнить полученный вместе с данными хеш с хешем, вычисленным заново. Если оба значения совпадут, то полученные данные не искажены и совпадают с тем, что отправлял клиент. Хеш вычисляется при помощи одного из хеш-алгоритмов таких, как MD5 или SHA-1. По хешу невозможно реконструировать исходные данные, поскольку хеширование – необратимый процесс. Цифровая подпись представляет собой не что иное, как зашифрованный хеш.

XML Signature

Цифровая подпись XML Signature представляет собой технологию, сочетающую в себе усилия консорциума W3C и группы IETF (Internet Engineering Task Force – проблемная группа проектирования Internet). Стандарт XML Signature обеспечивает цифровую подпись частей XML-документов, и, тем самым, сквозную защиту целостности данных, проходящих через многие разнородные системы. Цифровая подпись XML подтверждает обязательства и доказывает целостность сообщения при передаче его через Web-службы.

XML Signature является основой для XKMS, WS-Security, SAML и других XML-технологий, где аутентификация производится на основе цифровой подписи. Цифровая подпись XML – это цифровая подпись, доступная в формате XML и удостоверяющая аутентичность и происхождение документа-носителя. Важной особенностью этой подписи является способность подписывать отдельные части документа, а не весь документ в целом. XML Signature может подписывать ресурсы различного рода по отдельности, например, часть документа, представляющую собой HTML-текст, и часть документа, содержащую двоичные данные (например, JPG).

КАК ЦИФРОВАЯ ПОДПИСЬ XML ОБЕСПЕЧИВАЕТ ПОДТВЕРЖДЕНИЕ ОБЯЗАТЕЛЬСТВ И ЦЕЛОСТНОСТЬ ДАННЫХ

Используя XML Signature, вы можете быть уверены в том, что полученное сообщение действительно и не было искажено по пути следования. Отправитель, формируя запрос к службе, подписывает документ своим секретным ключом и посылает его (документ) вместе со всеми остальными данными, заключенными в сообщении. Сервер может верифицировать подпись, воспользовавшись открытым ключом отправителя, и, таким образом, удостоверить целостность документа.

Цифровая подпись создается при помощи секретного ключа, хранящегося у отправителя. Никто кроме отправителя не имеет доступа к секретному ключу. Следовательно, отправитель отвечает за хранение своего ключа в секрете. Получатель верифицирует подпись при помощи соответствующего открытого ключа. Этот открытый ключ работает только в том случае, если использованный секретный ключ был аутентичен.

Таким образом, применяя XML Signature, вы обеспечиваете как целостность сообщения, так и подтверждение обязательств, поскольку никто кроме отправителя не мог создать успешно верифицированную подпись. Обычно подпись накладывается не на весь документ, а на небольшую его часть (хеш или дайджест). Алгоритмы хеширования очень чувствительны к малейшему изменению исходных данных. Таким образом, получатель может увериться в том, что документ не был изменен, сравнивая хеш, отправленный вместе с документом, с хешем, вычисленным заново у получателя.

ТИПЫ ЦИФРОВЫХ ПОДПИСЕЙ XML SIGNATURE

- **Enveloped Signature** – включенная подпись: сгенерированная подпись включается в состав подписываемого документа.
- **Enveloping Signature** – включающая подпись: сгенерированная подпись включает подписываемый документ в свой состав.
- **Detached Signature** – внешняя подпись: сгенерированная подпись пересылается отдельно от документа.

ПРИМЕР ЦИФРОВОЙ ПОДПИСИ XML SIGNATURE

```
<?xml version="1.0"?>
<!-- Detached XML Signature>
<Signature Id="MySignature"
xmlns="http://www.w3.org/xmldsig#">
<SignedInfo>
<CanonicalizationMethod Algorithm=
  "http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
<SignatureMethod Algorithm=
  "http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
<Reference URI=
  "http://www.w3.org/TR/2000/REC-xhtml1-20000126"/>
<Transforms>
<Transform Algorithm=
  "http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
<DigestMethod Algorithm=
  "http://www.w3.org/TR/2000/09/xmldsig#sha1"/>
<DigestValue>ak2r9u45n1045gan3435es503hkk8543fh68...
   </DigestValue>
</Reference>
</SignedInfo>
<SignatureValue>kf84jflk40klfk030klkdf0g55tdghdh6...
   </SignatureValue>
<KeyInfo>
   <KeyValue>
            <DSAKeyValue>
                  <p>...</p><X>...</X><A>...</A><Y>..</Y>
            </DSAKeyValue>
```

```
   </KeyValue>
</KeyInfo>
</Signature>
```

ЭЛЕМЕНТЫ ЦИФРОВОЙ ПОДПИСИ XML SIGNATURE

Элемент Signature является корневым элементом всех видов цифровой подписи XML Signature. Он включает в себя три основных элемента:

- элемент SignedInfo (<SignedInfo>…</SignedInfo>);
- элемент SignatureValue (<SignatureValue>…</SignatureValue>);
- элемент KeyInfo (<KeyInfo>…</KeyInfo>).

Давайте рассмотрим элементы подписи более подробно.

- Атрибут **<SignatureId>**: этот атрибут идентифицирует подпись.
- Элемент **<SignedInfo>**: содержит в себе всю необходимую информацию о подписываемом ресурсе.
- Элемент **<CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>**: этот элемент определяет метод канонизации и применяется в случаях, когда два сходных XML-документа содержат идентичные данные, но отличаются только незначительными деталями, скажем, пробелами, переводами строк и способами представления элементов. Канонизация – это процесс, нивелирующий все несущественные отличия в документах. Благодаря канонизации два логически эквивалентных документа дадут одинаковые хеши в результате хеширования несмотря на отличия в форматировании. Следовательно, XML-данные перед наложением подписи необходимо канонизировать, это поможет избежать ошибочного результата при проверке подписи. Приведенная выше строка задает алгоритм канонизации.
- Элемент **<SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>**: это алгоритм, который будет использован для генерации электронной подписи (то есть преобразования канонизированного содержимого **SignedInfo** в содержимое **SignatureValue**).
- Атрибут **<Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">**: этот необязательный атрибут указывает на местоположение подписанных данных.
- Элемент **<Transforms>**: это элемент указывает на список шагов обработки, которым должны подвергнуться данные в указанном источнике, прежде чем из них может быть сформирован дайджест.
- Атрибут **<DigestMethod Algorithm="http://www.w3.org/TR/2000/09/xmldsig#sha1"/>**: этот обязательный атрибут задает алгоритм создания дайджеста.
- **<DigestValue>ak2r9u45n1045gan3435es503hkk8543fh68...</DigestValue>**: содержит дайджест сообщения, сгенерированный по методу, определенному атрибутом **DigestMethod Algorithm**.

- **<SignatureValue>kf84jflk40klfk030klkdf0g55tdghdh6...</SignatureValue>**: содержит подпись сообщения, сгенерированную методом, определенным атрибутом **SignatureMethod Algorithm**.
- Элемент **<KeyInfo>**: этот элемент содержит ссылку на открытый ключ отправителя, который получатель может использовать для верификации подписи (необязательно). Обычно **<KeyInfo>** содержит открытые ключи, имена ключей, сертификаты и т. д. В предыдущем примере мы использовали ключ типа DSA.

Защита данных и конфиденциальность

Под защитой данных понимается процесс, обеспечивающий секретность и конфиденциальность данных. «Секретность и конфиденциальность» означает защиту чувствительной информации от доступа посторонних лиц. Такая защита достигается шифрованием данных при помощи того или иного крипто-алгоритма. При использовании SSL мы шифруем все данные, передаваемые по защищенному каналу. Однако если мы хотим зашифровать лишь отдельные части XML-документа, нам понадобится технология XML Encryption. При помощи XML Encryption мы можем зашифровать только чувствительные данные (отдельные части документа) такие, как, например, информация о кредитной карточке, лицензионную информацию и другие данные. Тем самым реализуется сквозная защита данных на пути от отправителя до получателя.

XML Encryption

Рабочая группа W3C XML Encryption, объединяющая усилия консорциума W3C и группы IETF (Internet Engineering Task Force – проблемная группа проектирования Internet), выпустила стандарты и спецификации XML Encryption, шифрования XML. В марте 2002 года рабочая группа выпустила спецификацию (на правах рекомендации-кандидата) шифрования XML. Эта спецификация обуславливает синтаксис, используемый при шифровании, и описывает процесс шифрования XML-документа или его частей. XML Encryption – это процесс шифрования и дешифрования цифрового содержимого XML, основанный на определенных алгоритмах. Основным элементом синтаксиса XML Encryption является элемент **EncryptedData**, который совместно с элементом **EncryptedKey** используется для транспортировки ключей шифрования от их создателя к известному получателю.

Данные, подлежащие шифрованию, могут быть произвольными, это может быть XML-документ или элемент документа или содержимое элемента. При шифровании элемента или содержимого элемента, этот элемент или, соответственно, его содержимое, замещается элементом **EncryptedData**. При шифровании XML-документа целиком элемент **EncryptedData** становится корневым элементом нового документа или дочерним элементом в документе (определяется приложением).

СИНТАКСИС XML ENCRYPTION

Вот краткое описание синтаксиса элемента **EncryptedData**, здесь:

- ? означает ноль или одно вхождение;
- + означает одно или несколько вхождений;
- * означает ноль или несколько вхождений.

Пустой тег элемента означает, что элемент пуст.

```
<EncryptedData Id? Type?>
   <EncryptionMethod/>?
     <ds:KeyInfo>
              <EncryptedKey>?
              <AgreementMethod>?
              <ds:KeyName>?
              <ds:RetrievalMethod>?
              <ds:*>?
     </ds:KeyInfo>?
     <CipherData>
              <CipherValue>?
              <CipherReference URI?>?
     </CipherData>
   <EncryptionProperties>?
</EncryptedData>
```

Элемент **CipherData** может включать в себя зашифрованные данные или же ссылаться на них.

ПРИМЕР СИНТАКСИСА XML ENCRYPTION

```
<?xml version='1.0'?>
   <AuthorInfo xmlns='http://objectinnovations.com/Author'>
              <CompanyName>Object Innovations</CompanyName>
                     <Author>
                            <Name>G.GNANA ARUN GANESH</Name>
                            <Age>23</Age>
                            <Salary>50000</Salary>
                            <Department>.NET</Department>
                     </Author>
   </AuthorInfo>
```

ШИФРОВАНИЕ СОДЕРЖИМОГО XML-ЭЛЕМЕНТА (ЭЛЕМЕНТОВ)

Следующий код иллюстрирует XML-структуру с зашифрованным элементом <**Salary**>.

```
<?xml version='1.0'?>
   <AuthorInfo xmlns='http://objectinnovations.com/Author'>
     <CompanyName>Object Innovations</CompanyName>
```

```
      <Author>
        <Name>G.GNANA ARUN GANESH</Name>
        <Age>23</Age>

          <EncryptedData Type=
               'http://www.w3c.org/2001/04/xmlenc#Element'
             xmlns='http://www.w3c.org/2001/04/xmlenc#'>
             <CipherData>
                <CipherValue>A23d42U56N</CipherValue>
              </CipherData>
          </EncryptedData>

        <Department>.NET</Department>
      </Author>
  </AuthorInfo>
```

ЗАШИФРОВАННЫЙ ДОКУМЕНТ С ПОЛНОСТЬЮ СКРЫТЫМ СОДЕРЖИМЫМ

```
<?xml version='1.0'?>
   <EncryptedData
    xmlns='http://www.w3c.org/2001/04/xmlenc#'
    Type=
'http://www.isi.edu/in-notes/iana/assignments/mediatypes/text/xml'
   >
             <CipherData>
               <CipherValue>A23d42U56N</CipherValue>
             </CipherData>
   </EncryptedData>
```

СУПЕР-ШИФРОВАНИЕ: ШИФРОВАНИЕ УЖЕ ЗАШИФРОВАННЫХ ДАННЫХ

Супер-шифрование допускает повторное шифрование XML-документа, уже содержащего в себе зашифрованные части. При супер-шифровании документа с элементами **EncryptedData** или **EncryptedKey** вы должны зашифровать весь документ целиком. Повторное шифрование только этих элементов или их дочерних элементов будет недействительным.

Различные элементы в XML-документе можно зашифровать отдельно при помощи XML Encryption. В результате шифрования данных создается шифрованный текст. Он закодирован в формате base64, где для представления двоичных данных используется 64 символа. Как во включающей, так и во включенной подписи, шифрованный текст хранится в элементе **CipherValue**, который в свою очередь содержится в элементе **CipherData**. При использовании внешней подписи на местоположение шифрованного текста указывает **CipherReference**.

Спецификация управления ключами XML (XKMS – XML Key Management Specification)

XKMS – это спецификация, определяющая способы получения ключевой информации (то есть значений ключей, сертификаты и др.) от Web-службы. Спецификация XKMS состоит из двух частей:

- X-KISS – XML Key Information Service Specification (спецификация службы информации об XML-ключах);
- X-KRSS – XML Key Registration Service Specification (спецификация службы регистрации XML-ключей).

XKMS обеспечивает XML-интерфейс к инфраструктуре PKI (Public Key Infrastructure – инфраструктура открытых ключей)[1], включая распределение ключей, подтверждение ключей и управление ключами. PKI может основываться на различных спецификациях таких, как X.509/PKIX, SPKI или PGP. XKMS обуславливает метод, которым клиенты, основывающиеся на технологии XML, могут безопасно получать криптографические ключи. Главная цель протокола X-KISS состоит в том, чтобы минимизировать сложность приложений, использующих подпись XML. Протокол X-KRSS определяет порядок регистрации информации об открытом ключе.

Язык разметки утверждений безопасности SAML (Security Assertion Markup Language)

Спецификация SAML – языка разметки утверждений безопасности (или «языка разметки допуска к информации» – соответствующий русский термин еще не устоялся) представляет собой XML-стандарт и протокол для обмена между бизнес-партнерами аутентификационной и авторизационной информацией, обмена, не зависящего от используемых различными партнерами систем защиты. Стандарт SAML устанавливает организация продвижения стандартов структурированной информации (The Organization for the Advancement of Structured Information Standards – OASIS).

SAML заменяет две предыдущие инициативы этой организации, протоколы аутентификации и авторизации S2ML и AuthXML. Протокол SAML предназначен для аутентификации и авторизации пользователей. SAML предусматривает не только регистрацию пользователя в системе, но и автоматизированные транзакции, необходимые для безопасного обмена. SAML обеспечивает принцип однократной регистрации, то есть система принимает регистрацию пользователя только один раз, а затем предоставляет ему доступ к различным ресурсам, основываясь на уже предъявленных свидетельствах.

---

[1] PKI представляет собой сетевую архитектуру, которая обеспечивает улучшенную защиту при помощи криптографических ключей. PKI включает в себя криптографию с открытым ключом и цифровые подписи для аутентификации пользователей, участвующих в транзакции.

Глобальная архитектура XML Web-служб (Global XML Web Services Architecture – GXA)

Web-службы строятся на основе спецификаций XML, SOAP, WSDL и UDDI (Universal Description, Discovery and Integration – «универсальное описание, раскрытие и интеграция»). Эти базовые спецификации определяют основы интеграции и агрегации приложений. Однако функциональность более высокого уровня, например, безопасность, маршрутизация, обмен сообщениями и транзакции, необходимо надстраивать над архитектурой Web-служб для того, чтобы реализовать сценарии реального времени для нужд предприятия. В апреле 2001 года корпорации Microsoft и IBM предложили проект архитектуры для разработки XML Web-служб в W3C Workshop, предназначенную для разрешения проблем, стоящих перед программистами, как основу для дополнительных спецификаций[1].

Данный проект был прототипом архитектуры Microsoft Global XML Web Services. Microsoft GXA[2] представляет собой рамочный протокол, предназначенный для создания последовательной модели построения инфраструктурных протоколов для Web-служб и Web-приложений. Microsoft планирует представить спецификации GXA для утверждения в качестве стандартов, что сделает GXA открытой архитектурой. GXA является просто набором модульных, дополняющих спецификаций, расширяющих SOAP так, чтобы сделать возможным разработку усовершенствованных Web-служб реального времени. Спецификация GXA применима для самых разных сценариев, от решений B2B и EAI, до одноранговых приложений и служб B2C.

СПЕЦИФИКАЦИИ GXA

Спецификации GXA и определения, предложенные Microsoft, выглядят следующим образом:

- **WS-Security:** гибкая система, предназначенная для использования в качестве основы для конструирования разнообразных моделей защиты, включая PKI, Kerberos и SSL. WS-Security обеспечивает поддержку множественных маркеров безопасности, множественных доверенных доменов, множественных форматов цифровой подписи и множественных технологий шифрования.
- **WS-Routing:** это простой, основанный на SOAP, протокол для маршрутизации сообщений SOAP асинхронным методом через различные протоколы-носители такие, как TCP, UDP и HTTP.
- **WS-Inspection:** эта спецификация определяет XML-формат для инспекции сайтов на предмет наличия доступных служб, а также набор правил относительно способов, какими такая информация должна

[1] Подробней о проекте «Security in a Web Service World: A Proposed Architecture and Roadmap» см. *http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnwssecur/html/securitywhitepaper.asp*.

[2] Подробней о Microsoft GXA см. *http://msdn2.microsoft.com/en-us/library/ aa479664.aspx*.

предоставляться. Документ WS-Inspection определяет способы агрегирования документов-описаний существующих служб, которые могут существовать в разных форматах.

- **WS-Refferal:** данный протокол определяет динамически конфигурируемые стратегии маршрутизации, используемые узлами SOAP для передачи сообщений.
- **WS-Coordination:** эта рамочная и расширяемая спецификация служит для определения протоколов координации действий между распределенными приложениями. Подобные протоколы координации используются для поддержки многих приложений, включая такие, где необходимо достижение одновременного согласия сторон на результат транзакции.
- **WS-Transaction:** эта спецификация описывает типы координации, используемые при реализации расширенной координации на основе WS-Coordination.
- **WS-ReliableMessaging:** описывает протокол надежной доставки сообщений между распределенными приложениями при наличии отказов приложений, систем и сети.
- **WS-Addressing:** описывает независимые от способа транспортировки механизмы адресации Web-служб и сообщений.
- **WS-Attachment:** эта спецификация определяет абстрактную модель вложений в сообщения SOAP и, основываясь на этой модели, определяет механизм инкапсуляции сообщения SOAP и нескольких или ни одного вложения в сообщения DIME (Direct Internet Message Encapsulation – прямая инкапсуляция сообщений Internet).

Архитектура Web-служб Microsoft (Расширенная) схематически изображена на рисунке 10.8.

WS–Security

WS-Security[1] – это спецификация, определяющая принципы создания безопасных Web-служб. 11 апреля 2002 года IBM, Microsoft и VeriSign совместно предложили «спецификацию безопасности Web-служб» – (Web Services Security) – WS-Security.

Эта спецификация призвана помочь предприятиям в построении безопасных и широко взаимодействующих друг с другом Web-служб и приложений. Хотя WS-Security и не предлагает окончательных решений проблем безопасности, она обеспечивает способ построения других спецификаций, которые будут на ней основываться. На рисунке 10.9 изображена «дорожная карта» спецификации WS-Security.

WS-Security является базовой спецификацией и помогает обеспечивать безопасность пересылаемых сообщений. Она объясняет, как следует присоединять к сообщениям SOAP маркеры безопасности, включая двоичные маркеры такие, как сертификаты X.509 и «билеты» Kerberos.

[1] Подробней о спецификации WS-Security см. *http://msdn.microsoft.com/webservices/understanding/default.aspx?pull=/library/en-us/dnglobspec/html/wssecurspecindex.asp*.

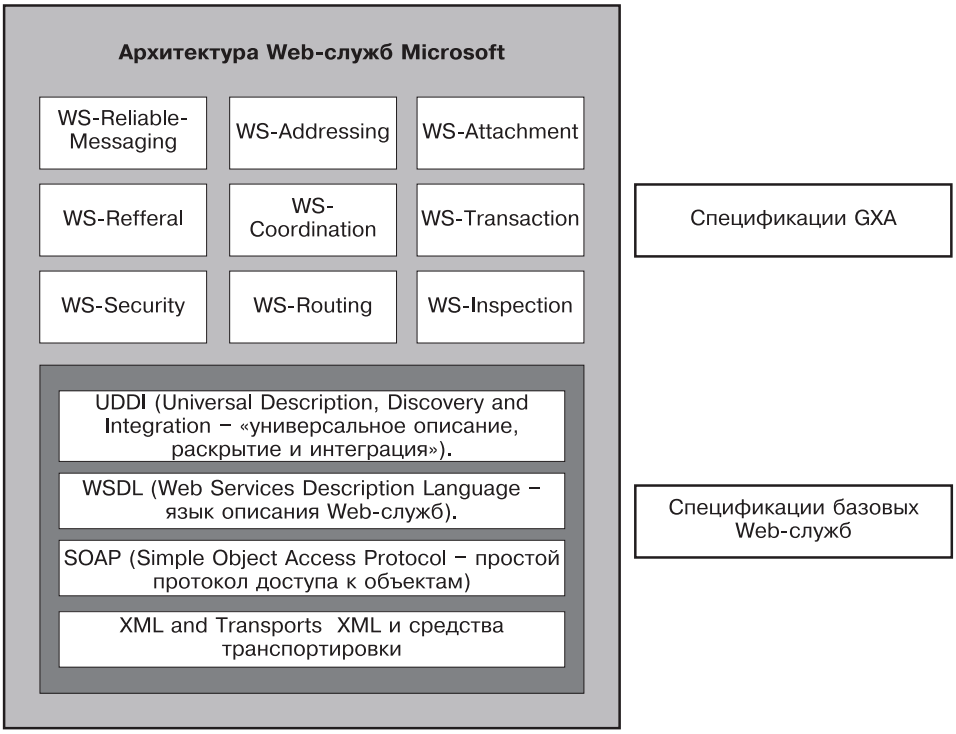


Рис.10.8. Расширенная архитектура Web-служб Microsoft

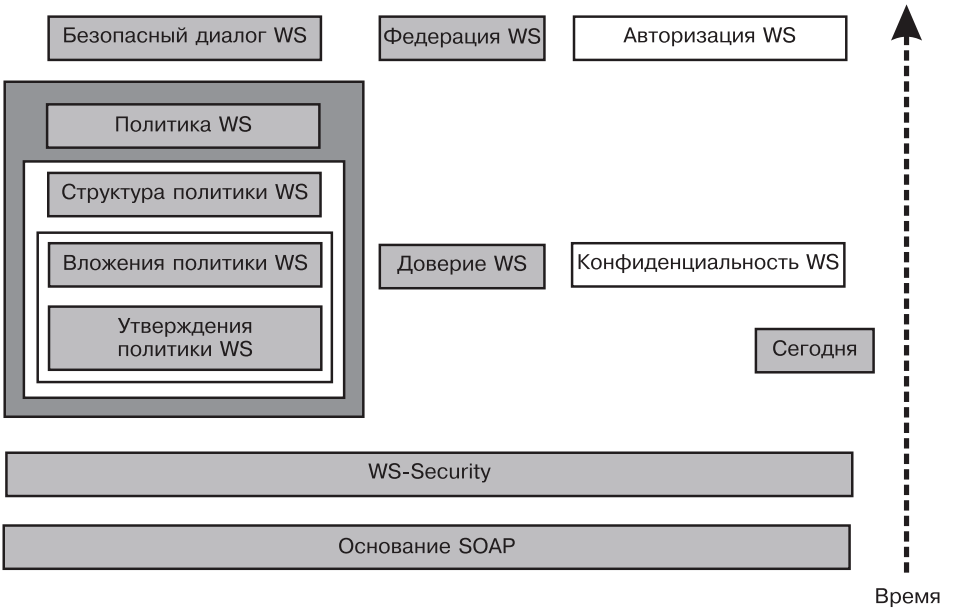


Рис.10.9. Развертывание «дорожной карты» WS-Security

Начальная спецификация WS

WS-SECURITY

WS-Security – это спецификация, определяющая различные способы аутентификации присоединением цифровой подписи и зашифрованного заголовка к сообщению SOAP таким образом, чтобы обеспечить защиту целостности и конфиденциальности сообщения, передаваемого между различными бизнес-приложениями.

