Практическая криптография

Practical Cryptography

Niels Ferguson
Bruce Schneier

Wiley Publishing, Inc.

Практическая криптография

Нильс Фергюсон
Брюс Шнайер

Диалектика

Москва • Санкт-Петербург • Киев
2005

ББК 32.973.26–018.2.75
Ф43
УДК 681.3.07

Компьютерное издательство “Диалектика”

Главный редактор *С.Н. Тригуб*

Зав. редакцией *В.Р. Гинзбург*

Перевод с английского *Н.Н. Селиной*

Под редакцией *А.В. Журавлева*

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:
info@dialektika.com, http://www.dialektika.com

Фергюсон, Нильс, **Шнайер**, Брюс.

Ф43 Практическая криптография. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2004. — 432 с. : ил. — Парал. тит. англ.

ISBN 5–8459–0733–0 (рус.)

В современном деловом мире вопрос безопасности компьютерных систем приобретает решающее значение. Проигнорировав его, вы лишаете себя возможности заработать деньги, расширить свой бизнес, а следовательно, ставите под угрозу само существование вашей компании. Одной из наиболее многообещающих технологий, позволяющих обеспечить безопасность в киберпространстве, является криптография. Данная книга, написанная всемирно известными специалистами в области криптографии, представляет собой уникальное в своем роде руководство по практической разработке криптографической системы, устраняя тем самым досадный пробел между теоретическими основами криптографии и реальными криптографическими приложениями.

ББК 32.973.26–018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства JOHN WILEY&Sons, Inc.

Copyright © 2005 by Dialektika Computer Publishing.

Original English language edition Copyright © 2003 by Niels Ferguson and Bruce Schneier

All rights reserved including the right of reproduction in whole or in part in any form. This translation is published by arrangement with Wiley Publishing, Inc.

ISBN 5–8459–0733–0 (рус.)
ISBN 0–4712–2357–3 (англ.)

© Компьютерное изд-во “Диалектика”, 2005
© Niels Ferguson and Bruce Schneier, 2003

Оглавление

Содержание

Предисловие

На протяжении последнего десятилетия криптография гораздо больше способствовала разрушению безопасности цифровых систем, чем ее усовершенствованию. В начале 90-х годов прошлого века криптография считалась настоящей панацеей, способной обеспечить безопасность в Internet. Одни воспринимали криптографию как грандиозный технологический "уравнитель" — математический аппарат, позволяющий уравнять права и возможности защиты данных среднестатистического обывателя и крупнейших государственных разведывательных служб. Другие видели в ней оружие, применение которого может привести к гибели наций, если будет потерян контроль за поведением людей в киберпространстве. Третьи представляли, что это настоящий рай для наркоторговцев, террористов и распространителей детской порнографии, которые смогли бы общаться между собой в атмосфере полной секретности. Даже неисправимым реалистам стало казаться, что криптография — то самое средство, которое приведет к расцвету глобальной коммерции в новом интерактивном сообществе.

Прошло 10 лет, и ожидания не оправдались. Несмотря на распространение криптографии, наличие государственных границ в Internet стало ощутимым более чем когда-либо. Способность обнаруживать и прослушивать переговоры членов криминальных группировок гораздо больше зависит от политики и человеческих ресурсов, нежели от математического аппарата. У частных лиц нет никаких шансов дотянуться до уровня хорошо финансируемых государственных разведслужб. И наконец, наблюдавшийся расцвет глобальной коммерции никак не связан с внедрением криптографии в широкие массы.

В большинстве случаев применение криптографии не дало пользователям Internet практически ничего, кроме ложного ощущения безопасности. Криптография обещала обеспечить безопасность обмена данными, но сделать это так и не удалось. И это плохо для всех (за исключением, разумеется, злоумышленников).

Причины подобного провала кроются не столько в криптографии как в математической науке, сколько в криптографии как в инженерной дисциплине. На протяжении последнего десятилетия мы разработали, реализовали и выпустили в свет массу криптографических систем. Как ни печально, превратить математические перспективы криптографической безопасности

в реальную безопасность оказалось намного сложнее, чем можно было предположить. Это и есть самая трудная часть программы.

Многие разработчики относятся к криптографии как к некоему волшебному порошку. Стоит только "посыпать" им аппаратное или программное обеспечение, как оно тут же приобретет то самое магическое свойство безопасности. Слишком многие потребители полагаются на волшебное действие этого порошка, слепо доверяя слову "зашифрованный" в громких рекламных кампаниях. Недалеко от них ушли и серьезные аналитики, которые на основании длины ключей шифрования провозглашали один продукт более безопасным, нежели другой.

Каждая система безопасна настолько, насколько безопасно ее самое слабое звено, а математический аппарат криптографии никогда не был ее слабым звеном. Фундаментальные концепции, лежащие в основе криптографии, безусловно, важны, однако гораздо важнее то, как они реализуются и используются на практике. Спорить о том, какой должна быть длина ключа — 112 или 128 бит, все равно что вкопать в землю огромный столб и надеяться, что злоумышленник в него врежется. Вы можете долго выяснять, какой высоты должен быть столб — километр или полтора километра, а злоумышленник просто обойдет его стороной. Однако безопасность — это не столб, а настоящий забор: т.е. целый комплекс неких вещей, делающих криптографию действительно эффективной.

Практически во всех книгах по криптографии, изданных на протяжении последних 10 лет, ощущается стойкий привкус "волшебного порошка". Все восхваляют преимущества "тройного" DES, скажем, со 112-битовым ключом шифрования, не упоминая о том, как следует генерировать или использовать ключи этого алгоритма. Книга за книгой описывает сложные протоколы, не затрагивая общественных или бизнес-ограничений, в рамках которых приходится применять эти протоколы, и рассматривает криптографию как чисто математическую дисциплину, не тронутую рамками и реалиями земного мира. Однако именно эти рамки и реалии определяют различие между мечтами о криптографическом чуде и суровыми буднями цифровой безопасности.

Книга *Практическая криптография* тоже посвящена этой дисциплине, однако здесь речь идет вовсе не о той "незапятнанной" криптографии, которая упоминалась выше. Назначение этой книги состоит в том, чтобы открыто описать все ограничения и аспекты применения криптографии в реальной жизни, а также рассказать о разработке действительно безопасных криптографических систем. В некотором смысле данная книга является продолжением книги Брюса Шнайера *Applied Cryptography* [86], впервые опубликованной более 10 лет назад. Однако, в отличие от книги *Applied Cryptography*, дающей широкое представление о криптографии и тысячах ее возможностей, *Практическая криптография* охватывает достаточно узкую, строго определенную

область знаний. Мы не предлагаем вам десятки вариантов; мы рассматриваем *один* вариант, описывая то, как его правильно реализовать. Книга *Applied Cryptography* демонстрирует поразительные возможности криптографии как математической науки — что возможно и что достижимо, в то время как *Практическая криптография* содержит конкретные советы, предназначенные для людей, которые разрабатывают и реализуют криптографические системы.

Настоящая книга — это попытка сократить разрыв между теорией криптографии и ее применением в реальной жизни, а также научить разработчиков использовать ее для повышения безопасности систем.

Мы позволили себе взяться за написание этой книги, поскольку оба являемся опытными специалистами в области криптографии. Брюса хорошо знают по его книгам *Applied Cryptography* (она упомянута выше) и *Secrets and Lies* [88], а также по информационному бюллетеню *Crypto-Gram*. Нильс отточил свое криптографическое мастерство, разрабатывая криптографические системы платежей в институте CWI (Национальный исследовательский институт математики и информатики Нидерландов) в Амстердаме и позднее в нидерландской компании DigiCash. Брюс разработал знаменитый алгоритм шифрования Blowfish, и мы оба принимали участие в разработке алгоритма Twofish. Исследования Нильса привели к появлению первого представителя нового поколения эффективных протоколов анонимных платежей. Общее количество написанных нами научных статей выражается трехзначным числом.

Что еще более важно, мы оба имеем обширный опыт в разработке и построении криптографических систем. С 1991 по 1999 годы консалтинговая компания Брюса Counterpane Systems предоставляла услуги по проектированию и анализу для нескольких крупнейших компьютерных и финансовых корпораций мира. В последние годы компания Counterpane Internet Security, Inc. занимается предоставлением услуг по слежению за безопасностью и ее обеспечению для больших корпораций и правительственных служб по всему миру. Нильс тоже работал в Counterpane перед тем, как основал собственную консалтинговую компанию MacFergus. Мы живем и дышим криптографией. Мы наблюдаем, как она "прогибается" под тяжестью реалий разработки программных систем и, что еще хуже, под тяжестью реалий бизнеса. Мы позволили себе издать эту книгу, поскольку уже десятки раз писали ее для клиентов, которых консультируем.

Как читать эту книгу

Книгу *Практическая криптография* вряд ли можно назвать справочником. В ней прослеживается процесс проектирования криптографической си-

стемы от выбора конкретных алгоритмов через всевозможные наслоения вопросов безопасности до построения инфраструктуры, необходимой для работы этой системы. На протяжении книги обсуждается одна-единственная проблема криптографии, лежащая в основе практически каждой криптографической системы: как обеспечить безопасность общения двух людей. Сконцентрировав все внимание на одной проблеме и одной философии проектирования, применяемой для ее решения, мы верим, что можем рассказать больше о реальных аспектах разработки криптографических систем.

Мы оба уже издавали книги и прекрасно знаем, что это не имеет никакого отношения к точным наукам. Как бы мы ни старались, нам не удастся избежать ошибок. Простите за откровенность, но мы просто реально смотрим на вещи. (Что интересно, криптографические системы страдают от той же проблемы; это обсуждается в нескольких главах.) Конечно же, мы приложили все усилия к тому, чтобы довести свою книгу до совершенства, и вместе с тем разработали процедуру, гарантирующую, что все наши ошибки (а они, увы, неизбежны) когда-нибудь будут исправлены.

- Прежде чем приступать к чтению книги, посетите Web-узел `http://www.macfergus.com/pc` и загрузите текущий список исправлений.
- Если вы обнаружите в книге ошибку, проверьте, не встречается ли она в списке исправлений.
- Если ее там нет, пожалуйста, сообщите о ней по адресу:

  `practical-cryptography@macfergus.com`

  Мы непременно добавим ее к списку.

На наш взгляд, криптография — это самая интересная вещь во всей математике. Мы постарались наполнить книгу этим ощущением и надеемся, что вам понравится результат. Спасибо, что вы с нами.

Январь 2003 года

Нильс Фергюсон
Амстердам
Нидерланды
`niels@macfergus.com`

Брюс Шнайер
Миннеаполис, Миннесота
США
`schneier@counterplane.com`

Глава 1

Наша философия проектирования

Эта книга посвящена безопасности, а точнее, построению безопасных криптографических систем. Работая над книгой, мы просто помешались на безопасности, и на то есть серьезные причины. За все годы работы в сфере защиты информации мы еще не видели ни одной системы, которая была бы полностью безопасна. Это действительно правда. В защите абсолютно каждой системы, которую мы анализировали, находились те или иные "прорехи". Разумеется, в каждой системе есть и хорошие компоненты, однако они всегда используются небезопасными способами.

Если общество хочет обезопасить свое цифровое будущее, каждый должен осознать реалии окружающего мира и стать лучше. Надеемся, что наша книга внесет свою лепту в это благое дело.

Эта книга содержит массу практической информации о криптографических системах. К сожалению, она не принесет никакой пользы, пока вы не осознаете одну простую вещь: безопасность слишком важна, чтобы пренебрегать ею. Но заботиться о безопасности — значит, быть безжалостным во многих других областях. К этому сложно привыкнуть. Прошли многие годы, прежде чем мы смогли стать достаточно безжалостными. Невозможно обеспечить "чуть-чуть" безопасности. Это все равно что построить забор только с одной стороны дома или, например, запереть входную дверь и оставить открытым черный ход. Безопасность — это свойство, которое не знает компромиссов. Одна небольшая дыра в заборе — и о безопасности можно забыть. Таким образом, все должно быть принесено в жертву безопасности. По своему опыту мы знаем, что подобные идеи крайне сложно "продать" руководству. Тем не менее без них не обойтись, чтобы действительно чувствовать себя спокойно в цифровом мире.

1.1 Обратная сторона производительности

Для того чтобы поверить в существование моста через Ферт-оф-Форт в Шотландии, его нужно увидеть собственными глазами. Настоящее инженерное чудо XIX века, он просто невероятно огромен по сравнению с идущими по нему поездами. На строительство моста ушло так много материала и затрачены такие огромные деньги, что в это просто невозможно поверить. Кроме того, проектировщики моста столкнулись с проблемой, которую еще никому не удавалось решить успешно: построить большой стальной мост. Они проделали поразительный объем работ и весьма преуспели: их мост используется и сегодня, более ста лет спустя. Именно так и должны поступать настоящие разработчики.

Со временем инженеры научились строить такие мосты намного эффективнее и с меньшими затратами средств. Между тем главным приоритетом в мостостроении было и остается получение безопасно функционирующего моста, а сокращение стоимости — это уже второстепенный вопрос.

К сожалению, современная компьютерная безопасность характеризуется прямо противоположными приоритетами. Основная цель проектирования информационных систем, как правило, предполагает соответствие жестким требованиям эффективности. Главным приоритетом всегда является скорость даже в тех областях, где она не важна. Стремясь к повышению скорости, экономят на безопасности. Между тем безопасность — это та область проектирования, в которой у нас недостаточно навыков для построения хорошей системы даже при наличии неограниченного бюджета. Результатом такого подхода неизбежно является создание системы, наверное в чем-то и эффективной, однако отнюдь не безопасной.

История моста через Ферт-оф-Форт не обошлась и без отрицательного опыта. В 1878 году Томас Бауч (Thomas Bouch) завершил строительство моста через залив Ферт-оф-Тей близ города Данди, в те времена это был самый большой мост в мире. Бауч использовал новую технологию, скомбинировав чугун и кованое железо, и его мост считался настоящим чудом техники. К сожалению, не прошло и двух лет, как в сильную бурю 28 декабря 1879 года мост упал, когда по нему ехал поезд с 75 пассажирами. Все погибли. Это была самая крупная техногенная катастрофа того времени[1]. И когда через несколько лет началось проектирование моста через Ферт-оф-Форт, его разработчики не пожалели металла — не только для того, чтобы сделать мост

[1] Об этом происшествии Вильям Мак-Гонагалл (William McGonagall) написал знаменитую поэму, которая оканчивается словами: “For the stronger we our houses do build/The less chance we have to be killed” (чем крепче мы будем строить свои дома, тем менее вероятно, что нас убьют). Этот совет весьма актуален и сегодня.

действительно прочным, но и чтобы он *выглядел* прочным для тех, кто по нему ездит.

Все мы знаем, что проектировщики иногда ошибаются, особенно когда разрабатывают что-то новое. Порой эти ошибки приводят к гибели людей. Инженеры викторианской эпохи преподали нам хороший урок: если что-то не получается, оглянитесь назад и станьте более консервативными. Современная компьютерная индустрия напрочь забыла этот урок. Несмотря на все новые и новые сбои в системах безопасности, которые обнаруживаются чуть ли не каждую неделю, мы упорно рвемся вперед, списывая все неудачи на происки судьбы. Мы не возвращаемся к стадии разработки, чтобы спроектировать что-нибудь более консервативное, а просто выпускаем несколько исправлений и надеемся, что они решат проблему. Весьма неблагоразумно!

Должно быть, вы уже поняли, что в любых обстоятельствах мы всегда отдаем предпочтение безопасности перед эффективностью. Сколько процессорного времени мы бы согласились потратить на обеспечение безопасности? Да практически все. Мы нисколько не огорчимся, если 90% мощности процессора будет потрачено на поддержку надежной системы безопасности. Нехватка компьютерной безопасности является серьезной помехой для нас, как и для большинства остальных пользователей. Именно из-за нее все еще приходится рассылать твердые копии документов с подписями и печатями и беспокоиться о вирусах и прочих атаках на компьютерные системы. С каждым годом "цифровые" мошенники будут углублять свои знания и наращивать арсенал применяемых средств, поэтому проблема компьютерной безопасности будет становиться все более актуальной. То, что мы сейчас наблюдаем, — это только зарождение цифровой преступности. Если мы хотим и дальше использовать Internet для осуществления бизнес-транзакций, то должны намного лучше обезопасить свои компьютеры.

Существует множество способов обеспечения безопасности. Тем не менее, как отметил Брюс в книге *Secrets and Lies*, хорошая система безопасности — это всегда сочетание предупреждения, обнаружения и реакции [88]. Функции криптографии заключаются в предупреждении и должны быть реализованы как можно лучше, чтобы не слишком перегружать части системы, относящиеся к обнаружению и реакции (которые могут и должны включать в себя ручное вмешательство). Как бы там ни было, эта книга посвящена криптографии, поэтому мы сконцентрируем внимание именно на ней.

Конечно, мы знаем, что большинству из вас не понравится идея насчет 90% мощности процессора. Но зачем же еще нужны наши компьютеры? Никто не сможет набрать больше 10 символов в секунду (в хороший день), а с этим прекрасно справлялись и старые машины, применявшиеся еще лет 10 назад. Современные машины в тысячи раз быстрее. Если 90% мощности процессора пойдет на обеспечение безопасности, компьютер будет работать

всего лишь в 10 раз медленнее — примерно с такой же скоростью, какую имели компьютеры около пяти лет назад. А тех компьютеров было более чем достаточно для повседневной работы.

Существует всего несколько операций, которые выполняются относительно медленно: загрузка Web-страниц, печать документов, запуск некоторых приложений, загрузка компьютера и т.п. Хорошая система безопасности не приведет к замедлению этих процессов. Современные компьютеры настолько быстры, что придумать адекватную нагрузку для них очень и очень сложно. Конечно же, мы можем применять альфа-сопряжение, трехмерную анимацию или даже голосовое управление. Тем не менее компоненты этих приложений, ответственные за выполнение сложных математических расчетов, не осуществляют никаких действий, касающихся обеспечения безопасности, поэтому функционирование системы безопасности никак не повлияет на их работу. Основная нагрузка ложится на оставшуюся часть системы, которая уже быстра настолько, насколько это вообще может осознать человек. Что из того, что она станет работать всего лишь в 10 раз медленнее? В большинстве случаев нагрузка на систему будет вообще незаметна. Впрочем, даже если она окажется значительной, что поделаешь — бизнес тоже требует жертв.

Если вас когда-нибудь начнут уговаривать поступиться безопасностью в пользу эффективности, повторите про себя несколько раз: “У нас уже куча быстрых и небезопасных систем. Зачем нам еще одна?” Как показывает практика, это простое объяснение действует на людей гораздо эффективнее, чем многочасовые рассказы о различных степенях безопасности.

Вы наверняка уже поняли, что нашим главным приоритетом является безопасность, безопасность и еще раз безопасность, а производительность стоит где-то в конце списка. Разумеется, мы тоже хотим, чтобы система была как можно более эффективной, но только не за счет безопасности. Мы понимаем, что наша философия проектирования не всегда применима в реальном мире. Зачастую реалии рынка компьютерных систем заглушают робкие требования обеспечения безопасности. Системы редко разрабатываются “с нуля” и зачастую требуют пошагового обеспечения безопасности или на каждом этапе разработки, или после завершения их развертывания. Они должны обладать обратной совместимостью с существующими небезопасными системами. Нам обоим приходилось разрабатывать системы безопасности в подобных ограничениях, и мы можем подтвердить, что разработать хорошую систему безопасности в таких условиях практически невозможно. Философия проектирования, которую мы пропагандируем в этой книге, такова: безопасность прежде всего и превыше всего. Вот какими мы хотели бы видеть все коммерческие системы.

1.2 Обратная сторона оснащенности

Не существует сложных систем, которые были бы безопасными. Сложность — главный враг безопасности; она практически всегда выражается в количестве дополнительных средств, возможностей и параметров, доступных пользователю программы.

Мы еще вернемся к вопросу о том, почему сложность автоматически исключает безопасность. Пока же приведем только основной аргумент. Представьте себе компьютерную программу с 20 параметрами, каждый из которых может быть включен либо выключен. Это предполагает наличие более миллиона различных конфигураций программы. Чтобы убедиться в работоспособности программы, достаточно протестировать лишь несколько наиболее распространенных сочетаний значений параметров. А чтобы убедиться в ее безопасности, необходимо оценить каждую из миллиона возможных конфигураций и проверить устойчивость этих конфигураций к каждому из возможных видов атак. Это, конечно же, невозможно. Между тем количество параметров большинства программ на несколько порядков превышает 20. Итак, чтобы создать что-нибудь безопасное, необходимо сделать его простым.

Простая система не обязательно должна быть маленькой. Всегда можно построить большую систему, которая при этом останется относительно простой. Сложность измеряется количеством объектов, взаимодействующих друг с другом в определенный момент времени. Если влияние параметра ограничивается только небольшой частью приложения, он не будет взаимодействовать с другим параметром, влияние которого распространяется на другую часть приложения. Чтобы построить большую и вместе с тем простую систему, необходимо создать очень простой и понятный интерфейс между различными частями системы. Программисты называют это разбивкой на модули. Хороший простой интерфейс изолирует детали реализации модуля от остальных частей системы. И это должно касаться всех параметров или средств, реализованных в данном модуле.

Вообще-то можно было и не говорить о необходимости разбивки на модули. Все это прописные истины программной инженерии. К сожалению, их слишком редко соблюдают в реальном мире.

В этой книге мы постарались определить простые интерфейсы для базовых криптографических объектов. Никаких средств, никаких параметров, никаких частных случаев, никаких примечаний или исключений, требующих запоминания, — только самые простые определения из всех возможных. Некоторые из этих определений новые, разработанные в процессе написания книги. Они помогли нам оформить свои размышления о хороших системах безопасности и, надеемся, помогут и вам.

Глава 2

Криптография в контексте окружающего мира

Криптография — это искусство и наука шифрования. По крайней мере так она начиналась. Сегодня понятие криптографии значительно расширилось и включает в себя аутентификацию, цифровые подписи и множество других элементарных функций безопасности. Криптография все еще остается искусством и наукой: чтобы построить хорошую криптографическую систему, необходимо обладать глубокими научными знаниями и приличным багажом той самой "черной магии", которую называют опытом.

Криптография представляет собой весьма обширную область науки. На конференциях по криптографии поднимаются самые разные темы, связанные с компьютерной безопасностью, высшей математикой, экономикой, квантовой физикой, гражданским и уголовным правом, статистикой, проектированием микросхем, экстремальным программированием, политикой, проблемами пользовательского интерфейса и т.п. Эта книга посвящена очень небольшой области криптографии, а именно ее практической стороне. Мы надеемся продемонстрировать вам, как реализовать криптографические методы в реальных системах.

Многообразие аспектов криптографии — вот то, что придает ей поистине неповторимое очарование. По большому счету, криптография представляет собой смесь самых разных областей науки. В ней всегда можно чему-нибудь поучиться, а новые идеи приходят буквально отовсюду. С другой стороны, такое многообразие является и одной из причин сложности криптографии. Ее невозможно понять *целиком*. На свете нет ни одного человека, который бы знал все или хотя бы почти все о криптографии. Разумеется, не знаем этого и мы. Итак, вот вам первый урок криптографии: относитесь ко все-

му критически. Не нужно слепо верить всему написанному, даже если это научная книга.

2.1 Роль криптографии

Сама по себе криптография довольно бесполезна. Она должна быть частью гораздо более крупной системы. Мы любим сравнивать криптографию с дверными замками. Замок как таковой никому не нужен. Его использование обретает смысл тогда, когда он становится одной из составляющих более крупной системы, будь то дверь в здании, цепь, сейф или еще что-нибудь. Замок — это просто крохотная часть большой системы безопасности. Это же справедливо и в отношении криптографии: она представляет собой лишь маленькую часть большой системы безопасности.

Тем не менее эта часть, несмотря на свою малость, весьма и весьма критическая. Криптография — это часть системы безопасности, которая разрешает доступ к чему-либо одним людям, но не разрешает другим. Тут и начинаются первые сложности. Большинство элементов системы безопасности напоминают стены и заборы тем, что не пропускают внутрь *никого*. Криптография же выполняет роль замка: она должна отличать "хороший" доступ от "плохого". Это намного сложнее, чем просто никого не пускать. Таким образом, криптография и ее сопутствующие элементы образуют естественную мишень для атаки в любой системе безопасности.

Из сказанного выше отнюдь не следует, что криптография всегда является "слабым местом" в системе безопасности. Напротив, это случается крайне редко. В реальной жизни даже плохо реализованный криптографический метод всегда оказывается намного лучше остальных компонентов системы безопасности. Вы, вероятно, видели дверь в банковское хранилище (по крайней мере в кино) — этакую толстую тридцатисантиметровую дверь из закаленной стали с огромными болтами. Конечно же, она выглядит весьма внушительно. В цифровом мире, однако, обеспечение безопасности зачастую напоминает установку подобной двери в туристической палатке. Многие стоят возле двери и спорят, насколько толстой она должна быть, но никто не додумывается взглянуть на саму палатку. Люди любят обсуждать длину ключа в криптографических системах, а вот устранять переполнение буферов на Web-серверах нравится им куда меньше. Результат вполне предсказуем: злоумышленники добиваются переполнения буфера и не обременяют себя излишними заботами по поводу криптографии. Криптография действительно полезна только тогда, когда оставшаяся часть системы также безопасна.

Существует, однако, причина, по которой правильная реализация криптографии очень важна даже в системах, имеющих другие слабые стороны. Если

злоумышленнику удастся взломать криптографическую систему, его вряд ли обнаружат. В системе не останется следов атаки, потому что доступ к ней злоумышленника будет выглядеть как еще один "хороший" доступ. Это можно сравнить со взломом в реальной жизни. Если грабитель взломает дверь с помощью бензопилы, вы по крайней мере увидите, что ограбление произошло. Если же грабитель подберет ключ, вы можете никогда не узнать о самом факте ограбления. Большинство видов атак оставляют следы или каким-либо образом "беспокоят" систему. В отличие от них, атака на криптографическую систему может пройти быстро и незаметно, позволяя взломщику возвращаться вновь и вновь...

2.2 Правило слабого звена

Наберите следующую сентенцию самым большим шрифтом, распечатайте и приклейте над монитором:

Система безопасности надежна настолько, насколько надежно ее самое слабое звено.

Читайте эту фразу изо дня в день, пытаясь осознать все выводы, которые из нее следуют. Правило слабого звена — это и есть основная причина, затрудняющая реализацию хорошей системы безопасности.

Каждая система безопасности состоит из огромного количества составных частей. Мы должны исходить из предположения, что наш противник очень умен, а потому попытается нацелить свои действия на самое слабое место системы. Неважно, насколько сильны ее остальные части. Как и в обычной цепи, самое слабое звено всегда рвется первым. Прочность остальных звеньев уже не имеет значения.