WS-POLICY

WS-Policy – спецификация, которая описывает функциональные возможности политик безопасности в среде передачи и в конечных пунктах передачи сообщений (например, требуемые маркеры безопасности, поддерживаемые алгоритмы шифрования, правила конфиденциальности и т. п.). Она описывает деловые политики, политики безопасности, политики доверия и политики конфиденциальности, которые определяют, каким образом бизнес-приложения интегрируются друг с другом. Впоследствии спецификация WS-Policy была уточнена дополнением четырех новых документов – Web Services Policy Framework (WS-Policy), Web Services Policy Attachment (Вложения политики WS), Web Services Policy Assertions Language (Язык утверждений политики WS) и Web Services Security Policy (Политика безопасности WS). Документ Web Services Security Policy был опубликован, как открытая спецификация, 18 декабря 2002 года.

WS-TRUST

WS-Trust определяет расширения WS-Security, описывающие порядок запроса и выдачи маркеров безопасности, также она описывает порядок работы с доверительными отношениями между бизнес-приложениями. Эта спецификация задает основу для построения моделей доверия, которые обеспечивают безопасное взаимодействие Web-служб. Документ WS-Trust был опубликован 18 декабря 2002 года.

WS-SECURE CONVERSATION

WS-Secure Conversation (Безопасный диалог WS) – это спецификация, описывающая контроль и аутентификацию сообщений, которыми обмениваются Web-служба и ее клиент, включая безопасный контекстный обмен в сложных бизнес-транзакциях. WS-Secure Conversation был опубликован 18 декабря 2002 года.

WS-FEDERATION

WS-Federation – спецификация, определяющая механизмы для реализации федераций идентификации, учета, атрибутов, аутентификации и авторизации, простирающихся через области с различными степенями доверия. WS-Federation описывает модель интеграции несовпадающих механизмов безопасности (например, когда одна сторона использует систему RKI, а другая – Kerberos) или сходных механизмов (например, обе стороны используют Kerberos), которые функционируют в различных доменах. WS-Federation и профили федерации (WS-Federation Active Requestor Profile – профиль активной запрашивающей стороны и Federation Passive Requestor Profile – профиль пассивной запрашивающей стороны) были опубликованы 8 июля 2003 года.

Следующие шаги спецификаций

Следующие спецификации находятся в стадии разработки в группе OASIS.

WS-PRIVACY

Спецификация WS-Privacy описывает, как Web-служба и запрашивающая сторона специфицируют политики и предпочтения безопасности.

WS-AUTHORIZATION

WS-Authorization посвящена управлению авторизационными данными и политиками авторизации.

Почему WS-Security?

Как мы обсуждали ранее, защита на уровне HTTP и HTTPS обеспечивает защиту только на уровне «точка-точка». Однако в Web-приложениях реального времени требуется сквозная защита. Защиту и целостность данных необходимо обеспечивать на многоэтапных маршрутах. WS-Security описывает, как обеспечивать сквозную безопасность на многоточечных маршрутах сообщений. Более того, WS-Security является гибкой и расширяемой спецификацией. Она включает в себя широкое разнообразие существующих моделей безопасности и технологий шифрования.

WS-SECURITY В ДЕТАЛЯХ

WS-Security использует множество знакомых и существующих стандартов и спецификаций в области безопасности. WS-Security реализует защиту Web-служб при помощи текущих, существующих стандартов таких, как Kerberos, PKI, XML Encryption, XML Signature и SSL. WS-Security предлагает рамочную основу для внедрения существующих технологий в сообщения SOAP вне зависимости от способа их транспортировки. Так-

же, WS-Security поддерживает распространение маркеров безопасности таких, как сертификаты X.509 или «билеты» Kerberos и многих других через множественные домены с разным уровнем доверия, используя множественные форматы цифровой подписи и множественные технологии шифрования.

В спецификации WS-Security элемент заголовка SOAP используется для транспортировки данных, необходимых для реализации технологий защиты. Например, при использовании подписи XML в заголовок может включаться информация о типе использованного ключа и само значение подписи. Аналогичным образом, при использовании шифрования XML в заголовке может помещаться информация о шифровании. Таким образом, WS-Security вместо определения формата указывает, как внедрить в сообщение SOAP информацию, заданную другими спецификациями.

Кроме внедрения информации в заголовки SOAP, WS-Security задает метод, которым мы можем передавать простые пользовательские свидетельства через элемент **UsernameToken**. Также она определяет, как передавать двоичные маркеры. Поскольку вся необходимая для технологий безопасности информация заключена в SOAP-части сообщения, WS-Security обеспечивает сквозную защиту для Web-служб. WS-Security обеспечивает расширение возможностей существующих сообщений SOAP, повышая тем самым качество защиты. На рисунке 10.10 изображен формат сообщения SOAP.

Рис.10.10. Формат сообщения SOAP по WS-Security

WS-Security использует три ключевых механизма для защиты Web-служб:

- распространение маркеров безопасности;
- целостность сообщения;
- конфиденциальность сообщения.

Распространение маркеров безопасности

В этом стандартном механизме маркер безопасности включается в сообщение SOAP. «Распространение маркеров безопасности» означает, что свидетельства безопасности передаются от отправителя к получателю. Отправитель и получатель могут быть клиентом, XML Web-службой или посредником. Поскольку маркер включается в сообщение SOAP, он прозрачен для внешнего мира. Возможность подделки маркера посредником остается до тех пор, пока маркер передается в форме открытого текста. Для обеспечения целостности и конфиденциальности сообщения при этом необходимо принять надлежащие защитные меры. WS-Security предлагает для этого лишь механизм общего назначения, ассоциирующий маркер с сообщением SOAP. Эта спецификация не требует, чтобы маркер относился к какому-то определенному типу. Это придает ей гибкость и совместимость со многими типами маркеров безопасности.

Необходимая для технологий защиты и адресованная получателю информация, включая маркеры безопасности, добавляется в заголовок SOAP <**Security**> отправителем или посредником. Если представленная в заголовке SOAP информация действительна, сообщение принимается; в противном случае оно будет отвергнуто. Сообщение SOAP, прежде чем достичь получателя, может включить в себя ноль или несколько заголовков <**Security**>, поскольку оно может маршрутизироваться через несколько посредников.

ПРИМЕР WS-SECURITY

Ниже приведен пример сообщения SOAP с заголовком <**Security**>, иллюстрирующий передачу свидетельств отправителя (сертификата X.509) для получателя **http://www.arunmicrosystems.netfirms.com/Gnv.asmx**.

```
<S:Envelope xmlns:S="http://www.w3c.org/2001/12/soap-envelope"
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
  <S:Header>
    ...
    <wsse:Security S:actor=
            "http://www.arunmicrosystems.netfirms.com/Gnv.asmx">
      <wsse:BinarySecurityToken
            xmlns:wsse=
                  "http://schemas.xmlsoap.org/ws/2002/04/secext"
            ValueType="wsse:X509v3"
            wsu:Id="X509Token"
            EncodingType="wsse:Base64Binary">
            AwSaRgguTQmVkopX...
      </wsse:BinarySecurityToken>
    </wsse:Security>
            ...
  </S:Header>
  ...
</S:Envelope>
```

Пространства имен, использованные в этом документе, приведены в таблице 10.2.

Таблица 10.2. Пространства имен, используемые в WS-Security

Префикс	Пространство имен
S	http://www.w3c.org/2001/12/soap-envelope
Ds	http://www.w3c.org/2000/09/xmldsig#
Xenc	http://www.w3c.org/2001/04/xmlenc#
M	http://schemas.xmlsoap.org/rp
Wsse	http://schemas.xmlsoap.org/ws/2002/07/secext
Wsu	http://schemas.xmlsoap.org/ws/2002/07/utility

Любой получатель может использовать информацию, содержащуюся в заголовке <**Security**>, если атрибут actor отсутствует. Блок заголовка <**Security**> содержит информацию о сообщении, связанную с безопасностью.

```
<wsse:BinarySecurityToken
...
</wsse:BinarySecurityToken>
```

Этот код указывает на маркер безопасности, ассоциированный с данным сообщением. В нашем случае мы указываем на сертификат X.509, закодированный при помощи base64.

Целостность сообщения

WS-Security определяет механизм, при помощи которого получатель может удостовериться, как в подлинности отправителя сообщения, так и в том, что сообщение не было искажено по маршруту следования. Для этой цели WS-Security определяет использование спецификации XML Signature. Механизм защиты целостности предусматривает использование многих видов подписи, потенциально наложенных различными сторонами, и допускает расширение с целью использования дополнительных форматов подписи. Как мы уже говорили, спецификация XML Signature определяет элемент <**Signature**>, а также его субэлементы для реализации подписи.

WS-Security надстраивает над этой основой механизм XML Signature, добавляя элемент <**SecurityTokenReference**> и элемент <**Signature**> для ссылки на маркер безопасности, указанный в заголовке SOAP <**Security**>. Подпись XML криптографически вычисляется с использованием содержимого сообщения SOAP и маркера безопасности. Получатель сообщения верифицирует подпись при помощи криптографического алгоритма дешифрования.

Ниже приведено сообщение SOAP с элементом <**SecurityTokenReference**>, оно иллюстрирует добавление подписи XML Signature к сообщению SOAP.

```
<S:Envelope
      xmlns:S="http://www.w3c.org/2001/12/soap-envelope"
      xmlns:ds="http://www.w3c.org/2000/09/xmldsig#"
      xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext" >
   <S:Header>

     <wsse:Security>
              <wsse:BinarySecurityToken
                     ValueType="wsse:X509v3"
                     EncodingType="wsse:Base64Binary">
                 wsu:Id="X509Token"
                 AwSaRgguTQmVkopX...
              </wsse:BinarySecurityToken>
              <ds:Signature>
                     ...
                <ds:KeyInfo>
                  <wsse:SecurityTokenReference>
                    <wsse:Reference URI="#X509Token"/>
                  </wsse:SecurityTokenReference>
                </ds:KeyInfo>
              </ds:Signature>
     </wsse:Security>

   </S:Header>
   ...
</S:Envelope>
```

Подпись вычисляется на основе сертификата X.509, специфицированного в заголовке <**Security**>.

Конфиденциальность сообщения

Мы обсуждали, как при помощи WS-Security реализуется защита целостности сообщения, и как распространяются маркеры безопасности. Однако целостность и аутентификация, сами по себе, еще недостаточны. Представьте, что вы отправляете сообщение, содержащее чувствительные данные. Оно аутентифицировано и подписано, но не зашифровано. Что может с ним произойти? Злоумышленник может его перехватить, прочесть и даже убедиться, что оно действительно исходит от вас и не искажено каким-нибудь другим злоумышленником! Итак, сообщение необходимо, кроме всего прочего, зашифровать. Для шифрования вы можете использовать симметричный или асимметричный алгоритм, в зависимости от ситуации. WS-Security определяет механизм защиты конфиденциальности

сообщений SOAP. Согласно WS-Security эта защита реализуется при помощи стандарта XML Encryption шифрованием частей сообщения SOAP. Спецификация описывает, каким образом элементы <**ReferenceList**>, <**EncryptedData**>, <**EncryptedKey**> и <**DataReference**>, определенные стандартом XML Encryption, следует использовать в заголовке <**Security**>.

Следующий код шифрует тело сообщения SOAP при помощи секретного ключа, разделяемого между отправителем и получателем.

```
<S:Envelope
  xmlns:S="http://www.w3c.org/2001/12/soap-envelope"
  xmlns:ds="http://www.w3c.org/2000/09/xmldsig#"
  xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
  xmlns:xenc="http://www.w3c.org/2001/04/xmlenc#">
   <S:Header>
            <wsse:Security>
                  <xenc:ReferenceList>
                        <xenc:DataReference URI="#enc1"/>
                  </xenc:ReferenceList>
            </wsse:Security>
   </S:Header>
   <S:Body>
     <xenc:EncryptedData
       xmlns:xenc="http://www.w3c.org/2001/04/xmlenc#"
       xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
       Type="http://www.w3c.org/2001/04/xmlenc#Element"
       wsu:Id="enc1">
       <xenc:EncryptionMethod
         Algorithm=
              "http://www.w3c.org/2001/04/xmlenc#3des-cbc"/>
              <xenc:CipherData>
                    <xenc:CipherValue>dwSaRgguTQmVkopX...
                    </xenc:CipherValue>
              </xenc:CipherData>
     </xenc:EncryptedData>
   </S:Body>
</S:Envelope>
```

Обзор стандартов в области защиты Web-служб приведен в таблице 10.3.

РАСШИРЕНИЯ WEB-СЛУЖБ – БИБЛИОТЕКА WSE (WEB SERVICES ENHANCEMENTS 1.0) ДЛЯ MICROSOFT .NET

Корпорация Microsoft выпустила WSE 1.0 для Microsoft .NET 5 декабря 2002 года. Этот пакет заменяет WSDK Technical Preview. WSE 1.0 для Microsoft .NET – это новая библиотека классов .NET, при помощи которой вы можете реализовать последние протоколы Web-служб, включая WS-Security, WS-Routing, DIME и WS-Attachments.

При помощи библиотеки WSE и .NET Framework разработчики могут встроить в свои Web-приложения такие функции, как защита, маршрутизация и вложения. Библиотека WSE является надстройкой над .NET Framework и предоставляет разработчикам возможности использования новейших протоколов в XML Web-службах. Ключевой частью WSE является класс **Microsoft.Web.Services.SoapContext**, который обеспечивает интерфейс для обследования заголовка WS-Security и других заголовков на предмет входящих сообщений SOAP, и для добавления сообщений SOAP в заголовок WS-Security и другие заголовки исходящих сообщений.

Если вы установили WSE 1.0, то у вас установлены следующие элементы:

- сборка Microsoft.Web.Services (Microsoft.Web.Services.dll) в каталоге приложения;
- в файл **Machine.config** при установке WSE 1.0 добавляется следующее:

```
<mscorlib>
  <cryptographySettings>
    <cryptoNameMapping>
      <cryptoClasses>
        <cryptoClass Sha1=
              "System.Security.Cryptography.SHA1Managed" />
      </cryptoClasses>
      <nameEntry name=SHA1" class="Sha1" />
     <nameEntry
             name="System.Security.Cryptography.SHA1"
             class="Sha1" />
    </cryptoNameMapping>
  </cryptographySettings>
</mscorlib>
```

- документация WSE;
- примечания к примерам для «быстрого старта» (quick start samples) в WSE и примечания по релизу.

Библиотека WSE обеспечивает следующие функциональные возможности для реализации защиты передаваемых сообщений SOAP:

- **Добавление свидетельств безопасности в сообщение SOAP:** при помощи WSE вы можете добавить одно или несколько свидетельств в сообщение SOAP. Благодаря этому защита будет обеспечена не на уровне сеанса, а на всем пути сообщения от отправителя к получателю, даже если сообщение будет проходить по маршруту следования через приложения-посредники. Таким образом, вместо предъявления свидетельств на транспортном уровне, вы переносите их на уровень сообщения.
- **Наложение цифровой подписи на сообщение SOAP:** при помощи WSE вы можете наложить на сообщение SOAP цифровую подпись,

которая докажет получателю, что сообщение отправлено вами, и не было искажено по пути следования.

- **Шифрование сообщений SOAP:** при помощи WSE вы можете зашифровать сообщение SOAP и гарантировать тем самым, что прочесть его сможет только получатель.

Таблица 10.3. Стандарты в области защиты Web-служб

Технология	Функция защиты	Описание
Базовый HTTP	Аутентификация	HTTP и его защитные механизмы предназначены для соединений «точка-точка»
HTTPS (SSL 3.0/ TLS 1.0	Конфиденциальность, целостность данных и аутентификация (шифрование)	Защищенные сеансы «точка-точка» между клиентом и сервером
XML Signature	Аутентификация, целостность данных и подтверждение обязательств (аутентификация)	Необходимая предпосылка для реализации WS-Security. Защищает сообщение от искажения злоумышленником
XML Encryption	Целостность данных и конфиденциальность (шифрование)	Необходимая предпосылка для реализации WS-Security. Предотвращает чтение злоумышленником содержимого XML-сообщения
XML Key Management Specification (XKMS)	Аутентификация, конфиденциальность и целостность данных (обмен ключами)	XKMS обеспечивает XML-интерфейс к PKI, включая распределение и подтверждение ключей, управление ключами
Security Assertion Markup Language (SAML)	Аутентификация и авторизация	Обеспечивает универсальную аутентификацию и авторизацию (принцип однократной регистрации)
WS-Security	Аутентификация, шифрование и целостность данных	WS-Security – это рамочная спецификация, направленная на обеспечение безопасности пересылаемых сообщений. Она иллюстрирует использование XML Encryption и XML Signature для заголовков SOAP

ОБРАБОТКА СООБЩЕНИЙ SOAP, ПОДПИСАННЫХ ПРИ ПОМОЩИ USERNAMETOKEN

Давайте рассмотрим, как XML Web-служба обрабатывает сообщение SOAP, подписанное при помощи **UsernameToken**. Процедура состоит из следующих шагов:

1. Добавить в проект ASP.NET Web-службы ссылку на сборку Microsoft.Web.Services.SoapContext.
2. Добавить раздел конфигурации **microsoft.web.services** в конфигурационный файл. Добавьте элемент <**section**> в раздел <**configuration**> требуемого файла **Web.config**. Приведенный ниже код показывает, как настроить этот раздел. Атрибут «type» элемента <**section**> должен помещаться в одной строке (в тексте на книжной странице он разбит на несколько строк для удобства чтения).

```
<configuration>
    <configSections>
          <section name=" microsoft.web.services"
type = "Microsoft.Web.Services.Configuration.
WebServicesConfiguration, Microsoft.Web.Services,
Version=1.0.0.0, Culture-neutral,
PublicKeyToken=31bf3856ad364e35" />
    </configSections>
</configuration>
```

3. В файле **Web.config** вашей XML Web-службы добавьте элемент <**add**> в раздел <**soapExtensionTypes**>. Следующий код необходимо добавить в конфигурацию в файле **Web.config**, для того чтобы библиотека WSE работала в вашей XML Web-службе. Атрибут «type» элемента <**add**> должен помещаться в одной строке (в тексте на книжной странице он разбит на несколько строк для удобства чтения).

```
<configuration>
    <system.web>
          <webServices>
                <soapExtensionTypes>
type="Microsoft.Web.Services.WebServicesExtension,
Microsoft.Web.Services Version 1.0.0.0,
Culture-neutral, PublicKeyToken=31bf3856ad364e35"
                      priority="1"
                      group="0"/>
                </soapExtensionTypes>
          </webServices>
    </system.web>
</configuration>
```

Сборка Microsoft.Web.Services должна быть доступна из приложения ASP.NET; в противном случае приведенная выше конфигурация работать не будет. Таким образом, сборка Microsoft.Web.Services должна находиться каталоге bin приложения или в глобальном кеше сборок GAC (Global Assembly Cache). В среде Visual Studio .NET после задания ссылки на сборку Microsoft.Web.Services установите флажок Copy Local (копировать локально). Тем самым сборка будет скопирована в каталог bin приложения.

Давайте посмотрим теперь, как разрабатывается Web-служба, обрабатывающая сообщения SOAP, подписанные при помощи **UsernameToken**.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using Microsoft.Web.Services.Security;
using Microsoft.Web.Services;
using System.Security.Cryptography;

namespace WSEUsernameToken
{
   public class Service1 : System.Web.Services.WebService
   {
   [WebMethod]
   public string Hello()
   {
              SoapContext requestContext =
                     HttpSoapContext.RequestContext;
   //Проверяем, получен ли SOAP-запрос
     if (requestContext == null)
     {
              throw new
                     ApplicationException("Не SOAP-запрос или WSE
                            не установлена надлежащим образом.");
     }
     UsernameToken Token1 =
              GetToken( requestContext.Security );
     if ( Token1 != null)
     {
              value = "Hello";
     }
     return value;
}
private UsernameToken GetToken( Security sec )

   {
   UsernameToken value = null;
```

```
            if ( sec.Tokens.Count > 0 )
            {
                  foreach ( SecurityToken tok in sec.Tokens )
                  {
                        value = tok as UsernameToken;
                        if ( value != null )
                        {
                        return value;
                        }
                  }
            }
   return value;
   }
}
}
```

Поскольку каждый заголовок WS-Security может содержать ноль или несколько маркеров безопасности, мы просматриваем их все при помощи цикла foreach и возвращаем **UsernameToken.**

Для обработки SOAP-сообщения, подписанного при помощи **Username-Token** необходимо выполнить два основных шага. Во-первых, добавить класс, реализующий интерфейс **IPasswordProvider.** Соответствующий код приведен ниже.

```
/* Рекомендуется, чтобы сборка, имеющая доступ к этому классу, уже
имела разрешение на вызов неконтролируемого кода, поскольку сборка
Microsoft.Web.Services является единственной сборкой,обращающейся
К этому классу. Таким образом, примените атрибут
SecurityPermissionAttribute
к классу, реализующему IPasswordProvider, затребовав разрешение
UnmanagedCode */

[SecurityPermission(SecurityAction.Demand,
 Flags= SecurityPermissionFlag.UnmanagedCode)]
public class PasswordProvider: IPasswordProvider
   //Реализовать метод GetPassword для
   //интерфейса IPasswordProvider
public string GetPassword(UsernameToken userName)
{
   //код, приведенный ниже, служит только для целей обучения.
   //В реальных приложениях такой код обычно запрашивает внешнюю
   //базу данных, где получает имена пользователей и хеши паролей.
   //Здесь, для простоты, мы используем имена пользователей
   //в кодировке UTF-8.
   byte[] encodeUserName =
            System.Text.Encoding.UTF8.GetBytes
                  (username.Username);
   return System.Text.Encoding.GetString(encodedUserName)
}
```

Когда класс будет зарегистрирован в файле Web.config, библиотека WSE вызовет метод **GetPassword** этого класса, если сообщение подписано при помощи **UsernameToken** и из него необходимо извлечь пароль для введенного имени пользователя.

Теперь сконфигурируем класс, реализующий **IPasswordProvider**, в конфигурационном файле **Web.config** для нашей XML Web-службы, как показано ниже. Значение атрибута «type» должно помещаться в одной строке (в тексте на книжной странице оно разбито на несколько строк для удобства чтения).

```
<microsoft.web.services>
   <security>
              <passwordProvider type =
              "MyNamespace.PasswordProvider, MyAssemblyName,
              Version=1.0.0.0,
              Culture=neutral,
              PublicKeyToken=81f0828a1c0bb867" />
   </security>
</microsoft.web.services>
```

Мы конфигурируем тип **MyNamespace.PasswordProvider** так, чтобы он был вызван в случае, когда XML Web-служба, использующая данный файл **Web.config**, получит сообщение SOAP, подписанное при помощи **UsernameToken**.

НАЛОЖЕНИЕ ПОДПИСИ НА СООБЩЕНИЕ SOAP ПРИ ПОМОЩИ USERNAMETOKEN

Теперь посмотрим, как клиент XML Web-службы подписывает SOAP-запрос, используя **UsernameToken**.

1. Добавьте в проект Web-ссылки на **Microsoft.Web.Services.dll** и **System.Web.Services.dll**.
2. Добавьте в проект Web-ссылку на ту Web-службу, которой будет адресован наш SOAP-запрос.
3. Отредактируйте класс proxy так, чтобы он производился из **WebServicesClientProtocol**. Для этого щелкните правой кнопкой мыши на файле **Reference.cs** в окне просмотра Solution Explorer и выберите команду View Code. Отредактируйте код класса – измените:

```
public class Service1 :
System.Web.Services.Protocols.SoapHttpClientProtocol
```

на

```
public class Service1 :
Microsoft.Web.Services.WebServicesClientProtocol.

/* В среде Visual Studio .NET, если вы щелкнете на Update
```

```
Web Reference, proxy-класс будет сгенерирован заново, исходный
класс будет возвращен к SoapHttpClientProtocol, и вам,
соответственно, придется
редактировать proxy-класс снова. */
```

4. Добавьте в код приложения-клиента следующие директивы:

```
using Microsoft.Web.Services;
using Microsoft.Web.Services.Security;
using System.Security.Cryptography;
```

Код клиентского приложения должен выглядеть следующим образом:

```
localhost.Service1 proxy = new localhost.Service1();
/* Создайте новый экземпляр UsernameToken,
указав имя пользователя, пароль,
и способ пересылки пароля в сообщении SOAP. */
UsernameToken userToken = new UsernameToken(username,
password, PasswordOption.SendHashed);
proxy.RequestSoapContext.Security.Tokens.Add(userToken);
//Подписывает сообщение SOAP при помощи UsernameToken.
Proxy.RequestSoapContext.Security.Elements.Add(new
Signature(userToken));
//Вызывать Web-службу
label1.Text = proxy.Hello();
```

РАСШИРЕНИЯ WEB-СЛУЖБ – БИБЛИОТЕКА WSE 2.0 ДЛЯ MICROSOFT .NET

Корпорация Microsoft выпустила WSE 2.0 для Microsoft .NET 15 июля 2003[1] года. Она заменяет WSE 1.0 Technical Preview, которая была выпущена 5 декабря 2002 года. При помощи WSE 2.0 вы можете реализовать последние протоколы из области защиты XML Web-служб, включая WS-Addressing, WS-Policy, WS-SecurityPolicy, WS-Trust и WS-SecureConversation.

Вот новые возможности, появившиеся в WSE 2.0 Technical Preview:

1. Библиотека WSE версии 2.0 поддерживает новые спецификации Web-служб, включая WS-Addressing, WS-Policy, WS-SecurityPolicy, WS-Trust и WS-SecureConversation.
2. В WSE версии 2.0 вы можете задать требования к отправке и приему сообщений при помощи конфигурационных файлов (декларация политик при помощи конфигурации).
3. Библиотека WSE версии 2.0 предоставляет возможность программно запрашивать маркер безопасности при помощи сообщения SOAP, и этот маркер можно использовать для серии сообщений между отправителем запросов и Web-службой.

[1] Подробней о WSE 2.0 см. *http://msdn.microsoft.com/webservices/building/wse/default.aspx*.

4. Библиотека WSE версии 2.0 обеспечивает для SOAP-сообщений авторизацию, основанную на ролях, конструируя «пользователя» на основе маркера безопасности из сообщения SOAP.
5. Библиотека WSE версии 2.0 поддерживает маркеры Kerberos. Эта поддержка зависит от операционной системы.
6. При работе с сообщениями SOAP, библиотека WSE версии 2.0 обеспечивает гибкий механизм отправки и приема сообщений. Эта легкая программная модель, ориентированная на сообщения, позволяет приложениям легко переключаться между транспортными протоколами TCP и HTTP.
7. Библиотека WSE версии 2.0 обеспечивает поддержку маркеров безопасности XML таких, как маркеры безопасности XrML и SAML.
8. Примеры для «быстрого старта» (Quick Start Samples) представлены на языках C# и Visual Basic .NET.

Заметим, что релиз Technical Preview предназначен только для целей тестирования; это программное обеспечение не должно использоваться в промышленных целях, не должно распространяться и не поддерживается Microsoft. Познакомьтесь с WSE 2.0 Technical Preview, прежде чем выйдет финальная версия WSE 2.0.

Организации

В процессе поиска решений для устранения недостатков Web-служб участвуют различные организации:

- Консорциум W3C;
- Организация продвижения стандартов структурированной информации OASIS (The Organization for the Advancement of Structured Information Standards);
- Проблемная группа проектирования Internet – IETF (Internet Engineering Task Force);
- Организация взаимодействия Web-служб WS-I (Web services Interoperability Organization);
- Производители программного обеспечения такие, как Microsoft, IBM, Sun, BEA, e-Speak, IONA и Hewlett-Packard.

Рабочие группы W3C совершенствуют спецификации как SOAP 1.1, так и WSDL 1.1. Рабочая группа протокола XML и рабочая группа описания Web-служб параллельно с этим работают над стандартизацией SOAP 1.2 и WSDL 1.2, соответственно. В дополнение к цифровой подписи XML и XML Encryption W3C разрабатывает спецификацию управления ключами XML (XML Key Management Specification) и архитектуру Web-службы, которая включает в себя рамочную спецификацию протоколов защиты.

Ниже приведено описание целей, которые ставят перед собой важные организации, занятые разработкой стандартов в области защиты Web-служб.

- **Технический комитет по защите Web-служб OASIS – WSS TC (OASIS Web Services Security Technical Committee):** 11 апреля 2002 года IBM, Microsoft и VeriSign представили набор спецификаций, получивший наименование WS-Security, который расширяет защиту SOAP и основывается на существующих стандартах безопасности Web-служб. WSS TC продолжает работу над основами безопасности Web-служб, как описано в спецификации WS-Security. Работа WSS TC составит необходимое техническое основание для служб защиты более высоких уровней, которые должны быть определены другими спецификациями.
- **Рабочая группа XML Signature:** цель этой рабочей группы состоит в развитии XML-совместимого синтаксиса, используемого для представления подписи Web-ресурсов и частей сообщений (всего, на что можно сослаться при помощи URI) и процедур для вычисления и проверки цифровой подписи. Это объединенная рабочая группа IETF и W3C.
- **Рабочая группа XML Encryption:** цель этой рабочей группы состоит в разработке процесса шифрования/дешифрования цифрового содержимого (включая XML-документы и их части) и синтаксиса XML, позволяющего представлять, как зашифрованные данные, так и информацию, необходимую получателю для дешифрования.
- **Технический комитет по службам защиты на основе XML – SSTC (XML-Based Security Services Technical Committee):** этот технический комитет работает над SAML, основанным на XML стандартом для обмена аутентификационной и авторизационной информацией.
- **Организация взаимодействия Web-служб – WS-I (Web services Interoperability Organization):** WS-I является открытой промышленной организацией, призванной развивать взаимодействие между Web-службами, основывающихся на разных платформах, операционных системах, языках программирования. Эта организация взаимодействует с промышленностью и службами стандартизации, обеспечивая руководство, практические примеры и ресурсы для разработки эффективных решений в области Web-служб.

OASIS и W3C фокусируются на ключевых темах, перечисленных в таблице 10.4, с тем чтобы создать рамочную основу для построения защиты Web-служб.

Таблица 10.4. Стандарты в области защиты Web-служб и их состояние

Стандарт (предлагаемый или принятый)	Орган, разрабатывающий стандарт и состояние
WS-Security	OASIS: формирование технического комитета, WSS TC
XML Digital Signature	W3C: завершено
XML Encryption	W3C: финальное голосование в комитете
XKMS	W3C: рабочий проект
SAML	OASIS: финальное голосование
XACML	OASIS: пересмотр в комитете
SSL/TLS	IETF: RFC 2246
Kerberos	IETF: RFC 1510

Итоги главы

В этой главе мы изучали проблемы безопасности Web-служб. Вначале мы рассмотрели некоторые современные методы защиты Web-служб такие, как SSL/TLS и брандмауэры. Затем мы исследовали криптографические технологии XML: подпись XML, шифрование XML, XKMS и SAML. Наконец, мы обсудили спецификацию WS-Security и ее реализацию при помощи библиотеки WSE. Протоколы SSL/TLS и другие существующие технологии являются временными решениями, которые работают удовлетворительно при определенных ограничительных условиях. Если вы используете Web-службы прямо сейчас, мы советуем вам воспользоваться этими решениями. Но также мы советуем следить за развитием библиотеки WSE, которая реализует спецификацию WS-Security, и использовать ее, когда выйдет ее финальный релиз.

Приложение А

Пример атаки на код: перекрытие стека

Атака типа «перекрытие стека» – это хороший пример использования уязвимостей, часто становящихся мишенью злоумышленников. Самый знаменитый в истории Internet пример подобной атаки – это так называемый «червь Морриса». Это лишь один из многих типов уязвимостей, о которых нужно заботиться при написании безопасных программ. Атака такого типа крайне затруднительна в такой среде управляемого кода, как .NET или Java, но она много раз успешно реализовывалась против программ, написанных на C/C++, включая такие, как SQL Server и IIS. Изучение этого простого примера атаки поможет вам почувствовать и понять настроение и ход мысли вашего потенциального противника.

Пример программы **Win32ProjectBufferOverflow**, описанный в этом приложении, иллюстрирует общую концепцию атак типа «перекрытие стека»[1]. Техника этой атаки сводится к написанию такого кода, который в стеке попадает в область параметров, но при этом перекрывает собой адрес возврата из функции таким образом, что при возврате из функции управление получает код злоумышленника, а не тот код, который вызвал функцию. В приведенном далее листинге код злоумышленника представлен функцией **AttackerInsertedCode**. Если вы запустите программу, то увидите, что функция **AttackerInsertedCode** действительно выполняется, хотя нигде в коде программы обращений к ней нет.

Чтобы упростить понимание примера, здесь в одной программе совмещены атакующий и атакуемый коды. Собственно, оба кода совмещаются в одном файле исходного текста, что очень удобно для демонстрации принципа, но конечно совершенно не соответствует реальным сценариям. В реальном мире атакующий код, конечно, содержится в совершенно другой программе, которая сообщается с атакуемой программой через сокет. Тем не менее, принцип действия в демонстрационном примере будет тем же самым, что и в реалистичном сценарии. Вот что еще следует помнить при реализации данного примера программы, – если вы компилируете программу заново при помощи компилятора другой версии, или вы как-то меняете код, а затем компилируете программу заново, то, вероятнее

[1] Заметим, что в программах Visual C/C++ для обнаружения переполнений буфера можно использовать ключ /GS, что эффективно предотвратит появление большинства уязвимостей для атак такого типа.

всего, необходимо будет изменить данные, предназначенные для перекрытия стека, поскольку компоновка кода и стека может измениться. В данном проекте проверка стека в период выполнения отключена, и если вы включите эту опцию компилятора, программа надлежащим образом работать не будет.

```
// Win32ProjectBufferOverflow.cpp
...
#define BUFLENGTH 16
//следующий фрагмент кода написан небрежным программистом
void VulnerableFunction(char * str)
{
   char buf[BUFLENGTH];
   strcpy(buf, str); //вот, она, уязвимость!

   printf("Вызов уязвимой функции.\n\n");
}

//следующий фрагмент кода написан злонамеренным программистом
void AttackerInsertedCode()
{
   printf("Оп-па! ... Атакующий код получил управление!\n");
   while (true); //возврата не будет
}
void main()
{
   //вызов уязвимой функции безопасным способом
   printf("Вызвать уязвимую функцию безопасным способом.\n");
   VulnerableFunction("hello");

   //вызов уязвимой функции опасным способом
   printf("Вызвать уязвимую функцию негодным способом.\n");
   char bufNasty[25];
   //убедиться, что нет промежуточных нулей
   for (int i=0; i<20; i++)
            bufNasty[i] = 0x01;
   //переписать адрес возврата адресом AttackerInsertedCode
   int * pRetAddres = (int *)&(bufNasty[20]);
   *pRetAddres = 0x004119A0;
   //маркер конца
   bufNasty[24] = (char)0x00; //фактически, излишне
   //делаем свое черное дело
   VulnerableFunction(bufNasty);
}
```

Суть уязвимости, которая здесь используется, заключается в небрежно написанном коде функции **VulnerableFunction**, где обращение к strcpy производится без учета соображений безопасности. Функция **strcpy** копирует содержимое буфера (включая ограничивающий null), указанного вторым параметром, в буфер, указанный первым параметром. Однако это копирование производится без контроля длины буфера.

В данных функции **VulnerableFunction** локальный буфер **buf** располагается в сегменте стека и занимает 16 байт. Если вы вызовете **VulnerableFunction** с параметром, указывающим на ограниченный значением null буфер, размер которого не превышает 16 байт, то все будет нормально. Если же вы передадите указатель на буфер, содержащий более 16 байт, то избыточные байты будут записаны в следующие доступные адреса в стеке, где среди прочего содержится адрес возврата функции, по которому она должна вернуть управление после завершения своей работы.

Если вы запустите программу, то получите приведенный ниже вывод. Обратите внимание, что при первом вызове функция **VulnerableFunction** отрабатывает нормально, поскольку ей был передан корректный буфер размером менее 16 байт. Но при втором вызове функции **VulnerableFunction** картина становится совершенно иной. Ей передается буфер размером 25 байт. В общем случае это была бы просто ошибка в программе, приводящая к переписыванию стека случайными данными, что привело бы к генерации исключения или даже просто никак не повлияло бы на работу программы. Но в этом случае данные, которые записываются в стек, специально подготовлены для достижения специфического результата. Вместо возврата управления в точку вызова и вместо генерации исключения произойдет следующее: управление будет передано функции **AttackerInsertedCode**, которая и будет выполнена.

```
Вызвать уязвимую функцию безопасным способом.
Вызов уязвимой функции.

Вызвать уязвимую функцию негодным способом.
Вызов уязвимой функции.

Оп-па! ... Атакующий код получил управление!
```

Но как же мы можем точно определить, какой длины должен быть наш буфер, переписывающий стек, и какие данные нужно в него поместить, чтобы наш злонамеренный код получил управление, вместо того чтобы функция нормально вернула управление в точку вызова? Чтобы выяснить это, необходимо использовать отладчик, устанавливать точки прерывания, анализировать дизассемблированный код, а также изучать работу некоторых регистров процессора и просматривать содержимое памяти в области стека. Только таким образом вы сможете выяснить адреса тех четырех байтов, где будет содержаться адрес возврата из функции **VulnerableFunction**. Также вам потребуется выяснить адрес вашей собственной функции **AttackerInsertedCode**.

Давайте посмотрим, как можно узнать адрес функции **AttackerInsertedCode**. Установите в отладчике точку прерывания на функции **AttackerInsertedCode** и запустите программу в отладчике. Затем щелкните на вкладке Disassembly и запишите адрес функции. Как видно на рисунке А.1, в нашем примере этот адрес равен 004119A0.

Далее посмотрим, где будет записан адрес возврата из функции **VulnerableFunction**. Тут требуются определенные навыки, но основной принцип сводится к следующему. Вначале установите точку прерывания на вызываемую функцию (рисунок А.2), а затем запустите программу в

отладчике. Когда вы достигнете точки прерывания, переключите режим просмотра на дизассемблированный код, чтобы увидеть ассемблерный код вызова функции.

Здесь вы увидите тот адрес возврата, который вам требуется подменить (**pRetAddress = 0x004119A0**), и также положение этого параметра в стеке, то есть место, куда необходимо поместить адрес вашего кода, который должен получить управление в результате атаки.

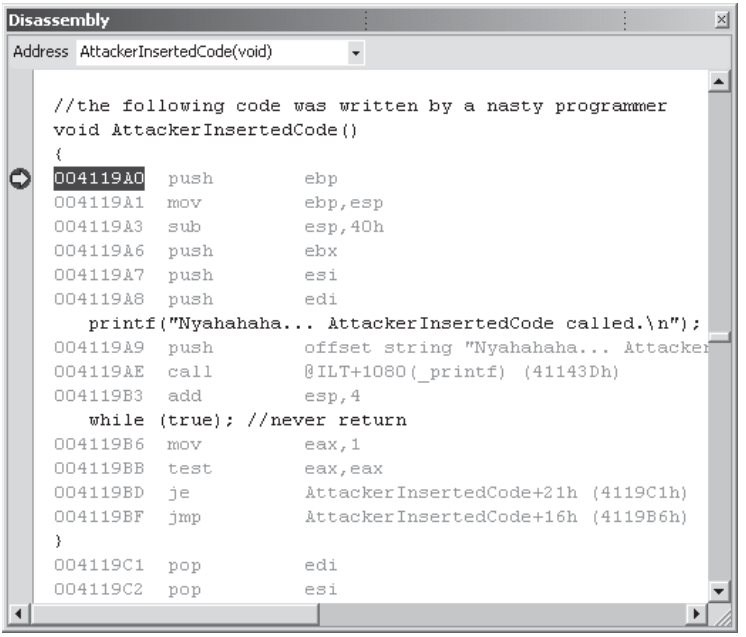


Рис. А.1. Определение адреса функции AttackerInsertedCode

```
//the following code was written by a nasty programmer
void AttackerInsertedCode()
{
   printf("Nyahahaha... AttackerInsertedCode called.\n");
   while (true); //never return
}
void main()
{
   //call the vulnerable function in a safe way
   printf("Call VulnerableFunction in a safe way.\n");
   VulnerableFunction("hello");

   //call the vulnerable function in a nasty way
   printf("Call VulnerableFunction in nasty way.\n");
   char bufNasty[25];
   //make sure there are no intervening nulls
   for (int i=0; i<20; i++)
      bufNasty[i] = 0x01;
   //overwrite return address with AttackerInsertedCode
   int * pRetAddres = (int *)&(bufNasty[20]);
   *pRetAddres = 0x004119A0;
   //final null terminator
   bufNasty[24] = (char)0x00; //actually redundant
```

Рис. А.2. Установка точки прерывания на функцию VulnerableFunction

Приложение B

Как работает шифр RSA

Для того чтобы оценить ту внутреннюю работу, которая производится внутри таких криптографических систем, как RSA, полезно ознакомиться с простой реализацией такой системы в коде. В этом приложении мы заглянем во «внутренний мир» алгоритма RSA. В чисто практическом смысле для реализации криптографических решений на основе классов .NET в этом нет никакой необходимости. Однако, даже самым заядлым «практикам» понимание принципов работы алгоритма окажется полезным.

Это приложение посвящено примеру программы **BigRSA**, которая иллюстрирует внутреннее устройство алгоритма RSA. Вряд ли вам когда-либо доведется собственноручно реализовать алгоритм RSA «с нуля», поскольку большинство криптографических библиотек, не исключая и .NET Security Framework, предоставляют превосходные его реализации. Данный пример программы приводится здесь лишь для более глубокого понимания работы алгоритма.

Модульная арифметика

Прежде чем знакомиться с кодом программы, давайте немного познакомимся с арифметикой «по модулю» (модульной арифметикой). Модульная арифметика используется во многих криптографических системах, включая RSA, DSA и SSL. Алгебраическая структура, которая здесь используется, обозначается Z_n и представляет собой множество неотрицательных целых чисел по модулю n. То есть Z_n состоит из множества чисел $\{0, 1, 2, \ldots n-1\}$ и операций сложения и умножения над ними. Заметим, что Z_n является конечной (циклической) целочисленной арифметикой, а не бесконечной числовой осью. Z_n иногда называют «циферблатной» арифметикой, поскольку двенадцатичасовой циферблат реализует арифметику Z_{12}.

Например, Z_{12} будет множеством чисел от 0 до 11 и определенной над ними операцией сложения, которая аналогична обычному сложению, но с одним отличием – если результат сложения превышает 11 (то есть число, на единицу меньшую модуля), счет продолжается с 0. Таким образом, в арифметике по модулю 12, как и в обычной, $1 + 0 = 1$, $1 + 1 = 2$ и $3 + 5 = 8$. Однако, когда результат сложения превысит 11, появится отличие: на-

пример, 10 + 5 = 3, а не 15. В обычной арифметике сумма 10 + 5 = 15, но, поскольку 15 больше 11, то при сложении по модулю 12 сумма будет равна 15 – 12, то есть 3. Другой пример: сумма 5 + 7 по модулю 12 будет равна 0, а не 12, поскольку 12 – 12 = 0. Идея модульной арифметики состоит в том, что задавшись определенным модулем, вы никогда не получите величин, равных модулю или превышающих его. Как ни удивительно, но этот прием дает огромные преимущества в реализации асимметричных алгоритмов. Конечно, в таких алгоритмах, как RSA, вы никогда не будете иметь дело с такими малыми числами, как 12, вам потребуются модули огромной величины, требующие значительного объема вычислений.

Пример программы BigRSA

Из главы 4 вы можете помнить основные вычислительные шаги, реализующие алгоритм RSA. В примере программы **BigRSA** вы заметите черты сходства с программой **TinyRSA** из главы 4. Основное отличие программы **BigRSA** состоит в использовании математической библиотеки GnuMP. Подробней об установке и использовании этой библиотеки см. приложение С.

Обратите внимание, в программе **BigRSA** число e выбирается случайным образом. Это сделано для того, чтобы показать: любое значение e такое, что $e < phi$ и НОД(e, phi) = 1, будет работать. На практике, впрочем, чаще всего используются величины e, равные 3, 17 и 65535 (0xFFFF).