В свое время Нильс работал в здании, где все двери запирались на ночь. Звучит вроде бы вполне безопасно, правда? Единственная проблема состояла в том, что в здании был подвесной, "фальшивый" потолок. Любой человек мог приподнять потолочные панели и пролезть над дверью или стеной. Если снять потолочные панели, весь этаж выглядел бы как нагромождение очень высоких перегородок с дверьми. И на этих дверях висели замки. Разумеется, наличие замков немного затрудняло задачу грабителя, но в то же время не позволяло охраннику проверять офисы во время ночных обходов. Вообще непонятно, к чему в данном случае привело запирание дверей: к улучшению общей безопасности здания или, напротив, к ее ухудшению. В этом примере правило слабого звена значительно снизило эффективность запирания дверей. Наличие замков могло улучшить прочность конкретного звена (двери), однако в системе все равно оставалось слабое звено (потолок). Таким обра-

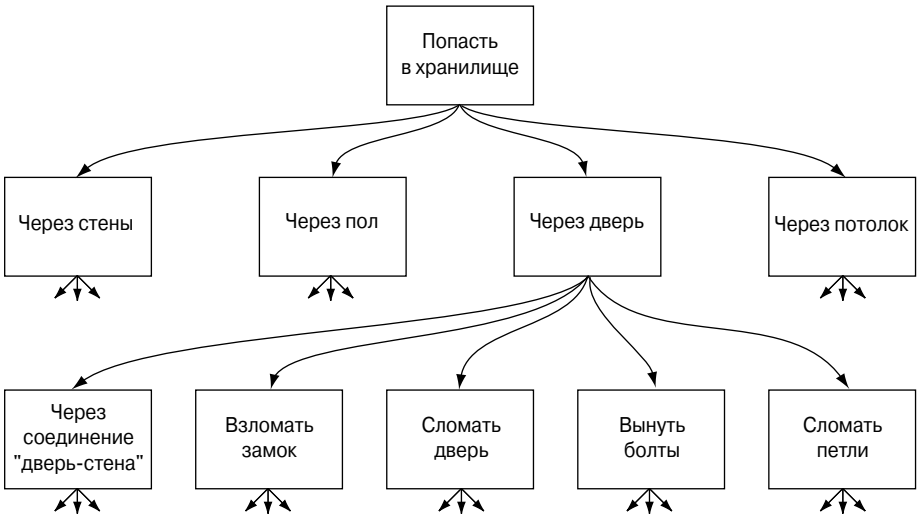

Рис. 2.1. Пример построения дерева атак для банковского хранилища

зом, общий эффект был небольшим, а его отрицательная сторона могла легко перевесить положительную.

Чтобы повысить безопасность системы, мы должны улучшить ее самое слабое звено. Вначале следует определить, из каких звеньев состоит система безопасности и какое из них самое слабое. Для этого лучше всего воспользоваться иерархической древовидной структурой. Каждая часть системы безопасности состоит из нескольких звеньев, а каждое из них, в свою очередь, состоит из вложенных звеньев. Все эти звенья можно организовать в так называемое *дерево атак (attack tree)*. В качестве примера рассмотрим дерево, представленное на рис. 2.1. Предположим, необходимо проникнуть в банковское хранилище. Звеньями первого уровня являются стены, пол, дверь и потолок. Взлом любого из этих звеньев позволит проникнуть внутрь. Теперь давайте повнимательнее приглядимся к двери. Дверная система состоит из собственных звеньев: соединения дверной рамы со стенами, замка, самой двери, болтов, которые удерживают дверь в петлях дверной рамы, и наконец петель. Это дерево можно продолжить, рассмотрев возможности взлома замка, одна из которых подразумевает получение ключа. Последнее, в свою очередь, дает начало дереву способов кражи ключа и т.п.

Каждое звено можно анализировать и разбивать на более мелкие звенья до тех пор, пока не будут получены элементарные компоненты. Выполнение этой процедуры для реальных систем отнимает массу времени и усилий, однако обеспечивает весьма ценное понимание возможных путей атак. Попытка обезопасить систему без проведения подобного анализа зачастую оканчивает-

ся неудачей. В этой книге, однако, рассматривается работа с ограниченным числом компонентов, т.е. только с теми, которые могут быть реализованы с помощью криптографии, поэтому дерево атак будет столь простым, что не потребуется описывать его явно.

Существует много способов влияния правила слабого звена на нашу работу. Например, принято предполагать, что пользователи системы имеют надежные пароли, однако на практике это не так. Обычно пользователи выбирают простые и короткие пароли. Если же пароли будут назначены "свыше", пользователи сделают все, чтобы поменьше утруждать свою память. Один из наиболее распространенных способов — написать пароль на бумажке и приклеить ее на монитор. Подобное поведение всегда влияет на конечный результат, поэтому его нельзя не учитывать. Если вы разработаете систему, которая каждую неделю будет снабжать пользователя новым 12-значным паролем, можете быть уверены — он обязательно напишет пароль на бумажке и приклеит ее к монитору. Это существенно ослабит и без того слабое звено и плохо скажется на общей безопасности системы.

С формальной точки зрения, укрепление любого звена кроме самого слабого — пустая трата времени. На практике же все обстоит далеко не так просто. Злоумышленник может не знать, какое звено является самым слабым, и нападет на более сильное звено. Самое слабое звено может различаться для разных типов злоумышленников. Прочность любого звена зависит от навыков злоумышленника и имеющихся у него средств. Поэтому самое слабое звено во многом определяется той или иной ситуацией. Необходимо укрепить любое звено, которое в определенной ситуации может оказаться самым слабым.

2.3 Противоборствующее окружение

Одно из основных различий между проектированием систем безопасности и практически всеми остальными типами проектирования состоит в наличии такого фактора, как *противоборствующее окружение (adversarial setting)*. Большинству инженеров приходится сталкиваться с такими проблемами, как ураганы, жара и изнашивание. Все эти факторы влияют на проектирование систем, однако для опытного инженера это влияние предсказуемо. В системах безопасности все совершенно по-другому. Наши противники умны, сообразительны, коварны и изворотливы; они делают то, что нам и не снилось. Они не играют по правилам, и их поведение совершенно непредсказуемо. В таком окружении гораздо сложнее работать.

Возможно, вы видели фильм, в котором подвесной мост Такома–Нэрроуз раскачивался и гнулся на сильном ветру, пока в конце концов не сломался и не упал в реку. Этот фрагмент фильма, основанный на реальных событи-

ях, стал знаменит во всем мире, а разрушение моста преподало инженерам ценный урок. При наличии сильного ветра легкие подвесные мосты могут войти в резонанс, в результате чего они начнут колебаться и в конце концов сломаются. Как уберечь от подобного разрушения новые мосты? Для сопротивления колебаниям можно было бы существенно укрепить весь мост, однако это обошлось бы слишком дорого. Наиболее распространенным приемом является изменение аэродинамики моста. Мостовое полотно делается толще, в результате чего ветру становится труднее прогибать его вверх-вниз. Иногда для гашения колебаний используются перила, благодаря которым мостовое полотно перестает напоминать крыло самолета, поднимаемое ветром. Данный метод срабатывает, поскольку поведение ветра достаточно предсказуемо и не изменяется для того, чтобы разрушить мост.

Проектировщики систем безопасности должны отталкиваться от того, что "ветер" коварен. Что, если бы он начал дуть не из стороны в сторону, а вверх-вниз и притом постоянно изменял свое направление с такой частотой, чтобы мост вошел в резонанс? Проектировщики мостов сочтут подобные разговоры нелепыми: "Ветер никогда *так* не дует". В этом плане проектировщикам мостов, конечно же, намного легче. Криптографы лишены и этого. Системы безопасности подвергаются нападению умных и коварных злоумышленников, поэтому необходимо уметь защищаться от всех типов атак.

Работать в противоборствующем окружении очень трудно. Это игра без правил, в которой карты всегда ложатся не так, как того хотелось бы. Ведь речь идет о некоем абстрактном "злоумышленнике"; не известно, кем он является в действительности, какими знаниями и ресурсами обладает, какова его цель и когда он собирается напасть. Поскольку атака нередко случается через много лет после создания системы, злоумышленник обладает преимуществом в пять-десять лет исследований и может воспользоваться новыми технологиями, которых не существовало на момент проектирования системы. И что гораздо печальнее, злоумышленник, имея все эти преимущества, может найти всего лишь одно слабое звено в нашей системе, в то время как мы должны защищать ее всю. Итак, наша цель состоит в том, чтобы построить систему, способную выдержать весь этот чудовищный натиск. Добро пожаловать в замечательный мир криптографии!

2.4 Практическая паранойя

Чтобы заниматься проектированием безопасных систем, необходимо самому стать хитрым и изворотливым. Найти слабые места в собственной работе сможет только тот, кто сам начнет думать как злоумышленник. Разумеется, это повлияет и на остальные аспекты вашей жизни. Каждый, кто работал

с криптографией на практике, испытал это чувство. Начав думать о том, как нападать на системы, вы будете применять это ко всему, что вас окружает. Вашу голову заполонят кошмарные мысли о том, как можно перехитрить людей и как они могут перехитрить вас. Криптографы — это профессиональные параноики. Через некоторое время вы либо научитесь отделять профессиональную паранойю от реальной жизни, либо просто сойдете с ума. К счастью, большинству из нас удалось сохранить хоть какие-то остатки здравомыслия... как нам кажется[1].

Паранойя очень полезна при работе в области практической криптографии. Представьте себе, что вы разрабатываете систему электронных платежей. В работу системы вовлечено несколько участников: покупатель, продавец, банк покупателя и банк продавца. Определить потенциальные источники угрозы очень трудно, поэтому мы воспользуемся параноидальной моделью. Для каждого участника будем предполагать, что остальные участники процесса объединились с целью его обмануть. Если криптографическая система успешно пройдет испытание параноидальной моделью, она будет иметь неплохой шанс выжить в реальном мире.

2.4.1 Критика

Профессиональная паранойя — это неотъемлемая составляющая коммерции. Увидев на рынке новую систему, мы первым делом думаем о том, как бы ее взломать. Чем быстрее вы отыщете в системе слабое место, тем больше вы о ней узнаете. Нет ничего хуже, чем работать над системой долгие годы только затем, чтобы однажды услышать от случайного знакомого: "А что, если я нападу вот так?.." Наверное, никому бы не хотелось испытать подобный конфуз.

Работая в области криптографии, необходимо четко различать критику работы и критику самого человека. Любая работа — это всего лишь спорт, в котором есть свои правила. Если кто-нибудь предлагает что-нибудь, он автоматически приглашает общественность покритиковать свое изобретение. Если вы взломаете одну из наших систем, мы громко поаплодируем и расскажем об этом всему миру[2]. Мы постоянно ищем слабые места везде и во всем, потому что это единственный способ научиться создавать безопасные системы. Запомните еще один урок: критика системы — это не критика вас лично. Без этого правила вы не выживете в мире криптографии, поэтому привыкайте к критике. Точно так же, отыскивая в чужой системе слабое ме-

[1] Но помните: тот факт, что **вы** не параноик, еще не означает, что вы не попадете в сети, расставленные другим параноиком.

[2] В зависимости от ситуации, мы можем поругать себя за то, что сами не обнаружили слабое место в своей системе, но это уже совсем другая история.

сто, критикуйте систему, а не ее разработчиков. В криптографии нападки на личности будут восприняты так же негативно, как и везде.

Кажущийся враждебный настрой криптографов то и дело приводит к недоразумениям. Люди, далекие от криптографии, часто воспринимают критику работы, выполненной человеком, как критику самого человека и затем жалуются на наши дурные манеры. В 1999 году на одной конференции рассматривались блочные шифры, использующие алгоритм AES. Создатель шифра Magenta не смог приехать на конференцию и прислал вместо себя своего студента. Между тем у шифра сразу же обнаружился серьезный недостаток, и через каких-нибудь пять минут после начала доклада Magenta был, что называется, разгромлен в пух и прах сразу несколькими слушателями. Это было очень интересно. Нескольким лучшим криптографам мира представили новый блочный шифр, и они раскритиковали его на глазах у всей остальной публики. Это обычный порядок вещей. Еще один слушатель, до этого практически не имевший дела с криптографией, написал очерк о своем посещении конференции. Ему показалось, что все нападки были направлены в адрес самого студента, и он был поражен невероятно грубым поведением аудитории. Реакция этого человека вполне понятна. Во многих кругах нашего общества критика идеи или предложения автоматически означает критику самого автора. В криптографии, однако, мы просто не можем себе этого позволить.

Еще хуже, когда под прицелом критиков оказываются разработчики, которые случайно забрели в область криптографии. Такие люди воспринимают критику своей работы как нападки на них лично со всеми вытекающими отсюда последствиями. Иногда приходится вести себя тактично, однако это мешает донести до собеседника свою мысль. Если мы скажем что-то наподобие: “Здесь могут быть небольшие недочеты в системе безопасности”, то непременно получим ответ: “О, не волнуйтесь, мы это исправим”, даже если проблема заключается в самой структуре системы. Практика показывает, что донести до оппонента можно только конкретизированную мысль: “Если мы сделаем то-то и то-то, безопасность будет нарушена”. С другой стороны, сказав это разработчику системы, вы обрекаете себя на неприятности. К сожалению, гармоничного решения этой проблемы не существует.

Итак, когда в следующий раз кто-нибудь будет критиковать вашу работу, постарайтесь не воспринимать это как личное оскорбление. А нападая на чужую систему, критикуйте ее саму, а не тех, кто за ней стоит.

2.5 Модель угроз

Каждая система может подвергнуться нападению. Смысл любой системы безопасности состоит в том, чтобы разрешить доступ к чему-либо одним людям и не разрешить другим. В конце концов, человек всегда должен кому-нибудь доверять, хотя этот кто-то, в свою очередь, может на него напасть.

Очень важно знать, от чего вы пытаетесь защититься. Решить этот на первый взгляд простой вопрос намного сложнее, чем кажется. В чем именно состоит угроза? Большинство компаний защищают свои локальные сети брандмауэрами, в то время как наиболее разрушительные атаки обычно исходят изнутри. Тут брандмауэр бесполезен. Каким бы хорошим он ни был, он не защитит вас от коварного сотрудника. Налицо неудачное построение модели угроз.

В качестве еще одного примера можно привести протокол защищенных электронных транзакций (Secure Electronic Transaction — SET), который применяется для защиты электронных платежей, выполняемых в Internet с помощью кредитных карт. Одна из особенностей протокола SET заключается в шифровании номера кредитной карты, чтобы злоумышленник, перехвативший номер, не мог его скопировать. Это хорошая идея. Вторая же особенность, состоящая в том, что даже продавец не видит номера кредитной карты покупателя, гораздо менее удачна.

Некоторые продавцы используют номера кредитных карт для поиска сведений о покупателе или для взимания с него дополнительных сборов. Целые системы электронной коммерции были основаны на предположении о том, что у продавца есть доступ к номеру кредитной карты покупателя. Разумеется, запретить этот доступ нереально. Когда Нильс работал с протоколом SET несколько лет назад, последний предоставлял возможность отправлять в зашифрованном виде номер кредитной карты дважды банку и продавцу, чтобы у продавца тоже был номер кредитной карты покупателя. (Спецификации протокола SET настолько сложны и запутанны, что мы решили не утруждать себя проверкой того, существует ли данная возможность и сегодня.)

Даже несмотря на наличие такой возможности, протокол SET был направлен на решение совершенно не той проблемы. В большинстве случаев номера кредитных карт воруют вовсе не во время их передачи от покупателя продавцу. Их крадут из базы данных продавца. SET же защищает информацию только во время ее передачи.

Протокол SET обладает еще одним, гораздо более серьезным изъяном. Недавно один голландский банк, услугами которого пользуется Нильс, предложил своим клиентам пластиковые карты с поддержкой SET. В качестве одного из главных аргументов в пользу приобретения такой карты банк называла улучшенную безопасность при совершении покупок через Internet.

Однако данный аргумент оказался не более чем фальшивкой. Покупать товары через Internet с помощью обычной электронной карты вполне безопасно. Номер кредитной карты не является каким-либо секретом: вы сообщаете его каждому продавцу, у которого что-нибудь покупаете. Настоящим гарантом покупки является ваша подпись, так как именно она авторизует транзакцию. Если у продавца украдут номер вашей кредитной карты, вам могут приписать несуществующие покупки, однако, поскольку эти транзакции не будут подтверждены вашей собственноручной подписью (или PIN-кодом), законного основания для снятия денег с вашего счета не будет. В большинстве подобных случаев достаточно просто обратиться с жалобой, чтобы получить свои деньги обратно. Разумеется, процедура получения новой кредитной карты с другим номером может доставить некоторые неудобства, однако на этом все и заканчивается. С протоколом SET все обстоит по-другому. Для авторизации транзакций SET применяет цифровую подпись пользователя (более подробно это рассматривается в главе 13, "Алгоритм RSA"). На первый взгляд это более безопасно, чем использование только номера кредитной карты. Но взгляните на данную ситуацию с другой стороны. Теперь вся ответственность по выполнению транзакций с компьютера пользователя ложится на самого пользователя. А что, если его компьютер подвергнется нападению вируса, который взломает программное обеспечение SET? Такое программное обеспечение может подписать не ту транзакцию, и пользователь потеряет деньги.

Итак, с точки зрения пользователя, протокол SET обеспечивает безопасность *хуже*, чем обычная кредитная карта. Используя обычную кредитную карту, пользователь всегда сможет вернуть несправедливо снятые с его счета деньги. Применение протокола SET делает пользователя более уязвимым. Таким образом, хотя SET повышает безопасность системы электронных платежей в целом, он переносит остаточный риск с продавца и/или банка на пользователя. Модель угроз пользователя, которую можно было описать следующим образом: "я потеряю деньги только тогда, когда кто-нибудь слишком хорошо подделает мою подпись", изменится и будет выглядеть так: "я потеряю деньги тогда, когда кто-нибудь слишком хорошо подделает мою подпись или мой компьютер подвергнется нападению вируса".

Правильное построение модели угроз очень важно. Прежде чем приступать к разработке криптографической системы безопасности, хорошенько обдумайте, откуда могут исходить потенциальные угрозы. Ошибка в анализе угроз может свести на нет смысл всего проекта. В этой книге рассматривается очень ограниченная область криптографии, поэтому модели угроз не уделяется должного внимания, однако при построении реальной системы не забудьте провести анализ угроз для каждого из ее участников.

2.6 Криптография — это не решение

Разумеется, криптография не решит проблем безопасности. Она может стать как частью решения, так и частью самой проблемы. Во многих ситуациях появление криптографии только усугубляет проблему, а улучшения от ее применения весьма и весьма сомнительны.

Как защитить автомобиль от угонщика? Самый простой способ — непосредственно защитить машину физически. Просто не подпускайте к ней ни одного прохожего. Это не слишком удобно, поэтому мы предпочитаем установить на машине замок и носить с собой ключ. Поскольку машина закрыта, ее можно оставить на улице. Теперь у злоумышленника есть два способа проникнуть в машину. Он может сломать систему блокировки дверей (в том числе разбить окно) или же украсть ключи от машины, которые необходимо защищать физически. Как видите, теперь у нас два объекта нападения: система блокировки дверей и ключи. Появление ключей помогло угонщику, создав новый объект нападения. Основное преимущество ключей состоит в том, что защитить их физически гораздо легче, чем машину. Поскольку данное преимущество перевешивает дополнительный риск, связанный с возможностью поломки замка, применение ключей повышает безопасность машины.

В цифровом мире все происходит несколько иначе. Предположим, у вас на компьютере есть секретный файл, который никто не должен прочитать. Вы можете просто защитить файловую систему от неавторизованного доступа или же зашифровать файл и защитить ключ. Зашифрованный файл больше не представляет собой какого-либо секрета, поэтому вы, как обычный человек, скорее всего, не будете охранять этот файл слишком уж тщательно (в этом он сходен с машиной, оставленной на улице). Но где же хранить ключ? Хорошие ключи слишком сложны для запоминания. Некоторые программы сохраняют ключ на диске — в том самом месте, где хранился зашифрованный файл. Теперь у злоумышленника появляется два способа добраться к содержимому файла. Любая атака, которая в первом случае могла привести к похищению секретного файла, теперь с таким же успехом может привести к похищению ключа, который, в свою очередь, поможет расшифровать файл. Любая атака, которая могла сработать в первом случае, сработает и во втором, однако теперь у нас появился новый объект нападения: сама система шифрования. Очевидно, безопасность системы в целом снизилась. Таким образом, шифрование файла — это не полноценное решение. Оно может быть частью другого решения, однако само по себе больше проблем создает, нежели решает.

Криптография имеет множество областей применения. Она является неотъемлемой частью многих хороших систем безопасности. Неправильное применение криптографии, напротив, существенно ослабляет безопасность.

Во многих случаях криптография обеспечивает не реальную безопасность, а только видимость безопасности. Зачастую, впрочем, это все, что требуется клиентам. Они хотят *чувствовать* себя в безопасности, но вовсе не хотят ввязываться во все тонкости, которые свойственны реальным механизмам безопасности. В подобных ситуациях (а они случаются слишком часто) криптография сама по себе может стать решением проблемы, однако с таким же успехом, как и какой-нибудь амулет, который носят на шее или в кармане.

2.7 Криптография очень сложна

Криптография ужасно сложна. Даже опытные эксперты разрабатывают системы, которые через несколько лет оказываются безжалостно взломанными. Это случается настолько часто, что уже никого не удивляет. Правило слабого звена и противоборствующее окружение существенно усложняют и без того нелегкую жизнь криптографов.

Еще одной важной проблемой является недостаточный объем тестирования. Никто не знает, как убедиться в том, что система действительно безопасна. Лучшее, что можно сделать, — это опубликовать систему, чтобы на нее взглянули другие эксперты. Обратите внимание, что вторая часть этого метода вовсе не выполняется автоматически: существует масса опубликованных систем, на которые никто так и не обратил внимания. Даже конференции и научные статьи удостаиваются лишь поверхностных обзоров. Эти обзоры направлены на то, чтобы отличить хорошие статьи от плохих, а не на проверку безопасности систем или правильности результатов. Разумеется, обозреватели все же проводят некоторую проверку, однако на более тщательный анализ статьи у них просто не остается времени. (В конце концов, обозревателям редко платят за их работу, и, когда в свободное время им приходится анализировать по 20-30 статей, они, скорее всего, не будут делать это слишком тщательно.)

Существует несколько небольших областей криптографии, в которых мы разбираемся довольно хорошо. Это вовсе не означает, что они просты; это лишь говорит о том, что мы работаем над ними уже несколько десятилетий и — смеем надеяться — знаем их проблематику. Данная книга в основном посвящена этим областям криптографии. Здесь мы попытались собрать и упорядочить все то, что знаем о разработке и построении практических криптографических систем.

К сожалению, многие, почти ничего не зная о криптографии, почему-то думают, что она очень проста. Время от времени нам попадается очередной электротехник или программист, который прочитал половину книги о криптографии — в большинстве случаев это первая книга Брюса *Applied Cryptog-*

raphy [86] — и решил разработать собственную систему. На нашей памяти это *никогда* не приводило к хорошим результатам, по крайней мере за последнее десятилетие. Представьте себе студента архитектурного института, который после года обучения решил спроектировать мост качественно новой конструкции через Берингов пролив. Стали бы мы строить и использовать этот мост без более глубокого изучения предложенной модели? Конечно, нет. Между тем, когда дело касается криптографии, люди почему-то охотно платят за реализацию систем, разработанных новичками. А поскольку плохая криптографическая система ничем не отличается от хорошей криптографической системы до тех пор, пока на нее не нападут, некоторые клиенты впадают в заблуждение и покупают подобные продукты.

Не верьте распространенному мифу о том, что криптография проста. Это не так.

2.8 Криптография — это самая простая часть

Тем не менее, несмотря на сложность самой криптографии, она остается одной из наиболее простых частей системы безопасности. Криптографический компонент, как и дверной замок, имеет довольно четкие границы и требования. Дать четкое определение системе безопасности в целом намного сложнее, поскольку она включает в себя слишком много аспектов. Вопросы наподобие организационных процедур, применяемых для предоставления доступа, и процедур, которые применяются для проверки соответствия другим процедурам, весьма сложны в решении, поскольку связаны с постоянно меняющейся ситуацией. Еще одной серьезной проблемой компьютерной безопасности является отвратительное качество практически всего программного обеспечения. О какой безопасности может идти речь, если программы, установленные на компьютере пользователя, содержат тысячи дефектов, которые приводят к появлению “дыр” в системе безопасности!

Криптография — одна из наиболее простых частей системы безопасности, поскольку на свете существуют люди, которые в ней разбираются. При необходимости вы можете нанять экспертов, которые разработают для вас криптографическую систему. Разумеется, их услуги недешевы, а работать с ними крайне сложно. Они то и дело будут настаивать на изменении других частей системы для достижения желаемого уровня безопасности. Тем не менее, каково бы ни было практическое применение криптографии, оно затрагивает проблемы, которые мы умеем решать.

Реализация остальных частей системы безопасности касается проблем, которые мы не умеем решать. Управление ключами и хранение ключей являются основополагающими аспектами любой криптографической системы, одна-

ко у большинства компьютеров нет безопасного места для хранения ключей. Плохое качество программного обеспечения вообще стало притчей во языцех, не говоря уже о защите сетей. Ну а если к этому добавить еще и несознательных пользователей, проблема обеспечения безопасности становится практически нерешаемой.

2.9 Рекомендуемая литература

Все, кто интересуются криптографией, должны прочитать книгу Дэвида Кана (David Kahn) *The Codebreakers* [45]. В ней содержится история криптографии с древнейших времен до XX века. Помимо всего прочего, автор описывает множество проблем, с которыми сталкивались и продолжают сталкиваться разработчики криптографических систем во всем мире.

Книга, которую вы сейчас держите в руках, в некотором смысле является продолжением первой книги Брюса Шнайера *Applied Cryptography* [86], которая охватывает намного больший диапазон тем и содержит спецификации всех рассматриваемых в ней алгоритмов. Тем не менее в ней отсутствуют детали разработки реальных систем, которые обсуждаются в книге *Практическая криптография*.

По части цифр и фактов ничто не может сравниться с книгой Менезеса (Menezes), ван Ооршота (van Oorschot) и Вэнстоуна (Vanstone) *Handbook of Applied Cryptography* [64]. Этот невероятно полезный справочник — настоящая энциклопедия криптографии, но, как и все энциклопедии, едва ли подходит для глубокого изучения темы.

Предыдущая книга Брюса *Secrets and Lies* [88] содержит хорошее объяснение общих проблем компьютерной безопасности, а также роли криптографии в ее обеспечении. И разумеется, лучшим пособием по проектированию систем безопасности является книга Росса Андерсона (Ross Anderson) *Security Engineering* [1]. Обе книги весьма важны для понимания криптографии в контексте окружающего мира.

Глава 3

Введение в криптографию

В этой главе рассматриваются основные понятия криптографии, а также предоставлена дополнительная информация, которая понадобится вам при изучении последующих глав книги.

3.1 Шифрование

Шифрование — это и есть первоначальное назначение криптографии. На рис. 3.1 изображена общая схема обмена данными, требующая применения шифрования. Пользователи А и Б хотят общаться друг с другом. (В англоязычной литературе по криптографии для обозначения пользователей и злоумышленников принято применять личные имена, в частности Alice, Bob и Eve.) К сожалению, в общем случае канал общения не является безопасным. Он прослушивается злоумышленником Е. Каждое сообщение m, которое пользователь А отсылает пользователю Б, получает и злоумышленник Е. (Это же справедливо и для сообщений, которые пользователь Б отсылает пользователю А, за исключением того, что пользователи А и Б меняются местами. Если удастся защитить сообщения пользователя А, то это же решение можно применить и для защиты сообщений пользователя Б, поэтому в дальнейшем будем говорить только о сообщениях, отсылаемых пользователем А пользователю Б.) Что же нужно сделать, чтобы злоумышленник Е не смог читать сообщения?

Чтобы злоумышленник Е не смог понять сообщения, которыми обмениваются пользователи А и Б, применяется схема шифрования, представленная на рис. 3.2. Вначале пользователи А и Б договариваются о применении секретного ключа K_e. Для этого им следует воспользоваться каналом общения, который не может прослушиваться злоумышленником Е (например,

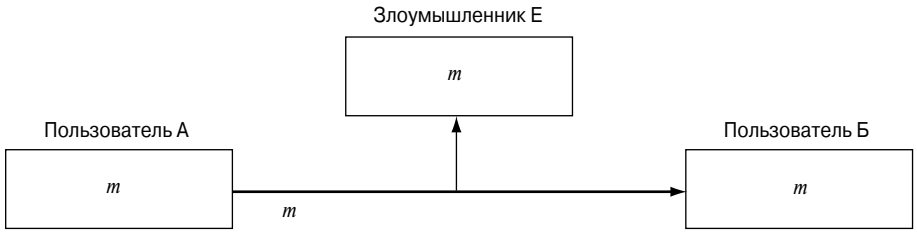

Рис. 3.1. Как обеспечить безопасность общения пользователей A и Б?

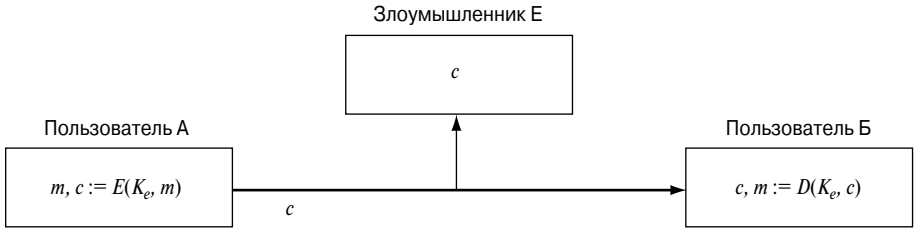

Рис. 3.2. Общая схема шифрования

пользователь A может отослать копию ключа пользователю Б по почте или поступить другим подобным образом).

Когда пользователь A хочет отослать сообщение m, он вначале зашифровывает его с помощью функции шифрования. Обозначим функцию шифрования как $E(K_e, m)$, а результат применения этой функции назовем *шифрованным текстом (ciphertext)* c. (Исходное сообщение m называется *открытым текстом (plaintext).*) Вместо того чтобы посылать пользователю Б сообщение m, пользователь A посылает ему шифрованный текст $c := E(K_e, m)$. Когда пользователь Б получает сообщение c, он может расшифровать его с помощью функции дешифрования $D(K_e, c)$ и получить исходный открытый текст m, который ему хотел отослать пользователь A.

Злоумышленник E не знает ключ K_e, поэтому если он перехватит шифрованный текст c, то не сможет его расшифровать. Хорошая функция шифрования делает невозможным определение открытого текста m по шифрованному тексту c без ключа K_e.

Описанная схема шифрования может применяться не только для обмена сообщениями электронной почты (наиболее очевидная область применения), но и для хранения данных. Хранение информации можно рассматривать как процесс передачи сообщения не в пространстве, а во времени. В этом случае пользователи A и Б зачастую являются одним и тем же лицом, однако сама схема шифрования остается неизменной.

3.1.1 Принцип Кирхгофа

Для расшифровки шифрованного текста пользователю Б нужны две вещи: алгоритм дешифрования D и ключ K_e. Знаменитый принцип Кирхгофа формулируется следующим образом: надежность схемы шифрования должна зависеть только от секретности ключа K_e и не зависеть от секретности алгоритмов шифрования и дешифрования.

Для появления данного правила были весьма серьезные основания. Изменять алгоритмы сложно. Они встраиваются в аппаратное или программное обеспечение, которое может с трудом поддаваться обновлению. На практике один и тот же алгоритм используется довольно долгое время. Это всего лишь суровая правда жизни. Кроме того, даже простой ключ трудно сохранить в секрете. Обеспечить же секретность целого алгоритма намного сложнее (и потому намного дороже). Никто не станет разрабатывать криптографическую систему только для двух пользователей. Каждый участник системы (а таких могут быть миллионы) использует один и тот же алгоритм. Злоумышленнику Е достаточно всего лишь получить алгоритм от одного из них, а найти незащищенного пользователя среди миллионов не так уж трудно. В конце концов, можно просто украсть ноутбук с установленным на нем алгоритмом. А помните нашу параноидальную модель? Злоумышленник Е может оказаться одним из остальных пользователей системы или даже одним из ее разработчиков.

Существует еще несколько аргументов в пользу публикации алгоритмов шифрования. Опыт подсказывает, что достаточно сделать маленькую ошибку — и созданный криптографический алгоритм окажется совершенно никудышным. Если алгоритм не является открытым, никто не обнаружит эту ошибку, пока на систему не нападет злоумышленник. Последний сможет использовать найденный недостаток для взлома системы. Как консультантам, нам не раз доводилось анализировать секретные алгоритмы шифрования, и *все* они были слабыми. Вот почему в наших кругах распространено здоровое недоверие ко всем закрытым, конфиденциальным или другим в той или иной мере секретным алгоритмам. Не поддавайтесь старому заблуждению: “Если мы сохраним в секрете и алгоритм, то только увеличим безопасность”. Это неправда. Потенциальное повышение безопасности, вызванное сохранением алгоритма в секрете, невелико, а вот потенциальное снижение — огромно. Большинство секретных алгоритмов настолько плохи, что любые громкие утверждения наподобие приведенного выше граничат с демонстрацией собственной некомпетентности.

3.2 Аутентификация

Схема общения пользователей А и Б (см. рис. 3.1) имеет еще одно слабое место. Злоумышленник Е может не просто прочитать сообщение. Он может его изменить. Для этого ему понадобится иметь немного больше контроля над каналом общения, но это вполне осуществимо. Например, на рис. 3.3 показано, что пользователь А пытается послать сообщение m, однако его перехватывает злоумышленник Е. Вместо того чтобы получить сообщение m, пользователь Б получает сообщение m'. Также предполагается, что злоумышленник Е прочитал сообщение, которое отправил пользователь А. Помимо этого, злоумышленник может удалить сообщение, чтобы пользователь Б вообще его не получил, вставить вместо исходного новое сообщение с другим текстом, записать сообщение и отправить его пользователю Б позднее или же изменить порядок отправки сообщений.

Предположим, пользователь Б только что получил сообщение. Почему он должен поверить, что оно было действительно отправлено пользователем А? На это нет никаких оснований. А если пользователь Б не знает, кто в точности отправил сообщение, оно становится довольно-таки бесполезным.

Для решения этой проблемы применяется аутентификация. Как и в схеме шифрования, в аутентификации используется секретный ключ, который знают только пользователи А и Б. Чтобы отличать этот ключ от ключа шифрования K_e, назовем его ключом аутентификации и будем обозначать как K_a. Процесс аутентификации сообщения m представлен на рис. 3.4. Когда пользователь А посылает сообщение m, он подсчитывает так называемый *код аутентичности сообщения* (*Message Authentication Code — MAC*). Значение MAC a вычисляется по формуле $a := h(K_a, m)$, где h — функция вычисления MAC, а K_a — ключ аутентификации. Теперь пользователь А отправляет пользователю Б сообщение m и значение a. Когда пользователь Б получает m и a, он вычисляет, каким должно быть значение a при заданном ключе K_a, и проверяет, таково ли полученное им значение a.

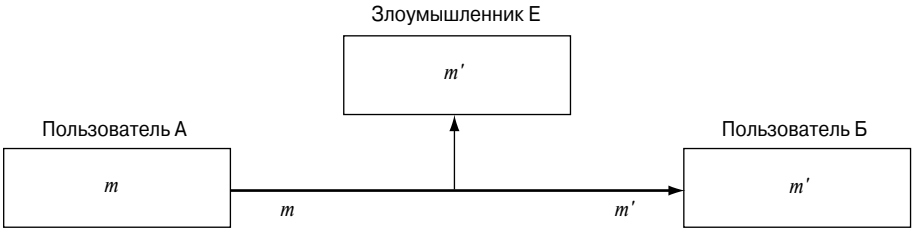

Рис. 3.3. Пользователь Б не знает, кто в действительности послал сообщение

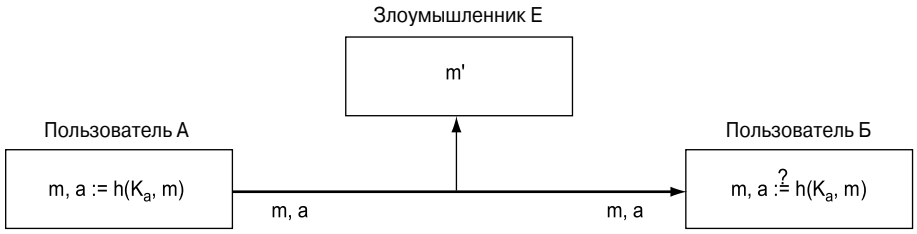

Рис. 3.4. Общая схема аутентификации

Пусть злоумышленник Е хочет изменить сообщение m, чтобы получить сообщение m'. Если злоумышленник просто заменит сообщение m на m', пользователь Б подсчитает значение функции $h(K_a, m')$ и сравнит его со значением a. Хорошая функция вычисления MAC никогда не дает один и тот же результат для двух разных сообщений, поэтому пользователь Б увидит, что сообщение тем или иным образом было искажено, и просто его отбросит.

Если предположить, что злоумышленник Е не знает ключа аутентификации K_a, у него остается единственный способ получить сообщение и правильное значение MAC — прослушивать систему пользователя А, когда тот отсылает сообщения пользователю Б. Тем не менее даже в этом случае злоумышленник Е способен нанести некоторый ущерб. Он может записывать сообщения и их значения MAC, чтобы впоследствии воспроизвести их, отправив пользователю Б когда-либо позднее.

Сама по себе аутентификация лишь частично решает проблему. Злоумышленник Е все еще может удалять сообщения, которые посылает пользователь А, а также воспроизводить старые сообщения или изменять порядок сообщений. По этой причине аутентификацию практически всегда сочетают со схемой нумерации сообщений, при которой каждому сообщению присваивается порядковый номер. Если сообщение m будет содержать порядковый номер, злоумышленнику Е не удастся обмануть пользователя Б, отослав ему старое сообщение. Пользователь Б сразу же увидит, что сообщение содержит правильное значение MAC, однако его порядковый номер соответствует номерам старых сообщений, и отбросит полученное сообщение.

Аутентификация в сочетании с нумерацией сообщений решает бóльшую часть проблемы. Злоумышленник Е все еще может помешать общению пользователей А и Б или задержать доставку сообщений, удаляя их и пересылая пользователю Б позднее. Однако этим круг возможностей злоумышленника ограничивается.

Для большей наглядности представьте себе случай, когда пользователь А отсылает последовательность сообщений $m_1, m_2, m_3, \ldots$. Пользователь Б принимает только те сообщения, которые имеют правильное значение MAC и по-

рядковый номер которых строго больше[1] порядкового номера последнего принятого сообщения. Таким образом пользователь Б получает последовательность сообщений, которая является *подпоследовательностью* последовательности сообщений, отправленных пользователем А. Подпоследовательность — это та же последовательность, из которой были удалены нуль или более сообщений.

Описанная схема аутентификации и нумерации сообщений — это все, чем криптография может помочь в данной ситуации. Пользователь Б будет получать подпоследовательность сообщений, отправленных пользователем А, однако злоумышленник Е не сможет нанести трафику сообщений никакого ущерба, кроме удаления некоторых или всех сообщений. Чтобы избежать потери информации, пользователи А и Б могут применить схему повторной отправки утерянных сообщений, однако это зависит от особенностей конкретного приложения, применяемого для отправки сообщений, и не относится к криптографии.

Разумеется, во многих случаях пользователям А и Б придется применять и шифрование и аутентификацию. Более подробно сочетание шифрования и аутентификации рассматривается несколько позднее, а пока хотим вас предупредить: никогда не путайте эти понятия! Шифрование сообщения не мешает злоумышленнику манипулировать его содержимым, а аутентификация сообщения не обеспечивает его секретности. Одна из классических ошибок криптографии — думать, что шифрование сообщения помешает злоумышленнику изменить его. Это не так.

3.3 Шифрование с открытым ключом

Чтобы использовать схему шифрования, описанную в разделе 3.1, пользователи А и Б должны обладать общим ключом K_e. А как договориться о применении этого ключа? Пользователь А не может отослать ключ пользователю Б по каналу общения, потому что ключ тут же попадет в руки злоумышленника Е. В действительности проблема распространения ключей и управления ими является одной из наиболее сложных проблем криптографии, для которой существуют лишь частичные решения.

Пользователи А и Б могут обменяться ключом, встретившись где-нибудь за чашечкой кофе. Однако, если они являются частью группы из 20 друзей, которые хотят общаться между собой, каждому члену группы понадобится обменяться 19 ключами с остальными ее членами. Всего для такой группы понадобится 190 ключей. Это слишком сложно, а с возрастанием количества людей в группе сложность обмена ключами еще более возрастет.

[1] "Строго больше" означает "больше и не равно".

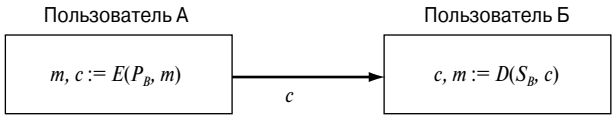

Рис. 3.5. Общая схема шифрования с открытым ключом

Проблема распространения криптографических ключей стара как мир. Одной из важных попыток ее решения является применение криптографии с открытым ключом. Рассмотрим схему шифрования с открытым ключом (рис. 3.5). На схеме не показан злоумышленник Е; в дальнейшем будем просто предполагать, что сообщения, которыми обмениваются пользователи А и Б, всегда доступны злоумышленнику Е. Во всем остальном эта схема весьма напоминает схему, представленную на рис. 3.2. Разница состоит лишь в том, что пользователи А и Б больше не применяют один и тот же ключ шифрования. Они используют *разные* ключи. В этом и состоит революционная идея криптографии с открытым ключом.

Вначале пользователь Б с помощью специального алгоритма генерирует пару ключей (S_B, P_B). Это секретный ключ S_B и открытый ключ P_B. Затем пользователь Б делает удивительную вещь: он публикует ключ P_B. (Иначе почему бы этот ключ назывался открытым?)

Когда пользователь А хочет послать сообщение пользователю Б, он находит открытый ключ P_B, опубликованный пользователем Б. Пользователь А зашифровывает сообщение m с помощью открытого ключа P_B и посылает пользователю Б полученный шифрованный текст c. Пользователь Б применяет к полученному тексту свой секретный ключ S_B и алгоритм дешифрования, в результате он получает исходное сообщение m.

Чтобы этот метод действительно работал, алгоритмы генерации пары ключей, шифрования и дешифрования должны гарантировать, что в процессе расшифровки всегда будет получено исходное сообщение. Другими словами, равенство $D(S_B, E(P_B, m)) = m$ должно выполняться для всех возможных значений m. Мы еще вернемся к этому вопросу немного позднее.

Пользователи А и Б применяют не только разные ключи, но и совершенно разные алгоритмы шифрования и дешифрования. Все схемы шифрования с открытым ключом имеют серьезную математическую основу. Одно из очевидных требований к подобным схемам состоит в том, чтобы на основе открытого ключа нельзя было вычислить соответствующий секретный ключ.

Этот тип шифрования называется *асимметричным шифрованием (asymmetric-key encryption)* или *шифрованием с открытым ключом (public-key encryption)* в противоположность *симметричному шифрованию (symmetric-key*

encryption) или *шифрованию с секретным ключом (secret-key encryption)*, о котором шла речь ранее.

Применение криптографии с открытым ключом значительно облегчает проблему распространения ключей. Теперь пользователю Б достаточно распространить единственный открытый ключ, который могут применять все желающие. Пользователь А точно так же публикует свой открытый ключ, после чего пользователи А и Б смогут безопасно обмениваться сообщениями. Даже при наличии большой группы собеседников каждому члену группы необходимо опубликовать всего лишь один открытый ключ, что вполне поддается управлению.

Так для чего же нам шифрование с секретным ключом, если есть такое замечательное шифрование с открытым ключом, спросите вы? Все очень просто: шифрование с секретным ключом на несколько порядков эффективнее в плане ресурсов. Использование шифрования с открытым ключом во всех возможных ситуациях обошлось бы нам слишком дорого. В реальных системах, использующих криптографию с открытым ключом, практически всегда применяется сочетание алгоритмов шифрования с открытым и с секретным ключом. Алгоритмы шифрования с открытым ключом используются для передачи секретного ключа, который, в свою очередь, используется для шифрования сообщений. Это позволяет совместить гибкость криптографии с открытым ключом и эффективность симметричной криптографии.

3.4 Цифровые подписи

Цифровые подписи — это эквивалент открытого ключа для кодов аутентичности сообщений (MAC). Общая схема применения цифровых подписей приведена на рис. 3.6. На этот раз пользователь А применяет специальный алгоритм для генерации пары ключей (S_A, P_A). Если пользователь А хочет отослать пользователю Б подписанное сообщение m, он генерирует цифровую подпись $s := \sigma(S_A, m)$. Затем m и s отсылаются пользователю Б. Пользователь Б применяет принадлежащий пользователю А открытый ключ P_A и алгоритм верификации $\nu(P_A, m, s)$ для проверки цифровой подписи. Цифровая подпись полностью аналогична коду аутентичности сообщения. Различие состоит лишь в том, что для проверки цифровой подписи используется открытый ключ, а секретный ключ требуется только для создания новой подписи.

Для проверки того, что сообщение было действительно отправлено пользователем А, пользователю Б достаточно иметь лишь открытый ключ пользователя А. Что интересно, все остальные тоже могут применить открытый ключ пользователя А и убедиться, что сообщение пришло именно от него.

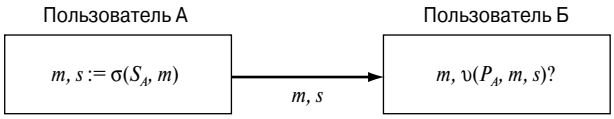

Рис. 3.6. Общая схема применения цифровых подписей

Вот почему подобный способ аутентификации принято называть *цифровой подписью (digital signature)*. Пользователь А будто бы подписывает свое сообщение. В случае какого-нибудь конфликта пользователь Б может показать судье значение m и s и подтвердить, что сообщение было подписано именно пользователем А.

Все это хорошо в теории, да и работает чудесно... тоже в теории. В действительности же цифровые подписи далеко не так полезны, как кажется. Основная проблема заключается в том, что цифровая подпись генерируется не самим пользователем А, а его компьютером. Таким образом, наличие цифровой подписи еще не доказывает, что пользователь А подтвердил свое сообщение или вообще видел его на экране компьютера. Учитывая, с какой легкостью современные вирусы захватывают компьютеры, цифровые подписи вообще мало что доказывают. Тем не менее при правильном использовании цифровая подпись может весьма пригодиться.

3.5 Инфраструктура открытого ключа

Применение криптографии с открытым ключом значительно упрощает управление ключами, однако пользователю А все равно нужно получить открытый ключ пользователя Б. Как гарантировать, что опубликованный ключ P_B является именно ключом пользователя Б, а не кого-нибудь другого? Злоумышленник Е может сгенерировать свою пару ключей и опубликовать открытый ключ от имени пользователя Б. В общем случае для решения этой проблемы применяется *инфраструктура открытого ключа (public key infrastructure — PKI)*.

Основная идея этой инфраструктуры состоит в обращении к некоему доверенному субъекту, называемому *центром сертификации (certificate authority — CA)*. Каждый пользователь регистрируется в центре сертификации и передает ему свой открытый ключ. Центр сертификации подписывает открытый ключ пользователя своей цифровой подписью. Подписанное сообщение, или *сертификат (certificate)*, гласит: “Я, центр сертификации, удостоверяю, что открытый ключ P_B принадлежит пользователю Б”. Помимо этого, сер-

тификат может включать в себя дату окончания срока действия и другую полезную информацию.

С помощью сертификатов пользователю А гораздо легче проверить правильность ключа пользователя Б. Предположим, пользователь А знает открытый ключ центра сертификации и убедился, что этот ключ правильный. Теперь пользователь А может узнать ключ P_B пользователя Б из базы данных или получить его по почте. Чтобы проверить сертификат, прилагаемый к ключу P_B, пользователь А может применить открытый ключ центра сертификации. Этот сертификат гарантирует, что полученный ключ P_B является корректным и может применяться для общения с пользователем Б. Точно так же пользователь Б может узнать открытый ключ пользователя А и будет уверен в том, что общается с нужным человеком.

В инфраструктуре открытого ключа каждому участнику нужно всего лишь сертифицировать свой открытый ключ в центре сертификации, а также получить открытый ключ центра сертификации, чтобы иметь возможность проверять сертификаты других участников. Это намного проще, чем обмениваться ключами с каждым участником общения. Именно в этом и заключается основное преимущество инфраструктуры открытого ключа: регистрируйся однажды, используй везде.

На практике в инфраструктуре открытого ключа обычно создают несколько уровней центров сертификации. Центр сертификации верхнего уровня, называемый *корневым (root)*, выдает сертификаты ключей дочерним центрам сертификации, а те, в свою очередь, сертифицируют ключи пользователей. Общая схема проверки ключа не изменяется, только теперь пользователю А приходится проверять не один, а два сертификата.

Инфраструктура открытого ключа — решение далеко не идеальное; она тоже имеет свои недостатки. Прежде всего, центр сертификации должен пользоваться доверием всех участников общения. Иногда это несложно. Например, отдел кадров предприятия знает всех сотрудников и может выполнять роль центра сертификации. Не существует, однако, такого субъекта, которому бы доверяли абсолютно *все* люди мира. Сама идея того, что одна инфраструктура открытого ключа может охватить весь мир, в корне неверна.

Вторая проблема касается ответственности. Что делать, если центр сертификации выдаст фальшивый сертификат или злоумышленник украдет секретный ключ центра сертификации? Пользователь А будет доверять фальшивому сертификату и потеряет из-за этого много денег. Кто за это заплатит? Очевидно, пользователь А сможет доверять сертификату только в том случае, если будет каким-то образом застрахован центром сертификации. Это потребует более тесных деловых отношений между пользователем А и центром сертификации.

В настоящее время существует масса компаний, которые пытаются утвердить себя в качестве мирового центра сертификации. Наиболее известной из них, пожалуй, является VeriSign. Тем не менее эта компания открыто ограничивает ответственность за некорректное выполнение своих обязанностей. В большинстве случаев размер компенсации ограничивается суммой в 100 долларов. Это намного меньше, чем каждый из нас заплатил во время своей последней покупки книг через Internet (защита платежных транзакций осуществлялась именно посредством сертификатов, подписанных VeriSign). В нашем случае это не было проблемой, так как оплата с помощью кредитной карты вполне безопасна для покупателя. Тем не менее мы ни за что не стали бы покупать автомобиль, используя сертификат, подкрепленный компенсацией всего лишь в 100 долларов.

Более подробно об инфраструктурах открытого ключа и проблемах, связанных с их применением, речь идет в главах 19, "PKI: красивая мечта", 20, "PKI: жестокая реальность", и 21, "Практические аспекты PKI".

3.6 Типы атак

Выше описаны наиболее распространенные функции, применяемые в криптографии. Пришло время обсудить вопросы, связанные с атаками. Существует множество типов атак, каждый из которых обладает той или иной степенью сложности.

3.6.1 Только шифрованный текст

Говоря о взломе системы шифрования, многие имеют в виду *атаку с использованием только шифрованного текста (ciphertext-only attack)*. В этом случае пользователи А и Б зашифровывают свои данные, а злоумышленник видит только шифрованный текст. Попытка расшифровать сообщение при наличии только шифрованного текста и называется атакой с использованием только шифрованного текста. Это наиболее трудный тип атаки, поскольку злоумышленник обладает наименьшим объемом информации.

3.6.2 Известный открытый текст

При *атаке с известным открытым текстом (known plaintext attack)* мы знаем и открытый и шифрованный текст. Цель такой атаки, очевидно, состоит в том, чтобы найти ключ. Вначале это кажется невероятным: откуда взять открытый текст? Однако на практике существует множество ситуаций, в которых злоумышленник может узнать открытый текст сообщения. Иногда содержимое сообщения легко угадать. Представьте себе такую ситуацию:

пользователь А уехал в отпуск и установил в своем почтовом ящике автоответчик, который отвечает на каждое входящее письмо сообщением: “Я уехал в отпуск”. Вы отправляете сообщение пользователю А и получаете точную копию этого ответа. Когда пользователь Б отсылает сообщение пользователю А, он получает точно такой же ответ, однако на сей раз зашифрованный. В этом случае у вас окажется и открытый и шифрованный текст. Если вам удастся определить ключ, вы сможете расшифровывать все сообщения, которыми обмениваются пользователи А и Б.

В качестве еще одного примера можно привести распространенную ситуацию, когда пользователь А рассылает сообщение сразу нескольким адресатам, включая и вас. В этом случае у вас появится открытый текст сообщения, а также копии его шифрованного текста, которые были разосланы всем остальным.

Пользователи А и Б могут обмениваться черновиками пресс-релиза. Как только пресс-релиз будет опубликован, у вас появится и открытый и шифрованный текст.

Даже если в вашем распоряжении не окажется полного открытого текста сообщения, довольно легко получить хотя бы его часть. Например, сообщения электронной почты часто имеют стандартное начало или фиксированную подпись в конце. С высокой степенью вероятности можно предугадать содержимое заголовка IP-пакета. Подобные прогнозируемые данные позволяют получить частично известный открытый текст, а соответствующий тип атак также принято относить к атакам с известным открытым текстом.

Атака с известным открытым текстом намного результативнее атаки с использованием только шифрованного текста. При наличии известного открытого текста у злоумышленника оказывается больше информации, чем при наличии только шифрованного текста, а вся дополнительная информация в данном случае идет только на пользу.

3.6.3 Избранный открытый текст

Следующий уровень контроля подразумевает, что злоумышленник сам может подбирать открытый текст. Это еще более мощный тип атаки, чем атака с известным открытым текстом. Теперь вы сами можете отсылать специально подготовленные простые тексты, подобранные таким образом, чтобы облегчить нападение на систему. Вы можете отослать любое количество простых текстов и получить в ответ соответствующие шифрованные тексты. И вновь следует отметить, что на практике данный тип атаки отнюдь не так невозможен, как кажется. Во многих случаях нападающий сам может выбирать данные, которые будут подвергнуты шифрованию. Довольно часто пользователь А получает информацию из некоторых внешних источников

(например, из находящихся под влиянием злоумышленника) и пересылает ее в зашифрованном виде пользователю Б.

Возможность проведения *атаки с избранным открытым текстом (chosen plaintext attack)* ни в коем случае не следует сбрасывать со счетов. Хороший алгоритм шифрования без труда выдержит это испытание. Если кто-нибудь попытается убедить вас, что атака с избранным открытым текстом не имеет применения на практике, а значит, не стоит внимания специалистов, будьте уверены: его новенький алгоритм шифрования настолько слаб, что не выдержит даже элементарной атаки с избранным открытым текстом.

Существует два вида атак с избранным открытым текстом: *автономная (offline)* и *оперативная (online)*. В первом случае набор открытых текстов, которые должны подвергнуться шифрованию, подготавливается заранее, еще до получения шифрованных текстов. Во втором случае выбор каждого последующего открытого текста осуществляется, исходя из уже полученных шифрованных текстов. В большинстве случаев различием между этими типами атак можно пренебречь. В дальнейшем речь будет идти, как правило, об оперативной атаке с избранным открытым текстом, поскольку она более результативна, чем автономная.