```
unsafe static void Main(string[] args)
{
   //инициализация генератора случайных чисел
   gmp_randstate_struct state =
              new gmp_randstate_struct();
   GmpWrapper.__gmp_randinit_default(&state);

   //для задания начального числа используем дату и время
   DateTime dt = DateTime.Now; //текущая дата и время
   GmpWrapper.__gmp_randseed_ui(
              &state, (ulong)dt.Ticks);

  //получаем 256-битовое равномерно распределенное простое число p
   mpz_struct p;
   GmpWrapper.__gmpz_init(&p);
   GmpWrapper.__gmpz_urandomb(&p, &state, 256);
   GmpWrapper.__gmpz_nextprime(&p, &p);

   //отображаем простое число p
   Console.WriteLine(
              "p: " +
              GmpWrapper.__gmpz_get_str(null, 10, &p));
```

```
//получаем 256-битовое равномерно распределенное простое число q
 //теоретически, мы должны убедиться, что p != q
 //но здесь мы не будем себя этим обременять...
 //...если, конечно, вы вчера не выиграли в лотерею джек-пот.
mpz_struct q;
GmpWrapper.__gmpz_init(&q);
GmpWrapper.__gmpz_urandomb(&q, &state, 256);
GmpWrapper.__gmpz_nextprime(&q, &q);

//отображаем простое число q
Console.WriteLine(
          "q: " +
          GmpWrapper.__gmpz_get_str(null, 10, &q));

//pq = p*q
mpz_struct pq;
GmpWrapper.__gmpz_init(&pq);
GmpWrapper.__gmpz_mul(&pq, &p, &q);

//отображаем произведение pq
Console.WriteLine(
          "\npq:  " +
          GmpWrapper.__gmpz_get_str(null, 10, &pq));

//инициализируем эйлерову функцию phi = (p-1)*(q-1)

mpz_struct phi;
GmpWrapper.__gmpz_init(&phi);
mpz_struct one;
GmpWrapper.__gmpz_init(&one);
GmpWrapper.__gmpz_set_str(&one, "1", 10);
mpz_struct pminusone;
GmpWrapper.__gmpz_init(&pminusone);
GmpWrapper.__gmpz_sub(&pminusone, &p, &one);
mpz_struct qminusone;
GmpWrapper.__gmpz_init(&qminusone);
GmpWrapper.__gmpz_sub(&qminusone, &q, &one);
GmpWrapper.__gmpz_mul(&phi, &pminusone, &qminusone);

//отображаем phi
Console.WriteLine(
          "phi: " +
          GmpWrapper.__gmpz_get_str(null, 10, &phi));

//получаем значение e, взаимно простое с phi и < phi
mpz_struct e;
GmpWrapper.__gmpz_init(&e);
while(true)
{
```

```
            //получаем простое число - кандидат на роль e
            GmpWrapper.__gmpz_urandomb(&e, &state, 256);

            //e < phi?
            bool goodsize =
              GmpWrapper.__gmpz_cmp(&e, &phi) < 0;

            //НОД (наибольший общий делитель) для e и phi = 1?
            mpz_struct gcd;
            GmpWrapper.__gmpz_init(&gcd);
            GmpWrapper.__gmpz_gcd(&gcd, &e, &phi);
            bool relprime =
              GmpWrapper.__gmpz_cmp(&gcd, &one) == 0;

            //выход, если мы нашли искомое
            if (goodsize && relprime)
              break;
}

//отображаем e
Console.WriteLine(
            "\ne: " +
            GmpWrapper.__gmpz_get_str(null, 10, &e));

//вычисляем d = e^-1 mod phi
mpz_struct d;
GmpWrapper.__gmpz_init(&d);
GmpWrapper.__gmpz_invert(&d, &e, &phi);

//отображаем d
Console.WriteLine(
            "d: " +
            GmpWrapper.__gmpz_get_str(null, 10, &d));

//создаем открытый текст m
mpz_struct m;
GmpWrapper.__gmpz_init(&m);

while (true)
{
            //получаем 256-битовое равномерно распределенное
            //случайное сообщение m
            GmpWrapper.__gmpz_urandomb(&m, &state, 256);

            //необходимо убедиться, что m < pq,
            //иначе алгебра нас подведет
            if (GmpWrapper.__gmpz_cmp(&m, &pq) < 0)
            break;
}
```

```
   //отображаем открытый текст m
   Console.WriteLine(
              "\nm: " +
              GmpWrapper.__gmpz_get_str(null, 10, &m));

   //вычисляем шифрованный текст c = m^e mod pq
   mpz_struct c;
   GmpWrapper.__gmpz_init(&c);
   GmpWrapper.__gmpz_powm(&c, &m, &e, &pq);

   //отображаем шифрованный текст c
   Console.WriteLine(
              "c: " +
              GmpWrapper.__gmpz_get_str(null, 10, &c));

   //вычисляем дешифрованное сообщение x = c^d mod pq
   mpz_struct x;
   GmpWrapper.__gmpz_init(&x);
   GmpWrapper.__gmpz_powm(&x, &c, &d, &pq);

   //отображаем дешифрованное сообщение x
   Console.WriteLine(
              "x: " +
              GmpWrapper.__gmpz_get_str(null, 10, &x));
}
```

Пример программы CrackRSAWorkFactorDemo

Пример **CrackRSAWorkFactorDemo** продемонстрирует экспоненциальный рост времени, требуемого для раскрытия шифра RSA. Вывод программы изображен на рисунке B.1. Использовался компьютер с тактовой частотой процессора 900 МГц.

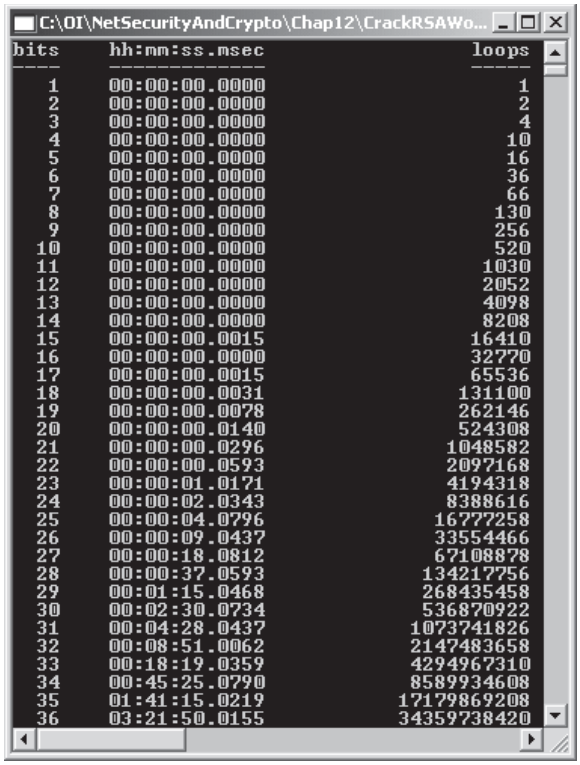
Если вы хотите сопоставить эти результаты с результатами на собственном компьютере, то запустите редактор реестра **Regedt32.exe** и посмотрите на значение тактовой частоты в ключе реестра:

```
HKEY_LOCAL_MACHINE/HARDWARE/DESCRIPTION/System/CentralProcessor/0
```

Для чисел размером до 13 бит требуется менее одной миллисекунды. Для 22-битового числа потребуется почти целая секунда; 28-битовое число займет час, и если у вас хватит терпения, то вы убедитесь, что для 38-битового числа потребуется более одного дня. Сколько же бит должно быть в числе, чтобы раскрытие шифра заняло целую человеческую

жизнь? Предположим, что вы проживаете 2^15 = 32768 дней, что составляет 89 лет и около 9 месяцев, это чуть больше средней продолжительности жизни в развитых странах.

Если 38 бит занимает более одного дня, 53 = 38 + 15 (умножение осуществляется сложением показателей степени) займет время, существенно превышающее человеческую жизнь.



bits	hh:mm:ss.msec	loops
1	00:00:00.0000	1
2	00:00:00.0000	2
3	00:00:00.0000	4
4	00:00:00.0000	10
5	00:00:00.0000	16
6	00:00:00.0000	36
7	00:00:00.0000	66
8	00:00:00.0000	130
9	00:00:00.0000	256
10	00:00:00.0000	520
11	00:00:00.0000	1030
12	00:00:00.0000	2052
13	00:00:00.0000	4098
14	00:00:00.0000	8208
15	00:00:00.0015	16410
16	00:00:00.0000	32770
17	00:00:00.0015	65536
18	00:00:00.0031	131100
19	00:00:00.0078	262146
20	00:00:00.0140	524308
21	00:00:00.0296	1048582
22	00:00:00.0593	2097168
23	00:00:01.0171	4194318
24	00:00:02.0343	8388616
25	00:00:04.0796	16777258
26	00:00:09.0437	33554466
27	00:00:18.0812	67108878
28	00:00:37.0593	134217756
29	00:01:15.0468	268435458
30	00:02:30.0734	536870922
31	00:04:28.0437	1073741826
32	00:08:51.0062	2147483658
33	00:18:19.0359	4294967310
34	00:45:25.0790	8589934608
35	01:41:15.0219	17179869208
36	03:21:50.0155	34359738420

Рис. В.1. Демонстрация роста рабочего фактора раскрытия RSA

Ниже приведен исходный код примера программы.

```
unsafe static void Main(string[] args)
{
   //инициализируем простое число p
   mpz_struct p;
   GmpWrapper.__gmpz_init(&p);

   //инициализируем простое число  q
   mpz_struct q;
   GmpWrapper.__gmpz_init(&q);

   //инициализируем pq
```

```
   mpz_struct pq;
   GmpWrapper.__gmpz_init(&pq);

   //строковое представление n-битного двоичного числа
   String strBits = "1";

   Console.WriteLine(
             "bits    hh:mm:ss.msec            loops");
   Console.WriteLine(
             "----    -------------            -----");

   //циклически проходим n-битные двоичные числа от 2^1 до 2^256
   for (ulong n= 1; n<=256; n++)
   {
             //получить p, следующее простое число после 2^n
             GmpWrapper.__gmpz_set_str(&p, strBits, 2);
             GmpWrapper.__gmpz_nextprime(&p, &p);

             //получить q, следующее простое число после  p
             GmpWrapper.__gmpz_nextprime(&q, &p);

             //получить произведение чисел pq = p*q
             GmpWrapper.__gmpz_mul(&pq, &p, &q);

             //отобразить число битов n
             Console.Write("{0,4}    ", n);

             //разложить на множители и отобразить счетчик цикла
             WorkFactor(&pq);

             //при следующем проходе добавляем бит
             strBits = strBits + "0";
   }
}
unsafe static void WorkFactor(mpz_struct* pq)
{
   //инициализировать число-кандидат
   mpz_struct candidate;
   GmpWrapper.__gmpz_init(&candidate);
   GmpWrapper.__gmpz_set_ui(&candidate, 2);

   DateTime dtStart = DateTime.Now;

   //поиск методом грубой силы
   while (true)
   {
             if (GmpWrapper.__gmpz_divisible_p(
                    pq, &candidate) != 0)
                    break; //множитель найден
```

```
            GmpWrapper.__gmpz_add_ui (
                  &candidate, &candidate, 1L);
   }

   DateTime dtEnd = DateTime.Now;
   TimeSpan ts = dtEnd-dtStart;
   Console.Write(
            "{0,2:d2}:{1,2:d2}:{2,2:d2}.{3,2:d4}    ",
            ts.Hours,
            ts.Minutes,
            ts.Seconds,
            ts.Milliseconds);

   GmpWrapper.__gmpz_sub_ui(
            &candidate, &candidate, 1);
   String str = GmpWrapper.__gmpz_get_str(
            null, 10, &candidate);

   Console.WriteLine("{0,18}", str);
}
```

Приложение C

Использование библиотеки GNU GMP

Как объяснялось в главе 4, криптография с открытым ключом использует некую математическую функцию, которая относительно легко вычисляется в прямом направлении, но требует огромных вычислительных затрат при вычислении в направлении обратном, если только в распоряжении вычислителя нет некоторой добавочной информации, то есть ключа. Все такие математические функции, пригодные для реализации криптографии с открытым ключом, требуют использования арифметики с произвольной точностью.

Поскольку .NET Framework в явном виде не поддерживает в настоящее время такую арифметику, для экспериментов с реализацией асимметричных алгоритмов в нем придется использовать стороннюю библиотеку. Доступно множество библиотек арифметики с произвольной точностью, однако, самым очевидным выбором будет библиотека GNU MP, также известная под названием GMP, – это библиотека из мира программного обеспечения с открытым кодом. Подробней об этой библиотеке см. *http://gmplib.org/*.

Библиотека GMP выпущена под лицензией GNU (General Public License). Это означает, что вы можете свободно копировать исходный код библиотеки и использовать его в своих программах, но при этом вы должны делать код своих программ также открытым, на условиях лицензии GPL. Подробней о лицензии GPL см. *http://www.gnu.org*.

Установка Cygwin

Прежде чем вы сможете использовать библиотеку GMP, вам потребуется платформа, на которой она может работать. Как обычно в случае программного обеспечения с открытым кодом, предполагаемой платформой является Linux или одна из версий UNIX. Данная книга адресована программистам на платформе .NET, что неявно подразумевает использование одной из версий операционной системы Windows. Получается, что без принятия дополнительных мер, библиотека GMP не будет работать в среде Windows.

Для решения этой проблемы мы используем программный продукт с открытым кодом от Red Hat, именуемый Cygwin. Cygwin обеспечивает для Microsoft Windows среду выполнения, аналогичную среде UNIX. Этот эффект достигается эмуляцией поддержки POSIX API и набором стандартных утилит UNIX, включая gcc, make и так далее. Библиотека Cygwin.dll обеспечивает очень полезное подмножество стандартной среды UNIX. Подробней об этой библиотеке см. *http://cygwin.com*.

Библиотека Cygwin также находится под действием лицензии GNU GPL. Это означает, что исходный код любого приложения, написанного при помощи Cygwin, также должен распространяться свободно, если только вы не купили специальную лицензию (buy out license) у Red Hat.

Программу установки Cygwin (файл **setup.exe**) можно загрузить по ссылке *http://www.cygwin.com/setup.exe*. Затем вы можете установить Cygwin, запустив файл **setup.exe**. Однако стандартный вариант установки предусматривает только базовый набор пакетов. В частности, в стандартном варианте не будут установлены утилиты, которые вам потребуются для разработки программ такие, как gcc, binutils и make. Эти утилиты устанавливаются при выборе категории установки Devel. Следовательно, после запуска setup.exe, выберите категорию установки Devel, как описано в следующем абзаце.

Когда вы запустите **setup.exe**, на экране появится серия диалоговых окон мастера установки, как изображено на рисунках С.1...С.9. Обратите внимание, в диалоге на рисунке С.7 необходимо щелкнуть на узле Devel, для того чтобы выбрать эту категорию установки. В результате будут установлены также и утилиты, необходимые для разработки программ.

Рис. С.1. Установка Cygwin, шаг 1

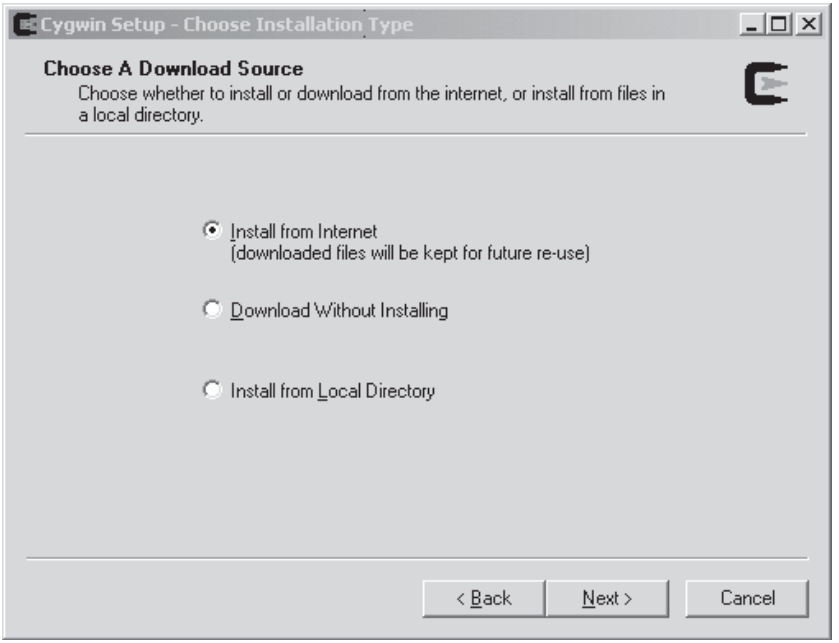


Рис. C.2. Установка Cygwin, шаг 2

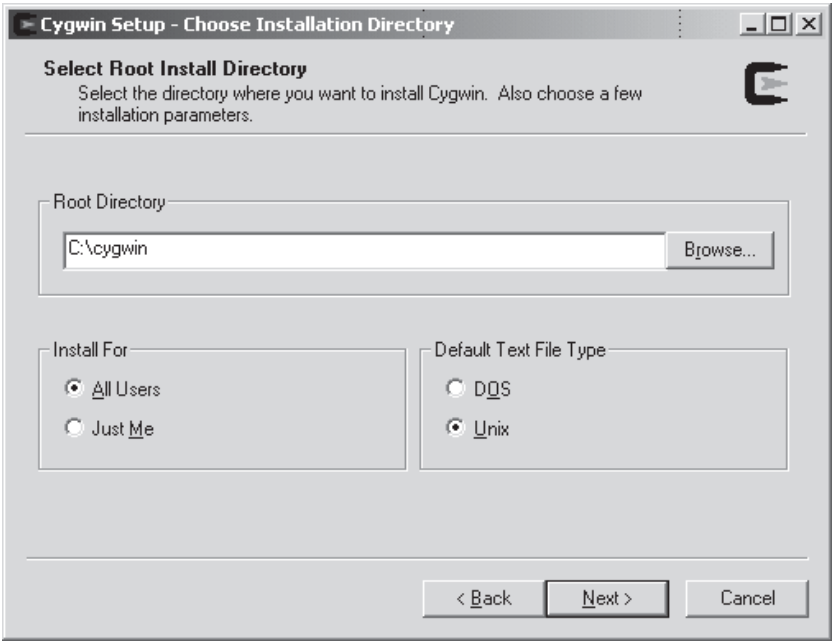


Рис. C.3. Установка Cygwin, шаг 3

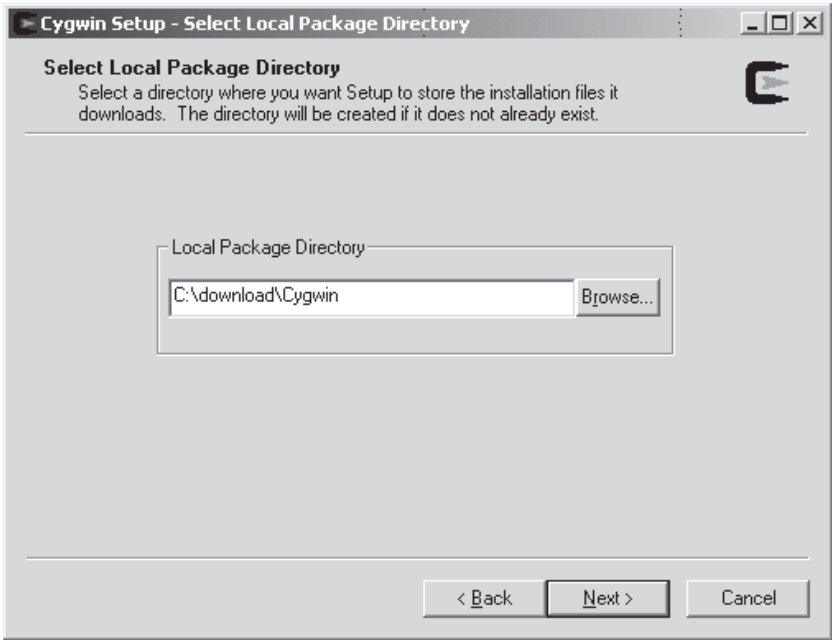


Рис. С.4. Установка Cygwin, шаг 4

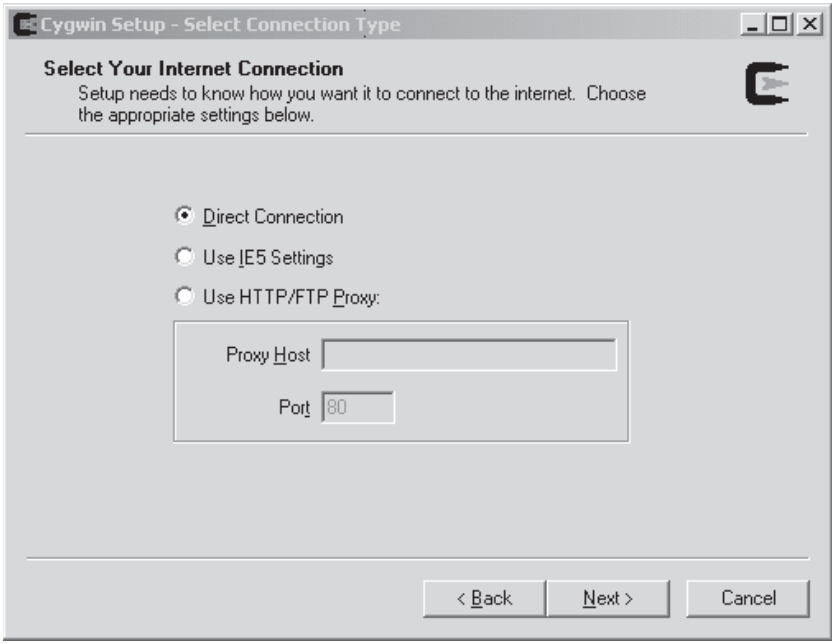


Рис. С.5. Установка Cygwin, шаг 5

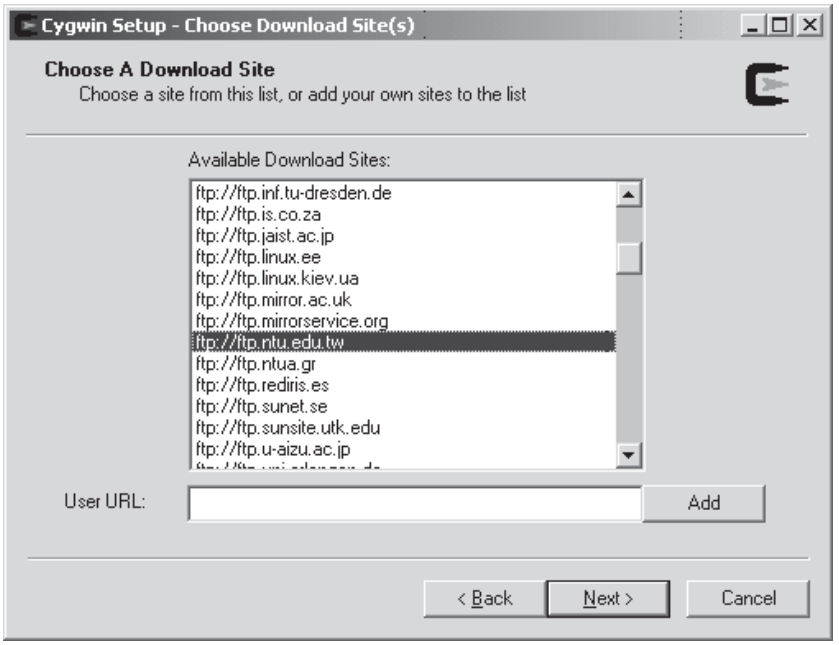


Рис. C.6. Установка Cygwin, шаг 6

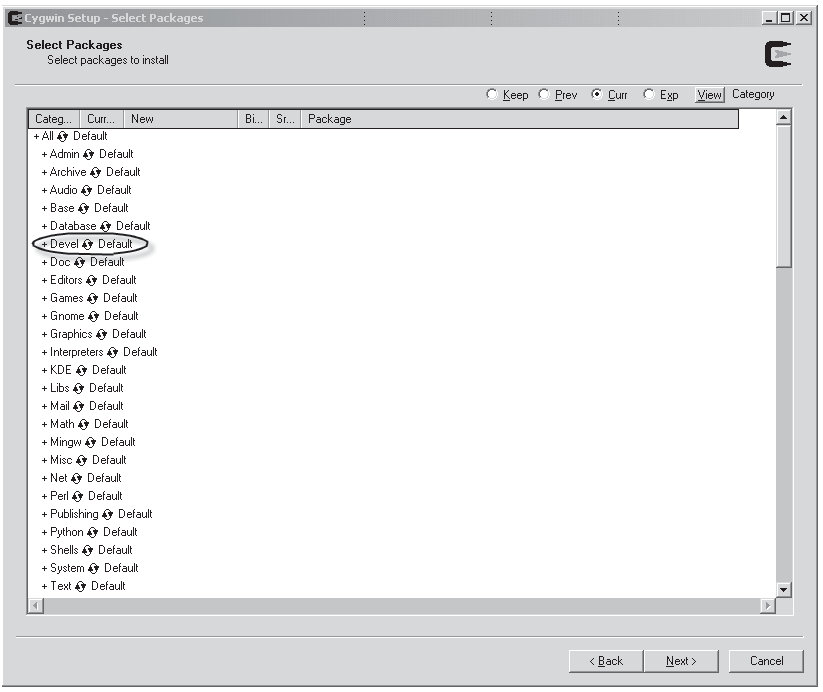


Рис. C.7. Установка Cygwin, шаг 7

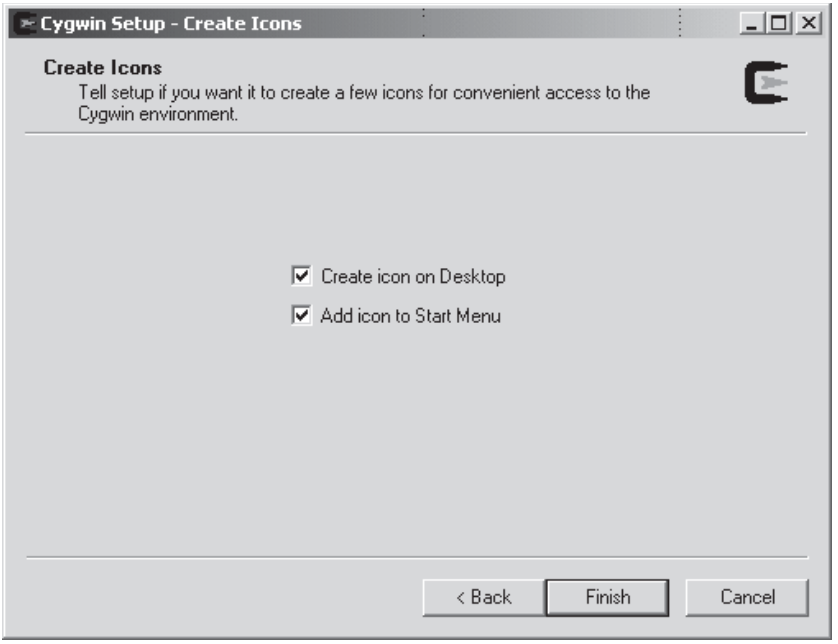


Рис. С.8. Установка Cygwin, шаг 8

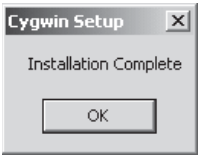


Рис. С.9. Установка Cygwin, шаг 9

Библиотека Cygwin предоставляет окно командной строки в стиле UNIX. Все инструментальные программы GNU можно запускать из этой командной строки, используя обычный синтаксис путей Windows. Для того чтобы эта возможность была доступна из любого каталога, вам необходим соответствующий путь (по умолчанию, C:\cygwin\bin) в переменную среды PATH, как показано на рисунке С.10, тогда вы сможете обращаться к утилитам bin из любого каталога.

Тестирование библиотеки Cygwin

Теперь вы можете проверить установку Cygwin, скомпилировав простую DLL при помощи компилятора gcc и вызвав ее из программы C# в среде .NET.

Создайте следующий файл исходного кода на языке C с именем **MyDll.c**. Он будет скомпилирован утилитой gcc для создания DLL-библиотеки для среды Windows.

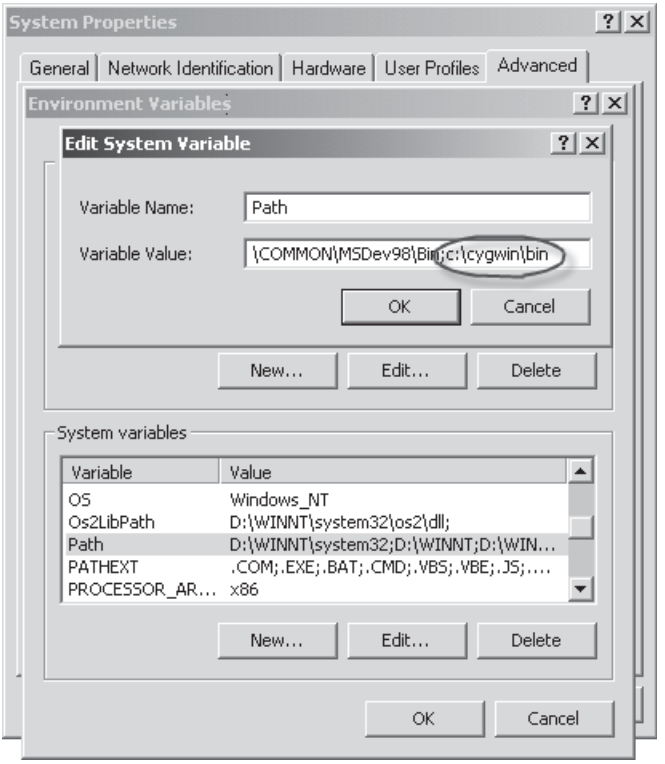


Рис. C.10. Формат SOAP-сообщения в WS-Security