3.6.4 Избранный шифрованный текст

Название *атака с избранным шифрованным текстом (chosen ciphertext attack)* не совсем точно отражает реальный смысл этого понятия. Ее следовало бы называть “атакой с избранным шифрованным и открытым текстом”, однако это слишком неудобно. При проведении атаки с избранным открытым текстом у злоумышленника есть возможность самому подбирать открытый текст. В свою очередь, при проведении атаки с избранным шифрованным текстом подбирать можно как открытый, так и шифрованный текст. Для каждого подобранного открытого текста вы получаете соответствующий шифрованный текст, а для каждого подобранного шифрованного текста — соответствующий открытый текст.

Очевидно, атака с избранным шифрованным текстом является еще более мощной, чем атака с избранным открытым текстом, потому что у атакующего появляется больше свободы в действиях. Цель такой атаки, как и прежде, состоит в том, чтобы найти ключ. А хорошая схема шифрования без труда выдержит и этот тип атаки.

3.6.5 Различающие атаки

Описанные выше типы атак направлены на восстановление открытого текста или ключа шифрования. Существуют атаки, которые не восстанавли-

вают ключ шифрования, однако позволяют расшифровать другое сообщение. Кроме того, есть атаки, которые не восстанавливают сообщение, но позволяют получить некоторую информацию о нем. Вообще говоря, с каждым днем в мире появляется так много разнообразных типов атак, что перечислить их все на страницах этой книги невозможно. Так с чем же нам следует бороться?

Чтобы лучше понять эту проблему, введем понятие различающей атаки. *Различающая атака (distinguishing attack)* — это любой нетривиальный метод, позволяющий обнаружить различие между идеальным и реальным шифром. Данное понятие охватывает все рассмотренные нами типы атак, а также атаки, которые появятся в будущем. Разумеется, нам также понадобится определить, что следует считать идеальным шифром.

Не слишком ли это определение искусственно, скажете вы? Вовсе нет. Опыт подсказывает, что каждый компонент криптографической системы должен быть как можно более совершенным. Некоторые функции шифрования имеют определенные недостатки, что теоретически позволяет взломать их посредством различающей атаки. Во всем остальном, однако, они являются абсолютно удовлетворительными функциями шифрования. Используя такие функции, необходимо всякий раз проверять, не приводят ли эти недостатки к каким-либо проблемам безопасности. В системе, состоящей из множества компонентов, необходимо также проверять, не приводят ли к проблемам безопасности все возможные комбинации недостатков ее компонентов. На практике реализовать данное требование почти невозможно, в результате чего на рынке появляются системы, обладающие массой слабых мест из-за известного несовершенства их компонентов.

3.6.6 Атаки, в основе которых лежит парадокс задачи о днях рождения

Такие атаки получили свое название в честь парадокса задачи о днях рождения, суть которого такова: если в комнате находятся 23 человека, вероятность того, что два из них родились в один и тот же день, превышает 50%. Это на удивление высокая вероятность, учитывая то, что в году может быть 365 разных дней рождения.

Итак, что же такое атака, в основе которой лежит парадокс задачи о днях рождения (birthday attack)? Это тип атаки, основанный на том факте, что одинаковые значения, называемые также *коллизиями (collisions)*, появляются намного быстрее, чем можно было ожидать. Представьте себе систему финансовых транзакций, в которой для обеспечения безопасности каждой транзакции применяется новый 64-битовый ключ аутентификации. (Для простоты будем предполагать, что шифрование не используется.) Существует 2^{64} возможных значения ключа (это больше, чем $18 \cdot 10^{18}$, т.е. 18 миллиардов

миллиардов), поэтому взломать такую систему на первый взгляд довольно сложно, не правда ли? Отнюдь нет! Отследив около 2^{32} транзакций, злоумышленник может предположить, что две из них используют один и тот же ключ. Предположим, что первое сообщение, передаваемое в ходе каждой транзакции, всегда одно и то же: "Готовы ли вы принять транзакцию?" Если две транзакции используют один и тот же ключ аутентификации, тогда значения MAC первых сообщений этих транзакций будут совпадать, что легко отследит злоумышленник. Зная, что обе транзакции используют один и тот же ключ аутентификации, он сможет вставлять сообщения из более старой транзакции в более новую транзакцию во время выполнения последней. Поскольку ложные сообщения успешно пройдут аутентификацию, они будут приняты, что является очевидным взломом системы финансовых транзакций.

В общем случае, если элемент может принимать N различных значений, ожидать первой коллизии можно после случайного выбора приблизительно $\sqrt{N}$ элементов. Не вдаваясь в подробности, можно сказать, что значение $\sqrt{N}$ достаточно близко к истинному. Для парадокса задачи о днях рождения $N = 365$ и $\sqrt{N} \approx 19$. На самом деле количество людей, при котором вероятность совпадения дней рождения превышает 50%, равно 23, однако для наших целей вполне достаточно и приближения $\sqrt{N}$. Для большей наглядности рассмотрим следующее. Если случайным образом выбрать k элементов из N, они могут образовать $k(k-1)/2$ различных пар. Вероятность того, что элементы одной пары совпадут, равна $1/N$. Следовательно, вероятность того, что хотя бы два элемента из k совпадут, равна $k(k-1)/2N$. Если $k \approx \sqrt{N}$, эта вероятность приближается к 50%[2].

В большинстве случаев мы будем говорить об n-битовых значениях. Поскольку n-битовый элемент может иметь 2^n возможных значения, необходимо извлечь $\sqrt{2^n} = 2^{n/2}$ элементов множества, чтобы надеяться на возникновение коллизии. Назовем это оценкой $2^{n/2}$ или *оценкой парадокса задачи о днях рождения (birthday bound).*

3.6.7 Двусторонняя атака

Другой разновидностью атак, в основе которых лежит парадокс задачи о днях рождения, являются так называемые *двусторонние атаки* или *атаки "встреча на середине" (meet-in-the-middle attacks).* (В совокупности оба типа атак называются *атаками на основе коллизий (collision attacks).*) Этот тип атак более распространен и более результативен. Вместо того чтобы пассивно

[2]В действительности это лишь приближенные значения, однако для наших целей их вполне достаточно.

ожидать повторения ключа, вы можете построить таблицу ключей, которые выбрали сами.

Давайте вернемся к предыдущему примеру о системе финансовых транзакций, в которой для каждой транзакции используется новый 64-битовый ключ. Используя двустороннюю атаку, злоумышленник может еще более продвинуться во взломе системы. Для этого он случайным образом выбирает 2^{32} различных 64-битовых ключа. По каждому из них он подсчитывает значение MAC для сообщения: "Готовы ли вы принять транзакцию?" Полученное значение MAC вместе с соответствующим ключом помещается в таблицу. Затем злоумышленник прослушивает каждую транзакцию и проверяет, не окажется ли значение MAC первого сообщения этой транзакции в его таблице. Если такая транзакция находится, значит, с высокой долей вероятности ее ключом аутентификации является тот самый ключ, который был сгенерирован злоумышленником и помещен в таблицу вместе с соответствующим значением MAC. Теперь, когда злоумышленник обладает ключом аутентификации транзакции, он может вставлять в нее собственные сообщения с любым нужным ему текстом. (Напомним, что предыдущий тип атаки позволял злоумышленнику вставлять только сообщения, взятые из более старой транзакции.)

Сколько транзакций придется прослушать злоумышленнику? Он подсчитал значения MAC для 2^{32} из всех возможных ключей. По этой причине каждый раз, когда система выбирает ключ, он с вероятностью 1 к 2^{32} окажется в таблице злоумышленника, а следовательно, после 2^{32} транзакций злоумышленник может ожидать появление транзакции, использующей ключ из его таблицы. Таким образом, злоумышленнику придется проделать 2^{32} предварительных подсчета и прослушать 2^{32} транзакции. Это намного меньше, чем перебирать все 2^{64} возможных ключа.

Различие между атакой, в основе которой лежит парадокс задачи о днях рождения, и двусторонней атакой состоит в следующем. В первом случае мы ждем, когда одно и то же значение появится дважды в одном множестве элементов. В двусторонней атаке у нас есть два множества элементов, и мы ждем, когда эти множества пересекутся. И в том и в другом случае ожидать первого результата можно примерно через одинаковое количество элементов.

Двусторонняя атака является более гибкой, чем атака, в основе которой лежит парадокс задачи о днях рождения. Давайте рассмотрим суть двусторонней атаки в более абстрактной форме. Предположим, что элементы обоих множеств могут принимать N возможных значений. Пусть в первом множестве содержится P элементов, а во втором — Q элементов. Из них можно образовать PQ различных пар, в которых один элемент будет принадлежать первому множеству, а другой — второму множеству. Для каждой пары вероятность того, что значения ее элементов совпадут, равна $1/N$. Мы можем ожи-

дать возникновения коллизии, когда значение PQ/N приближается к единице. В этом случае наиболее эффективным выбором будет $P \approx Q \approx \sqrt{N}$. Как видите, мы вновь получили оценку парадокса задачи о днях рождения. Отметим, что двусторонняя атака обладает определенной гибкостью в отношении выбора P и Q. Иногда элементы одного множества легче получить, чем элементы второго, поэтому размер множеств может быть и неодинаков. Единственным требованием является соблюдение условия $PQ \approx N$. Мы можем выбрать $P \approx N^{1/3}$, а $Q \approx N^{2/3}$. В приведенном выше примере злоумышленник может составить таблицу из 2^{40} значений MAC и ожидать первого совпадения уже после 2^{24} прослушанных транзакций.

Проводя теоретический анализ того, насколько легко взломать систему, мы часто принимаем размер обоих множеств равным $\sqrt{N}$, поскольку это минимизирует количество шагов, которые должен предпринять злоумышленник. На практике необходимо анализировать, насколько элементы одного множества труднее получить, чем элементы другого. Проводя двустороннюю атаку в реальной жизни, величины P и Q следует выбирать таким образом, чтобы удовлетворить условие $PQ \approx N$ при наименьших возможных затратах.

3.6.8 Другие типы атак

До сих пор в основном речь шла об атаках на функции шифрования. Аналогичным образом можно определить атаки на другие криптографические функции, такие, как аутентификация, цифровые подписи и т.п. Они будут рассматриваться по мере необходимости.

3.7 Уровень безопасности

Теоретически, приложив достаточно усилий, злоумышленник может взломать любую криптографическую систему. Вопрос заключается в том, *сколько* работы ему необходимо проделать. Чтобы получить количественную оценку нагрузки на злоумышленника, атаку можно сравнить с поиском, выполняемым путем полного перебора вариантов. Если для осуществления атаки необходимо проделать 2^{235} шагов, это эквивалентно полному перебору вариантов для 235-битового значения.

Много раз отмечая, что злоумышленнику необходимо проделать определенное количество шагов, мы до сих пор не уточнили, что же такое шаг. Это продиктовано не только свойственной нам — чего греха таить — ленью, но и стремлением упростить анализ. Если речь идет о попытке взлома функции шифрования, одним шагом можно считать одну процедуру шифрования заданного сообщения с заданным ключом. Иногда шаг атаки включает в себя всего лишь одно обращение к таблице. Все зависит от конкретной ситуации.

Тем не менее, что бы ни включало в себя понятие шага, он всегда может быть выполнен компьютером за очень короткое время. Иногда на выполнение шага требуется одна временна́я единица, а иногда — миллион единиц, однако в терминах нагрузки, необходимой для осуществления криптографических атак, порядок в один миллион не играет сколько-нибудь значимой роли. Простота выполнения пошагового анализа атак существенно перевешивает его погрешности. При необходимости вы всегда сможете провести подробный анализ, чтобы определить, сколько именно работы включает в себя выполнение одного шага. Для получения быстрых оценок принято полагать, что на выполнение одного шага требуется одна временна́я единица.

Любой современной системе вполне достаточно 128-битового уровня безопасности. Это означает, что осуществление любой атаки на систему потребует, как минимум, 2^{128} шагов. Среднестатистическая криптографическая система, созданная в наши дни, скорее всего, проработает около 30 лет и должна обеспечивать конфиденциальность данных на протяжении приблизительно еще 20 лет с момента окончания использования системы. Таким образом, нам нужно гарантировать безопасность системы на протяжении следующих 50 лет. Это было бы нелегкой задачей, если бы на помощь не пришел закон Мура, адаптированный к криптографии. Согласно ему для современных систем достаточно 128-битового уровня безопасности [62]. Вообще говоря, нам могло бы хватить 110 или даже 100 бит, однако криптографических алгоритмов, рассчитанных на такие ключи, не существует, поэтому мы воспользуемся 128-битовыми ключами.

Приведенная выше концепция уровня безопасности построена на приближениях. Мы измеряем только объем работы, который должен проделать злоумышленник, и не учитываем такие аспекты, как обмен данными с памятью или взаимодействие с атакуемой системой. Анализ нагрузки на злоумышленника — задача отнюдь не простая. Дальнейшее усложнение модели еще более затрудняет анализ и существенно повышает вероятность упустить какой-нибудь важный момент. Поскольку стоимость использования простого и консервативного подхода относительно невелика, мы отдаем предпочтение простой концепции уровня безопасности.

3.8 Производительность

Каждый раз при разработке новой системы мы получаем массу жалоб по поводу высоких затрат, связанных со скоростью и производительностью. Вот что мы думаем по этому поводу.

Прежде всего, безопасность не бывает бесплатной. Если вы хотите обеспечить безопасность своих систем, готовьтесь хорошо заплатить. Те, кто не

в состоянии позволить себе такие расходы, могут не рассчитывать на высокий уровень безопасности. Все очень просто. Современное программное обеспечение затрачивает громадное количество циклов работы процессора на рисование красивых трехмерных окошек с альфа-сопряжением, которые нисколько или почти нисколько не улучшают функциональность. Безопасность — одна из главных проблем современной индустрии. Несмотря на это, люди упорно не желают тратить на нее столько процессорного времени, сколько уходит на рисование красивых окошек.

Разрабатывая криптографическую систему, мы пытаемся сделать ее как можно более эффективной. Тем не менее, начиная с определенного момента, каждая единица прироста производительности обходится слишком дорого. Любое уклонение от проторенной дорожки в сфере обеспечения безопасности оборачивается необходимостью проведения масштабного анализа, гарантирующего, что мы не допустили ошибку и не занимаемся разработкой заведомо слабой системы. Проведение такого анализа требует привлечения опытных специалистов по криптографии, а их услуги обходятся весьма недешево. В большинстве случаев клиентам намного дешевле купить быстрый компьютер, чем ввязываться во все проблемы и расходы, касающиеся разработки и реализации более эффективной системы безопасности.

Для большинства компьютеров системные требования криптографических алгоритмов не составляют проблемы. Современные процессоры настолько быстры, что могут справиться практически с любым потоком данных. Например, шифрование канала передачи данных, обладающего пропускной способностью 100 Мбит/с, с помощью алгоритма AES отнимает всего лишь 20% процессорного времени старенького процессора Pentium III 1 ГГц. (В реальной жизни эта величина еще меньше, поскольку добиться скорости передачи данных 100 Мбит/с невозможно из-за особенностей протокола обмена данными.) Подождав еще пару месяцев, вы сможете купить по той же цене гораздо более мощный процессор, а значит, затраты на повышение производительности будут равны нулю. Одна компания когда-то жаловалась на высокую стоимость шифрования канала передачи данных с пропускной способностью 1 Гбит/с. Машина, которая обрабатывала эти данные, стоила более полумиллиона долларов. Думается, эта компания могла бы позволить себе установить дополнительный процессор или обновить материнскую плату для ускорения обработки криптографии.

Иногда применение криптографии действительно приводит к появлению “узких мест” в производительности системы. Хорошим примером являются Web-серверы, использующие большое число SSL-соединений. Процедура инициализации SSL-соединения предполагает применение шифрования с открытым ключом и требует большого объема вычислительной мощности на стороне сервера. Разумеется, мы могли бы разработать замену протоколу SSL,

которая бы более эффективно использовала ресурсы сервера. Однако это никому не нужно. Гораздо дешевле купить несколько аппаратных ускорителей и работать с существующим протоколом SSL, нежели платить эксперту за разработку нового протокола. А затем за стандартизацию этого протокола. А затем за его реализацию в серверах и обозревателях. А затем убеждать каждого в необходимости перейти на новую версию обозревателя.

Недавно мы придумали хороший аргумент для тех, кто не желает поступаться производительностью в пользу безопасности: "У нас уже есть куча быстрых и небезопасных систем. Зачем нам еще одна?" Это очень верное высказывание. Обеспечение безопасности "наполовину" обходится почти в такую же сумму, как и обеспечение безопасности "целиком", а вот практической пользы от такой безопасности нет. Мы свято верим, что безопасность нужно либо обеспечивать хорошо, либо не обеспечивать вообще.

3.9 Сложность

Не существует сложных систем, которые были бы безопасными.

Правило проектирования 1. *Сложность — главный враг безопасности.*

Это очень простое правило, тем не менее нам понадобилось немало лет для его полного осознания. Современные IT-разработчики практически не справляются с построением сложных систем. Они что-то придумывают, что-то реализуют и приводят это в рабочее состояние... вот, собственно говоря, и все. Продукты, которые бóльшую часть времени находятся в работоспособном состоянии, считаются хорошими, и практически никогда не проводятся тесты на попытку их преднамеренного разрушения.

Описанная ситуация связана с общепринятым принципом разработки программного обеспечения: построить систему, протестировать ее на наличие ошибок, вернуться назад и исправить ошибки, снова протестировать ее на наличие новых ошибок и т.п. Протестировать, исправить, повторить. Так продолжается до тех пор, пока руководство компании не прикажет издать продукт или же пока программистам не надоест с ним возиться. Разумеется, в результате такого процесса будет получена вполне работоспособная система — работоспособная до тех пор, пока будет применяться так, как это было задумано. Это может быть хорошо с точки зрения функциональности, однако совершенно не годится для разработки систем безопасности.

Основная проблема данного метода состоит в следующем. Тестирование обнаруживает только наличие ошибок, притом только таких ошибок, которые ищут тестировщики. Между тем системы безопасности должны работать даже под натиском умных и коварных злоумышленников. Систему нельзя протестировать на все ситуации, в которые ее может поставить нападающий.

Поэтому система должна быть безопасной не в результате тестирований и исправлений, а с самого начала разработки.

Представьте себе такую аналогию. Вы пишете довольно большое приложение на одном из популярных языков программирования. При этом вы старательно исправляете все синтаксические ошибки до тех пор, пока приложению не удастся пройти первую компиляцию. Как только это происходит, вы без какого-либо дальнейшего тестирования упаковываете диск с приложением в красивую коробку и продаете покупателю. Разумеется, никто не ожидает получить действительно функциональный продукт подобным образом.

Между тем именно это и происходит в наши дни с системами безопасности. Их невозможно протестировать, так как никто не знает, на что их нужно тестировать. При этом, если в системе безопасности обнаружится хотя бы одна ошибка, она сведет на нет назначение всей системы. Таким образом, единственный способ получить безопасную систему — это с самого начала делать ее надежной и безопасной. Это, в свою очередь, требует, чтобы система была простой.

Единственный известный нам способ построить простую систему — разбить ее на модули. Это прописная истина программной инженерии. В нашем случае, однако, нельзя позволить себе вообще ни одной ошибки, поэтому к разбивке на модули необходимо подойти со всей возможной строгостью. Вот наше основное правило.

Правило проектирования 2. *Корректность работы должна быть локальным свойством.*

Другими словами, одна часть системы должна функционировать корректно, независимо от того, как функционирует вся остальная система. Только не пытайтесь сказать нам что-нибудь наподобие: “Это не проблема, потому что оставшаяся часть системы просто не может дать сбой”. Оставшаяся часть системы может иметь ошибку или же измениться в какой-нибудь будущей версии. Каждая часть системы должна отвечать за свою собственную функциональность.

Как вы вскоре убедитесь, мы еще не раз будем применять это правило на протяжении последующих глав книги. Именно оно стало причиной того, что большинство наших решений оказались более консервативными, чем те, которые принято применять в разработке современных систем.

Часть I

Безопасность сообщений

Глава 4

Блочные шифры

Блочные шифры — одна из фундаментальных составляющих криптографических систем. Существует обширная литература, посвященная блочным шифрам, а сами они являются одной из наиболее хорошо изученных областей криптографии.

4.1 Что такое блочный шифр?

Блочный шифр (block cipher) — это функция шифрования, которая применяется к блокам текста фиксированной длины. Текущее поколение блочных шифров работает с блоками текста длиной 128 бит (16 байт). Такой шифр принимает на вход 128-битовый открытый текст и выдает 128-битовый шифрованный текст. Блочный шифр является обратимым: существует функция дешифрования, которая принимает на вход 128-битовый шифрованный текст и выдает исходный 128-битовый открытый текст. Открытый и шифрованный текст всегда имеет один и тот же размер, который называется *размером блока (block size)*.

Чтобы зашифровать сообщение, нужен секретный ключ. Невозможно скрыть сообщение, не сохраняя что-нибудь в секрете. Подобно открытому и шифрованному тексту, ключ шифрования также представляет собой строку битов. Наиболее распространены ключи размером 128 и 256 бит. Шифрование открытого текста p при помощи ключа K принято обозначать как $E(K, p)$ или $E_K(p)$, а расшифровку шифрованного текста c при помощи ключа K — $D(K, c)$ или $D_K(c)$.

Как правило, блочные шифры применяются для шифрования информации. К коротким сообщениям блочный шифр можно применять непосредственно. Если же длина сообщения превышает длину блока (обычно так и бы-

вает), необходимо использовать один из режимов работы блочного шифра, рассматриваемых в главе 5, “Режимы работы блочных шифров”.

Мы всегда следуем принципу Кирхгофа и предполагаем, что алгоритмы шифрования и дешифрования являются публично известными. Многим трудно смириться с таким подходом, и они предпочитают сохранять алгоритмы в секрете. Никогда не доверяйте секретным блочным шифрам (или любым другим секретным криптографическим функциям)!

Иногда блочный шифр было бы удобно представить в виде большой таблицы, построенной на основе конкретного ключа. Для каждого фиксированного ключа можно построить таблицу соответствий, которая бы отображала все варианты открытого текста в соответствующий шифрованный текст. Разумеется, это была бы *очень* большая таблица. Для блочного шифра с размером блока 32 бит понадобилась бы таблица размером 16 Гбайт, для шифра с размером блока 64 бит — 150 млн. Тбайт, а для шифра с размером блока 128 бит — $5 \cdot 10^{39}$ байт. Такому большому числу еще даже не придумали названия! Конечно же, на практике построить такую таблицу нереально, однако ее удобно использовать в качестве концептуальной модели. Мы также знаем, что блочный шифр является обратимым. Другими словами, в таблице не существует двух одинаковых элементов шифрованного текста, соответствующих разным элементам открытого текста. В противном случае функция дешифрования не смогла бы однозначно восстановить открытый текст по шифрованному тексту. Из этого можно сделать вывод, что в таблице содержится ровно по одному экземпляру всех возможных вариантов шифрованного текста. Как видите, набор элементов шифрованного текста совпадает с набором элементов открытого текста, порядок расположения которых изменен. В математике это называется *перестановкой (permutation)*. Блочный шифр с размером блока k бит задает перестановку k-битовых элементов для каждого из заданных значений ключа.

4.2 Типы атак

На первый взгляд, имея определение блочного шифра, совсем несложно дать определение безопасному блочному шифру: это блочный шифр, который позволяет сохранить открытый текст в секрете. Однако очевидно, что данное требование является необходимым, но отнюдь не достаточным для построения действительно хорошего шифра. Оно предполагает всего лишь устойчивость блочного шифра по отношению к атаке с использованием только шифрованного текста, в котором нападающий видит лишь зашифрованный текст сообщения. В литературе опубликовано несколько примеров атак такого типа [53, 91], однако они крайне редки. Большинство известных атак принадлежат

к типу атак с избранным открытым текстом. В разделе 3.6 рассматриваются основные типы атак. Все они применимы и к блочным шифрам. Существует также несколько типов атак, специфичных для блочных шифров.

Один из этих типов атак называется *атакой со связанным ключом (related-key attack)*. Впервые представленная Эли Бихемом в 1993 году [7], атака со связанным ключом предполагает, что злоумышленник имеет доступ к нескольким функциям шифрования. Все они работают с неизвестными ключами, однако эти ключи связаны определенным отношением, которое известно злоумышленнику. На первый взгляд такая атака выглядит несколько странно, однако, как оказалось, она дает весьма неплохие результаты по отношению к реальным системам. Существует множество реальных систем, которые используют разные ключи, связанные известным отношением. В одной закрытой системе для каждого нового сообщения предыдущее значение ключа увеличивалось на единицу. Таким образом, сообщения, идущие друг за другом, шифровались с помощью последовательных значений ключей. Оказывается, что подобные соотношения между ключами могут использоваться для атак на блочные шифры.

Существуют еще более экзотические типы атак. Команда разработчиков блочного шифра Twofish представила концепцию *атаки с избранным ключом (chosen key attack)*, в которой злоумышленник задает часть ключа и затем выполняет атаку со связанным ключом на оставшуюся часть ключа [85][1].

Зачем обращать внимание на какие-то неправдоподобные типы атак, такие, как атаки со связанным ключом или с избранным ключом? На это есть ряд причин. В своей практике мы видели реальные системы, которые вполне могли подвергнуться атаке со связанным ключом, поэтому данный тип атак вообще нельзя назвать неправдоподобным. Блочные шифры довольно часто используются в криптографических системах и потому подвергаются всем мыслимым и немыслимым нападениям. Одним из стандартных приемов построения функции хэширования для блочного шифра является метод Дэвиса–Мейера [95]. Оказалось, что при использовании функции хэширования Дэвиса–Мейера злоумышленник получает возможность выбирать ключ блочного шифра, что позволяет ему осуществлять атаки со связанным ключом и с избранным ключом. Любое определение безопасности блочного шифра, которое не учитывает эти или любые другие типы атак, является неполным. Блочный шифр — это модуль, который должен иметь простой интерфейс. Наиболее простым этот интерфейс будет в том случае, если он включает в себя все свойства, которые кто-либо может ожидать от блочного шифра.

[1] Дальнейшие исследования показали, что этот тип атаки не позволяет взломать Twofish [31], однако может оказаться успешным при нападении на другие блочные шифры.

Наличие некоторого несовершенства блочного шифра лишь усложняет использующую его систему многочисленными перекрестными зависимостями.

4.3 Идеальный блочный шифр

Чтобы определить безопасность блочного шифра, необходимо вначале дать определение идеальному блочному шифру. Как должен выглядеть идеальный блочный шифр? Очевидно, это должна быть случайная перестановка вариантов открытого текста. Немного уточним: для каждого значения ключа блочный шифр должен представлять собой случайную перестановку вариантов открытого текста, причем перестановки для различных вариантов ключа должны выбираться независимо друг от друга. Как уже отмечалось, 128-битовый блочный шифр (одна перестановка 128-битовых значений) можно представить себе в виде большой таблицы соответствий, содержащей 2^{128} элементов по 128 бит в каждом. В идеальном блочном шифре каждому значению ключа соответствует одна из таких таблиц, причем она выбирается случайным образом из набора всех возможных таблиц (т.е. всех возможных перестановок).

С формальной точки зрения, данное определение идеального блочного шифра является неполным, поскольку оно не задает соответствия таблиц различным значениям ключей. С другой стороны, как только будет задано соответствие таблиц, идеальный блочный шифр станет фиксированным и больше не будет случайным. Чтобы формализовать это определение, мы не можем говорить об одном конкретном идеальном шифре. Мы должны рассмотреть идеальный шифр как равномерное вероятностное распределение на множестве всех возможных блочных шифров. Имея дело с идеальным блочным шифром, необходимо размышлять в терминах вероятностей. Это приводит в восторг математиков, однако существенно усложняет и без того непростые объяснения. Поэтому будем придерживаться неформальной, но в то же время более простой концепции случайно выбранного блочного шифра.

4.4 Определение безопасности блочного шифра

В литературе дано множество определений безопасности блочного шифра (например, [52]). Большинство этих определений сформулированы с математической точки зрения, и ни одно из них не отражает аспектов, которые затронутых в предыдущих разделах. Мы же предпочитаем простое, хотя и неформальное определение.

Определение 1 *Безопасный блочный шифр — это шифр, для которого не существует атак.*

Небольшая тавтология, не так ли? Теперь необходимо определить, что такое атака на блочный шифр.

Определение 2 *Атака на блочный шифр — это нетривиальный метод обнаружения различия между блочным шифром и идеальным блочным шифром.*

Что же подразумевается под обнаружением такого различия? Речь идет о сравнении некоторого блочного шифра X с идеальным блочным шифром, имеющим такой же размер блока и такой же размер ключа. Различитель — это алгоритм, которому свойственна функция "черного ящика", вычисляющая для заданного открытого текста блочный шифр X либо идеальный блочный шифр. (Функция "черного ящика" — это функция, которую можно оценить, однако точная внутренняя структура которой неизвестна.) Алгоритму-различителю доступны и функция шифрования, и функция дешифрования. Кроме того, для каждой операции шифрования или дешифрования различитель может применять любой выбранный им ключ. Задача различителя состоит в том, чтобы определить, что именно реализует функция "черного ящика": блочный шифр X или же идеальный блочный шифр. Различителю необязательно быть совершенным — достаточно лишь, чтобы правильные ответы давались значительно чаще неправильных.

Описанная выше задача, конечно же, имеет тривиальные решения. Можно зашифровать открытый текст 0 с помощью ключа 0 и посмотреть, соответствует ли результат шифрования тому, что мы ожидаем получить от блочного шифра X. Данный алгоритм также подходит под определение различителя, однако, чтобы осуществить реальное нападение на систему, различитель должен быть нетривиальным. Именно этот аспект и затрудняет определение безопасности блочного шифра. Мы не можем формализовать понятия "тривиальный" и "нетривиальный". Это все равно что пытаться определить непристойное поведение: мы сможем понять, непристойно себя ведет человек или нет, только непосредственно столкнувшись с его поведением[2]. Различитель является тривиальным, если мы можем найти аналогичный различитель практически для каждого блочного шифра. В описанном случае различитель является тривиальным, потому что мы можем построить такой же различитель для каждого блочного шифра, даже для идеального.

Можно построить и более совершенный тривиальный различитель. Давайте зашифруем открытый текст 0 с помощью всех ключей в диапазоне $1, \ldots, 2^{32}$ и подсчитаем, как часто среди полученных результатов будет повторяться каждое конкретное значение первых 32 бит шифрованного текста.

[2] В 1964 году верховный судья США Поттер Стюарт (Potter Stewart) использовал это выражение для определения того, что следует считать непристойным поведением: "Сегодня я больше не буду пытаться определить данный тип события... но я пойму, оно это или нет, когда его увижу".

Пусть для шифра X оказалось, что значение t встречается пять раз вместо ожидаемого одного. Это свойство вряд ли характеризует идеальный шифр. Данный алгоритм также является тривиальным различителем, поскольку аналогичный метод можно применить к любому шифру X. (Маловероятно, чтобы для конкретного блочного шифра не нашлось подходящего значения t.)

Ситуация усложняется, если построить различитель следующим образом. Составим список из 1000 различных статистических показателей, которые мы можем подсчитать для шифра. Затем подсчитаем все эти показатели для шифра X и построим различитель на основе того показателя, который даст наиболее значимый результат. Мы ожидаем найти показатель с уровнем значимости, примерно равным 0,001. Разумеется, с помощью этого алгоритма можно построить различитель для любого конкретного шифра, поэтому данный метод является тривиальным, однако теперь эта тривиальность зависит не только от самого различителя, но и от того, как он был построен. Вот почему еще никому не удалось формализовать определение тривиальности и безопасности блочного шифра. Криптографическое сообщество еще не настолько хорошо знает криптографию, чтобы корректно сформулировать данное определение. Использование более формального определения, которое не учитывает целый ряд типов атак, никак не поможет в построении безопасных систем.

Допустимое количество вычислений, выполняемых различителем, должно быть ограничено. Мы не упомянули об этом в самом определении, чтобы не усложнять его. Если блочный шифр обладает явно заданным уровнем безопасности в n бит, различитель должен быть более эффективен, чем поиск путем полного перебора n-битовых значений. Если же уровень безопасности явно не указан, он принимается равным размеру ключа. Данная формулировка несколько расплывчата, однако на то есть причина. Во многих источниках просто отмечается, что различитель должен выполнить свою работу менее чем за 2^n шагов. Это, безусловно, верно, однако некоторые типы различителей дают только вероятностный результат, более похожий на поиск ключа с частичным перебором вариантов. Атака не всегда обладает прямой зависимостью между количеством проделанной работы и вероятностью обнаружить различие между шифром и идеальным шифром. Взять хотя бы такой пример: поиск путем полного перебора на половине пространства ключей требует выполнения 2^{n-1} шагов и дает правильный ответ в 75% случаев. (Если злоумышленник находит ключ, он уже знает ответ. Если же он не находит ключ, у него все еще есть 50%-ная вероятность правильно угадать этот ключ. Таким образом, общая вероятность получить правильный ответ равна $0{,}5 + 0{,}5 \cdot 0{,}5 = 0{,}75$.) Сравнивая различитель с подобным частичным поиском на пространстве ключей, мы учитываем эту особенность и не рассматриваем частичный поиск как атаку.

Наше определение безопасности блочного шифра охватывает все возможные типы атак. Атака с использованием только шифрованного текста, с известным открытым текстом, с избранным открытым текстом (в том числе и оперативная или, как ее еще называют, адаптивная), со связанным ключом и все другие типы атак реализуют нетривиальный различитель. Вот почему нам так нравится наше определение.

Вас может удивить, что мы потратили столько страниц, пытаясь дать определение безопасному блочному шифру? Это определение является очень важным, поскольку описывает простой и понятный интерфейс между блочным шифром и оставшимися частями системы. Такой тип разбивки на модули — важная особенность качественного проектирования. В системах безопасности, одним из главных врагов которых является сложность, удачная разбивка на модули имеет более существенное значение, чем в остальных областях проектирования. Если блочный шифр удовлетворяет нашему определению безопасности, мы можем считать его идеальным шифром. В конце концов, если он не ведет себя как идеальный шифр, мы можем найти для него различитель, а значит, шифр не удовлетворяет нашему требованию безопасности. Используя безопасный блочный шифр, нет необходимости помнить о его особенностях или недостатках; наш шифр будет обладать всеми свойствами, которых мы ожидаем от блочного шифра. Поскольку концепция идеального шифра очень проста, это значительно облегчает работу проектировщиков.

4.4.1 Четность перестановки

К сожалению, есть еще одно затруднение. Как уже отмечалось, шифрование блока текста при заданном значении ключа соответствует поиску соответствия в таблице перестановок. Предположим, что построение этой таблицы осуществляется в два этапа. Вначале мы инициируем таблицу, присваивая элементу с индексом i значение i. Затем создаем нужную перестановку, меняя местами два соседних элемента таблицы и повторяя эту операцию нужное количество раз. Оказывается, существует два типа перестановок: одни могут быть получены путем четного количества таких обменов (они называются четными перестановками), другие — путем нечетного количества обменов (они называются нечетными перестановками). Думаем, вы не удивитесь, узнав, что половина всех перестановок относится к четным, а половина — к нечетным.

Большинство современных блочных шифров имеют размер блока 128 бит, однако применяются к 32-битовым словам. Их функции шифрования построены на основе многократного применения 32-битовых операций. Данный метод получил заслуженное признание, однако у него есть один недостаток.

Применяя такие операции, довольно сложно получить нечетную перестановку. В результате практически все известные блочные шифры генерируют только четные перестановки.

Упомянутый выше факт позволяет построить простой различитель практически для каждого блочного шифра. Мы называем такой алгоритм *атакой с проверкой четности (parity attack)*. Для заданного значения ключа постройте перестановку, зашифровав по порядку все возможные варианты открытого текста. Если перестановка является нечетной, значит, перед нами идеальный блочный шифр, так как реальный блочный шифр никогда не генерирует нечетную перестановку. Если же перестановка четная, соответствующий блочный шифр рассматривается как реальный. Такой различитель будет давать правильные ответы в 75 случаях из ста. Он ошибется только тогда, когда ему попадается идеальный блочный шифр, генерирующий четную перестановку. Чтобы повысить вероятность получения правильного ответа, этот же алгоритм можно применить и к другим значениям ключа.

Несмотря на всю привлекательность атаки с проверкой четности, она не имеет практического применения. Чтобы определить четность перестановки, необходимо с помощью функции шифрования вычислить все пары "открытый текст–шифрованный текст", кроме одной. (Последняя пара определяется тривиально: единственный оставшийся вариант открытого текста отображается на единственный оставшийся вариант шифрованного текста.) В реальных системах никогда не следует допускать такое количество запросов к блочному шифру, поскольку успешное завершение других типов атак будет происходить еще быстрее. В частности, как только злоумышленник получит бóльшую часть пар "открытый текст–шифрованный текст", ему больше не понадобится ключ: он сможет расшифровать сообщение или хотя бы его основную часть, просто воспользовавшись таблицей соответствий для этих пар.

По определению атаку с проверкой четности можно было бы назвать тривиальной, однако это было бы не совсем честно с нашей стороны. Вместо этого мы изменим определение идеального блочного шифра и ограничим его случайно выбранными *четными* перестановками.

Определение 3 *Для каждого значения ключа идеальный блочный шифр реализует случайную четную перестановку, выбранную независимо от перестановок, соответствующих другим значениям ключа.*

Данное определение несколько усложняет нашу концепцию "идеального" шифра, однако в противном случае нам пришлось бы дисквалифицировать практически все известные науке блочные шифры. Для подавляющего большинства случаев ограничение допустимых перестановок четными перестановками является несущественным. Поскольку мы никогда не позволим злоумышленнику вычислить все пары "открытый текст–шифрованный текст",

определить различие между четными и нечетными перестановками будет невозможно.

Если у вас когда-нибудь появится блочный шифр, который *будет* генерировать нечетные перестановки, вам понадобится вернуться к первоначальному определению идеального блочного шифра. На практике атаки с проверкой четности влияют скорее на формальное определение безопасности, нежели на реальные системы, поэтому о четности перестановок можно вообще забыть.

4.5 Современные блочные шифры

На протяжении последних десятилетий мировой общественности были представлены сотни блочных шифров. Создать новый блочный шифр проще простого. Создать *хороший* новый блочный шифр чрезвычайно сложно. Мы не имеем в виду безопасность: блочный шифр по умолчанию должен быть безопасным. Хвастаться безопасностью нового шифра — это все равно что кичиться новой крышей, которая не протекает, или машиной, у которой есть фары. Самая большая сложность заключается в том, чтобы создать блочный шифр, который бы оказался эффективным во многих областях применения.

Разработка нового шифра — весьма интересное и поучительное занятие, но очень просим вас: ПОЖАЛУЙСТА, не используйте неизвестный шифр в реальных системах! Мы ни за что не станем доверять шифру до тех пор, пока его тщательно не исследуют другие эксперты. Основным требованием к новому шифру является его повсеместная публикация, однако этого недостаточно. Существует так много шифров, что лишь некоторые из них подвергаются тщательным исследованиям. Намного проще использовать один из широко известных шифров, которые уже были изучены и получили массу положительных откликов.

Практически все блочные шифры представляют собой несколько последовательных применений слабого блочного шифра, называемого *раундом (round)*. Некоторые из таких слабых раундов в совокупности образуют весьма надежный блочный шифр. Подобные структуры значительно облегчают разработку и реализацию шифра, как, впрочем, и его анализ. Большинство атак на блочные шифры начинаются с атаки на версии шифров с минимальным количеством раундов. По мере усовершенствования атаки могут применяться для нападения на шифры с все бóльшим и бóльшим количеством раундов.

Далее в главе рассматривается несколько наиболее популярных блочных шифров. Наш обзор будет не слишком обширным — полные спецификации этих шифров можно найти по указанным нами ссылкам в Internet. Мы же сконцентрируемся на общей структуре и свойствах каждого из них.

4.5.1 DES

Алгоритм шифрования DES (Data Encryption Standard — стандарт шифрования данных) [69], незаменимая рабочая лошадка криптографии, наконец-то перешагнул черту пенсионного возраста. Ограничения на размер ключа в 56 бит и размер блока в 64 бит делают DES непригодным для современных высокоскоростных компьютеров и огромных объемов данных. Он все еще может применяться в виде "тройного" DES или 3DES [72] — блочного шифра, образованного путем трех последовательных применений алгоритма DES. Это решает наиболее острую проблему малого размера ключа, однако не позволяет снять ограничение на малый размер блока. По современным стандартам DES — не особенно быстрый шифр, а 3DES работает еще в три раза медленнее. Несмотря на все это, DES до сих пор используется во многих существующих системах, однако применять DES или 3DES в новых разработках не рекомендуется.

На рис. 4.1 приведена схема одного раунда DES. Это линейная диаграмма вычислений, выполняемых в рамках алгоритма DES; подобные диаграммы часто встречаются в криптографической литературе. Каждый прямоугольник соответствует вычислению значения конкретной функции, а стрелки показывают, куда подается то или иное значение. Существует несколько стандартных соглашений по поводу обозначений на диаграммах. Операция XOR — "исключающее ИЛИ" (ее еще называют "побитовым сложением" или "сложением без переноса") — обозначается в формулах как оператор $\oplus$, а на рисунках — точно таким же символом, только большим. Иногда на схемах встречается и обычная операция сложения, которую обозначают символом $+$.

На вход алгоритма DES подается 64-битовый блок открытого текста, который разбивается на две 32-битовые половины: L (левая) и R (правая). Пе-

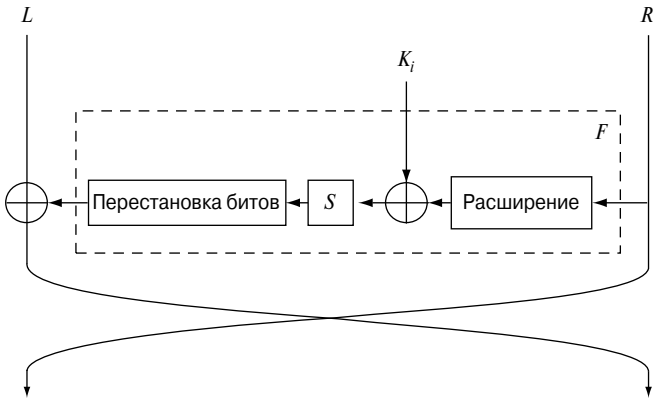

Рис. 4.1. Структура одного раунда DES

ред этим биты исходного блока текста подвергаются начальной перестановке по полуупорядоченному принципу. Никто не может толком объяснить, зачем разработчики шифра DES решили переставить биты открытого текста — ведь это не имеет никакого криптографического эффекта; однако алгоритм DES определен именно так. По окончании шифрования левая и правая половины вновь объединяются и подвергаются обратной перестановке, в результате чего вновь получается 64-битовый блок текста, однако на сей раз шифрованного.

Алгоритм DES состоит из 16 раундов, которые обозначаются цифрами от 1 до 16. Каждый раунд i преобразует пару (L, R) в новую пару (L, R) с помощью подключа K_i. Бóльшую часть преобразования выполняет функция раунда F (на рис. 4.1 она обведена штриховой рамкой). Как показано на рисунке, вначале к значению R применяется функция расширения. Она дублирует 16 битов значения R, в результате чего 32-битовое значение превращается в 48-битовое. Этот результат функции расширения складывается с помощью операции XOR с 48-битовым подключом K_i. Результат этой операции подается на вход S-матриц. По своей сути S-матрица (буква S означает *substitution*, т.е. *подстановка*) — это всего лишь таблица соответствий. Поскольку мы не можем построить таблицу соответствий для 48-битовых данных, S-матрицы состоят из восьми небольших таблиц соответствий, каждая из которых получает на вход 6 бит и выдает 4-битовый результат. Таким образом, после преобразования 48-битового значения с помощью S-матриц, мы вновь получаем 32-битовое значение. Последнее подвергается еще одной перестановке битов, после чего складывается с помощью операции XOR с левой половиной L. И наконец, значения правой и левой половины меняются местами. Эта процедура повторяется еще 15 раз.

В основе алгоритма DES лежит так называемый шифр Файстеля [29]. Идея шифра Файстеля очень проста и красива. Каждый раунд представляет собой побитовое сложение значения L со значением $F(K_i, R)$ (где F — это некоторая функция) и последующий обмен местами значений L и R. Прелесть этого алгоритма заключается в том, что расшифровка состоит из точно такого же набора операций. Необходимо поменять местами значения L и R и выполнить побитовое сложение значения L со значением $F(K_i, R)$. Это намного упрощает реализацию функций шифрования и дешифрования. Это также означает, что достаточно анализировать лишь одну из двух функций, поскольку они практически идентичны. В большинстве шифров Файстеля по окончании шифрования применяется еще один особый прием — отмена перестановки значений L и R, выполненной в последнем раунде. Благодаря этому функции шифрования и дешифрования становятся полностью идентичными за исключением порядка применения подключей. Это особенно удобно для реализации в аппаратном обеспечении, поскольку для шифрования и дешифрования может применяться одна и та же схема.

Для шифрования текста алгоритм DES использует шестнадцать 48-битовых подключей. Каждый подключ образуется путем выбора 48 бит из 56-битового ключа шифрования, причем для каждого раунда этот выбор выполняется по-своему[3].

Каждый из компонентов шифра DES имеет свое назначение. Алгоритм Файстеля упрощает структуру шифра и гарантирует перемешивание правой и левой половин текста. Сложение текста с подключом с помощью операции XOR гарантирует перемешивание ключа и данных, в чем, собственно, и заключается весь смысл шифрования. S-матрицы обеспечивают нелинейность. Без них процесс шифрования можно было бы представить в виде последовательности операций двоичного сложения, что очень легко взломать, используя методы линейной алгебры. И наконец, сочетание S-матриц, функции расширения и перестановки битов обеспечивает диффузию. Другими словами, если изменить один бит во входном значении функции F, в ее выходном значении изменится сразу несколько битов. В следующем раунде это изменение станет еще более обширным и т.п. При отсутствии диффузии незначительное изменение открытого текста приведет к незначительному изменению шифрованного текста, что можно легко отследить.

Алгоритм DES обладает рядом свойств, которые не позволяют считать его безопасным в соответствии с нашим определением безопасности. Каждый из подключей представляет собой не более чем выборку битов ключа шифрования. Если ключ шифрования равен 0, все подключи также будут равны 0. Это, в частности, означает, что все подключи будут одинаковыми. Напомним, что процедуры шифрования и дешифрования различаются лишь порядком применения подключей. Но в нашем случае все подключи будут равны. Таким образом, шифрование с помощью ключа 0 — это то же самое, что и дешифрование с помощью ключа 0. Данное свойство шифра очень легко обнаружить, а поскольку идеальный шифр таким свойством не обладает, это позволяет осуществить легкую и эффективную различающую атаку[4].

Алгоритм DES обладает и свойством комплементарности (или дополнения). Согласно этому свойству для любого ключа K и открытого текста P справедливо следующее:

$$E(\bar{K}, \bar{P}) = \overline{E(K, P)},$$

где $\bar{X}$ — это значение, каждый бит которого является дополнением соответствующего бита значения X. Другими словами, если зашифровать дополне-

[3]Выбор подключей осуществляется в соответствии с некоторым механизмом, описание которого содержится в спецификациях DES [69].

[4]Существует еще три ключа, которые обладают этим же свойством. Они называются слабыми ключами алгоритма DES.

ние открытого текста с помощью дополнения ключа, мы получим значение, которое является дополнением шифрованного (исходного) текста.

Доказать данное свойство нетрудно. Посмотрите на рис. 4.1 и подумайте, что случится, если изменить значения всех битов в L, R и K_i на противоположные. Функция расширения просто копирует биты, поэтому биты результата функции расширения также будут изменены на противоположные. У функции XOR будут изменены биты и первого и второго аргументов, поэтому их сумма останется неизменной. На вход S-матриц будет подано то же значение, что и раньше, а значит, и выходное значение не изменится. В последней операции XOR один аргумент останется неизменным, а второй будет изменен на противоположный. Поэтому в новом значении L (оно впоследствии поменяется местами с R) биты также окажутся измененными на противоположные. Другими словами, если в начале раунда заменить значения L, R и K_i их дополнениями, результатом выполнения раунда будет значение, являющееся дополнением того, которое было бы получено при использовании исходных значений. Данное свойство передается из раунда в раунд.

Идеальный блочный шифр никогда бы не обладал подобным курьезным свойством. Что еще более важно, это свойство шифра может привести к атакам на системы, использующие DES.

Иными словами, алгоритм DES больше не выдерживает проверки на профпригодность. Длина его ключа шифрования не отвечает современным требованиям. В мире уже совершено несколько успешных попыток определить ключ DES путем простого перебора вариантов.

Алгоритм 3DES работает с ключом шифрования большего размера. К сожалению, от своего предшественника DES он унаследовал и слабые ключи, и свойство комплементарности. Каждого из этих свойств вполне достаточно, чтобы по нашим стандартам шифр считался небезопасным. Кроме того, 3DES обладает ограничением на размер блока, который не может превышать 64 бит. Это существенно ограничивает объем данных, которые можно зашифровать с помощью одного ключа. (Более подробно это рассматривается в разделе 5.8.) Иногда 3DES все же приходится использовать для обеспечения совместимости с существующими системами, но будьте осторожны — он ведет себя отнюдь не как идеальный шифр.

4.5.2 AES

Несколько лет назад в США был принят новый правительственный стандарт шифрования, который получил название AES (Advanced Encryption Standard — улучшенный стандарт шифрования). Вместо того чтобы разрабатывать новый шифр или поручать эту работу конкретным людям, Национальный институт стандартов и технологий (National Institute of Standards

and Technology — NIST) обратился за помощью к криптографическому сообществу. В качестве кандидатов на лучший шифр были приняты 15 предложений [71], из которых затем отобрали пять финалистов [73]. Лучшим шифром был признан Rijndael, который и получил статус нового стандарта шифрования[5]. В целом этот процесс прошел гораздо лучше, чем ожидалось. Если кто-нибудь из вас захочет стандартизировать новую криптографическую систему, здесь определенно есть чему поучиться. Если вы получите хорошие предложения, устройте конкурс наподобие того, как это было сделано для AES, — это намного лучше, чем разрабатывать шифр всем комитетом. Если же у вас недостаточно экспертов, которые могут подавать хорошие предложения, то вам, вероятно, вообще не стоит заниматься стандартизацией.

Структура AES существенно отличается от DES. Алгоритм AES не относится к шифрам Файстеля. На рис. 4.2 показан один раунд алгоритма AES. Последующие раунды имеют аналогичную структуру. На вход алгоритма подается блок открытого текста длиной 16 байт. Вначале открытый текст складывается с помощью операции XOR с 16-байтовым (128-битовым) подключом. На рисунке этот процесс обозначен операторами $\oplus$ (каждый байт подключа складывается с соответствующим байтом открытого текста). Затем каждый из 16 байт полученного результата подается на вход таблицы S-матриц, которая отображает 8-битовые входные значения в 8-битовые выходные значения. Все S-матрицы одинаковы. Полученные байты переставляются в некотором заданном порядке. На рисунке он выглядит несколько запутанным, однако на самом деле имеет очень простую структуру. И наконец, каждая группа из 4 байт подвергается перемешиванию, которое осуществляется с помощью линейной функции перемешивания. Термин “линейная” означает лишь то, что каждый бит выходных данных функции перемешивания получен в результате применения операции XOR к нескольким входным битам.

Применение функции перемешивания завершает раунд. Полный процесс шифрования состоит из 10-14 раундов, в зависимости от размера ключа. Как и в DES, подключи AES генерируются на основе некоторого ключа шифрования, однако механизм генерации ключей полностью отличен от применявшегося в DES.

Алгоритм AES имеет свои преимущества и недостатки. Каждый шаг алгоритма состоит из нескольких операций, которые могут выполняться одновременно, что облегчает создание высокоскоростных реализаций AES. С другой стороны, операция дешифрования существенно отличается от операции шифрования. Для расшифровки текста необходимо использовать обратные S-

[5] Многие смущаются, потому что не знают, как произносить слово “Rijndael”. Не волнуйтесь: если вы не говорите по-голландски, то все равно не сможете прочитать его правильно, поэтому произносите его так, как вам больше нравится.

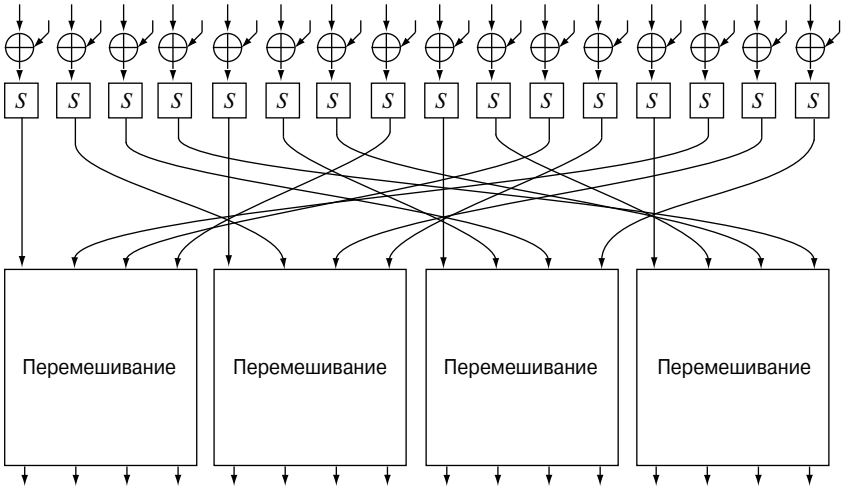

Рис. 4.2. Структура одного раунда AES

матрицы, да и функция, обратная перемешиванию, существенно отличается от самой функции перемешивания.

Как и в DES, в алгоритме AES можно выделить несколько функциональных блоков, каждый из которых имеет свое назначение. Операции XOR складывают значение ключа с данными, S-матрицы обеспечивают нелинейность, а функции перестановки и перемешивания гарантируют наличие диффузии. Шифр AES имеет очень четкую структуру, каждая часть которой выполняет строго определенную задачу.

Тем не менее полностью исключить сомнения насчет безопасности AES невозможно. Разработчики AES всегда проводили несколько агрессивную политику. На первой презентации своего алгоритма они продемонстрировали пример атаки на 6 раундов и объявили, что полный процесс шифрования должен состоять из 10–14 раундов, в зависимости от размера ключа [18]. В процессе отбора кандидатов на получение звания нового стандарта механизм атак был улучшен настолько, что стал справляться с 7 раундами алгоритма для 128-битовых ключей, 8 раундами для 192-битовых ключей и даже 9 раундами для 256-битовых ключей [30]. Казалось бы, у нас остается еще от 3 до 5 раундов на обеспечение безопасности. С другой стороны, наиболее результативная из известных атак на 128-битовые ключи покрывает уже 70% шифра. Другими словами, безопасность алгоритма AES основана на предположении, что будущие атаки на этот шифр не смогут продемонстрировать каких-либо существенных улучшений.