```
//MyDll.c

#include <windows.h>

int WINAPI DllMain(
   HANDLE hInst,
   ULONG reason,
   LPVOID lpReserved)
{
   return 1;
}

__declspec(dllexport) char* SomeDllFunction()
{
   return "SomeDllFunction called!";
}
```

Ниже приведена командная строка для компиляции и сборки **MyDll.dll**. Ее можно выполнить как в обычной командной строке Windows, так и в командной строке оболочки Shell, предоставляемой Cygwin.

```
gcc -shared MyDll.c -o MyDll.dll -e DllMain@12
```

Вот программа на языке C#, при помощи которой можно протестировать работу библиотеки **MyDll.dll**.

```
//MyDllClient.cs
using System;
using System.Runtime.InteropServices;

class MyDllClient
{
   //NOTE: MyDll.dll must be in dll search path

   [DllImport("MyDll.dll")]
   public static extern String SomeDllFunction();

   static void Main(string[] args)
   {
            String str = SomeDllFunction();
            Console.WriteLine(
                  "SomeDllFunction in MyDll.dll returned: " +
                  str);
   }
}
```

Результат выполнения этой программы изображен на рисунке С.11. Если вы достигли этого результата, значит пакет Cygwin установлен у вас надлежащим образом.

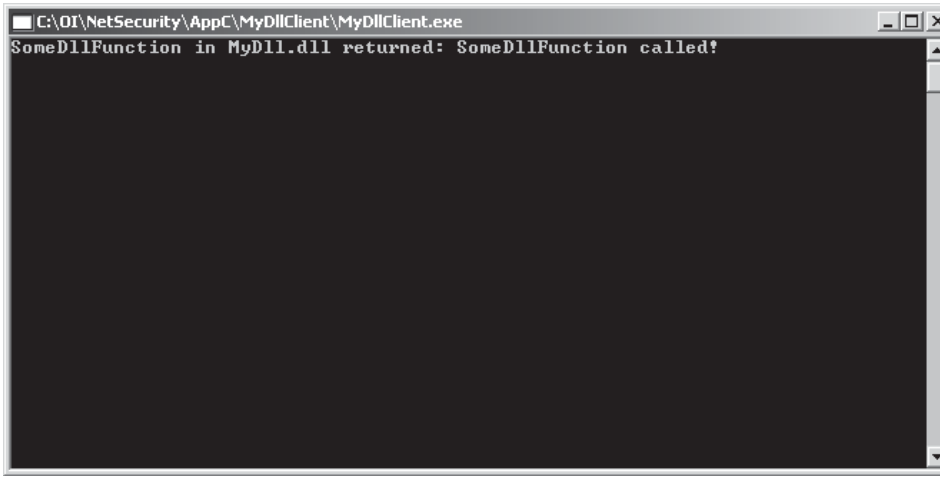


Рис. С.11. Результат выполнения клиентской программы на языке C#

Добившись работоспособности от этого простого примера, вы можете приступить к установке и тестированию библиотеки GMP.

Установка GMP

Теперь вам необходимо загрузить и установить библиотеку GMP. Последнюю версию этой библиотеки в форме исходного кода вы можете загрузить со страницы *http://gmplib.org/*. На момент написания этой книги последней версией библиотеки является **gmp-4.0.1.tar.gz**, как видно из рисунка C.12.

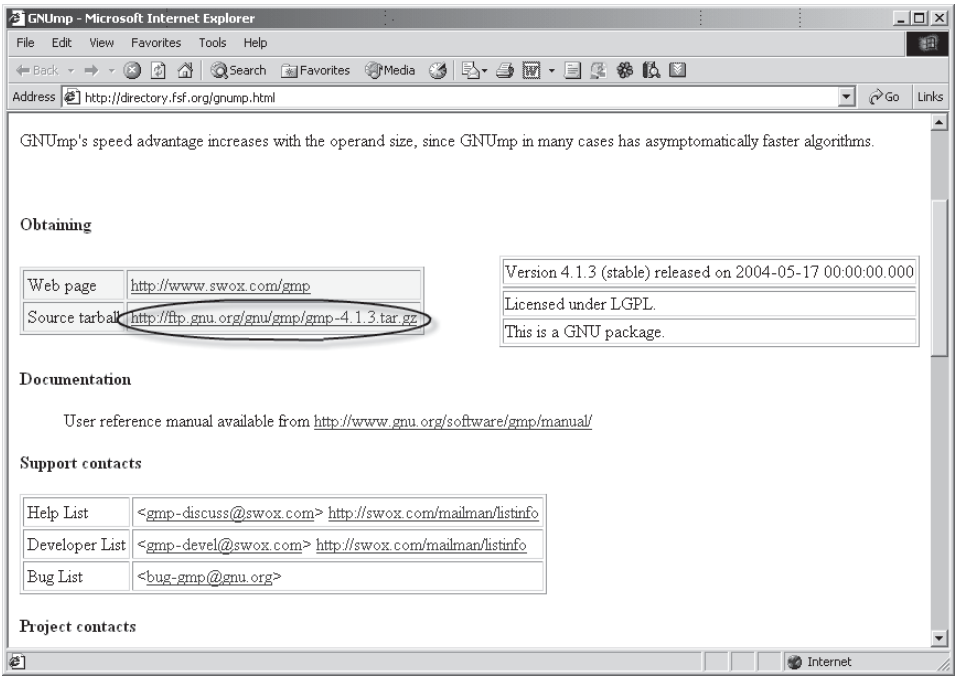


Рис. C.12. Загрузка Cygwin

После загрузки архива **gmp-4.0.1.tar.gz**, вам потребуется развернуть его при помощи архиватора WinZip или эквивалентного. В результате будет создан каталог **gmp-4.0.1** (цифры могут отличаться в зависимости от загруженной версии), который необходимо разместить в удобном для вас месте, например, в корневом каталоге диска C. На данном этапе каталог **C:\gmp-4.0.1** содержит древовидную структуру исходных файлов библиотеки, которая еще не скомпилирована и не скомпонована. Чтобы выполнить эти манипуляции, откройте окно командной строки Cygwin (Пуск | Программы | Cygwin | Cygwin Bash Shell).

Для построения библиотеки введите в командной строке окна Cygwin Bash Shell следующие команды:

```
cd c:/gmp-4.0.1
./configure --disable-static --enable-shared
make
make install
```

Имя рабочего каталога (c:/gmp-4.0.1) у вас может отличаться, в зависимости от версии библиотеки и от того, куда вы развернули архив. Нам не нужна статическая библиотека, которую необходимо использовать при компоновке исполняемых модулей программ на языке C/C++. Для программ, создаваемых в среде .NET, нам требуется динамическая библиотека DLL, которая предоставляет доступ к функциональности GMP через службы PInvoke. Это означает, что мы должны создать не статическую библиотеку (файл LIB), а динамическую, то есть файл типа DLL. Вот почему нам требуются в командной строке параметры **--disable-static** и **--enable-shared**. Первая команда **make** компонует проект GMP, а вторая команда make устанавливает его на вашу систему Cygwin. В результате выполнения этих команд файл заголовков **gmp.h** будет помещен в каталог **C:\gmp-4.0.1**, а DLL-библиотека с именем **cyggmp.dll** – в каталог **C:\gmp-4.0.1\.libs**. чтобы она была доступна приложениям, путь **C:\gmp-4.0.1\.libs** необходимо добавить в переменную окружения PATH.

Детали использования функциональности, заключенной в библиотеке GMP, выходят за рамки тем этой книги. Познакомиться с функциям, поддерживаемыми библиотекой GMP, можно при помощи документации, расположенной по ссылке *http://www.swox.com/gmp/manual/*. Тем не менее, для того чтобы положить начало знакомству и проверить вновь созданную библиотеку при помощи .NET-программы на языке C#, попробуйте создать клиентскую программу, описанную ниже. Если вам потребуется обратиться к любой другой функции библиотеки **cyggmp-3.dll**, просто добавьте соответствующие методы с атрибутом **DllImport** в класс **CygGmpWrapper**, согласно сигнатурам, которые вы обнаружите в **gmp.h**.

```
//MyCSharpGMPClient.cs

using System;
using System.Runtime.InteropServices;

//Структура GMP для целых чисел произвольной точности
[StructLayout(LayoutKind.Sequential)]
unsafe public struct mpz_struct
{
   public int _mp_alloc;
   public int _mp_size;
   public uint *_mp_d;
};
```

```
//ПРИМЕЧАНИЕ: библиотека cyggmp-3.dll должна присутствовать
//в пути поиска dll
class CygGmpWrapper //оболочка для обращений к библиотеке GMP
{
   [DllImport("cyggmp-3.dll")]
   unsafe public static extern
              void __gmpz_init(
              mpz_struct* val);
   [DllImport("cyggmp-3.dll")]
   unsafe public static extern
              int __gmpz_set_str(
              mpz_struct* val,
              String strval,
              int baseval);
   [DllImport("cyggmp-3.dll")]
   unsafe public static extern
              int __gmpz_mul(
              mpz_struct* valprod,
              mpz_struct* val1,
              mpz_struct* val2);
   [DllImport("cyggmp-3.dll")]
   unsafe public static extern
              String __gmpz_get_str(
              String str,
              int baseval,
              mpz_struct* val);
};

class MyCSharpGMPClient
{
   unsafe static void Main(string[] args)
   {
              //инициализация целых переменных gmp: a, b, и p
              mpz_struct mpzs_a;
              mpz_struct* a = &mpzs_a;
              CygGmpWrapper.__gmpz_init(a);
              mpz_struct mpzs_b;
              mpz_struct* b = &mpzs_b;
              CygGmpWrapper.__gmpz_init(b);
              mpz_struct mpzs_prod;
              mpz_struct* prod = &mpzs_prod;
              CygGmpWrapper.__gmpz_init(prod);

              //присвоить значения a и b, используя
              //строковые литералы
              CygGmpWrapper.__gmpz_set_str (
                     a, "123456789123456789", 10);
              CygGmpWrapper.__gmpz_set_str (
                     b, "123456789123456789", 10);
```

```
            //умножить a на b, и поместить результат в prod
            //result is 152415787806736785156226207501905 21
            CygGmpWrapper.__gmpz_mul(prod, a, b);

            //вывести prod, как строку
            String strProd =
                  CygGmpWrapper.__gmpz_get_str(null, 10, prod);
            Console.WriteLine(
                  "123456789123456789 * 123456789123456789 = \n"
+
                  strProd);
   }
}
```

После запуска этой программы вы должны получить следующий результат:

```
123456789123456789 * 123456789123456789 =
152415787806736785156226207501905 21
```

Удаление Cygwin из системы

Пакет Cygwin не снабжается автоматической утилитой удаления из системы. Для того чтобы удалить Cygwin, выполните следующие шаги вручную:

1. Удалите ярлыки Cygwin с рабочего стола и из меню Пуск.
2. Удалите ветку реестра **Software\Cygnus_Solutions** в разделах **HKEY_LOCAL_MACHINE** и **HKEY_CURRENT_USER**.
3. Удалите каталог Cygwin (обычно это **C:\Cygwin**).
4. Удалите файлы, созданные программой в каталоге для временных файлов.
5. Удалите путь **C:\Cygwin\bin** из переменной окружения **PATH**, если вы его туда записывали.

Приложение D

Ресурсы по криптографии и безопасности

В этом приложении вы найдете удобный для использования список ресурсов, которые могут оказаться полезными для более глубокого и широкого понимания тем, связанных с криптографией и вообще средствами обеспечения безопасности. Чуть ниже описываются наиболее примечательные книги, которые могут быть вам полезны (описания разбиты на несколько категорий). Далее вы обнаружите список полезных Web-сайтов, где содержатся ссылки и полезная информация о некоторых аспектах криптографии и систем безопасности.

Общетеоретические и концептуальные книги

Описанные ниже книги посвящены общим теоретическим основам и концепциям, относящимся к современной криптографии.

- *The Handbook of Applied Cryptography*, Alfred J. Menezes, Paul C. van Oorschot и Scott A. Vanstone. Также эта книга доступна по ссылке *http://www.cacr.math.uwaterloo.ca/hac/*. Это один из лучших и наиболее всесторонних технических справочников. Здесь описаны основы симметричных и асимметричных алгоритмов, и приведены великолепные математические определения. Также книга освещает такие темы, как генераторы псевдослучайных чисел, симметричные и асимметричные алгоритмы, потоковые и блочные шифры, хеш-функции и цифровая подпись. Также здесь описано несколько важных протоколов и стандартов.
- *Introduction to Cryptography*, Johannes A. Buchman. Это также отличная книга, показывающая, как работают некоторые важные криптографические технологии, как оценивается эффективность алгоритмов и стойкость шифров. В этой книге объяснены важные математические методы современной криптографии, при этом со стороны читателя требуются лишь базовые математические познания.

- *Applied Cryptography*, Bruce Scheier. Эта книга стала чрезвычайно популярной, особенно, в качестве первой книги для знакомства с темой. В ней содержится элегантный и легкий для восприятия обзор наиболее важных аспектов современной криптографии, и к ней прилагается набор файлов на языке C, иллюстрирующий все наиболее известные алгоритмы, что чрезвычайно удобно для начала изучения криптографии с точки зрения программиста.
- *Cryptography: Theory and Practice*. Second Edition. Douglas Stinson. Эта книга в весьма легкой для понимания форме описывает математические основы криптографии, включая алгоритмы DES, RSA, односторонние хеш-функции, генераторы псевдослучайных чисел и цифровые подписи.

Книги по криптографической математике

Если вы уже освоили общие теоретические основы криптографии, то вам, возможно, будет интересно более глубоко понять ее математические аспекты. Следующие книги фокусируются на математической основе криптографических алгоритмов.

- *A Course in Number Theory and Cryptography*, Neil I. Koblitz. В этой книге излагаются концепции теории чисел, имеющие отношение к криптографии. От читателя здесь не требуется предварительных познаний в алгебре и теории чисел, и книга методично и весьма подробно знакомит читателя с каждой из концепций. Наиболее важные алгоритмы здесь описываются параллельно с анализом их сложности. Также в этой книге описывается криптографическое использование эллиптических кривых, что является довольно новой и интересной темой. Каждая глава снабжена упражнениями и их решениями.
- *The Mathematics of Ciphers: Number Theory and RSA Cryptography*, S. C. Countinho. В этой небольшой по объему книге (менее 200 страниц) содержится легкое для понимания введение в те аспекты теории чисел, которые необходимы для понимания алгоритма RSA. «Мосты» от теории чисел здесь прокладываются к асимметричным алгоритмам вообще, а затем подробно разбирается собственно алгоритм RSA. Книга проведет вас через многие фундаментальные концепции для того, чтобы в последней главе полностью объяснить работу алгоритма RSA.

Освоение «криптографической» математики – лишь первый шаг на пути к полному пониманию теории криптографии. Следующий шаг состоит в изучении криптоанализа, то есть науки по раскрытию шифров.

Говоря вообще, обычному программисту, использующему в своих приложениях криптоалгоритмы, разбираться в криптоанализе необязательно. Однако для разработки новых алгоритмов необходимо хорошо понимать современную теорию криптоанализа. Криптоанализ – довольно сложная наука, и почти всегда проще использовать готовые, хорошо испытанные алгоритмы, чем изобретать что-то свое. Таким образом, разработку новых алгоритмов лучше предоставить профессиональным криптографам. Однако если вы заинтересованы в собственном развитии в этом направлении или планируете заняться криптоанализом профессионально, то вам будет полезно познакомиться со следующими книгами.

- *Differential Cryptanalysis of Data Encryption Standard*, Eli Biham, Asi Shamir. Это математический труд (требующий от читателя определенной математической подготовки), описывающий дифференциальный криптоанализ в применении к алгоритму DES. Здесь показывается, как можно проанализировать различия между отличающимися, но отчасти сходными сообщениями, зашифрованными одним и тем же ключом. Техника этого анализа позволяет раскрыть текст, зашифрованный 16-цикловым DES, гораздо быстрее, чем любым другим ранее опубликованным способом. Аналогичную технику можно применить и к любому другому симметричному шифру.
- *Elementary Cryptanalysis: A Mathematical Approach*, Abraham Sinkov. Эта книга объясняет, в легко доступной для понимания форме, некоторые фундаментальные подходы, используемые в криптоанализе. Здесь описываются несложные темы из статистики, модульной арифметики, теории чисел и элементарной линейной алгебры.
- *Cryptanalysis: A Study of Ciphers and Their Solutions*, Helen F. Gaines. Эта книга, опубликованная в 1939 году, является «библией» классического криптоанализа. Хотя классические шифры в наше время всерьез не воспринимаются, и в этом смысле книга устарела, все же прочесть ее весьма интересно. Она вводит вас во «внутренний мир» криптоанализа и учит криптоаналитическому взгляду на мир. В ней нет той устрашающей математики, которая характерна для современного криптоанализа, и прочесть ее интересно также и с исторической точки зрения.

Книги – руководства по безопасности

В нашей книге мы рассматривали криптографию и безопасность с точки зрения программиста. В конечном счете, однако, все эти технологии при-

меняются на практике администраторами и конечными пользователями. По этой причине программист должен был знаком с протоколами и административными аспектами обеспечения безопасности компьютерных систем. Ниже описаны полезные практические руководства, описывающие меры по обеспечению безопасности в реальных системах.

- *Cryptography and Network Security: Principles and Practice*, William Stallings. Это элементарный справочник по реально используемым протоколам и системам безопасности. Здесь описывается симметричная и асимметричная криптография и методы аутентификации. Также здесь описаны такие важные протоколы, как PGP, S/MIME, SSL и Kerberos.
- *Windows 2000 Security*, Roberta Bragg. Эта книга снабдит сетевых администраторов практическими сведениями об обеспечении безопасности систем Windows 2000, включая аспекты конфигурирования Active Directory, связанные с безопасностью.

Популярные книги по криптографии

Ниже описаны книги, отличающиеся легким в техническом смысле изложением и более историческим подходом к темам криптографии. Эти книги пригодятся вам в тот момент, когда вам захочется позабыть о программировании и математике, и просто посидеть с интересной книжкой у камина.

- *The Codebreakers*, David Kahn. Эта увлекательная книга об истории криптографии, хотя к современной криптографии она имеет мало отношения (в конце концов, написана она была в 1967 году). Тем не менее, она легко читается и написана увлекательным, совсем не техническим языком.
- *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*, Simon Singh. Эта книга также посвящена популярному описанию истории криптографии, в ней описаны также некоторые эпизоды, не упомянутые в предыдущей книге. Читается также легко и увлекательно.
- *Decrypted Secrets*, Friedrich Bauer. Эта книга напоминает *The Codebreakers*, однако написана она чуть более техническим языком, особенно, когда речь идет о раскрытии важных шифров Второй Мировой. В целом эта книга также достаточно интересна для обычного читателя.

Группы новостей по криптографии

- Группа news:sci.crypt – отличное место для того чтобы задавать вопросы по криптографии. Однако чтобы избежать «флейма», прочтите вначале FAQ этой группы по адресу *www.faqs.org/faqs/cryptography-faq/*. Несколько весьма квалифицированных людей являются частыми участниками этой группы и вы сможете получить ответы даже на самые сложные вопросы. Однако эти люди нетерпимы к нарушителям правил, описанных в FAQ. В частности, если вы не обладаете достаточным пониманием теории современной криптографии и криптоанализа, лучше не пытайтесь предложить для комментирования группой изобретенные вами алгоритмы, – вы станете объектом критического «флейма» и не получите ответов на свои вопросы. И в то же время даже самые примитивные вопросы полных новичков будут встречены с пониманием, если они соответствуют требованиям, изложенным в FAQ.
- Группа news:sci.crypt.research является модерируемой группой (и, как следствие, более цивилизованной и целенаправленной, чем news:sci.crypt) и предназначена для проведения серьезных дискуссий в среде экспертов. Новичкам здесь рекомендуется побольше слушать и пореже вмешиваться в разговор. Несмотря на высокий уровень «устрашения новичков» в этой группе, чтение ее может быть очень полезным ресурсом для изучения криптографии.

Полезные Web-сайты на темы криптографии и безопасности

- Сайт Counterpane Internet Security расположен по адресу *www.counterpane.com*. Основателем и руководителем сайта является Брюс Шайнер (Bruce Scheiner), автор популярной книги *Applied Cryptography* и изобретатель алгоритмов Blowfish и Twofish. На этом сайте размещена интересная литература, также он предлагает доступ к службе *Crypto-Gram*, ежемесячной бесплатной рассылке новостей на темы криптографии и компьютерной безопасности.
- Сайт RSA Security находится по адресу *www.rsasecurity.com*. На этом Web-сайте имеется много полезной информации, в частности, здесь есть рассылка *CryptoBytes*, документация, технические статьи и бюллетени на темы, связанные с криптографией RSA.
- Сайт координационного центра CERT расположен по адресу *www.cert.org*. Центр CERT является организацией, финансируемой федеральным бюджетом и управляемой университетом Карнеги Меллоун. Здесь находится много важной и регулярно обновляемой информации на известные темы, связанные с безопасностью. Этот

сайт публикует все последние новости о значительных инцидентах, связанных с безопасностью, об обнаруженных уязвимостях и т. п.

- Книга *The Handbook of Applied Cryptography* (также часто именуемая HAC) доступна бесплатно в электронном виде по адресу www.cacr.math.uwaterloo.ca/hac/. Мы уже упоминали эту книгу ранее в данном приложении, но упоминаем ее снова, просто чтобы подчеркнуть огромную полезность этого ресурса для изучения теории криптографии.
- Сайт Бюро Промышленности и Безопасности расположен по адресу *http://www.bis.doc.gov/*. Он предоставляет информацию о регулировании экспорта криптографических технологий правительством США. Заметьте, в каждой стране имеются свои ограничения, регулирующие распространение технологий шифрования и защиты информации, поэтому вам следует ориентироваться на правила, принятые в вашей стране.

Приложение E

Исследование Web-служб

Думаю, во всем мире можно будет продать от силы пять компьютеров.
Томас Уотсон, глава компании IBM, 1943

Выживают вовсе не самые сильные экземпляры, и даже не самые умные, а те, кто способен меняться.
Чарльз Дарвин

Сеть Internet и «всемирная паутина» WWW являются наиболее важными и значительными созданиями 20-го века. Эти технологии настолько сильно изменили мир программного обеспечения, что приходится гадать, имел ли Дарвин в виду человеческую расу или же расу Internet. Влияние, оказываемое сетью Internet на современные технологии, носит глобальный характер. В частности, огромные сдвиги происходят в области прикладных программ, использующих различные Web-службы. Являясь новым, революционным направлением расширения Сети, Web-службы становятся той элементарной структурой, которая призвана связать воедино все вычислительные устройства.

Вот, что говорится в документации Microsoft по системе ASP.NET о Web-службах: «Сеть Internet быстро развивается от сегодняшних Web-сайтов, которые лишь предоставляют пользовательским браузерам информацию в той или иной форме, к программируемым сайтам нового поколения, которые напрямую связывают между собой организации, приложения, службы и устройства».

Зачем нужны Web-службы

Вы можете спросить, что такого есть особенного в Web-службах в сравнении с другими, такими уже традиционными моделями построения распределенных приложений, как технологии DCOM (Distributed Component Object Model – распределенная модель компонентных объектов) компании Microsoft, COBRA (Common Object Request Broker Architecture – архитектура запрашиваемых распределенных объектов) или RMI (Remote Method Invocation – вызовы удаленных методов) от компании Sun.

Главные проблемы существующих программных моделей распределенных приложений заключаются в следующем:

- Межсистемное взаимодействие, то есть способность передавать и разделять данные между приложениями разных производителей, функционирующих на разных платформах.
- Во многих случаях организация не может позволить себе траты, связанные с разработкой и администрированием приложений на CORBA или даже Enterprise Java Beans (EJB).
- Все эти технологии вынуждены иметь дело с брандмауэрами. Поскольку большинство брандмауэров блокируют все порты кроме нескольких, включая стандартный порт 80 (HTTP), все существующие протоколы взаимодействия распределенных объектов сталкиваются с проблемами, поскольку зависят от динамически назначаемых портов для удаленного вызова процедур.

Технология DCOM является частной (закрытой), что в принципе делает невозможным межсистемное взаимодействие, основанное на открытых стандартах. Системы DCOM не могут взаимодействовать с системами CORBA или EJB без дополнительных (и притом весьма значительных) усилий. У CORBA в этом отношении дела обстоят чуть лучше. Эта система, основанная на стандартах, нейтральная к производителю и независимая от языка. Однако ее ограничение состоит в неспособности использовать возможности Internet. Компоненты CORBA и DCOM часто сообщаются между собой через специальный шлюз COM/CORBA. Но достаточно внести даже небольшое изменение в один из компонентов модели, и придется модифицировать всю модель, включая шлюз. Технологии DCOM, IIOP и Java/RMI требуют плотной интеграции между клиентом и сервером, а также использования конкретной платформы. Кроме того, эти технологии используют специфические двоичные форматы данных и нуждаются в специфических технологиях обращения к объектам. Таким образом, главным доводом в пользу развития по пути Web-служб является неспособность промышленности перейти к единым стандартам.

Web-службы сталкиваются с проблемами распределенных вычислений и решают их. В модели распределенных вычислений, основанной на Web-службах, клиенту нет нужды беспокоиться о языке или об операционной системе, как и вообще ни о каких деталях реализации сервера, поскольку Web-службы свободно сочетаются с моделями Web-программирования, основанными на стандартных форматах данных и стандартных протоколах таких, как XML, SOAP и HTTP. Для клиента Web-службы достаточно иметь адрес службы и знать методы службы, которыми он может воспользоваться. Единственное предварительное условие, существующее между службой и клиентом, заключается в том, получатель SOAP-сообщений должен быть способен их понимать. В результате приложения, написанные на различных языках программирования, основанные на разных компонентных моделях, работающие под управлением разных операционных систем и на разной аппаратуре, могут беспрепятственно работать с Web-службами.

Определение Web-служб

Существует множество определений понятия «Web-служба». Ниже приведены определения, данные ведущими компаниями, имеющими отношение к Web-службам.

1. **W3C:** Web-служба – это программное приложение, идентифицируемое его URL, чьи интерфейсы и общая организация могут быть определены и описаны при помощи артефактов XML, и которые поддерживают прямое взаимодействие с приложениями, использующими XML-сообщения, передаваемые при помощи протоколов Internet.
2. **Microsoft:** Web-службы обеспечивают функцию обмена сообщениями со свободно выбираемыми корреспондентами при помощи таких стандартных протоколов, как HTTP, XML, XSD, SOAP и WSDL.
3. **IBM:** Web-службы – это новое поколение Web-приложений. Это автономные, самоописывающие, модульные приложения, которые можно опубликовать, разместить и вызывать через Web.
4. **Rogue Wave:** Приложения, взаимодействующие со свободно выбираемыми системами через посредство Internet-протоколов.
5. **The Stencil Group:** Свободно взаимодействующие программные компоненты, которые заключают в себе определенную функциональность и доступны через посредство Internet-протоколов.
6. **Sun Microsystems:** Web-служба – это приложение, которое существует в такой распределенной среде, как Internet. Web-служба принимает запросы, выполняет некоторые действия, вытекающие из запросов, и возвращает ответы.

Фундамент Web-служб

Давайте посмотрим на те основные спецификации, на которых основываются Web-службы.

- **XML (Extensible Markup Language – расширяемый язык разметки):** XML является тем основанием, на котором строятся Web-службы. XML обеспечивает форматы описания, хранения и передачи данных, которыми Web-служба обменивается с клиентом. XML – это простой, не зависящий от платформы и повсеместно принятый стандарт.
- **SOAP (Simple Object Access Protocol – простой протокол доступа к объектам):** SOAP – это легкий и простой протокол, основанный на XML протокол, используемый для обмена структурированной и типизированной информацией в Web. Протокол SOAP определяет

способы доступа к службе, объекты и серверы при помощи HTTP и XML, не зависящим от платформы образом. SOAP использует XML для описания способа передачи данных и соответствующих им определений типов. Протокол SOAP разрабатывался компаниями Microsoft, IBM и другими, а после завершения разработки он был передан консорциуму W3C для дальнейшего развития. В настоящий момент в стадии рассмотрения в консорциуме W3C находится версия 1.2 этого протокола.

- **WSDL (Web Services Description Language – язык описания Web-служб):** это грамматика, основанная на XML и служащая для описания Web-служб и их функций, параметров и возвращаемых значений. WSDL определяет методы и данные, ассоциированные со службой. Поскольку WSDL основывается на XML, она понятна как для машин, так и для людей.
- **UDDI (Universal Description, Discovery and Integration – «универсальное описание, раскрытие и интеграция»):** это многосторонняя промышленная инициатива, позволяющая хранить информацию о Web-службах в разделяемом предприятиями едином и глобальном реестре. UDDI позволяет предприятиям динамически находить друг друга и взаимодействовать. UDDI обеспечивает нахождение описания протокола заданной Web-службы, основанной на SOAP (стандартный способ публикации и распространения информации о Web-службах). UDDI представляет собой деловой реестр, позволяющий предприятиям и разработчикам программно находить информацию о Web-службах. В настоящее время UDDI является не стандартом, а объединенной инициативой предприятий и производителей программного обеспечения. Последняя версия UDDI 3 обладает такими некоторыми новыми возможностями, как мультирегистровая топология, улучшенная безопасность, расширенная поддержка WSDL, новый API и усовершенствования в основной информационной модели.

Следующее поколение распределенных вычислений: Web-службы

Web-служба на основе XML представляет собой программируемый компонент приложения, который предоставляет некоторую полезную функциональность, как логика приложения, и доступен любому числу потенциально не конгруэнтных систем благодаря использованию стандартных протоколов таких, как XML и HTTP. Web-службы способны обмениваться сообщениями с произвольными клиентами (в то время как DCOM, IIOP и

Java/RMI требуют заданной интеграции между клиентом и сервером), благодаря использованию общих форматов данных и стандартных протоколов таких, как XML, SOAP и HTTP.

Поскольку Web-службы основываются на стандартных форматах данных и протоколах, приложения Web-служб могут сообщаться с широким кругом вариантов реализаций, платформ и устройств. Так как сообщения XML являются основным средством коммуникации в Web-службах, они перекрывают собой все различия между сообщающимися системами, которые могут использовать различные компонентные модели, языки программирования и операционные системы.

Поскольку Web-службы используют уже существующую инфраструктуру, работающую по протоколу HTTP, данные Web-служб свободно проходят через брандмауэры между различными сетями.

Следовательно, мы видим в Web-службах следующее поколение распределенных вычислений. Web-службы открывают новую эру в разработке распределенных приложений.

Преимущества Web-служб

Давайте перечислим весомые преимущества Web-служб.

1. Web-служб открывают новые деловые возможности, делая задачу интеграции между партнерами очень простой.
2. Web-службы сокращают затраты на разработку и поддержку приложений (экономится как время, так и деньги).
3. Вы можете создать новый источник дохода, превратив существующую функциональность в Web-службу.
4. Web-службы позволяют решить проблемы межсистемного взаимодействия.
5. Web-службы позволяют реализовать кросс-платформенную интеграцию приложений (коммуникации «программа-программа»).
6. Программное обеспечение используется повторно, что экономит время и деньги.
7. Web-службы связывают информацию, приложения, людей, системы и устройства.

Web-службы ASP.NET

Поскольку Web-службы ASP.NET строятся на основе ASP.NET, вы можете, при создании службы, использовать все функциональные возможности ASP.NET.В свою очередь, ASP.NET опирается на функциональные возможности и механизмы защиты, заключенные в .Net Framework и CLR. Соответственно, Web-служба ASP.NET может использовать множество функций .Net Framework таких, как аутентификация, кеширование, управление памятью, межсистемное взаимодействие и управление

состояниями. Visual Studio .NET может быть мощным инструментом для построения Web-приложений, Web-служб, а также обычных и мобильных приложений. Интегрированная среда разработки Visual Studio .NET поддерживает методологию RAD (Rapid Application Development – быстрая разработка приложений), благодаря чему разработчики могут создавать и распространять Web-службы очень быстро. Платформа ASP.NET и Visual Studio .NET составляют вместе программную модель очень быстрой и простой разработки Web-служб. Например, разработчикам не требуется генерировать документы WSDL, поскольку ASP.NET сгенерирует все необходимые WSDL-документы автоматически.

Архитектура Web-служб

Инфраструктура Web-службы состоит из следующих четырех разделов.

1. Коммуникационный формат (HTTP и SOAP).
2. Стек протоколов описания и обнаружения (WSDL и Disco).
3. Стек каталога (UDDI).
4. Служба запросов и ответов (сообщения SOAP).

На рисунке E.1 изображена базовая архитектура Web-службы. Рисунок E.2 иллюстрирует четыре раздела инфраструктуры Web-службы и их взаимоотношения.

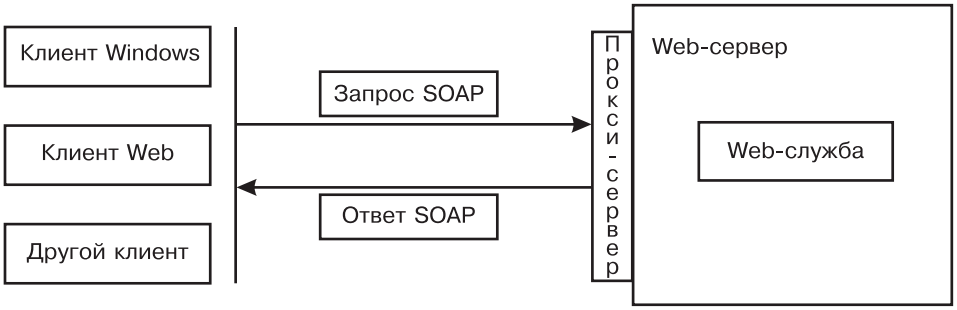


Рис. E.1. Базовая архитектура Web-службы

Рисунки E.3 и E.4 показывают взаимодействие Web-служб. Это взаимодействие включает в себя следующие шаги:

- поставщик службы для публикации информации о службе регистрирует ее определение (WSDL) в реестре UDDI;
- клиентское приложение запрашивает UDDI, используя, если необходимо, описание службы;
- если соответствующая описанию Web-служба нашлась, UDDI возвращает ссылку на WSDL-документ, описывающий службу;
- клиент, используя эту ссылку, запрашивает WSDL-документ у Web-службы;

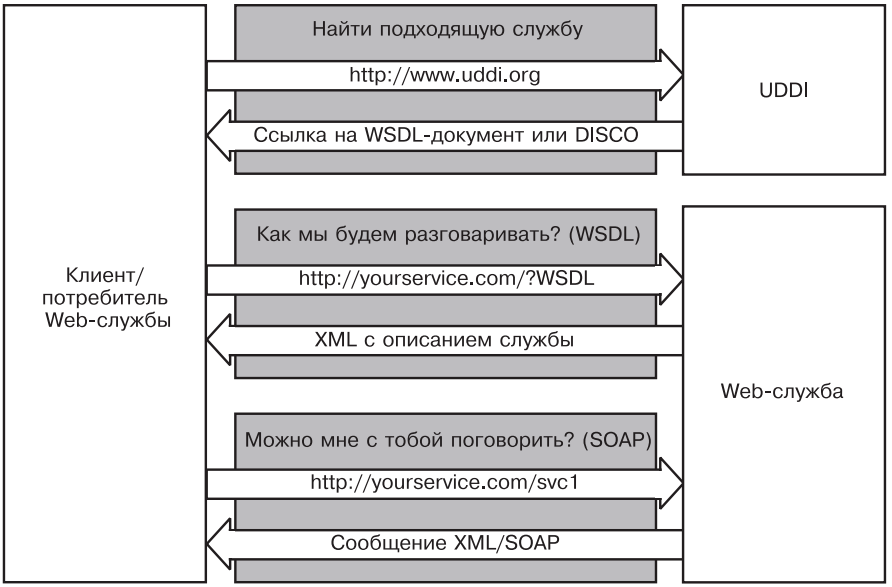


Рис. E.2. Архитектура Web-службы, представленная Microsoft

- Web-служба передает WSDL-документ, описывающий интерфейс(ы) для обращения к этой службе;
- клиент формирует SOAP-запрос, форматированный в соответствии с полученным документом;
- Web-служба возвращает SOAP-ответ.

Модель кода для Web-службы

Web-служба, созданная при помощи Visual Studio .NET, состоит из двух частей:

1. Первая часть – это файл с расширением **.asmx**, который играет роль адресуемой входной точки для Web-службы.
2. Вторая часть – это код в файле с расширением **.asmx.cs** (**.asmx.vb** для VB.NET), который реализует методы Web-службы.

Директива **WebService**, помещенная в **.asmx**-файл, указывает на публичный класс, реализующий логику Web-службы. В среде ASP.NET файл с расширением **.asmx** ссылается на прекомпилированные сборки, файлы с кодом или код, содержащийся в самом .asmx-файле. Класс Web-службы включает в себя один или несколько публичных методов для предоставления их через Web-службу. Методы Web-службы предваряются атрибутом **WebMethod**. По умолчанию, Visual Studio .NET использует файлы с кодом – например, **Service1.asmx.cs** или **Service1.asmx.vb** – когда вы разрабатываете Web-службу при помощи шаблона Web-службы ASP.NET.

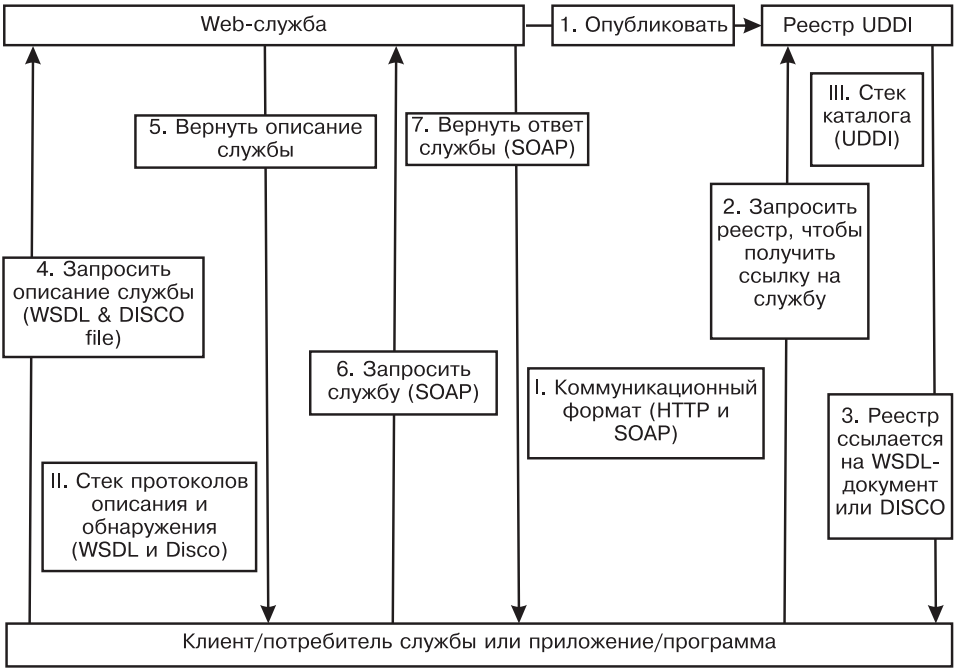


Рис. Е.3. Архитектура Web-службы

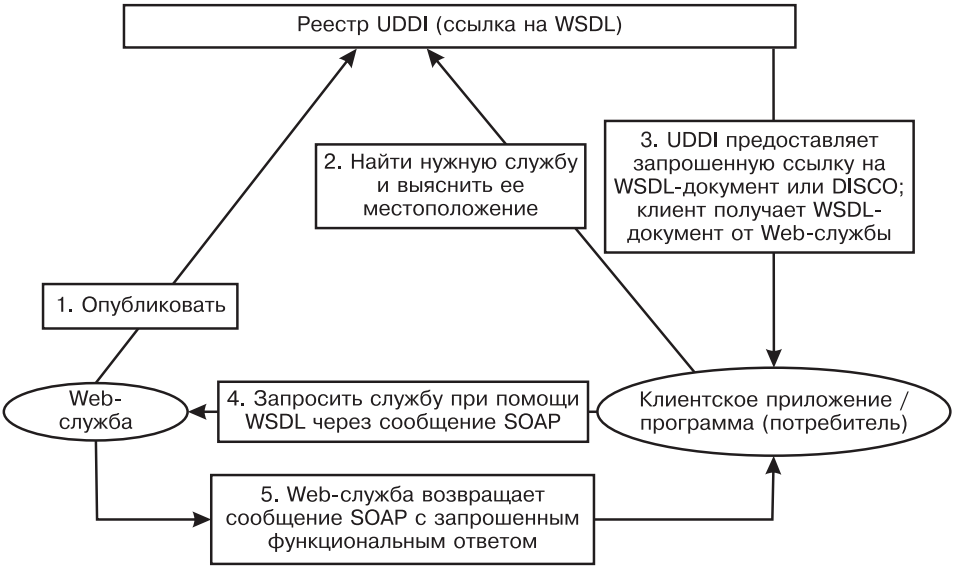


Рис. Е.4. Взаимодействие Web-службы

Разработка простой Web-службы

Выберите в меню File команду New | Project... В результате откроется диалоговое окно с различными типами проектов, как показано на рисунке E.5. Выберите ASP.NET Web service в разделе Visual C# Projects и укажите путь к локальному Web-серверу (или имя IIS-сервера, на котором будет располагаться служба). После этого щелкните на OK. Visual Studio автоматически создаст все необходимые файлы и ссылки. После щелчка на OK интегрированная среда отобразит файл **.asmx** в режиме просмотра Design View (поскольку Web-службы по природе своей – не визуальные объекты, вы не сможете в этом режиме манипулировать элементами управления или другими визуальными объектами при помощи мыши). Щелкните на ссылке Click here to switch to code view, чтобы перейти в режим исходного кода[1].

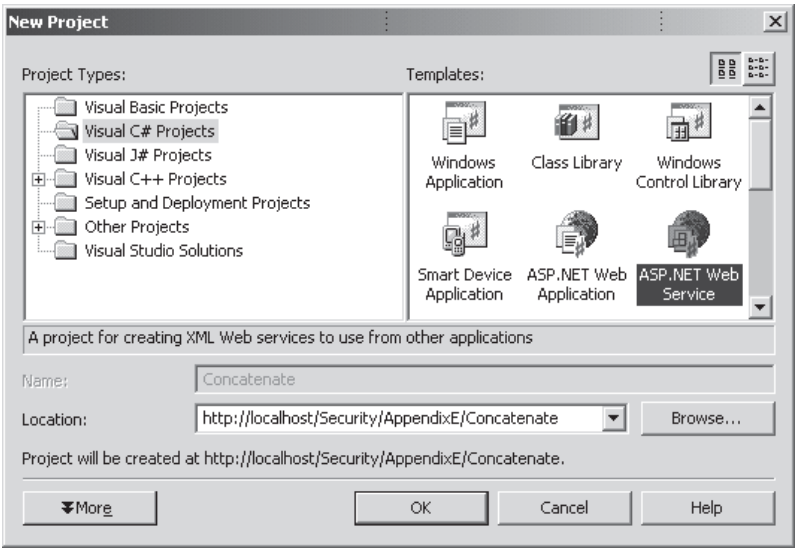


Рис. E.5. Создание проекта Web-приложения ASP.NET

После того как проект создан, Web-службе присваивается имя **Service1.asmx**. Переименуйте файл **Service1.asmx** в **Concatenate.asmx**. Адресуемая точка входа для Web-службы теперь специфицируется файлом **Concatenate.asmx**. Этот файл теперь – место для реализации методов, которые должна поддерживать Web-служба. Для того чтобы просмотреть **.asmx**-файл, щелкните на его имени правой кнопкой и выберите Source Code (Text) Editor в диалоговом окне Open With, а затем щелкните на Open, как показано на рисунке E.6.

[1] В проект, созданный Visual Studio, автоматически добавляется закомментированный код для примера – простая Web-служба типа «Hello World». Чтобы реализовать эту службу, просто раскомментируйте код.

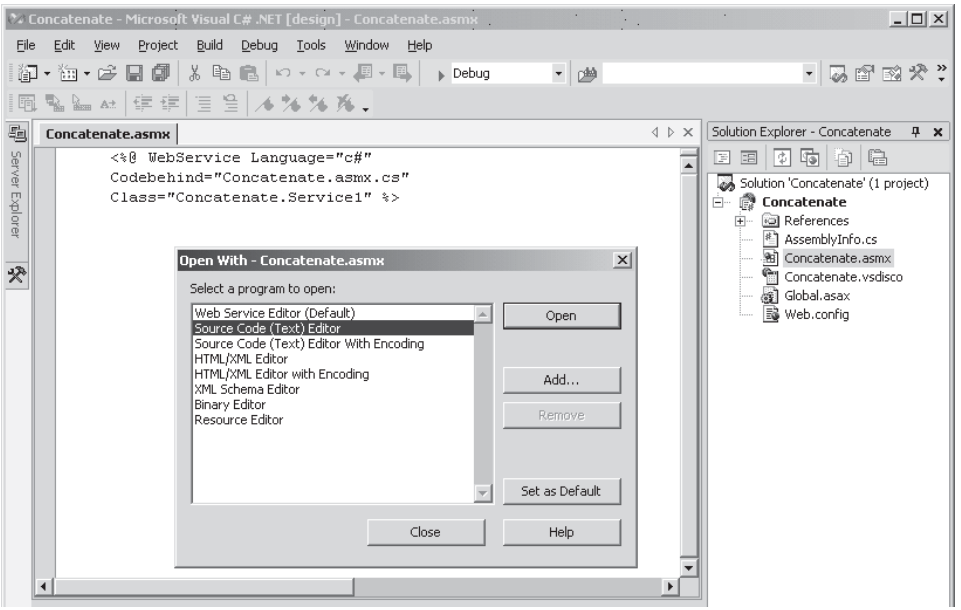


Рис. E.6. Диалоговое окно Open With (Открыть при помощи)

Используйте следующий URL для доступа к программе: *http://localhost/Security/AppendixE/Concatenate/Concatenate.asmx*.

Concatenate.cs и Concatenate.asmx.cs

Код файла **Concatenate.cs** включает просто директиву **WebService**:

```
<%@ WebService Language="c#"
Codebehind="Concatenate.asmx.cs"
Class="Concatenate.Service1" %>
```

Код, содержащийся в файле **Concatenate.asmx.cs**, приведен ниже.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace Concatenate
{
   [WebService(Namespace="http://www.phptr.com",
    Description="A Simple Web Service.", Name="Concatenation")]
```

```
   public class Service1 : System.Web.Services.WebService
   {
     public Service1()
     {
       InitializeComponent();
     }
   //Код, сгенерированный Component Designer

     protected override void Dispose( bool disposing )
     {
     }
//Если вы установите атрибут "WebMethodAttribute" для метода
//Web-службы, то этот метод будет доступен для вызова
//со стороны удаленного клиента

    [WebMethod (Description=
      "Web Service which provides Concatenation functionality.")]
     public string Concatenate(string s1, string s2)
     {
                    string s3="Concatenated String =";
                    return s3+s1+s2;
     }
    }
}
```

ТЕСТИРОВАНИЕ WEB-СЛУЖБЫ CONCATENATE

Даже без создания клиентского приложения для вашей Web-службы, вы можете проверить ее в действии средствами ASP.NET. После создания Web-службы в среде Visual Studio .NET просто щелкните на кнопке Run, чтобы увидеть службу Concatenate на тестовой странице Internet Explorer. Тестовая страница, на которой перечислены все доступные методы службы, изображена на рисунке E.7. (В нашем случае доступен только один, **Concatenate**.)[1]

Теперь давайте проверим нашу службу, щелкнув на ссылке Concatenate на тестовой странице. Вид страницы после щелчка на ссылке приведен на рисунке E.8. Страница состоит из двух частей. Первая часть состоит из двух текстовых полей и кнопки Invoke, позволяющей вызвать метод без того, чтобы создавать клиентское приложение. Вторая часть состоит из списка протоколов (HTTP POST, HTTP GET и SOAP), которые вы можете использовать для обмена с Web-службой и описания каждого из форматов сообщений. Если вы щелкнете на кнопке Invoke, результат отобразится в новом окне браузера в формате XML, как показано на рисунке E.9.

[1] Тестовая страница Internet Explorer – это автоматически созданная HTML-страница, при помощи которой вы можете выполнить и проверить методы Web-службы, а также просмотреть ее WSDL-документ.

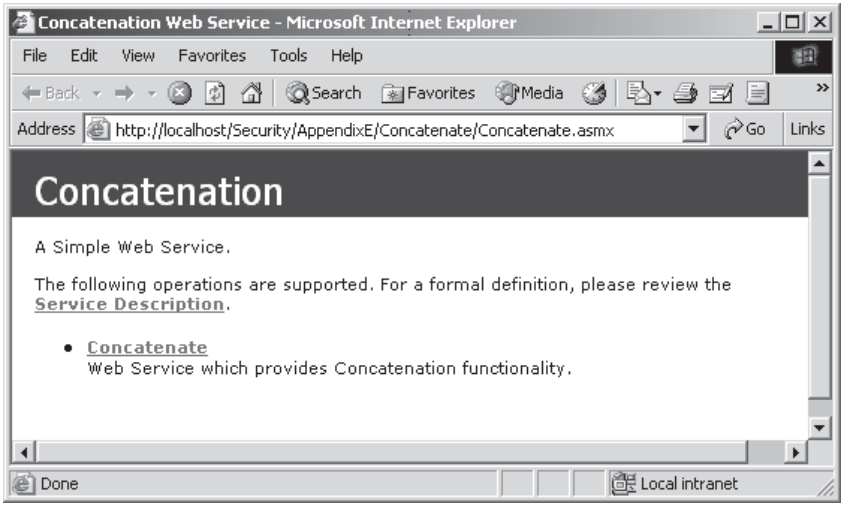


Рис. E.7. Тестовая страница Internet Explorer для службы Concatenate

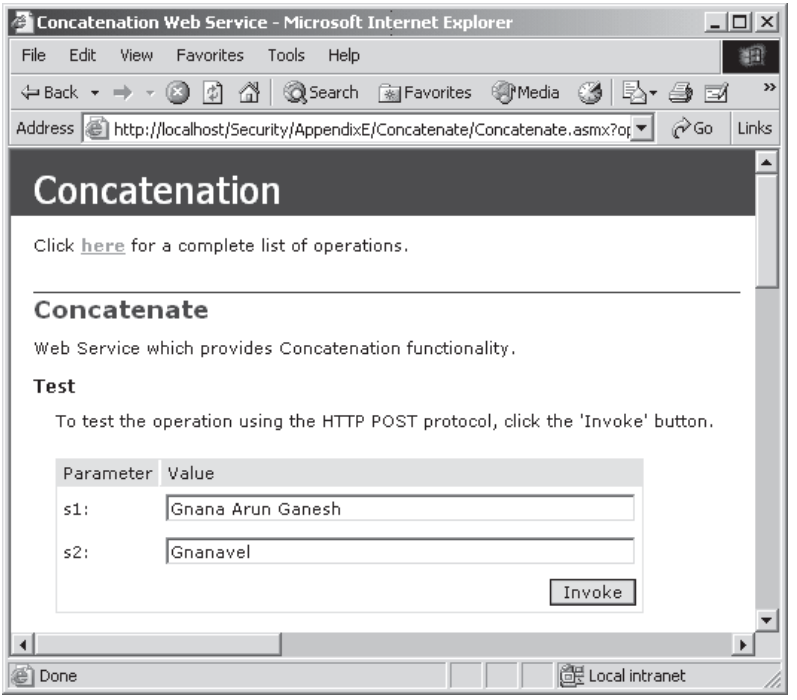


Рис. E.8. Вызов метода Concatenate

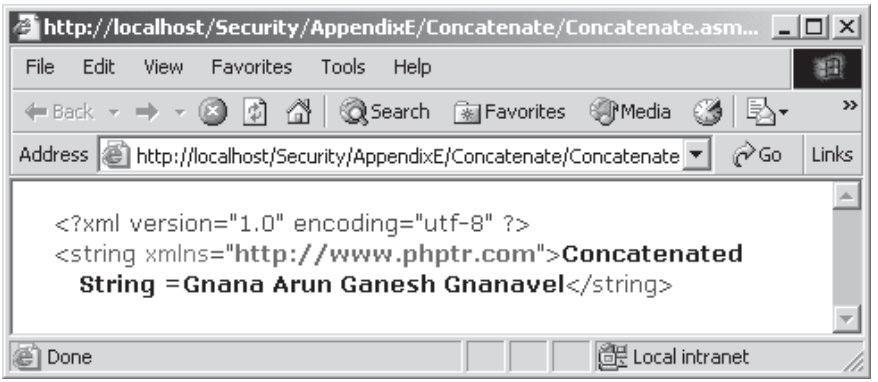


Рис. E.9. Результат работы метода Concatenate в формате XML

ОПИСАНИЕ СЛУЖБЫ

Если вы щелкнете на ссылке Service Description (Описание службы), отобразится WSDL-описание Web-службы, как показано на рисунке E.10.

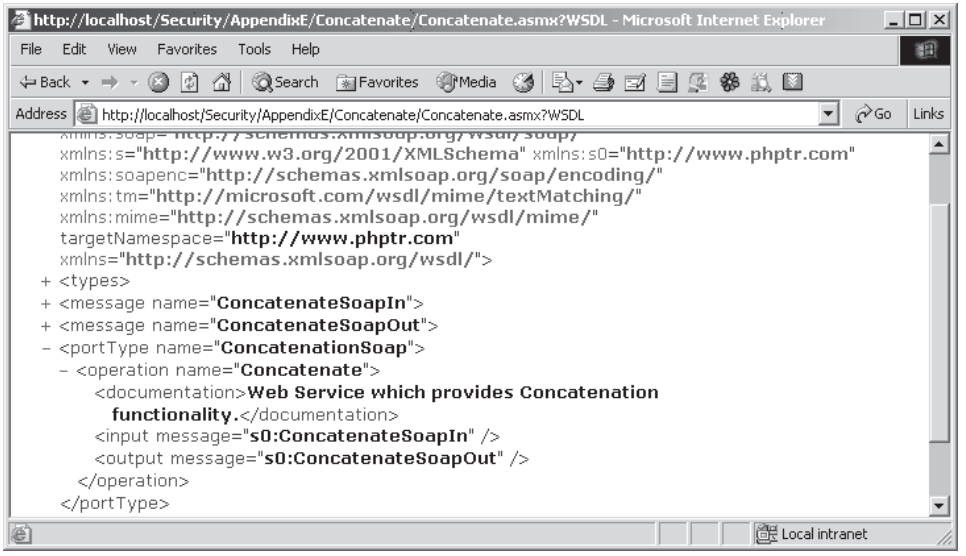


Рис. E.10. Просмотр описания службы Concatenate (WSDL-документ)

Документ WSDL можно логически разбить на две части.

- **Абстрактные определения:** типы, сообщения и типы портов.
- **Конкретные определения:** связи и службы.

- **Элемент WSDL: TYPE** • Этот элемент используется для определения типов данных в сообщениях SOAP. Элемент **<types>** содержит в себе XSD-схему. Например, раздел **<types>** в файле WSDL нашей Web-службы Concatenate определяет следующие типы:

- **Concatenate** используется, когда SOAP вызывает Web-службу.
- **ConcatenateResponse** используется, когда Web-служба отвечает на вызов. **Concatenate** состоит из двух элементов, **s1** и **s2**, которые определены при помощи строк типов XSD.

- **Элемент WSDL: MESSAGE** • Раздел <**message**> содержит описание сообщения.
- **Элемент WSDL: PORTTYPE** • Раздел <**portType**> описывает интерфейсы тех операций, которые поддерживает служба.
- **Элемент WSDL: SERVICE** • Раздел <**service**> описывает URL службы на сервере.
- **Элемент WSDL: BINDING** • Раздел <**binding**> определяет кодирование данных и используемые протоколы для каждой операции.

Позднее мы обсудим, как подключиться к уже созданной Web-службе. А сейчас давайте изучим подробней директиву **WebService** и пространство имен **System.Web.Services**.

Директива @ WebService

Для того чтобы объявить Web-службу, поместите директиву **WebService** в начале файла с расширением **.asmx**.

```
<%@ WebService class="Service1" Language="cs" %>
```

Директива **WebService** указывает класс, реализующий Web-службу и язык программирования, использованный для ее реализации. Вот важнейшие атрибуты этой директивы.

- **Атрибут Class:** этот атрибут указывает на класс, реализующий Web-службу. Он может указывать на класс, который определен непосредственно в этом же файле, как в нашем предыдущем примере, или же на класс в другом файле, помещенном в папку Bin в каталоге, где находится файл с расширением **.asmx**.
- **Атрибут Language:** этот атрибут указывает на язык программирования, использованный для создания службы. Вы можете создавать Web-службы на таких разных языках, совместимых со средой .NET, как C#, Visual Basic.Net и Jscript.NET.
- **Атрибут Codebehind:** если реализация службы находится в файле исходного кода, то этот файл задается атрибутом **Codebehind**.

Если реализация Web-службы находится в сборке, то объявить ее можно следующим образом:

```
<%@ WebService Language="cs" Class="ИмяКласса, ИмяСборки" %>
```

Пространство имен System.Web.Services

Пространство имен **System.Web.Services** состоит из классов, которые обеспечивают разработку и использование Web-службы. Давайте рассмотрим классы в пространстве **System.Web.Services** более детально. Эти классы описаны в таблице E.1.

Таблица E.1. Классы в пространстве имен System.Web.Services

Элемент	Описание
WebServiceAttribute (необязательный)	Может использоваться для передачи такой добавочной информации, как строка, описывающая функциональность службы
WebService (необязательный)	Очерчивает базовый класс для Web-служб, если вы хотите использовать такие общие объекты ASP.NET, как **Server.Session**, **User.Context** и **Application**
WebMethodAttribute	Если вы добавляете этот атрибут к методу в Web-службе, то метод становится доступным для вызова удаленным клиентом

Атрибут WebServiceAttribute

Вы можете использовать атрибут **WebServiceAttribute** для передачи дополнительной информации. Этот атрибут необязателен. Он используется в описании службы и в страничках Справки по службе. Для того чтобы опубликовать и выполнять Web-службу, этот атрибут не требуется. Свойства экземпляра класса **WebServiceAttribute** приведены в таблице E.2.

Таблица E.2. Свойства экземпляра класса WebServiceAttribute

Свойства	Описание
Description	Описательные комментарии о Web-службе
Name	Имя Web-службы ASP.NET
Namespace	Пространство имен XML по умолчанию для Web-службы

В нашем первом примере, **Concatenate.asmx**, вы могли заметить эти свойства в коде программы и в ее выходных данных:

- Description: "A Simple Web Service."
- Name: "Concatenation"
- Namespace: "http://www.phptr.com"

Значением по умолчанию для пространства имен Web-служб, которые находятся в стадии разработки, является **http://tempuri.org**, однако уже опубликованные Web-службы должны использовать более перманентное

пространство. Настоятельно рекомендуется персонализировать это пространство имен по умолчанию до того как делать Web-службу публично доступной. Это важно, потому что Web-службу необходимо отличать от других Web-служб. Для того чтобы применить атрибут **WebService**, вставьте его перед объявлением класса и задайте значения свойств **Namespace** и **Description**. Множественные свойства разделяйте запятыми.

```
[WebService(Namespace="http://www.phptr.com",
Description="A Simple Web Service.", Name="Concatenation")]
public class Service1 : System.Web.Services.WebService
   {
   //реализация
   }
```

Если вы запустите программу **Concatenate** без атрибута **WebService** и посмотрите на тестовую страницу, необходимость в атрибуте **WebService** станет вам яснее.

Класс WebService

Класс **System.Web.Services.WebService** является необязательным базовым классом для Web-служб и обеспечивает доступ к таким общим объектам ASP.NET, как **Session**, **Server**, **Application**, **Context** и **User**. Вы можете произвести Web-службу непосредственно из класса **System.Object**, однако по умолчанию, Web-службы, созданные при помощи Visual Studio .NET, автоматически производятся из базового класса **WebService**. Если вы создаете Web-службу, не производя ее класс из класса **WebService**, то вы не сможете получить доступ к объектам ASP.NET. Если Web-служба не наследует от базового класса **WebService**, она может получить доступ к внутренним объектам ASP.NET через **System.Web.HttpContext.Current**. Давайте рассмотрим повнимательней свойства класса **WebService**. Эти свойства описаны в таблице E.3.

Таблица E.3. Свойства класса WebService

Свойства	Описание
Application	Возвращает объект **Application** для текущего HTTP-запроса
Context	Возвращает ASP.NET **HttpContext** для текущего запроса, который инкапсулирует все специфичные для HTTP контексты, используемые сервером для обработки Web-запросов
Server	Возвращает **HttpServerUtility** для текущего запроса
Session	Возвращает экземпляр **HttpSessionState** для текущего запроса
User	Возвращает объект сервера ASP.NET **User**, который можно использовать для аутентификации текущего пользователя

Атрибут WebMethod

Если вы хотите сделать свой публичный метод доступным извне, как часть Web-службы, задайте этому методу атрибут **WebMethod**. Атрибут **WebMethod** предусматривает следующие свойства:

- **BufferResponse**;
- **Description**;
- **MessageName**;
- **CacheDuration**;
- **EnableSession**;
- **TransactionOption**.

АТРИБУТ WEBMETHOD: BUFFERRESPONSE

Свойство **BufferResponse** атрибута **WebMethod** отвечает за буферизацию ответов, которые должны отсылаться клиентам Web-службы. Если это свойство имеет значение true, то ответ на запрос накапливается в буфере в памяти, пока не иссякнет место в буфере или не будет накоплен весь ответ. После буферизации, ответ отправляется клиенту по сети. Значение по умолчанию – true. Техника буферизации улучшает производительность, сокращая обмен между рабочим процессом и процессом IIS. Вы можете установить это свойство в false, при этом ASP.NET будет буферизовать ответы порциями по 16 К. Если вам не нужно иметь размещенным в памяти весь ответ на запрос, или если метод службы возвращает клиенту огромный объем данных, то это свойство можно установить в false. Если это свойство имеет значение false, то расширения SOAP для соответствующего метода Web-службы отключаются.

```
[WebMethod (BufferResponse=false)]
public string GetData()
{
// реализация
}
```

АТРИБУТ WEBMETHOD: DESCRIPTION

Свойство **Description** атрибута **WebMethod** вы можете использовать для задания имени, под которым метод будет публиковаться. Это необязательное свойство. Оно используется в описании службы и в ее страницах Справки.

По умолчанию оно имеет значение **String.Empty**. В следующем примере для описания метода Web-службы используется строка **"Web Service which provides Concatenation functionality"** (Web-служба Concatenate предназначена для сцепления строк), и то же самое значение можно использовать в WSDL-файле, как показано на рисунке E.10.

```
[WebMethod (Description="Web Service which provides Concatenation
functionality")]
public string Concatenate(string s1, string s2)
{
   string s3="Concatenated string =";
   return s3+s1+s2;
}
```

АТРИБУТ WEBMETHOD: MESSAGENAME

Свойство **MessageName** атрибута **WebMethod** позволяет задать уникальное имя для перегруженного или полиморфного метода. Свойство **MessageName** можно использовать для создания псевдонимов для имен методов или свойств. Чтобы нагляднее увидеть роль этого свойства, давайте рассмотрим небольшой пример Web-службы, которая выполняет суммирование чисел и состоит из трех методов.

```
[WebMethod (Description="Add two floats.")]
public float Add(float s1, float s2)
{
   return s1 + s2;
}
[WebMethod (Description="Add two integers.")]
public int Add(int s1, int s2)
{
   return s1 + s2;
}
[WebMethod (Description="Add two three integers.")]
public int Add(int s1, int s2, int s3)
{
   return s1 + s2 + s3;
}
```

Эта программа откомпилируется без ошибок. Но если вы пошлете запрос к этой Web-службе, то столкнетесь со следующими ошибками перегруженных и полиморфных методов, соответственно.

- **System.Exception: Both Int32 Add(Int32, Int32) and Single Add(Single, Single) use the message name 'Add'. Use the MessageName property of the WebMethod custom attribute to specify unique message names for the methods.**
- **System.Exception: Both Int32 Add(Int32, Int32, Int32) and Int32 Add(Int32, Int32) use the message name 'Add'. Use the MessageName property of the WebMethod custom attribute to specify unique message names for the methods.**

(Обе функции Add используют одно имя сообщения. Используйте свойство **MessageName** атрибута **WebMethod**, для того чтобы задать уникальные имена этим методам.**)**

Если вы измените программу так, как показано ниже, при помощи свойства **MessageName**, то программа будет работать правильно.

```
[WebMethod (Description="Add two floats.",
MessageName="Add2Floats")]
public float Add(float s1, float s2)
{return s1 + s2;}

[WebMethod (Description="Add two integers.",
MessageName="Add2Integers")]
public int Add(int s1, int s2)
{return s1 + s2;}

[WebMethod (Description="Add two three integers.",
MessageName="Add3Integers")]
public int Add(int s1, int s2, int s3)
{return s1 + s2 + s3;}
```

Теперь, как видите, описания-«имена» WSDL стали уникальными.

КЕШИРОВАНИЕ В WEB-СЛУЖБЕ ASP.NET

Вы можете использовать кеширование для Web-службы ASP.NET, установив свойство **CacheDuration** атрибута **WebMethod** следующим образом:

```
[WebMethod (CacheDuration=30)]
```

Значение свойства **CacheDuration** указывает, сколько секунд система ASP.NET должна кешировать результаты. Вам следует задавать для свойства **CacheDuration** атрибута **WebMethod** значения больше нуля. По умолчанию в нем содержится значение нуль.

ТРАНЗАКЦИИ В WEB-СЛУЖБЕ ASP.NET

Вы можете задать «транзакционное» поведение (новую транзакцию) для Web-службы ASP.NET, для этого нужно выполнить следующие шаги:

1. Добавьте ссылку на **System.EnterpriseServices.dll**.
2. Используйте свойство **TransactionOption** атрибута **WebMethod**.

```
[WebMethod (TransactionOption=TransactionOptionRequiresNew)]
```

Пространство имен **System.EnterpriseServices** содержит методы и свойства, которые предоставляют для использования модель распределенных транзакций, заключенную в службах COM+. Если в процессе выполнения транзакционного процесса Web-службы генерируется исключение, автоматически отбрасывается вся транзакция (используется метод **SetAbort** класса **System.EnterpriseServices.ContextUtil**). Если исключений не было, то транзакция автоматически закрывается (используется метод **SetComplete** класса **System.EnterpriseServices.ContextUtil**). Свойство **TransactionOption** атрибута указывает, какой метод Web-службы участвует в транзакции. Перечисление членов **TransactionOption** приведено в таблице E.4.