Предсказать будущее, как всегда, невозможно, однако иногда имеет смысл заглянуть в прошлое. До сих пор наиболее хорошо изученными шифрами яв-

лялись DES, FEAL и IDEA. В каждом из этих случаев через много лет после первой публикации шифра наблюдалось значительное усовершенствование атак на него. Время идет, и криптография тоже не стоит на месте, однако нам и сейчас остается лишь верить в то, что мы знаем об атаках все и что в ближайшее время в этой области не произойдет сколько-нибудь заметных положительных изменений.

Нельзя не отметить, что данная проблема имеет значение в основном для специалистов по криптографии. Даже если бы современные атаки были усовершенствованы настолько, что могли бы взломать AES, они, вероятно, потребовали бы около 2^{120} шагов и 2^{100} байт памяти. Этого было бы достаточно для того, чтобы по нашим стандартам считать шифр "взломанным" или, точнее, чтобы уменьшить уровень его безопасности до 120 бит. Данный шифр уже бы не удовлетворял нашим требованиям безопасности, однако терять спокойный сон из-за него мы бы не стали. Подобные атаки еще невозможно осуществить на практике и вряд ли будет возможно на протяжении тех 50 лет жизни, которые мы отвели современным криптографическим системам (см. раздел 3.7).

Гораздо большее беспокойство вызывает простая алгебраическая структура алгоритма AES [33]. Весь процесс шифрования AES можно представить в виде относительно простой замкнутой алгебраической функции с конечным полем из 256 элементов. Это еще не атака, а лишь представление, однако, если кто-нибудь когда-нибудь сможет справиться с этими функциями, AES будет взломан, что открывает абсолютно новый подход к осуществлению атак. Еще ни один из известных блочных шифров не имел такого простого алгебраического представления. Мы не знаем, приведет это к нападениям на шифр или нет, но и этого вполне достаточно, чтобы скептически относиться к использованию AES. Наше воображение не раз рисовало пренеприятнейшую картину. Проходит пять лет. Алгоритм AES применяется во многих криптографических системах по всему миру. Мы сидим в аудитории и слушаем доклад какого-то старшекурсника, имеющего отношение к совсем другой области математики. Студент откашливается и начинает говорить: "Однажды от нечего делать я начал листать книгу своего друга. Это оказалась книга по криптографии. В ней было несколько формул, которые поразительно напоминали формулы, когда-то встречавшиеся мне совсем в другом месте. Мне стало интересно, и я..." Через 20 минут этот доклад завершается словами: "Итак, мой компьютер может вычислить этот ключ примерно за два часа".

Давайте говорить откровенно. Это абсолютно несправедливая критика AES. Пока что для этого шифра не существует атак. В будущем же атаке может подвергнуться абсолютно каждый шифр, включая и AES. Тем не менее простая алгебраическая структура AES делает его потенциально уязвимым перед абсолютно новым классом атак. У криптографов еще нет опыта

в этой области. Ни один из специалистов по криптографии, к которому мы обращались, не знает, что делать с такими функциями. Возможно, в мире существуют люди, которые умеют решать задачи подобного рода, однако у нас нет никаких причин предполагать, что они знают криптографию. Давайте будем оптимистами и предположим, что вероятность хоть сколько-нибудь успешных попыток осуществления атак такого типа равна 10%. Пусть вероятность того, что эти попытки приведут к реализации практической атаки на полную версию шифра, также равна 10%. Таким образом, вероятность практической атаки на AES составляет 1% — риск, которого можно было бы избежать, используя более традиционный шифр с более сложной алгебраической структурой.

Как бы там ни было, постепенно все перейдут на использование AES, поскольку это государственный стандарт США. Мы тоже будем рекомендовать своим клиентам использовать AES, потому что он *действительно* является стандартом, а использование стандарта позволяет избежать многочисленных споров и проблем. Даже если AES когда-нибудь и взломают, никто не посмеет упрекнуть вас за то, что вы отдали предпочтение стандартному шифру. К сожалению, агрессивная политика разработчиков AES в совокупности с простой алгебраической структурой несколько ухудшает общее впечатление от нового стандарта.

4.5.3 Serpent

Шифр Serpent — еще один из пяти финалистов, соревновавшихся за право носить гордое имя стандарта AES [2]. Своей массивностью и защищенностью он напоминает танк. Наиболее консервативный из всех участников конкурса, Serpent во многом противоположен AES. В то время как разработчики алгоритма AES делали упор на красоту и эффективность, Serpent полностью ориентирован на обеспечение безопасности. Наилучшая из известных атак способна взломать только 10 из 32 раундов [6]. Недостатком шифра Serpent является его скорость — он в три раза медленнее AES. Он также не очень подходит для эффективной реализации, поскольку S-матрицы должны быть преобразованы в булевы функции, подходящие для конкретного процессора.

Кое в чем Serpent все же сходен с AES. Его алгоритм шифрования состоит из 32 раундов. В каждом раунде выполняется сложение данных и 128-битового подключа с помощью операции XOR, применение к 128-битовому значению линейной функции перемешивания и наконец параллельное применение 32 четырехбитовых S-матриц. В каждом раунде применяются 32 одинаковые S-матрицы, однако из раунда в раунд они изменяются. Кроме того, есть восемь различных S-матриц, которые поочередно используются в каждом последующем раунде.

Существует интересный прием программной реализации шифра Serpent. Обычная реализация “в лоб” работала бы слишком медленно, потому что в каждом раунде необходимо выполнять поиск соответствий в 32 S-матрицах, а таких раундов тоже 32. В сумме необходимо 1024 раза проделать поиск соответствий, а проводить операции поиска одну за другой было бы слишком медленно. Вместо этого S-матрицы представляют в виде булевых функций. Каждый из четырех выходных битов представляется как результат выполнения булевой функции от четырех входных битов. После этого процессор непосредственно вычисляет значение булевой функции, используя команды AND, OR и XOR. Хитрость заключается в том, что 32-разрядный процессор может одновременно подсчитывать значения 32 таких функций, поскольку каждая позиция двоичного разряда в регистрах процессора вычисляет значение одной и той же функции, хотя и с разными входными данными. Такой тип реализации называется *разрядно-модульным (bitslice)*. Шифр Serpent специально спроектирован в расчете на разрядно-модульную архитектуру. Помимо S-матриц, она позволяет относительно легко вычислять значения функций перемешивания.

Если бы Serpent был таким же быстрым, как Rijndael (теперешний AES), он бы практически наверняка выиграл конкурс благодаря своей консервативной структуре. Но скорость — понятие относительное. В перерасчете на зашифрованный байт Serpent оказывается почти таким же быстрым, как DES, и намного быстрее, чем 3DES. Он кажется медленным только по сравнению с другими финалистами конкурса AES.

4.5.4 Twofish

Алгоритм Twofish также вошел в число финалистов AES. Он представляет собой некий компромисс между AES и Serpent — практически такой же быстрый, как и AES, но обладающий гораздо бóльшим “запасом прочности”. Но что еще важнее, он не имеет простого алгебраического представления. Наилучшая известная нам атака способна взломать лишь 8 раундов из 16. Основным недостатком Twofish является относительная дороговизна смены ключа шифрования. Это объясняется тем, что реализация алгоритма Twofish требует выполнения целого ряда предварительных операций над ключом.

Подобно DES, алгоритм Twofish основан на шифре Файстеля. Структура Twofish представлена на рис. 4.3[6]. На вход алгоритма подается 128-битовый текст. Он разбивается на четыре 32-битовых значения, и большинство операций выполняются над 32-битовыми значениями. Как видно из рисунка,

[6]Не удивляйтесь, что этот рисунок намного больше и подробнее предыдущих. Так уж получилось, что мы принадлежим к числу разработчиков Twofish, поэтому без всяких зазрений совести взяли этот рисунок прямо из нашей книги, посвященной Twofish [85].

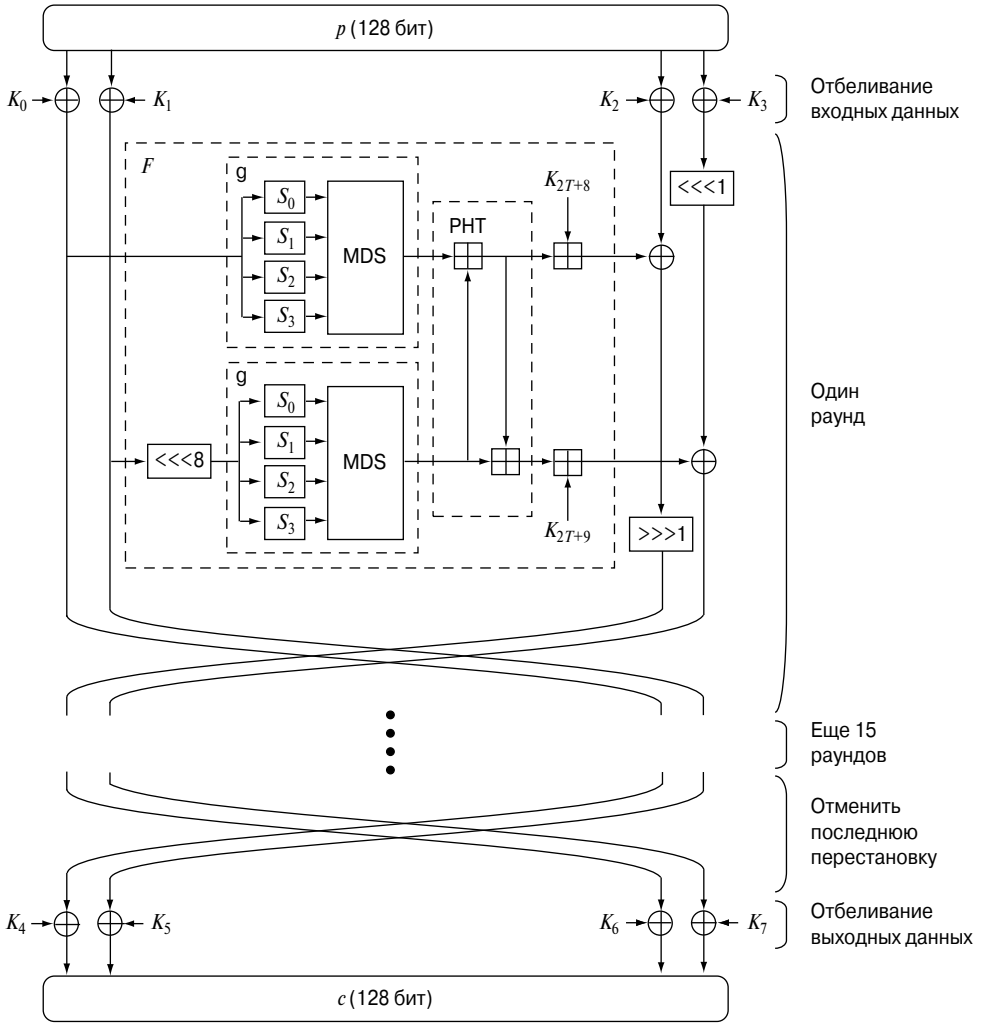

Рис. 4.3. Структура алгоритма Twofish

структура Twofish соответствует структуре шифра Файстеля. Здесь F — это функция раунда, которая состоит из двух одинаковых функций g, функции под названием PHT и операции сложения с подключом. Результат функции F складывается при помощи операции XOR с правой половиной текста (две вертикальные линии справа). Прямоугольники с символами <<< и >>> внутри означают циклический сдвиг битов 32-битового значения влево или вправо на указанное число позиций.

Каждая функция g состоит из четырех S-матриц, за которыми следует линейная функция перемешивания, очень похожая на ту, что используется

в AES. Однако S-матрицы здесь совсем другие. В отличие от всех блочных шифров, которые нам до сих пор доводилось видеть, S-матрицы Twofish не являются постоянными; их содержимое зависит от ключа. Существует алгоритм, который вычисляет S-матрицы для заданного ключа. S-матрицы были сделаны переменными, так как анализировать матрицы, зависящие от ключа, намного сложнее. Это также является причиной того, почему программные реализации Twofish выполняют ряд предварительных операций над ключом. Они вычисляют S-матрицы и сохраняют полученный результат в памяти.

Функция PHT перемешивает результаты двух функций g, используя 32-битовые операции сложения. В последней части функции F выполняется сложение данных с подключом. Обратите внимание, что обычная операция сложения обозначена как $\boxplus$, а операция "исключающее ИЛИ" — как $\oplus$.

Помимо описанных функций, в алгоритме Twofish используется так называемое *отбеливание (whitening)*. В начале и в конце шифрования данные складываются с дополнительными подключами. Это значительно затрудняет осуществление большинства типов атак, а расходы на выполнение операции отбеливания совсем небольшие.

Как и другие шифры, Twofish использует некоторый алгоритм генерации подключей раундов и двух дополнительных подключей, применяемых в начале и в конце шифрования, на основе фактического ключа шифрования.

Признаемся откровенно: наше мнение несколько субъективно. Как разработчики Twofish, мы очень любим свой шифр. Мы пытались сохранять объективность, однако в данной ситуации быть полностью объективными невозможно. Twofish был разработан частично затем, чтобы удовлетворить наши требования к хорошему блочному шифру. AES и Serpent были разработаны, чтобы удовлетворить те требования, которые казались наиболее важными их авторам. Например, в алгоритме Twofish мы сознательно добавили к шифру два однобитовых циклических сдвига. Такие сдвиги имеют два недостатка. Они вносят различие в операции шифрования и дешифрования, что требует гораздо бóльших расходов на реализацию. Они также замедляют реализацию программного обеспечения примерно на 5%. Единственная причина добавления циклических сдвигов состояла в том, чтобы нарушить ту четкую структуру разбивки на байты, которая, например, наблюдается в AES. Лично нам кажется, что алгоритм AES чрезмерно аккуратный и структурированный. Разработчики AES, в свою очередь, наверняка думают, что алгоритму Twofish не хватает структурированности и элегантности. У каждого из нас свое мнение, и мы, конечно же, глубоко уважаем мнения своих коллег. Но в этой книге все советы определяются нашим мнением.

4.5.5 Другие финалисты AES

Итак, три из пяти финалистов AES уже рассмотрены. Осталось еще два: RC6 [77] и MARS [13]. Эти алгоритмы нравятся нам гораздо меньше, а потому ограничимся лишь кратким обзором их особенностей.

Алгоритм RC6 интересен тем, что применяет умножение 32-битовых значений. В процессе состязания кандидатов на звание AES наиболее результативной атаке удалось взломать 17 раундов RC6. По сравнению с 20 раундами полной версии шифра это выглядит слишком угрожающе, чтобы ощущать себя в безопасности.

Алгоритм MARS обладает весьма запутанной структурой. В нем используется большое количество разных операций, в результате чего реализация MARS обходится дороже, чем реализация каждого из предыдущих алгоритмов. Помимо этого, нас беспокоят еще две ошибки. Ошибка в коде программы, которая генерировала S-матрицу для MARS, привела к тому, что полученная S-матрица не удовлетворяла требованиям, выдвинутыми самими разработчиками алгоритма. Но что гораздо важнее, в аргументе, приводимом авторами MARS как доказательство устойчивости алгоритма к линейному криптоанализу (особый тип атаки на шифр), обнаружилось серьезное упущение. На данный момент хорошего анализа чувствительности MARS по отношению к линейным атакам еще нет. Поскольку линейные атаки являются довольно мощным и весьма популярным инструментом нападения на шифр, такой анализ должен проводиться для каждого серьезного шифра.

Вообще-то и RC6 и MARS — неплохие шифры. Нам просто кажется, что AES, Serpent и Twofish намного лучше, поэтому советуем выбирать именно из этих трех шифров.

4.5.6 Атаки с помощью решения уравнений

Во время работы над этой книгой в мире стремительно набирал обороты новый тип атак. Он уже наделал довольно много шума в криптографическом сообществе. Основная идея этого метода заключается в том, чтобы представить блочное шифрование в виде системы линейных и квадратных уравнений над некоторым конечным полем, а затем решить эти уравнения, используя новые методы наподобие XL, FXL и XSL.

В 2002 году Николас Куртуа (Nicolas Courtois) и Йозеф Пьепжик (Josef Pieprzyk) объявили, что могут использовать указанные методы для нападения на Serpent и AES [17]. В криптографическом сообществе это заявление произвело эффект разорвавшейся бомбы. Наибольшую известность получило описание атаки на AES. Нас же гораздо больше потрясло сообщение об атаке на полную 32-раундовую версию Serpent, так как мы считали этот алгоритм самым надежным из всех кандидатов на AES.

Через некоторое время результаты Куртуа и Пьепжика получили несколько опровержений. Проблема заключается в том, что и те и другие заявления остаются чисто теоретическими. Алгоритм XSL очень сложен и слишком чувствителен к конкретной форме уравнений, для решения которых он предназначен. Оценка количества работы, необходимой для взлома AES и Serpent, построена на основе целого ряда эвристик. Так или иначе, пока что предполагаемое количество шагов намного превышает 2^{128}, поэтому наши системы, которые всегда обладают 128-битовым уровнем безопасности, могут не бояться XSL-атак. Пока XSL-атаки не претерпят значительных улучшений, они не представляют угрозы реальным системам. Тем не менее, как и все новые типы атак, они вполне могут быть усовершенствованы в самом недалеком будущем.

Насколько мы знаем, такие атаки еще не реализованы на практике. Хотелось бы увидеть компьютерную программу, которая использует эти методы для взлома сокращенных версий AES и Serpent. Получив хоть какие-то реальные данные, мы бы смогли оценить производительность алгоритма XSL по отношению к этим шифрам. Без этого мы не можем судить, действительно ли прямые атаки методом решения уравнений могут привести к взлому шифра.

Что из этого выйдет? Это не известно. Спросите нас через пять лет — возможно, тогда мы будем знать больше. На данный момент мы можем лишь высказывать свое мнение, а не приводить научно обоснованные факты. Это абсолютно новый тип атак, для которого пока что не придумано противоядия. Если XSL-атаки действительно работают, они взломают все существующие на данный момент шифры. Спасти шифр от взлома может лишь чистая случайность. С другой стороны, вполне возможно (а с нашей точки зрения, и наиболее вероятно), что XSL-атаки не применимы на практике или же применимы только к небольшому числу высокоструктурированных шифров.

А как насчет Twofish? Никто никогда не пытался применять эти методы к Twofish. Осуществить атаку подобного рода на Twofish гораздо сложнее, чем на AES или Serpent, так что никто и не пытался. Поэтому мы не знаем, насколько данные методы эффективны по отношению к нашему алгоритму. Если кому-нибудь удастся применить XSL-атаку для взлома AES или Serpent, она, без сомнения, будет применена и к Twofish.

4.5.7 Какой блочный шифр выбрать

Вот в чем вопрос. Не забывайте, что наше мнение несколько субъективно, потому что мы принадлежим к команде разработчиков Twofish. Мы также провели массу времени, пытаясь взломать другие шифры из числа финалистов AES, что еще сильнее повлияло на нашу точку зрения.

Заявления о возможности осуществления XSL-атак делают процесс выбора блочного шифра несколько неприятным. Мы просто не знаем, насколько такие атаки влияют на безопасность блочного шифра. Информации по этой теме пока еще очень немного, а та, что есть, постоянно оспаривается. Поскольку ни один блочный шифр не разрабатывался с учетом возможности XSL-атак, все эти шифры могут оказаться уязвимыми. На наш взгляд, в текущей ситуации выбор шифра не должен определяться восприимчивостью к XSL-атакам. Никто не может сказать, насколько один шифр более уязвим, нежели другой.

Самым безопасным выбором для карьеры, безусловно, является AES. Это официальный стандарт шифрования, принятый правительством США. Он будет использоваться всеми и каждым. Конечно же, мы не думаем, что это самый удачный выбор в плане безопасности данных. Тем не менее, даже если AES и взломают, это будет не ваша вина. Вы, вероятно, не раз слышали фразу: “Никого не оштрафуют за то, что он купил IBM”. Точно так же никто не упрекнет вас за то, что вы выбрали AES. Итак, если речь не идет о вашем семейном бюджете и/или спокойном сне, смело выбирайте AES.

Алгоритм AES имеет и другие преимущества. Он довольно прост в реализации и применении. Его поддерживают все криптографические библиотеки и любят все клиенты, потому что “это стандарт”. Исходя из этого, вы определенно не ошибетесь, выбрав AES.

Если вы серьезно обеспокоены безопасностью своих данных и готовы ради этого пожертвовать быстродействием, выбирайте Serpent. В процессе отбора кандидатов на AES все авторитетные криптографы признали, что Serpent был самым безопасным (или самым консервативным) из всех поступивших предложений.

Приведенные выше аргументы практически не оставляют места для Twofish. Выбирайте Twofish только тогда, когда хотите получить скорость AES без всех присущих тому недостатков в плане обеспечения безопасности. Разумеется, в этом случае все преимущества AES как официального стандарта шифрования будут направлены против вас. Если Twofish взломают, вас обвинят в неправильном выборе шифра.

В некоторых ситуациях наилучшим выбором все еще является 3DES. Если вам нужно обеспечить обратную совместимость с существующими системами или же если остальные части системы не поддерживают размеры блоков более 64 бит, выбирайте 3DES. Не забывайте, однако, что он не удовлетворяет нашим критериям безопасности, и будьте особенно бдительны, используя маленькие 64-битовые блоки.

4.5.8 Каким должен быть размер ключа

Все три шифра, рекомендуемые нами для использования в современных системах (AES, Serpent и Twofish), поддерживают ключи размером 128, 192 и 256 бит. Практически всем приложениям достаточно уровня безопасности 128 бит. Несмотря на это, нам очень не нравятся 128-битовые ключи.

Эти ключи хороши всем, за исключением одного: возможность осуществления атак на основе коллизий. Нам то и дело попадаются системы, чувствительные к атакам, в основе которых лежит парадокс задачи о днях рождения, или к двусторонним атакам. Одни разработчики просто игнорируют такие атаки, другие думают, что защищены от них, однако ни те, ни другие не застрахованы от появления новых, более изощренных способов нападения. Большинство режимов работы блочных шифров в той или иной форме допускают двусторонние атаки. Мы уже достаточно натерпелись от них, а потому предлагаем еще одно правило проектирования.

Правило проектирования 3. *Если уровень безопасности системы равен n бит, каждое криптографическое значение должно иметь длину, как минимум, $2n$ бит.*

Это правило сводит на нет все усилия по обнаружению коллизий, а расходы на его реализацию не так уж велики. В реальной жизни, однако, следовать ему довольно сложно. Например, для обеспечения 128-битового уровня безопасности необходимо использовать блочный шифр с размером блока 256 бит, однако все распространенные блочные шифры работают с блоками по 128 бит. Данная проблема намного серьезнее, чем кажется. Существует довольно большое количество атак на основе коллизий, направленных на режимы работы блочных шифров.

Несмотря на это, мы, как минимум, можем использовать ключи большего размера, которые поддерживают все рекомендуемые нами шифры. Отсюда урок: используйте 256-битовые ключи!

К сожалению, шифр AES (Rijndael) — единственный из всех финалистов AES, который работает с 256-битовыми ключами медленнее, чем со 128-битовыми[7]. Как следствие этого, во избежание потери производительности вас могут вынудить использовать ключи меньшего размера. Мы вовсе не утверждаем, что 128-битовые ключи небезопасны как таковые. Вполне возможно создать систему со 128-битовым уровнем безопасности, используя 128-битовые ключи, однако это будет крайне сложно. Например, вы не сможете просто воспользоваться одним из стандартных режимов работы блочных шифров.

[7]Serpent работает с одной и той же скоростью независимо от размера ключа. Twofish работает медленнее при выполнении предварительных вычислений над ключами большего размера, однако в программной реализации скорость шифрования не зависит от размера ключа.

Вам придется вводить дополнительные операции сложения данных с подключами, чтобы помешать осуществлению атак на основе коллизий. Это и есть тот вид сложности, который ведет к появлению "слабых мест". Поэтому рекомендуем использовать именно 256-битовые ключи.

Обратите внимание, что мы рекомендуем использовать 256-битовые ключи в системах с уровнем безопасности 128 бит. Другими словами, эти системы спроектированы так, чтобы выдержать атаку, состоящую из 2^{128} операций. Просто запомните, что для выбора размера параметров оставшейся части системы нужно использовать значение уровня безопасности (128 бит), а не размера ключа (256 бит).

Нельзя не отметить еще одно затруднение. Мы используем 256-битовый ключ для разработки систем со 128-битовым уровнем безопасности. Уже упоминавшиеся нами XSL-атаки применяются к алгоритмам AES и Serpent с ключами большего размера. Как утверждается, эти атаки более эффективны, чем поиск путем полного перебора *ключей большего размера*. Данное утверждение касается атаки на Serpent, которая требует выполнения более чем 2^{128} шагов. Даже если XSL-атака, как это и было заявлено, сработает против 256-битового шифра Serpent, она не будет атакой на 256-битовый Serpent, если установить уровень безопасности Serpent (как и всей остальной системы) равным 128 бит. В конце концов, если уровень безопасности равен 128 бит, тогда атака на шифр должна быть более эффективной, чем поиск путем полного перебора 2^{128} элементов, а текущие XSL-атаки на Serpent требуют выполнения гораздо большего количества шагов. Это же справедливо и для XSL-атак на AES.

Глава 5

Режимы работы блочных шифров

Блочные шифры применяются только к блокам текста фиксированного размера. Чтобы зашифровать какой-нибудь текст, размер которого не совпадает с размером блока, необходимо воспользоваться одним из *режимов работы блочных шифров (block cipher modes)*. Это еще одно название для функции шифрования, построенной на основе блочного шифра.

Прежде чем углубиться в изучение материала этой главы, обратите внимание вот на что. Режимы работы блочных шифров предназначены для того, чтобы не дать злоумышленнику читать поток сообщений. Они не обеспечивают никакой аутентификации, а потому злоумышленник все еще может изменять сообщение так, как ему вздумается. Как ни странно, но это действительно так. Функция дешифрования, соответствующая выбранному режиму работы, лишь расшифровывает данные. Ее результатом может оказаться бессмысленный набор символов, тем не менее она послушно расшифровывает (измененный) шифрованный текст в некоторый (измененный и, возможно, бессмысленный) открытый текст. Не думайте, что бессмысленные сообщения не могут причинить вреда. Они способны повлиять на остальные части системы, что порой приводит к печальным последствиям.

Практически во всех ситуациях измененные сообщения наносят гораздо больший ущерб, чем чтение злоумышленником открытого текста. Поэтому шифрование всегда следует сочетать с аутентификацией. Все режимы, рассмотренные в этой главе, нужно применять в совокупности с отдельной функцией аутентификации, о чем идет речь в главе 7, “Коды аутентичности сообщений”.

5.1 Дополнение

Режим работы блочного шифра — это способ зашифровать открытый текст P, преобразуя его в шифрованный текст C, причем и тот и другой текст имеет произвольную длину. Большинство режимов работы блочных шифров требуют, чтобы длина открытого текста P была кратной размеру блока. Для этого сообщение приходится дополнять до нужной длины. Существует множество способов дополнения открытого текста. Главное требование состоит в том, чтобы дополнение было обратимым, т.е. чтобы по дополненному сообщению можно было уникальным образом определить исходное сообщение.

Довольно часто открытый текст просто дополняют нулями до нужной длины. Это не очень удачная идея. Такое дополнение не является обратимым, поскольку открытый текст p при дополнении принимает ту же форму, что и открытый текст $p \parallel 0$. (Оператор $\parallel$ обозначает конкатенацию.)

В этой книге мы ограничимся рассмотрением открытого текста, состоящего из целого числа байтов. Некоторые криптографические алгоритмы предназначены для работы с сообщениями произвольной длины, в которых последний байт считается неполным. Подобное обобщение вряд ли может оказаться полезным, в то время как недостатков от его использования очень много. Большинство реализаций алгоритмов шифрования не принимают сообщений с неполными байтами, поэтому в дальнейшем все размеры приводятся в байтах.

В идеале правило дополнения не должно было бы увеличивать длину открытого текста, если она уже подходит для применения шифра. Зачастую, однако, это невозможно. Всегда найдется хотя бы несколько сообщений, длину которых, несмотря на то что она уже является подходящей, требуется еще более увеличить с помощью любой обратимой схемы дополнения. На практике все правила дополнения увеличивают длину открытого текста, как минимум, еще на один байт.

Итак, как же дополнить открытый текст? Пусть P — это открытый текст, $l(P)$ — его длина в байтах, b — размер блока используемого блочного шифра в байтах. Предлагаем воспользоваться одной из двух простых схем дополнения.

Добавьте к открытому тексту один байт со значением 128, а затем такое количество нулевых байтов, чтобы общая длина сообщения стала кратной b. Количество добавляемых нулевых байтов лежит в диапазоне $0, \ldots, b-1$.

Определите, сколько байтов нужно добавить к открытому тексту, чтобы его длина стала кратной b. Обозначим это количество байтов как n, где $1 \leq n \leq b$ и $n+l(P)$ кратно b. Присоедините к открытому тексту n байтов, каждый из которых будет иметь значение n.

Каждая из этих схем одинаково хорошо подходит для дополнения открытого текста. Никаких криптографических нюансов, связанных со схемами дополнения, не существует. Любая схема будет приемлема, если она обратима. Мы лишь привели примеры наиболее простых схем дополнения.

После того как открытый текст дополнен до длины, кратной размеру блока, он разбивается на блоки. В результате этого открытый текст P превращается в последовательность блоков $P_1, \ldots, P_k$. Количество блоков k может быть подсчитано по формуле $\lceil (l(P) + 1)/b \rceil$, где $\lceil \ldots \rceil$ — функция округления сверху, которая округляет число до следующего целого числа. В оставшейся части главы будем предполагать, что открытый текст P состоит из целого числа блоков $P_1, \ldots, P_k$.

Когда шифрованный текст будет расшифрован с использованием одного из рассмотренных далее режимов работы, добавленные байты следует удалить. Перед этим необходимо убедиться, что дополнение было выполнено корректно. Следует проверить правильность значения каждого из добавленных байтов. Неправильное дополнение должно быть обработано так же, как и ошибка аутентификации.

5.2 Электронная шифровальная книга (ECB)

Самый простой метод шифрования открытого текста, длина которого превышает длину блока, называется режимом *электронной шифровальной книги (electronic codebook — ECB)*. Он определяется следующим образом:

$$C_i = E(K, P_i) \quad \text{для } i = 1, \ldots, k.$$

Данный режим довольно прост: каждый блок сообщения шифруется отдельно. Разумеется, на практике все не может быть так просто, иначе зачем посвящать целую главу обсуждению режимов работы блочных шифров? Никогда не используйте режим ECB! Он обладает серьезными недостатками и включен в эту книгу только затем, чтобы посоветовать вам держаться от него подальше.

Так какими же недостатками обладает ECB? Если два блока простого текста одинаковы, соответствующие блоки шифрованного текста также будут одинаковы, что непременно заметит злоумышленник. В зависимости от структуры сообщения, подобный недостаток может привести к серьезной утечке информации.

Существует довольно много ситуаций, в которых повторяются большие блоки текста. Например, в данной главе часто повторяется словосочетание “блоки шифрованного текста”. Если сообщение содержит два повторяющихся

элемента с периодом повторения, кратным размеру блока, в открытом тексте появится два одинаковых блока. В большинстве строк Unicode каждый второй байт является нулевым, что значительно повышает вероятность возникновения повторяющихся блоков. Многие форматы файлов подразумевают наличие больших блоков символов, состоящих исключительно из нулей, что приводит к появлению повторяющихся блоков. К сожалению, данная особенность режима ECB делает его слишком слабым для сколько-нибудь серьезного использования.

5.3 Сцепление шифрованных блоков (CBC)

Наиболее популярным режимом работы блочных шифров является *сцепление шифрованных блоков (cipher block chaining — CBC)*. В этом случае во избежание недостатков, свойственных режиму ECB, каждый блок открытого текста складывается с помощью операции XOR с предыдущим блоком шифрованного текста. Стандартная формулировка режима CBC выглядит следующим образом:

$$C_i = E(K, P_i \oplus C_{i-1}) \quad \text{для } i = 1, \ldots, k.$$

Как видите, в открытый текст вносится "элемент случайности" путем его прибавления к предыдущему блоку шифрованного текста. В режиме CBC одинаковые блоки открытого текста преобразуются в разные блоки шифрованного текста, что значительно сокращает объем информации об открытом тексте, доступной злоумышленнику.

5.3.1 Фиксированный вектор инициализации

Следует заметить, что в качестве C_0 используется значение, называемое *вектором инициализации (initialization vector — IV)*. Использовать фиксированный вектор инициализации не следует. В противном случае в первом блоке каждого сообщения могут возникнуть проблемы, свойственные режиму ECB. Если два разных сообщения начинаются с одного и того же блока открытого текста, соответствующие шифрованные сообщения будут начинаться с одинаковых блоков шифрованного текста. На практике сообщения часто начинаются с одинаковых или похожих блоков шифрованного текста, поэтому не стоит помогать злоумышленнику.

5.3.2 Счетчик

В качестве вектора инициализации иногда используют счетчик. Например, для первого сообщения значение вектора инициализации выбирается

равным 0, для второго — 1 и т.д. Но это не совсем удачная идея. Как уже отмечалось, в реальной жизни многие сообщения начинаются примерно одинаково. Если первые блоки сообщений будут иметь простые отличия, использование в качестве вектора инициализации простого счетчика может нейтрализовать эти различия с помощью операции XOR и вновь привести к генерации одинаковых блоков шифрованного текста. В частности, значения 0 и 1 отличаются друг от друга ровно одним битом. Если начальные блоки открытого текста первых двух сообщений будут различаться только этим битом (что происходит гораздо чаще, чем можно было ожидать), то начальные блоки шифрованного текста обоих сообщений будут одинаковы. В этом случае злоумышленник сможет сразу же сделать определенные выводы относительно различий между сообщениями, чего никогда не должна допускать безопасная схема шифрования.

5.3.3 Случайный вектор инициализации

Все проблемы с режимом ECB, а также с использованием фиксированных векторов инициализации или счетчиков в режиме CBC связаны с тем, что реальные сообщения обладают высокой степенью неслучайности. Большинство сообщений имеют фиксированный заголовок или стандартную, легко предсказуемую структуру. В режиме CBC для внесения "элемента случайности" в блоки открытого текста используются блоки шифрованного текста, однако для шифрования первого блока текста применяется вектор инициализации. Вот почему этот вектор должен быть случайным.

Применение случайного вектора инициализации связано с определенной проблемой. Получатель сообщения должен знать значение вектора инициализации. Обычно для этого выбирают случайное значение IV и посылают его в качестве первого блока перед самим шифрованным сообщением. В результате процедура шифрования выглядит следующим образом:

$$C_0 := \text{случайное значение блока},$$
$$C_i := E(K, P_i \oplus C_{i-1}) \quad \text{для } i = 1, \ldots, k$$

при условии, что (дополненный) открытый текст $P_1, \ldots, P_k$ преобразуется в шифрованный текст $C_0, \ldots, C_k$. Обратите внимание, что шифрованный текст начинается с блока C_0, а не C_1; шифрованный текст на один блок длиннее, чем открытый. Из приведенных выше формул легко вывести соответствующую процедуру дешифрования:

$$P_i := D(K, C_i) \oplus C_{i-1} \quad \text{для } i = 1, \ldots, k.$$

Использование случайного вектора инициализации имеет два недостатка. Прежде всего алгоритм шифрования должен иметь доступ к источнику

случайных чисел. Реализация хорошего генератора случайных чисел весьма трудоемка, а потому данного требования рекомендуется всячески избегать. Кроме того, шифрованный текст на один блок длиннее, чем открытый. Это значительно увеличивает короткие сообщения, что всегда нежелательно.

5.3.4 Оказия

Существует еще одно — и, пожалуй, наилучшее — решение проблемы выбора вектора инициализации. Оно состоит из двух шагов. Вначале каждому сообщению, которое должно быть зашифровано с помощью заданного ключа, присваивается уникальное число, называемое *оказией (nonce)*. Название этого термина происходит от фразы "**n**umber used **once**" — "число, используемое только один раз". Основным свойством оказии является ее уникальность. Не допускается использовать одну и ту же оказию дважды с одним и тем же ключом. Обычно оказия представляет собой номер сообщения, присвоенный ему по некоторому принципу и, возможно, скомбинированный с какой-нибудь другой информацией. Большинство современных систем уже применяют нумерацию сообщений, чтобы сохранять сообщения в правильном порядке, распознавать повторяющиеся сообщения и т.п. Оказию не обязательно сохранять в секрете, но она может быть использована только один раз. Это особенно важно для получателя сообщений, который получает оказию вместе с каждым сообщением и должен убедиться в том, что ее текущее значение не повторяется дважды.

Вектор инициализации, необходимый для шифрования первого блока текста в режиме CBC, генерируется путем шифрования оказии.

Обычно отправитель последовательно нумерует свои сообщения и пересылает их вместе с соответствующим номером. В этом случае для отправки сообщения выполните приведенную ниже последовательность действий.

1. Присвойте сообщению номер. Обычно для нумерации сообщений используется счетчик, начинающийся с 0. Обратите внимание: значение счетчика никогда не должно вернуться к нулю, так как это нарушит свойство уникальности.
2. Используйте номер сообщения для построения уникальной оказии. Для заданного ключа оказия должна быть уникальной по всей системе, а не только в рамках текущего компьютера. Например, если один и тот же ключ используется для шифрования трафика в двух направлениях, оказия должна представлять собой сочетание номера сообщения и индикатора направления, в котором отсылается сообщение. Размер оказии должен быть равен размеру блока используемого блочного шифра.

3. Зашифруйте оказию с помощью блочного шифра, чтобы сгенерировать вектор инициализации.
4. Зашифруйте сообщение в режиме CBC, используя полученный вектор инициализации.
5. Добавьте к шифрованному тексту достаточно информации, чтобы получатель мог воссоздать значение оказии. Обычно для этого к началу шифрованного текста добавляется номер сообщения. Значение самого вектора инициализации (в наших обозначениях это C_0) пересылать не нужно.
6. Убедитесь, что получатель будет принимать сообщение с каждым конкретным номером только один раз. Обычно для этого получателю необходимо отклонять сообщения, номера которых меньше или равны номеру последнего принятого сообщения.

Как правило, при использовании оказии дополнительная информация, включаемая в сообщение, намного меньше, чем при пересылке случайного вектора инициализации. Для большинства систем вполне достаточно 32- или 48-битового счетчика сообщений, в то время как размер случайного вектора инициализации составляет не менее 128 бит. На практике большинству систем обмена информацией все равно требуются счетчики сообщений, поэтому использование оказии не приведет к дополнительным расходам.

5.4 Обратная связь по выходу (OFB)

До сих пор все рассмотренные режимы принимали на вход сообщение и шифровали его путем применения блочного шифра к блокам самого сообщения. Режим *обратной связи по выходу (output feedback — OFB)* отличается тем, что в качестве входных данных для блочного шифра не используется само сообщение. Вместо этого блочный шифр применяется для генерации псевдослучайного потока байтов (называемого ключевым потоком), который с помощью операции XOR складывается с открытым текстом для получения шифрованного текста. Схема шифрования, которая генерирует подобный ключевой поток, называется *поточным шифром (stream cipher)*. Некоторым почему-то кажется, что поточные шифры очень слабы. Ни в коем случае! Они не только весьма полезны, но и работают безотказно. Просто необходимо внимательно к ним относиться. Злоупотребление поточным шифром (как правило, в форме повторного использования оказии) легко нарушает безопасность системы. В этом смысле надежнее использовать режим наподобие CBC — последний даже при повторном использовании оказии остается относительно безопасным. Тем не менее преимущества поточных шифров зачастую перевешивают их недостатки.

Схема шифрования в режиме OFB определяется следующим образом:

$$\begin{aligned} K_0 &:= \text{IV} \\ K_i &:= E(K, K_{i-1}) \quad \text{для } i = 1, \ldots, k, \\ C_i &:= P_i \oplus K_i. \end{aligned}$$

Как видите, здесь также фигурирует вектор инициализации K_0, который применяется для генерации ключевого потока $K_1, \ldots, K_k$ путем постоянного шифрования предыдущего значения K_i. Затем ключевой поток с помощью операции XOR складывается с открытым текстом сообщения для получения шифрованного текста.

Значение вектора инициализации должно быть случайным. Как и в режиме CBC, оно может быть выбрано случайным образом и передано вместе с самим шифрованным текстом (см. раздел 5.3.3) или же сгенерировано на основе оказии (см. раздел 5.3.4).

Преимущество режима OFB состоит в том, что алгоритм шифрования полностью совпадает с алгоритмом дешифрования. Это позволяет сократить расходы на их реализацию. Что особенно приятно, в данном режиме необходимо использовать только функцию шифрования блочного шифра, а значит, реализовать функцию дешифрования не нужно вообще.

Второе преимущество режима OFB — отсутствие необходимости в дополнении. Поскольку ключевой поток может быть представлен в виде бесконечной последовательности байтов, мы можем использовать столько байтов, сколько содержит наше сообщение. Другими словами, если последний блок открытого текста неполный, мы посылаем только те байты шифрованного текста, которые соответствуют реальным байтам открытого текста. Отсутствие дополнения сокращает расходы на пересылку, что особенно важно для большого количества коротких сообщений.

Помимо этого, режим OFB наглядно демонстрирует одну из проблем использования поточных шифров. Если вы *когда-нибудь* используете один и тот же вектор инициализации для двух разных сообщений, последние будут зашифрованы с помощью одного и того же ключевого потока. Это очень и очень плохо. Предположим, что два открытых текста P и P' были зашифрованы с помощью одного и того же ключевого потока. Обозначим полученные шифрованные тексты как C и C' соответственно. Получив в свое распоряжение шифрованные тексты, злоумышленник сможет подсчитать значение выражения $C_i \oplus C'_i = P_i \oplus K_i \oplus P'_i \oplus K_i = P_i \oplus P'_i$. Другими словами, он сможет определить различие между двумя открытыми текстами. Предположим, что злоумышленник уже знает один из открытых текстов. (В реальной жизни это случается сплошь и рядом.) Тогда задача вычисления второго открытого текста становится тривиальной. Существует даже несколько атак, которые

восстанавливают информацию о двух неизвестных открытых текстах исходя из различия между ними [44].

Описанная выше проблема поточных шифров еще более усугубляется в режиме OFB. Если, к нашему ужасу, значение какого-нибудь ключевого блока повторится дважды, это приведет к циклическому повторению всей дальнейшей последовательности ключевых блоков, например в одном большом сообщении. Или, скажем, значение вектора инициализации одного сообщения может совпасть со значением ключевого блока, применяемого где-нибудь в середине второго сообщения, в результате чего части открытого текста обоих сообщений окажутся зашифрованными с помощью одного и того же ключевого потока. И в том и в другом случае разные блоки сообщений шифруются с помощью одного и того же ключевого блока, что делает применяемую схему шифрования небезопасной.

Чтобы подтвердить правдоподобность данного предположения, необходимо зашифровать большое количество данных. В нашем случае речь идет об обнаружении коллизий между блоками ключевого потока и начальными значениями этих блоков. Чтобы ожидать возникновения подобной коллизии, необходимо зашифровать, как минимум, 2^{64} блока данных. Если ограничить количество данных, которые могут быть зашифрованы с помощью одного и того же ключа, можно снизить вероятность повторения значения ключевого блока. К сожалению, риск остается всегда, и в случае неудачи вы можете утратить конфиденциальность всего сообщения.

5.5 Счетчик (CTR)

Наш любимый режим шифрования с использованием блочных шифров носит название *счетчика (counter — CTR)*. Хотя данный режим известен уже на протяжении многих лет, он не был стандартизирован как официальный режим работы шифра DES [68], а потому игнорируется авторами большинства учебников по криптографии. Недавно режим CTR был стандартизирован организацией NIST [26]. Как и OFB, режим счетчика основан на применении поточного шифра. Он определяется следующим образом:

$$K_i := E(K, Nonce \parallel i) \quad \text{для } i = 1, \ldots, k,$$
$$C_i := P_i \oplus K_i.$$

Как и в любом другом поточном шифре, в режиме CTR следует использовать оказию того или иного вида (в формуле она обозначена как Nonce). Большинство систем образуют значение оказии на основе номера сообщения и некоторых дополнительных данных, чтобы гарантировать ее уникальность.

Для генерации ключевого потока в режиме CTR используется удивительно простой метод. Он состоит в конкатенации значения оказии со значением счетчика и в последующем шифровании полученного значения для формирования одного блока ключевого потока. Данный метод требует, чтобы оказия и счетчик в совокупности умещались в рамки одного блока, однако в современных шифрах со 128-битовыми блоками это не составляет проблемы. Очевидно, размер оказии должен быть меньше размера одного блока, чтобы оставить место для значения счетчика i. Типичный 128-битовый блок может состоять из 48-битового номера сообщения, 16 бит дополнительной информации, входящей в оказию, а также 64 бит для счетчика i. Такая конструкция позволяет зашифровать с помощью одного ключа не более 2^{48} сообщений, а также ограничивает размер каждого сообщения до 2^{68} байт.

Как и в режиме OFB, в режиме счетчика необходимо гарантировать, что каждая конкретная комбинация "ключ–оказия" никогда не будет использована во второй раз. Данное требование принято относить к недостаткам режима CTR, однако оно же свойственно и режиму CBC. Если один и тот же вектор инициализации будет использован дважды, это приведет к утечке информации об открытых текстах. В этом плане режим CBC несколько надежнее, так как он с большей вероятностью ограничивает объем данных, которые может получить злоумышленник. Тем не менее, любая утечка информации нарушает требования к безопасности, а при использовании модульной архитектуры нельзя рассчитывать на то, что остальные части системы ограничат размер ущерба, даже если объем утечки весьма невелик. Таким образом, при использовании и CBC и CTR необходимо гарантировать, что оказия или вектор инициализации являются уникальными.

За исключением описанных факторов, режим CTR очень прост в использовании. Требуется только реализовать функцию шифрования блочного шифра, поскольку функции шифрования и дешифрования CTR одинаковы. Режим CTR позволяет легко осуществлять доступ к произвольным частям открытого текста, поскольку любой блок ключевого потока может быть подсчитан мгновенно, независимо от других блоков. В высокоскоростных приложениях вычисление блоков ключевого потока может выполняться с любой степенью параллелизма. Более того, безопасность режима CTR тривиальным образом связана с безопасностью блочного шифра. Любое слабое место режима CTR позволяет моментально осуществить атаку с избранным открытым текстом на соответствующий блочный шифр. Из этого следует и обратное утверждение: если не существует атак на блочный шифр, не существует и атак на режим CTR (кроме утечки информации, о которой вскоре пойдет речь).

5.6 Новые режимы

Все рассмотренные режимы работы блочных шифров появились в 70-х либо начале 80-х годов прошлого века. В последние годы криптографическому сообществу было предложено еще несколько новых режимов. Наиболее известным из них, пожалуй, является режим *шифровальной книги со сдвигом (offset codebook)*, или *OCB* [82, 83]. Этот и другие новые режимы сочетают в себе как шифрование, так и аутентификацию. Хотя данные режимы очень привлекательны, мы не можем рекомендовать их по двум причинам.

Во-первых, эти режимы еще слишком новы, а ко всему новому в криптографии относятся с подозрением. Большинство новых режимов, включая и OCB, обладают доказательствами своей безопасности. Однако доверять этим доказательствам можно не больше чем любому другому продукту человеческих рук. В доказательствах безопасности уже не раз встречались ошибки. Они будут встречаться и в будущем. Зачастую подобные доказательства очень сложны, и думать, что они не содержат ошибок, могут только самые закоренелые оптимисты. (Мы еще никогда не видели попыток доказать правильность самих доказательств.)

В действительности доказательство безопасности отнюдь не доказывает, что система является безопасной. Оно обеспечивает лишь *приведение безопасности (security reduction)* режима работы блочного шифра к безопасности самого блочного шифра. Доказательство может, например, утверждать, что если шифрование в режиме OCB можно взломать за время X, то соответствующий блочный шифр может быть взломан за время Y. Это очень ценный результат, который хорошо укладывается в нашу концепцию модульного дизайна криптографических систем. Тем не менее он доказывает лишь безопасность блочного шифра, а не безопасность шифрования с использованием режима работы блочного шифра. В этом нет ничего плохого: гораздо лучше иметь такое доказательство безопасности, чем не иметь его вообще, однако оно не является абсолютным доказательством, как нам зачастую пытаются внушить.

Во-вторых, текущее состояние лицензирования патентов на эти режимы весьма неопределенно, поскольку сразу несколько сторон претендуют на патентование различных режимов. На момент написания этой книги патенты еще даже не были выданы, поэтому точные рамки каждого патента еще не известны. Использовать один из этих режимов сейчас — все равно что ходить по минному полю. В любой момент можно напороться на огромные штрафы. Мы вас предупредили.

5.7 Какой режим выбрать

Итак, мы рассмотрели несколько режимов работы блочных шифров. Однако для практического использования стоит рекомендовать лишь два из них: CBC и CTR. Как уже отмечалось, режим ECB недостаточно безопасен. Режим OFB неплох, однако в определенных аспектах уступает режиму CTR, который к тому же лишен проблемы циклических повторений ключевых блоков. Поэтому отдавать предпочтение OFB, а не CTR нет смысла.

Так какой же режим выбрать — CBC или CTR? Давайте проведем небольшое сравнение.

- **Дополнение.** Режим CBC требует дополнения сообщений, а CTR — нет.
- **Скорость.** Оба режима предполагают один и тот же объем вычислений. Тем не менее режим CTR допускает параллельный подсчет любого количества ключевых блоков, в результате чего реализации CTR могут достигать более высоких скоростей.
- **Реализация.** Работая в режиме CTR, достаточно реализовать только функцию шифрования блочного шифра. Режиму CBC требуются обе функции — шифрования и дешифрования.
- **Надежность.** Если значение оказии будет использовано во второй раз, в режиме CBC может произойти небольшая утечка информации о начальном блоке открытого текста. В режиме CTR злоумышленник сможет получить информацию обо всем сообщении.
- **Оказия.** В режиме CBC можно использовать случайный вектор инициализации или оказию. В режиме CTR для генерации ключевых блоков требуется уникальная оказия. На практике, однако, в режиме CBC практически всегда используют оказию, поэтому в данном аспекте режимы CBC и CTR можно считать эквивалентными.

Как видите, CTR превосходит CBC по всем параметрам, кроме надежности. Если вы когда-нибудь используете значение оказии во второй раз, утечка информации в режиме CTR будет гораздо существеннее. В большинстве систем разработчику придется самому задавать принцип использования оказий, поэтому гарантировать уникальность последних нетрудно. На наш взгляд, преимуществ CTR вполне достаточно, чтобы отдать предпочтение именно этому режиму, кроме тех ситуаций, где вы не можете контролировать способ применения функции шифрования.

С одной стороны, мы отмечали, что каждая часть системы должна сама обеспечивать свою безопасность и не зависеть от остальных ее частей. С другой стороны, рекомендуем использовать режим CTR, уникальность оказий

которого обеспечивается другими частями системы. Разве мы не противоречим сами себе? И да и нет. В идеале было бы предпочтительнее использовать режим работы блочного шифра, который не зависит от остальных частей системы. К сожалению, ни один из рассмотренных режимов такими свойствами не обладает. Все они в той или иной степени зависят от остальных частей системы. Это легко объяснить. Режим работы блочного шифра не является ни завершенной системой шифрования, ни независимым структурным модулем. *Блочный шифр* — это модуль, который при необходимости можно заменить другим подходящим блочным шифром. *Режим работы* блочного шифра — это всего лишь метод, который используется в модуле следующего уровня, а именно модуле шифрования и аутентификации сообщения (он рассматривается в главе 8, "Безопасный канал общения"). Поскольку режим работы блочного шифра не является отдельным модулем, мы не можем гарантировать его безопасность независимо от других частей системы.

Не забывайте, что режим работы блочного шифра обеспечивает только конфиденциальность. Другими словами, злоумышленник не может получить никакой информации о данных, которыми вы обмениваетесь с собеседником, кроме информации о том, что вы *обмениваетесь* данными, а также *когда*, *в каком объеме* и *с кем* вы обмениваетесь данными[1]. Шифрование данных ни в коей мере не помешает злоумышленнику изменить эти данные.

5.8 Утечка информации

Вот мы и подобрались к страшному секрету режимов работы блочных шифров. Все режимы допускают утечку информации. Ни один из них не совершенен. Анализ данного аспекта крайне редко встречается в литературе по криптографии, поэтому мы решили включить его в нашу книгу.

Проводя анализ режимов, будем исходить из предположения, что у нас есть идеальный блочный шифр. Тем не менее даже в этом случае шифрованный текст, полученный в результате применения режима работы блочного шифра, будет в той или иной мере воспроизводить структуру открытого текста. Это связано с вопросами равенства и неравенства блоков открытого и шифрованного текста.

Начнем с режима ECB. Если для блоков открытого текста P_i и P_j выполняется равенство $P_i = P_j$, тогда для соответствующих блоков шифрованного текста справедливо $C_i = C_j$. Случайные блоки открытого текста редко бывают равны друг другу, однако на практике открытый текст не является

[1] Подобный тип криптоанализа называется *анализом потока данных (traffic analysis)*. Он может снабдить злоумышленника очень ценной информацией. Помешать анализу потока данных можно, однако в терминах пропускной способности это оказалось бы слишком дорого для всех нас (кроме, пожалуй, вооруженных сил).

случайным, наоборот — он высоко структурирован. В таком тексте одинаковые блоки встречаются довольно часто, а соответствующие одинаковые блоки шифрованного текста хорошо воспроизводят эту структуру для злоумышленника. Вот почему мы не рекомендуем использовать режим ECB.

А что же режим CBC? В этом случае одинаковые блоки открытого текста не будут преобразованы в одинаковые блоки шифрованного текста, поскольку перед шифрованием каждый блок открытого текста складывается с помощью операции XOR с предыдущим блоком шифрованного текста. Все блоки шифрованного текста можно представить себе в виде случайных значений; в конце концов, они были получены посредством блочного шифра, генерирующего случайные выходные данные по заданным входным данным. Но что, если у нас окажется два одинаковых блока шифрованного текста? Рассмотрим следующее равенство:

$$
\begin{aligned}
C_i &= C_j, \\
E(K, P_i \oplus C_{i-1}) &= E(K, Pj \oplus C_{j-1}) && \text{— из спецификации режима CBC,} \\
P_i \oplus C_{i-1} &= P_j \oplus C_{j-1} && \text{— расшифруем обе стороны,} \\
P_i \oplus P_j &= C_{i-1} \oplus C_{j-1} && \text{— один из фундаментальных законов алгебры.}
\end{aligned}
$$

Последнее равенство показывает, что разница между блоками открытого текста равна сумме XOR соответствующих блоков шифрованного текста, которые, как можно было предположить, известны злоумышленнику. Разумеется, это совсем не то, чего мы ожидали от совершенной системы шифрования. А если открытый текст многословен (как, например, обычный текст на английском языке), он, вероятно, содержит достаточно информации, чтобы злоумышленник мог восстановить оба блока открытого текста.