Таблица Е.4. Перечисление членов TransactionOption

Элемент	Описание
Disabled	Игнорировать любые транзакции в текущем контексте
NotSupported	Создать компонент в контексте без транзакций
Required	Разделить транзакцию, если она существует; при необходимости создать новую транзакцию
RequiredNew	Создать компонент с новой транзакцией вне зависимости от состояния текущего контекста
Supported	Разделить транзакцию, если она существует

Свойство TransactionOption может принимать любое из этих значений. Однако перечисление TransactionOption метода Web-службы предусматривает только два варианта поведения:

- не участвовать в транзакции **(Disabled, NotSupported, Supported);**
- создать новую транзакцию **(Required, RequiredNew).**

По умолчанию выбрано значение **TransactionOption.Disabled.**

Управление сеансом

Свойство **EnableSession** атрибута **WebMethod** позволяет управлять состоянием сеанса метода Web-службы. Для этого нужно выполнить следующее:

- добавьте ссылку на пространство имен **System.WebServices;**
- установите свойство **EnableSession** атрибута **WebMethod** в true, как показано ниже.

```
[WebMethod(EnableSession=true)]]
Public string SessionCount()
{
// код
}
```

Протоколы

Когда вы создаете Web-службу ASP.NET, она автоматически поддерживает протоколы SOAP, HTTP GET и HTTP POST для вызова методов Web-службы. В HTTP GET данные пересылаются в строке запроса (пары «имя\значение»), которая присоединяется к URL. В методе HTTP POST данные не присоединяются к строке запроса, а вместо пар «имя\значение» передаются в виде отдельной строки, включаемой в заголовок HTTP, и поэтому непосредственно внешнему миру они не видны.

Типы данных, поддерживаемые методами HTTP GET и HTTP POST, включают строки (Int16, Int 32, Int64, Boolean, Single, Double, Decimal, DateTime и др.), перечисления и простые массивы. Вы не можете использовать методы HTTP GET и HTTP POST для представления сложных типов данных. Впрочем, протокол SOAP поддерживает более богатый набор типов данных, включая экземпляры объектов, наборы данных ADO.NET, узлы XML и сложные массивы.

Доступ к Web-службе

В работе с Web-службой есть два основных шага – это создание Web-службы и получение доступа к Web-службе. Мы обсудили, как создать Web-службу и как протестировать ее при помощи тестовой страницы Internet Explorer. Давайте теперь разберемся, как разработать клиентское приложение Windows Form. Для того чтобы воспользоваться услугами Web-службы, клиентскому приложению нужно обнаружить, получить ссылку и вызвать некую функциональность Web-службы. Клиентом Web-службы может быть приложение .NET любого типа, Web- или Windows-приложение. Давайте создадим простое клиентское приложение на основе Windows Form для нашей Web-службы Concatenate. Основные необходимые шаги сводятся к следующему:

- определить местоположение Web-службы при помощи диалогового окна Add Web Reference (если вы используете Visual Studio .NET);
- сгенерировать класс proxy для Web-службы при помощи инструмента WSDL.exe или добавлением Web-ссылки в ваш проект;
- сослаться на класс proxy в коде клиентского приложения, включив туда пространство имен;
- создать экземпляр proxy-класса Web-службы и вызвать через него методы Web-службы.

Генерация прокси

Класс proxy действует, как посредник между Web-службой и клиентом. Он передает полученные от клиента данные Web-службе и наоборот возвращает ответы Web-службы клиенту. По умолчанию, класс proxy использует для доступа к методу Web-службы сообщения SOAP. Сгенерировать класс proxy очень легко. Его можно создать двумя способами:

- при помощи утилиты командной строки Wsdl.exe;
- при помощи Visual Studio .NET.

Создание прокси-класса при помощи Wsdl.exe

Давайте создадим класс proxy для **Concatenate.asmx** и детально изучим его. Вам необходимо указать имя файла прокси, который нужно создать, и URL, по которому расположено описание службы.

```
Wsdl /out:Concatenate.cs http://localhost/Security
/AppendixE/Concatenate/Concatenate.asmx?wsdl
```

Утилита Wsdl.exe по умолчанию создаст прокси-класс на языке C#. Для того чтобы создать класс на другом языке, таком как VB.NET или Jscript.NET, необходимо добавить необязательный параметр **/I:**.

КОНСТРУИРОВАНИЕ СБОРКИ

Для того чтобы воспользоваться созданным прокси-классом, вам придется построить сборку, его содержащую. Сконструируйте сборку командой в командной строке:

```
csc /r:system.xml.dll /r:system.web.services.dll
/out:c:\Concatenate.dll /t:library Concatenate.cs
```

Синтаксис использования утилиты Wsdl выглядит следующим образом:

```
Wsdl /language:Язык /protocol:Протокол /namespace:ПространствоИмен
/out:ИмяФайла /username:ИмяПользователя /password:Пароль
/domain:Домен <url или путь к файлу>
```

На рисунке E.11 изображен вид сборки **Concatenate.dll** в окне IL DASM[1].

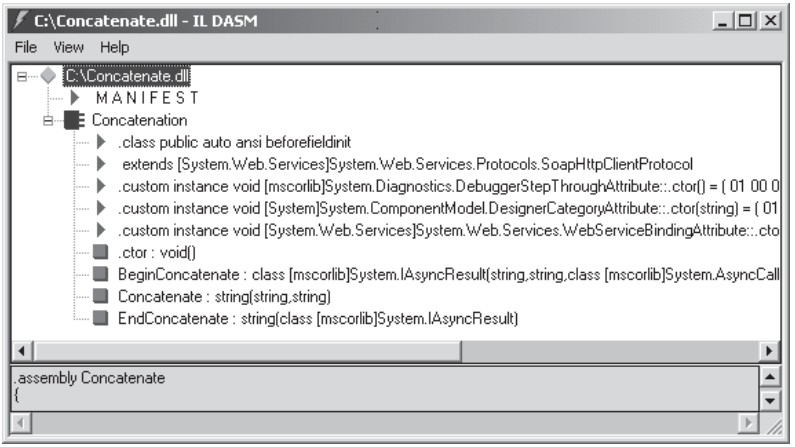


Рис. E.11. Просмотр сборки Concatenate.dll в окне IL DASM

[1] В пакете Microsoft .NET Framework SDK имеется инструмент, именуемый MSIL Disassembler (ILDasm.exe), который позволяет загрузить любую сборку Microsoft .NET (EXE или DLL) и исследовать ее содержимое, включая ее манифест, метаданные типов и набор инструкций.

Создание клиента Windows Form

Теперь мы создадим простое клиентское приложение на основе Windows Form при помощи Visual Studio.NET, которое будет взаимодействовать с Web-службой Concatenate.

Создайте проект типа Windows application и добавьте в него ссылку на сборку **Concatenate.dll**, а также на пространство **System.Web.Services**. Затем вы можете легко получить доступ к Web-службе, создав экземпляр объекта proxy в коде клиента следующим образом:

```
    private void button1_Click(object sender, System.EventArgs e)
{
            string s1,s2;
            Concatenation proxy = new Concatenation();
            s1=textBox1.Text.ToString();
            s2=textBox2.Text.ToString();
            textBox3.Text=proxy.Concatenate(s1,s2);
}
```

Экранный вывод программы приведен на рисунке E.12.

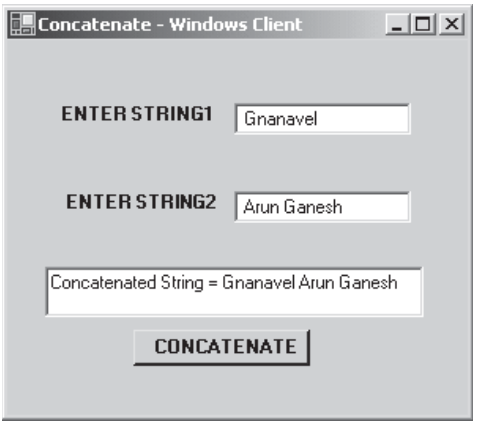


Рис. E.12. Экранный вывод программы Concatenationwinclient

СОЗДАНИЕ WEB-ССЫЛКИ В VISUAL STUDIO.NET

Создание Web-ссылки в Visual Studio.NET – простое дело. Единственное ограничение, связанное с использованием Visual Studio.NET, состоит в том, что в качестве протокола вы можете указать только SOAP. При создании ссылки с помощью утилиты Wsdl.exe вы можете задать протокол HTTP GET, HTTP POST или SOAP, используя параметр /**protocol**. Для того чтобы добавить ссылку на Web-службу, выберите команду Project | Add Web Reference. Затем щелкните на ссылке Web References on Local Web Server или введите путь к нужной службе в строку адреса.

Вы можете просмотреть тестовую страницу и файл WSDL, затем щелкните на кнопке Add Reference. Вы сможете просмотреть файлы **disco**[1] и WSDL, развернув в узел Web References в окне Solution Explorer[2]. Затем вы легко можете получить доступ к Web-службе, создав экземпляр объекта proxy в клиентском коде следующим образом:

```
private void button1_Click(object sender, System.EventArgs e)
   {
                    string s1,s2;
                    localhost.Concatenation proxy =
                          new localhost.Concatenation();
                    s1=textBox1.Text.ToString();
                    s2=textBox2.Text.ToString();
                    textBox3.Text=proxy.Concatenate(s1,s2);
   }
```

Асинхронное программирование Web-служб

Наряду с синхронными вызовами методов Web-службы вы можете использовать вызовы асинхронные. В чем состоит особенность асинхронного программирования? Изюминка этой техники состоит в том, что после отправки запроса к Web-службе клиент не должен дожидаться ответа.

Если предполагается, что Web-служба возвращает большие объемы данных, то есть основания для использования асинхронной технологии. Это значительно повысит производительность системы в целом. Клиент может выполнять последующие операции до тех пор, пока не вернется ответ Web-службы. Вот что здесь важно понимать: для того чтобы обращаться к Web-службе асинхронно, не требуется писать ее код каким-то специальным образом.

Когда прокси-класс создается для использования асинхронных вызовов, делается ли это при помощи Wsdl.exe или Visual Studio.NET, все методы, необходимые для реализации асинхронных вызовов, добавляются автоматически.

---

[1] Спецификация DISCO определяет алгоритм обнаружения описаний служб. Файл disco представляет собой XML-документ, содержащий ссылки на Web-службы, или он может содержать динамический список Web-служб по заданному пути.

[2] Если Web-служба изменилась, вы можете обновить класс прокси, щелкнув правой кнопкой мыши на имени сервера (в нашем случае это localhost) и выбрав команду Update Web References.

Два асинхронных метода (Begin и End)

Для каждого синхронного метода имеются асинхронные методы **Begin** и **End**. Например, в файле прокси **Concatenate.cs** для синхронного метода **Concatenate()** есть два асинхронных метода **BeginConcatenate()** и **EndConcatenate()**, как показано ниже.

```
public string Concatenate(string s1, string s2)
{
object[] results = this.Invoke("Concatenate", new object[]
{s1, s2});
return ((string)(results[0]));
}

public System.IAsyncResult BeginConcatenate(string s1, string s2,
System.AsyncCallback callback, object asyncState)
{
return this.BeginInvoke("Concatenate", new object[]
{s1, s2}, callback, asyncState);
}

public string EndConcatenate(System.IAsyncResult asyncResult)
{
   object[] results = this.EndInvoke(asyncResult);
   return ((string)(results[0]));
}
```

Метод **Begin** вызывается клиентом, для того чтобы инициировать процесс метода Web-службы (запрос). Метод **End** вызывается клиентом, для того чтобы получить результат запроса к методу Web-службы (ответ). Но когда, через какое время после вызова метода **Begin** мы должны вызвать метод **End**? Откуда нам знать, что запрашиваемый метод Web-службы завершил свою работу?

Существует четыре способа определить момент, когда асинхронный вызов метода Web-службы завершил свою работу.

1. Одновременно с методом **Begin** запускается функция обратного вызова, и прокси-класс обратится к этой функции, когда метод Web-службы завершит свою работу.

2. Клиент ожидает завершения работы вызванного метода при помощи методов класса **WaitHandles**.

3. Для проверки состояния процесса запрашивается значение **IAcyncResult.IsCompleted**. Результат вызова метода Web-службы будет доступен, когда это свойство будет равно **true**.

4. Напрямую вызывается метод **End**.

Первые два способа мы обсудим немного подробней.

Создание Web-службы ASP.NET «Калькулятор»

Создайте проект Web-службы на языке C# и укажите в качестве ее местоположения **http://localhost/Security/AppendixE/Calculator**. Напишите Web-метод **Calculator**, принимающий три аргумента: первым аргументом является оператор (_,-,*,/,Pow), затем следуют операнды, а возвращает функция результат выполнения операции, как видно из следующего кода.

```
[WebMethod(Description="Web-служба калькулятора.")]
public double Calculator(System.Double a,string c,
  System.Double b)
{
            switch (c)
            {
            case "+":
                  return a + b;
            case "-":
                  return a - b;
            case "/":
                  if (b == 0)
                  {
                  return 0;
                  }
                  return a /b;
            case "*":
                  return a * b;
            case "Pow":
                  return Math.Pow(a, b);
            default:
                  return 0;
            }
}
```

Протестируйте эти Web-службы при помощи тестовой страницы и проверьте правильность результатов вычислений.

ПРИМЕР: АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ (МЕТОД 1)

Для того чтобы проиллюстрировать технику асинхронного программирования по методу 1, мы рассмотрим простой пример. Создайте новое приложение Windows и добавьте в него Web-ссылку на Web-службу **http://localhost/Security/AppendixE/Calculator/Service1.asmx**. При использовании этого метода вместе с методом **Begin** передается функция обратного вызова **Callback**, которую вызовет прокси, когда прокси получит результат от метода Web-службы. Следующий пример иллюстрирует асинхронное программирование по методу 1.

```
private void button1_Click(object sender, System.EventArgs e)
   {
            localhost.Service1 proxy = new localhost.Service1();
            AsyncCallback cb = new AsyncCallback(AddCallback);
            double s1,s3;
            string s2;
            s1=Convert.ToDouble(textBox1.Text);
            s3=Convert.ToDouble(textBox3.Text);
            s2 = textBox2.Text;
            IAsyncResult ar =
                  proxy.BeginCalculator(s1,s2,s3,cb,proxy);
            /* Теперь вы можете заниматься какой-нибудь другой
полезной работой пока прокси не вызовет функцию Callback, что
будет означать завершение работы вызванного метода Web-службы */
   }

public static void AddCallback(IAsyncResult ar)
   {
            localhost.Service1 proxy = (localhost.Service1)
   ar.AsyncState;
            double a;
            // Вызов метода End.
            // EndCalculator (System.IAsyncResult asyncResult).
            a=proxy.EndCalculator(ar);
            MessageBox.Show(a.ToString(),"Output:");
   }
```

Результат обращения к **Calculator.cs** изображен на рисунке E.13.

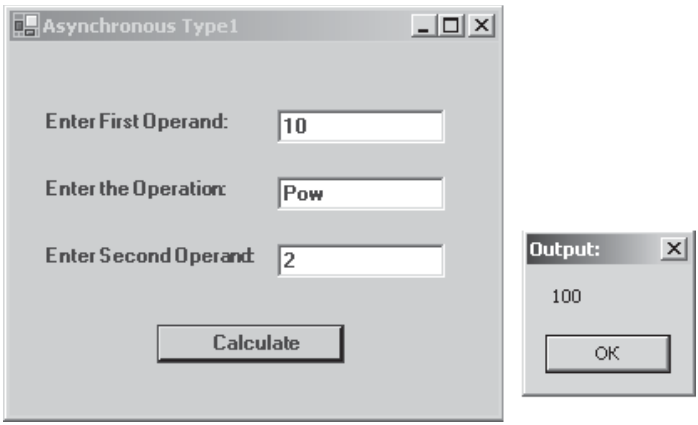


Рис. E.13. Результат обращения к Calculator.cs (асинхронный метод 1)

ПРИМЕР: АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ (МЕТОД 2)

Для иллюстрации второго метода вызова метода **End**, мы рассмотрим простой пример. Создайте новое Web-приложение ASP.NET и добавьте в него ссылку на Web-службу **http://localhost/Security/AppendixE/Calculator/Service1.asmx**. Включите также в проект пространство имен **System.Threading**. Это пространство имен вам потребуется для того, чтобы получить доступ к классу **WaitHandle**. При использовании этого метода асинхронного программирования клиент ожидает завершения работы метода Web-службы при помощи одного из методов класса **WaitHandle**.

После асинхронного вызова нужной Web-службы или служб этот метод ожидает:

- ответа одиночной Web-службы **(WaitHandle.WaitOne)**;
- ответа первой из нескольких Web-служб **(WaitHandle.WaitAny)**; этот вариант предпочтителен, если вы намерены обрабатывать ответы по мере их поступления;
- ответа всех из нескольких Web-служб **(WaitHandle.WaitAll)**; этот вариант предпочтителен, если вы намерены обрабатывать ответы после того, как все они будут получены.

В следующем примере мы продемонстрируем асинхронное программирование по методу 2.

```
private void Button1_Click(object sender, System.EventArgs e)
   {
              localhost.Service1 proxy = new localhost.Service1();
              double  s1,s3,Res;
              string s2;
              s1=Convert.ToDouble(TextBox1.Text);
              s2 = TextBox2.Text;
              s3=Convert.ToDouble(TextBox3.Text);
              IAsyncResult ar =
                     proxy.BeginCalculator(s1,s2,s3,null,null);
              //Теперь вы можете заниматься какой-нибудь
              //другой полезной работой .
              ar.AsyncWaitHandle.WaitOne();
              Res=proxy.EndCalculator(ar);
              TextBox4.Text=Res.ToString();
   }
```

Результат обращения к **Calculator.cs** изображен на рисунке E.14.

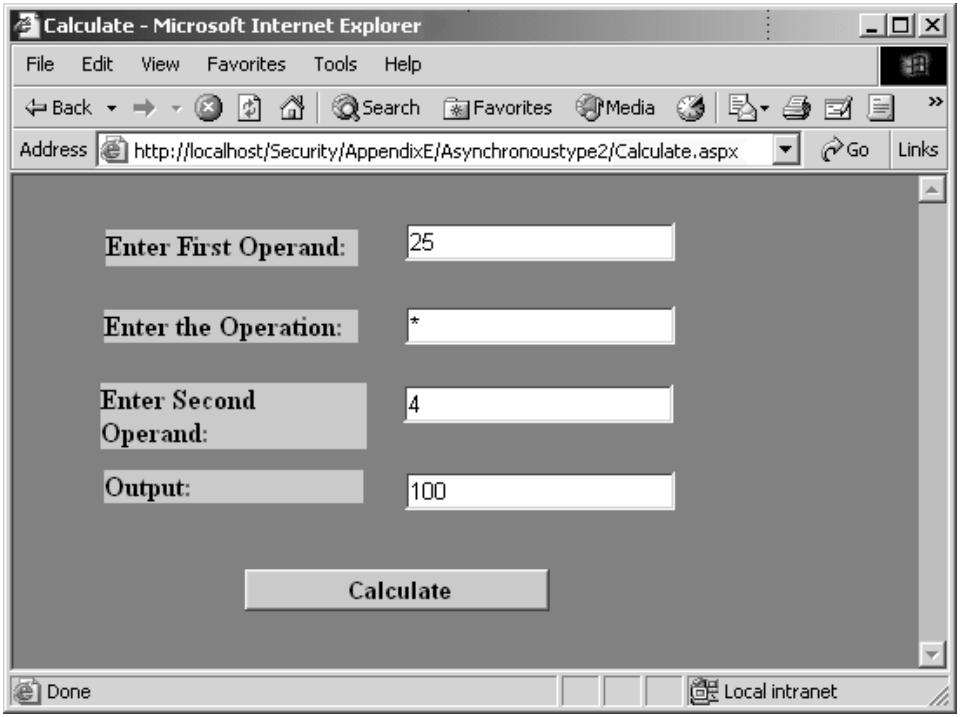


Рис. E.14. Результат обращения к Calculator.cs (асинхронный метод 2)

Web-службы все еще развиваются

Мы детально исследовали Web-службы, в том числе изучили различные техники программирования и узнали, как строятся распределенные приложения, но вам следует помнить об одном важном факте: Web-службы все еще находятся на этапе развития. С сегодняшним инструментарием Web-служб вы можете создать распределенные приложения, которые сообщаются при помощи сообщений SOAP.

Но вы не можете использовать Web-службы в таких критических, требующих высокой надежности приложениях, например, как финансовые задачи. Почему? Такие базовые спецификации, как WSDL, UDDI и SOAP все еще находятся в стадии формирования. Более того, у существующих Web-служб имеются некоторые слабые места такие, как безопасность, надежность, обработка транзакций, маршрутизация сообщений, качество обслуживания, межсистемное взаимодействие и управление эксплуатацией. Эти пробелы еще предстоит заполнить, прежде чем можно будет создавать приложения реального времени на основе Web-служб. Для решения этих проблем Microsoft, IBM и многие другие корпорации работают над архитектурой Global XML Architecture (GXA). Архитектура GXA представляет собой совокупность таких спецификаций, как WS-Security, WS-Routing, WS-Inspection, WS-Addressing, WS-Policy,

WS-Referral, WS-Coordination, WS-ReliableMessaging и WS-Transaction, которые расширяют функциональность SOAP и делают возможным создание качественных Web-служб реального времени.

Итоги

Web-службы – это развивающаяся архитектура распределенных вычислений, использующая такие стандартные протоколы, как HTTP, XML, XSD, SOAP и WSDL. Использование Web-служб сокращает затраты на разработку и обслуживание, позволяет обеспечить межсистемное взаимодействие и делает возможным разделение информации между бизнес-партнерами, причем новые функции интегрируются в уже существующие традиционные системы без необходимости создавать специализированные коммуникационные приложения. Платформа Microsoft.NET обеспечивает мощный инструментарий, позволяющий достаточно просто и быстро создавать Web-службы и клиентские приложения для них. В этом приложении мы вкратце исследовали тему Web-служб. Мы познакомились с директивой **WebService**, атрибутом **WebMethod**, атрибутом **WebService** и базовым классом **System.Web.Services.WebService**. Мы проиллюстрировали технику асинхронного программирования для Web-служб ASP.NET. на примерах. Наконец, мы обсудили потребность в более высоком уровне функциональности в том, что касается безопасности и надежности Web-служб реального времени.

Предметный указатель

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

A

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Х

Ц

Ч

Ш

Э

Оглавление

Минимальные системные требования определяются соответствующими требованиями программ Adobe Reader версии не ниже 11-й либо Adobe Digital Editions версии не ниже 4.5 для платформ Windows, Mac OS, Android и iOS; экран 10"

Справочное электронное издание

Серия: «Программисту»

Торстейнсон Питер
Ганеш Дж. Гнана Арун

КРИПТОГРАФИЯ И БЕЗОПАСНОСТЬ В ТЕХНОЛОГИИ .NET

Ведущий редактор *Л. Черепанова*
Художник *Ф. Инфантэ*
Компьютерная верстка *Л. Черепанова, Л. Катуркина*

Подписано к использованию 26.08.19.
Формат 145×220 мм

Издательство «Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-5272
e-mail: info@pilotLZ.ru, http://www.pilotLZ.ru

Эта книга позволит читателям:

- освоить базовые принципы криптографии, что даст возможность осмысленно применять криптографические средства .NET Framework;
- научиться применять симметричные и асимметричные алгоритмы, а также цифровые подписи;
- овладеть как традиционными способами программного шифрования, так и новыми технологиями, такими как шифрование XML и цифровая подпись XML;
- научиться применять все эти инструменты для обеспечения безопасности Web-служб и приложений ASP .NET.

Об авторах

Питер Торстейнсон — системный аналитик, занимавшийся программированием, обучением разработке программного обеспечения и созданием материалов для такого обучения более 10 лет. Он интересуется всеми аспектами C++, Java и C#, а также ATL, COM+, .NET и J2EE. Питер имеет степень бакалавра в области электротехники, а также является соавтором книг «.Net Architecture And Programming Using C++» и «Application Development Using Visual Basic .NET» (обе они выпущены издательством Prentice Hall PTR).

Дж. Гнана Арун Ганеш — разработчик, автор и консультант по .NET. Он руководит группой .NET Technology на Web Prodigies и отвечает за область .NET Reference Guide на InformIT. Арун — один из авторов материалов для курса обучения Object Innovations, обеспечивающих подготовку по фундаментальным программным технологиям. Он опубликовал более 50 статей по теме технологий .NET на многих Web-сайтах этой тематики.