Аналогичная ситуация случается и тогда, когда два блока шифрованного текста не равны друг другу. Зная, что $C_i \neq C_j$, получаем, что $P_i \oplus P_j \neq C_{i-1} \oplus C_{j-1}$, поэтому неравенство блоков шифрованного текста означает неравенство блоков открытого текста.

Подобными свойствами обладает и режим CTR. Известно, что все блоки K_i неодинаковы, так как получены путем применения шифрования к конкатенации значений оказии и счетчика. Поскольку все значения такого открытого текста будут разными, значения шифрованного текста (которые формируют ключевые блоки) тоже будут разными. Для двух любых блоков шифрованного текста C_i и C_j можно сказать, что $P_i \oplus P_j \neq C_i \oplus C_j$, так как в противном случае два ключевых блока были бы одинаковыми. Другими словами, режим CTR обеспечивает неравенство открытого текста для каждой пары блоков шифрованного текста.

В режиме CTR нет проблем с коллизиями. Два ключевых блока никогда не будут одинаковыми, а равенство блоков открытого или шифрованного текста не означает ровным счетом ничего. Единственное, что отличает CTR от идеального поточного шифра, — это отсутствие коллизий ключевых блоков.

Режим OFB хуже, чем CBC или CTR. Пока между блоками ключевого потока не возникает коллизий, режим OFB приводит к утечке такого же объема информации, как и режим CTR. Тем не менее, если между ключевыми блоками режима OFB когда-нибудь возникнет коллизия, она приведет к коллизии всех последующих ключевых блоков. Это катастрофа с точки зрения безопасности, а также основная причина того, почему CTR предпочтительнее режима OFB.

5.8.1 Вероятность коллизии

Итак, какова же вероятность того, что два блока шифрованного текста окажутся равными? Предположим, мы зашифровали M блоков открытого текста. То, как именно они были организованы — в несколько больших сообщений или в большое количество коротких сообщений, не имеет значения. Нас интересует только общее количество блоков. Из полученных блоков по довольно грубой оценке можно сформировать примерно $M(M-1)/2$ пар блоков. Вероятность того, что значения блоков в паре совпадут, равна 2^{-n}, где n — это размер блока используемого блочного шифра. Итак, ожидаемое количество пар одинаковых блоков шифрованного текста равняется $M(M-1)/2^{n+1}$, что приближается к единице при $M \approx 2^{n/2}$. Другими словами, когда будет зашифровано около $2^{n/2}$ блоков текста, можно ожидать появления двух одинаковых блоков шифрованного текста[2]. Если размер блока равен 128 бит, ожидать первого повторения блока шифрованного текста можно, зашифровав около 2^{64} блоков. Это и есть парадокс задачи о днях рождения, о котором идет речь в разделе 3.6.6. На сегодняшний момент 2^{64} блоков — это большое количество данных, однако не забывайте, что мы разрабатываем системы с расчетом на 30 лет службы. Возможно, в будущем кому-то понадобится шифровать сообщения и такой длины.

Меньшие объемы данных тоже подвергаются риску. Если обработать 2^{40} блоков текста (около 16 Тбайт данных), вероятность коллизии блоков шифрованного текста будет равна 2^{-48}. Это действительно очень малая вероятность, но давайте посмотрим на нее с точки зрения злоумышленника. Для конкретного ключа шифрования он собирает 2^{40} блоков шифрованного текста и проверяет, нет ли среди них повторений. Поскольку вероятность найти

[2] В действительности количество блоков, которые следует зашифровать прежде, чем ожидать первого повторения, близко к $\sqrt{\pi 2^{n-1}} = 2^{n/2}\sqrt{\pi/2}$, однако доказательство этого факта достаточно громоздко, а в нашей ситуации такой уровень точности не нужен.

повторяющиеся блоки очень мала, злоумышленник вынужден повторить этот процесс примерно для 2^{48} разных ключей. Общее количество работы, которую приходится проделать злоумышленнику, чтобы обнаружить коллизию, составляет $2^{40} \cdot 2^{48} = 2^{88}$, что намного меньше, чем заявленный нами 128-битовый уровень безопасности системы.

Давайте сконцентрируем внимание на режимах CBC и CTR. В режиме CTR мы получаем неравенство открытого текста для каждой пары блоков шифрованного текста. В режиме CBC мы получаем неравенство открытого текста, если блоки шифрованного текста не равны, и равенство, если они равны. Очевидно, равенство предоставляет злоумышленнику гораздо больше информации об открытом тексте, нежели неравенство, поэтому при обнаружении коллизии утечка информации в режиме CTR меньше.

5.8.2 Как бороться с утечкой информации

Как же удается достигнуть 128-битового уровня безопасности? Вообще-то не удается. Не существует легкого способа обеспечить 128-битовый уровень безопасности при использовании блочного шифра с размером блока, равным 128 бит. Вот почему нам так нужны блочные шифры с 256-битовыми блоками. К сожалению, подобных шифров пока еще нет. Все, что нам остается, — это приблизиться к желаемому уровню безопасности и попытаться ограничить размер ущерба, связанного с утечкой информации.

В режиме CTR объем утечки очень небольшой. Предположим, мы зашифровали 2^{64} блоков данных и получили шифрованный текст C. Для любого открытого текста P длиной 2^{64} блока злоумышленник может вычислить ключевой поток, который следует применить, чтобы преобразовать указанный текст P в C. Вероятность того, что полученный ключевой поток будет содержать коллизию, примерно 50%. Известно, что в режиме CTR никогда не случается коллизий, поэтому если коллизия все же возникнет, данный текст P может быть исключен из рассмотрения. Таким образом, злоумышленник может исключить из рассмотрения примерно половину всех возможных открытых текстов. Это соответствует утечке одного бита информации, что совсем немного. Если же мы ограничим размер шифруемого текста 2^{48} блоками, злоумышленник сможет исключить лишь $1/2^{32}$ всех открытых текстов, что практически не дает ему никакой информации. На практике подобная утечка несущественна. Итак, хотя режим CTR и не является совершенным, можно избежать ущерба, вызванного утечкой информации, просто ограничив объем данных, которые могут быть зашифрованы с помощью одного и того же ключа. Вполне разумно ограничить этот объем данных 2^{60} блоками, что позволит применить один ключ для шифрования 2^{64} байт данных и при этом снизит объем утечки до небольшой доли бита.

Использование режима CBC требует более жестких ограничений. Если в режиме CBC произойдет коллизия, злоумышленник получит около 128 бит информации об открытом тексте. Поэтому вероятность подобной коллизии должна быть как можно более низкой. Рекомендуем ограничить размер данных, которые могут быть зашифрованы с помощью одного и того же ключа в режиме CBC, примерно 2^{32} блоками. При этом остаточная вероятность утечки 128 бит информации составляет 2^{-64}, что вполне безобидно для большинства областей применения, но, конечно же, далеко от желаемого 128-битового уровня безопасности.

Еще раз отметим: приведенные выше ограничения касаются общего объема информации, которая может быть зашифрована с помощью одного и того же ключа. То, как именно организована эта информация — в виде одного очень большого сообщения или же множества небольших сообщений, значения не имеет.

Подобное положение дел сложно назвать удовлетворительным, но такова суровая правда жизни. Лучшее, что можно сделать в данной ситуации, — это использовать режим CTR или CBC и ограничить объем данных, которые могут быть зашифрованы с помощью одного и того же ключа. Позднее мы поговорим о протоколах согласования ключей. Установить новый ключ, когда лимит использования старого практически исчерпан, совсем нетрудно. Если вы уже использовали протокол согласования ключей для выбора ключа шифрования, обновить этот ключ не так уж сложно; это просто несколько затрудняет дело. Нельзя сказать, что нам это нравится, однако на данный момент лучшего решения еще не придумано.

5.8.3 О наших вычислениях

Читатели с математическим образованием наверняка шокированы тем, как легко мы обращаемся с вероятностями, не утруждая себя проверкой, являются ли те независимыми. С формальной точки зрения они, разумеется, правы. Тем не менее криптографы, как и физики, используют математический аппарат в своих собственных целях. Криптографические величины обычно ведут себя случайным образом. В конце концов, криптографы делают многое, чтобы разрушить зависимости между открытым и шифрованным текстом, так как наличие любых зависимостей делает системы уязвимыми. Практика показывает, что подобное “легкомысленное” обращение с вероятностями дает весьма точные результаты. Мы всячески поддерживаем математиков, пытающихся досконально вникнуть в детали и получить более точные результаты, но сами предпочитаем более грубые приближения по причине их простоты.

Глава 6

Функции хэширования

Функции хэширования поистине универсальны. Они могут применяться для шифрования, аутентификации и даже в простых схемах цифровых подписей[1].

Функция хэширования — это функция, которая принимает на вход строку битов (или байтов) произвольной длины и выдает результат фиксированной длины. Стандартной областью применения функций хэширования являются цифровые подписи. Для обеспечения аутентификации можно подписывать само сообщение m. Тем не менее в большинстве схем цифровых подписей операции с открытым ключом обходятся довольно дорого в плане вычислительных ресурсов. Поэтому, вместо того чтобы подписывать само сообщение m, можно применить к нему функцию хэширования и подписать значение $h(m)$. Результат применения функции h обычно составляет от 128 до 512 бит в длину, что совсем немного в сравнении с тысячами или миллионами бит сообщения m. Таким образом, подписать значение $h(m)$ гораздо быстрее, чем подписать непосредственно сообщение m. Чтобы данная конструкция была безопасной, функция хэширования должна удовлетворять следующему условию: невозможно построить два разных сообщения m_1 и m_2, результаты хэширования которых были бы одинаковы. Более подробно свойства безопасности функций хэширования рассматриваются несколько ниже.

Иногда функции хэширования называют функциями *профилей сообщений (message digest)*, а результат применения функции хэширования — *профилем (digest)* или *цифровым отпечатком пальца (digital fingerprint)*. Мы же предпочитаем более распространенное название *функция хэширования (hash function)*, поскольку, помимо создания профилей сообщений, эти функции имеют массу областей применения. Следует также предупредить вас о возмож-

[1] Самая первая схема цифровой подписи, разработанная Лесли Лэмпортом (Leslie Lamport), принадлежит именно к этому типу [21].

ной путанице: термином “функция хэширования” иногда называют функцию отображения, которая применяется для доступа к хэш-таблицам — структуре данных, используемой во многих алгоритмах. Эти так называемые функции хэширования схожи с криптографическими функциями хэширования, однако между ними есть существенное различие. Функции хэширования, используемые в криптографии, обладают особыми свойствами безопасности. К функциям отображения хэш-таблиц предъявляются более слабые требования. Никогда не путайте эти функции. В нашей книге речь идет только о криптографических функциях хэширования.

В криптографии функции хэширования имеют множество применений. Они выполняют роль “связующего звена” между различными частями криптографической системы. Если вас не устраивает значение с переменной длиной, вы можете применить к нему функцию хэширования, чтобы превратить его в значение с фиксированной длиной. Функции хэширования могут применяться в качестве криптографических генераторов псевдослучайных чисел для создания нескольких ключей на основе одного общего секретного ключа. И наконец, функции хэширования обладают свойством односторонности, благодаря которому разные части системы оказываются изолированными друг от друга, поэтому, даже если злоумышленнику удастся узнать какое-нибудь одно значение, он не сможет вычислить по нему другие.

Несмотря на повсеместное применение функций хэширования, мы знаем о них гораздо меньше, чем о блочных шифрах. Это одно из слабых мест криптографического сообщества. В сравнении с блочными шифрами функции хэширования крайне мало исследованы, а практические предложения функций хэширования весьма немногочисленны.

6.1 Безопасность функций хэширования

Как уже отмечалось, функция хэширования отображает строку m произвольной длины на значение $h(m)$ фиксированной длины. Обычно длина результата функции хэширования составляет от 128 до 512 бит. К функции хэширования предъявляется несколько требований. Самое простое из них — *односторонность (one-way property)*: для любого сообщения m легко вычислить значение $h(m)$, однако для любого значения x невозможно найти такое m, что $h(m) = x$. Другими словами, сама односторонняя функция вычисляется довольно легко, а вычисление функции, обратной по отношению к ней, осуществить практически невозможно (отсюда и название “односторонняя”).

Среди многих других свойств, которыми должна обладать хорошая функция хэширования, наиболее часто упоминают *сопротивляемость коллизиям (collision resistance)*. Последнее требование строже, чем свойство односторон-

ности. Коллизией по отношению к функциям хэширования называют два разных значения m_1 и m_2, для которых $h(m_1) = h(m_2)$. Разумеется, каждая функция хэширования обладает бесконечным числом подобных коллизий. (Существует бесконечное число возможных входных значений и только конечное число возможных выходных значений.) Таким образом, функция хэширования никогда не будет бесконфликтной. Свойство сопротивляемости коллизиям означает лишь то, что, хотя коллизии и существуют, их невозможно обнаружить.

Свойство сопротивляемости коллизиям делает функции хэширования пригодными для использования в схемах цифровых подписей. Тем не менее существует масса функций хэширования, обладающих сопротивляемостью к коллизиям и совершенно не подходящих для других областей применения, таких, как генерация ключей, односторонние функции и т.п. На практике разработчики криптографических систем нуждаются в функции хэширования, которая представляет собой случайное отображение. Поэтому необходимо, чтобы функция хэширования была неотличима от случайного отображения. Любое другое определение приводит к возникновению ситуации, в которой разработчик больше не может обращаться с функцией хэширования как с идеальным "черным ящиком" и должен просчитывать, как ее свойства будут взаимодействовать с остальными частями системы.

Определение 4 *Идеальная функция хэширования — это случайное отображение всех возможных входных значений на множество возможных выходных значений.*

Это определение, как и сформулированное ранее определение идеального блочного шифра (см. раздел 4.3), является неполным. С формальной точки зрения не существует такого понятия, как случайное отображение; мы можем говорить только о вероятностном распределении на множестве всех возможных отображений. Тем не менее для наших целей такое определение вполне подойдет.

Теперь можно дать определение атаке на функцию хэширования.

Определение 5 *Атака на функцию хэширования — это нетривиальный метод, позволяющий обнаружить различие между функцией хэширования и идеальной функцией хэширования.*

В этом определении идеальная функция хэширования, очевидно, должна иметь тот же размер выходных данных, что и функция хэширования, подвергающаяся атаке. Как и при рассмотрении блочных шифров, требование "нетривиальный" касается всех типов универсальных атак. Наши замечания относительно тривиальных атак на блочные шифры справедливы и здесь.

Осталось определить лишь то, какой объем работы придется выполнить злоумышленнику. В отличие от блочного шифра, у функции хэширования нет ключа. Кроме того, для функции хэширования не существует универсальной атаки наподобие перебора всех вариантов ключа. Основное внимание здесь следует уделить размеру выходных данных. Одной из универсальных атак на функцию хэширования является атака, в основе которой лежит парадокс задачи о днях рождения. Как известно, такая атака направлена на обнаружение коллизий. Для функции хэширования с n-битовыми выходными данными возникновения коллизии следует ожидать примерно через $2^{n/2}$ шагов. Между тем коллизии актуальны только для определенных областей применения функций хэширования. В других случаях целью злоумышленника может стать поиск прообраза (для заданного x найти такое m, что $h(m) = x$) либо выявление некоторых структурных закономерностей в выходных данных функции хэширования. Универсальная атака на прообраз требует выполнения около 2^n шагов. Мы не собираемся подробно рассматривать то, какие атаки являются актуальными для конкретной области применения функции хэширования и сколько времени может потратить различитель на осуществление конкретного типа атаки. Чтобы различитель имел практический смысл, он должен быть более эффективным, чем универсальная атака, направленная на достижение аналогичных результатов. К сожалению, это не точное определение, но мы не знаем, как сформулировать его точнее. Если кто-нибудь заявит об изобретении нового типа атаки, просто спросите себя, нельзя ли получить такой же или лучший результат с помощью универсальной атаки, не учитывающей специфику функции хэширования. Положительный ответ на этот вопрос будет означать, что новый различитель совершенно бесполезен. Если же ответ отрицателен, значит, новый различитель действительно имеет право на существование.

Как и при рассмотрении блочных шифров, мы допускаем наличие сниженного уровня безопасности, если это указано явно. Мы можем представить себе 512-битовую функцию хэширования, которая обеспечивает уровень безопасности 128 бит. В этом случае различитель должен добиться успеха не более чем за 2^{128} шагов.

6.2 Современные функции хэширования

Рассмотрим первую практическую проблему. На свете слишком мало хороших функций хэширования. На данный момент наш выбор ограничен лишь семейством функций SHA да еще, пожалуй, функцией MD5. Существуют и другие опубликованные предложения, но ни одно из них не привлекло внимания критиков настолько, чтобы ему можно было доверять. Вообще говоря,

даже функции SHA еще не были проанализированы так, как нужно, но они по крайней мере были разработаны Управлением национальной безопасности (National Security Agency — NSA) и стандартизированы NIST[2].

Практически все современные функции хэширования (и все функции, которые рассматриваются в этой главе) являются итеративными. Итеративные функции хэширования разбивают входное значение на последовательность блоков фиксированного размера $m_1, \ldots, m_k$, используя некоторое правило дополнения для заполнения последнего блока. Затем блоки сообщения обрабатываются по порядку с помощью функции сжатия h' и промежуточных состояний фиксированного размера. Этот процесс начинается с фиксированного значения H_0 и определяется как $H_i = h'(H_{i-1}, m_i)$. Последнее значение H_k и будет результатом функции хэширования.

Итеративный алгоритм хэширования имеет существенные практические преимущества. Во-первых, его намного легче определить и реализовать по сравнению с функциями, которые напрямую работают со значениями переменной длины. Во-вторых, итеративная структура позволяет начинать вычисление хэш-кода сообщения, как только у нас появится хотя бы часть этого сообщения. Благодаря этому в приложениях, работающих с поточными данными, хэшировать сообщение можно прямо "на лету", не сохраняя данные в буфере.

Как и при рассмотрении блочных шифров, не будем вдаваться в детали работы функций хэширования. Разработчики криптографических систем смогут найти полные спецификации функций хэширования в указанной нами литературе или же в Internet. Для всех остальных полные спецификации — это ненужные подробности, которые лишь отвлекают читателя от основной идеи книги.

6.2.1 MD5

Разработанная Роном Райвестом (Ron Rivest) [81] 128-битовая функция хэширования MD5 является дальнейшим усовершенствованием функции MD4 [78] и обладает дополнительной стойкостью к атакам[3].

Согласно алгоритму MD5 исходное сообщение разбивается на блоки по 512 бит. Последний блок дополняется до нужной длины, после чего к нему дописывается длина исходного сообщения в битах. Функция MD5 использует 128-битовое промежуточное состояние, которое разбивается на четыре 32-битовых слова. Функция сжатия h' состоит из четырех раундов, в каждом

[2] Что бы вы ни думали об Управлении национальной безопасности, его наработки в области криптографии довольно неплохи.

[3] Хотя функция хэширования MD4 очень быстрая, она довольно легко взламывается [24], поэтому не стоит ее использовать.

из которых выполняется перемешивание блока сообщения и промежуточного состояния. Перемешивание представляет собой комбинацию операции сложения, операций XOR, AND, OR и операций циклического сдвига битов над 32-битовыми словами. (Более подробно это рассматривается в [81].) В каждом раунде целый блок сообщения перемешивается с промежуточным состоянием, поэтому каждое слово сообщения фактически используется четыре раза. После четырех раундов функции h' входное промежуточное состояние и результат складываются для получения выходного значения функции h'.

Описанный алгоритм оперирования 32-битовыми словами весьма эффективен в системах с 32-разрядной архитектурой. Данный подход был впервые применен в алгоритме MD4 и сейчас широко используется во многих криптографических функциях.

В основе итеративного подхода к построению функций хэширования лежит следующая идея: если функция h' обладает сопротивляемостью коллизиям, тогда функция хэширования h, построенная на основе h', тоже будет обладать сопротивляемостью коллизиям. В конце концов, любая коллизия в h может произойти только при возникновении коллизии в h'. Недостатком алгоритма MD5 является то, что у его функции сжатия h' были выявлены коллизии [20]. На данный момент известных атак на саму функцию MD5 еще нет, однако наличие коллизий в функции сжатия делает использование MD5 потенциально небезопасным.

Для большинства областей применения 128-битового размера хэш-кода MD5 явно недостаточно. Используя парадокс задачи о днях рождения, можно найти реальную коллизию для функции MD5 примерно за 2^{64} оцениваний функции хэширования, что совершенно не подходит для современных систем. Поэтому мы не советуем использовать MD5.

6.2.2 SHA-1

Защищенный алгоритм хэширования (Secure Hash Algorithm — SHA) разработан Управлением национальной безопасности США (National Security Agency — NSA) и стандартизирован институтом NIST [70]. Первая версия этого алгоритма называлась просто SHA (теперь ее часто называют SHA-0) и имела весьма существенный недостаток. В NSA обнаружили этот недостаток и разработали метод его исправления. Это было опубликовано NIST в качестве улучшенной версии алгоритма SHA под названием SHA-1. Тем не менее NIST не привел никаких сведений о найденном недостатке. Через три года Шабо (Chabaud) и Жу (Joux) опубликовали статью о слабом месте алгоритма SHA-0 [16]. Эта ошибка была исправлена в алгоритме SHA-1, поэтому можно предположить, что речь идет именно о том загадочном недостатке, который был обнаружен NSA.

SHA-1 — это 160-битовая функция хэширования, основанная на алгоритме MD4. Наличие общего "предшественника" делает SHA-1 весьма схожей с MD5; однако SHA-1 обладает более консервативной структурой и работает в два-три раза медленнее, чем MD5. Тем не менее никаких проблем безопасности, связанных с SHA-1, пока не возникло, а потому данная функция получила самое широкое применение.

Функция SHA-1 использует 160-битовое промежуточное состояние, которое разбивается на пять 32-битовых слов. Как и MD5, она состоит из четырех раундов, представляющих собой комбинацию элементарных операций над 32-битовыми словами. Вместо того чтобы обрабатывать каждый блок сообщения по четыре раза, SHA-1 использует линейную рекуррентную функцию, чтобы "растянуть" 16 слов блока сообщения до нужных ей 80 слов. Это обобщение метода, используемого в MD4. В MD5 каждый бит сообщения используется функцией перемешивания по четыре раза. В SHA-1 наличие линейной рекуррентной функции гарантирует, что каждый бит сообщения используется функцией перемешивания по меньшей мере десяток раз. Что интересно, единственным отличием SHA-1 от SHA-0 стало добавление к линейной рекуррентной функции циклического сдвига на один бит.

Основным недостатком SHA-1 является 160-битовый размер результата функции хэширования. Для генерации коллизий достаточно выполнить лишь 2^{80} шагов, что значительно ниже уровня безопасности современных блочных шифров с размерами ключей от 128 до 256 бит. Этого недостаточно и для заявленного нами 128-битового уровня безопасности криптографических систем.

6.2.3 SHA-256, SHA-384 и SHA-512

Несколько лет назад NIST опубликовал черновой стандарт, содержащий три новых функции хэширования [74], которые выдают 256-, 384- и 512-битовые результаты соответственно. Эти функции разработаны для применения с 128-, 192- и 256-битовыми ключами алгоритма AES. Структура этих функций очень схожа со структурой SHA-1.

Эти функции хэширования слишком новы. Мы не хотели бы рекомендовать их для практического применения, однако альтернативы все равно нет. Чтобы достигнуть уровня безопасности, превышающего тот, который в состоянии обеспечить SHA-1, требуется функция хэширования с результатом большей длины. Ни один из опубликованных алгоритмов хэширования с результатами большего размера еще не был достаточно проанализирован в публичных источниках. Что же касается функций семейства SHA, они по крайней мере были исследованы в NSA, а эта организация вроде бы знает, что делает.

Функция SHA-256 работает намного медленнее, чем SHA-1. Хэширование длинных сообщений с помощью SHA-256 занимает примерно столько же времени, как шифрование сообщения с помощью AES или Twofish, а может, и немного больше. Это не так уж плохо. Поскольку для криптографического сообщества хэширование оказалось более сложной проблемой, чем шифрование, не удивительно, что функция хэширования работает медленнее, чем функция шифрования. Напротив, удивляет высокая скорость работы SHA-1 и MD5. С другой стороны, возможности осуществления атак на эти быстрые функции хэширования исследованы очень мало — во всяком случае несопоставимо меньше, чем возможности нападения на блочные шифры.

Функция SHA-384 довольно бесполезна. Для вычисления ее результата необходимо проделать столько же работы, как и для функции SHA-512, а затем отбросить некоторые биты. Непонятно, зачем для этого понадобилось вводить отдельную функцию. Рекомендуем придерживаться функций SHA-256 и SHA-512.

6.3 Недостатки функций хэширования

К сожалению, все описанные функции хэширования имеют ряд недостатков, которые вынуждают забраковать эти функции в соответствии с нашим определением безопасности.

6.3.1 Удлинение сообщения

Один из самых серьезных недостатков всех рассмотренных функций хэширования — удлинение сообщения. Этот недостаток приводит к реальным проблемам, а между тем его было бы так легко избежать. Вот в чем состоит проблема. Некоторое сообщение m разбивается на блоки $m_1, \ldots, m_k$ и хэшируется, в результате чего получается значение H. Теперь возьмем сообщение m', которое разбивается на блоки $m_1, \ldots, m_k, m_{k+1}$. Поскольку первые k блоков сообщения m' идентичны первым k блокам сообщения m, значение $h(m)$ соответствует промежуточному результату, полученному при вычислении $h(m')$ после обработки первых k блоков сообщения m'. Получается, что $h(m') = h'(h(m), m_{k+1})$. Используя MD5 или любую функцию семейства SHA, необходимо конструировать сообщение m' таким образом, чтобы в его состав включались биты дополнения и поле длины сообщения. Впрочем, это не проблема, поскольку метод построения таких полей широко известен.

Проблема удлинения сообщения возникла из-за того, что в конце вычисления результата функции хэширования не выполняется никакой специальной обработки данных. Как следствие этого, значение $h(m)$ снабжает злоумышленника самой непосредственной информацией относительно промежуточного состояния, полученного после сжатия первых k блоков сообщения m'.

Данное свойство, очевидно, никак не укладывается в концепцию случайного отображения, которым, как мы привыкли думать, является функция хэширования. Что еще более печально, это свойство автоматически выбраковывает все рассмотренные функции хэширования в соответствии с нашим определением безопасности. Все, что нужно различителю, — это построить несколько подходящих пар (m, m') и проверить функцию хэширования на наличие упомянутого свойства. Данное свойство, конечно же, не может быть присуще идеальной функции хэширования, поэтому соответствующий различитель можно классифицировать как атаку. Сама же атака потребует от злоумышленника вычисления всего нескольких значений функции хэширования, а следовательно, является очень быстрой.

Как данное свойство может повредить системе? Представим себе систему, в которой пользователь А отсылает сообщение пользователю Б и хочет аутентифицировать его, послав значение $h(X \parallel m)$, где X — это некий секретный ключ, известный только пользователям А и Б, а m — сообщение. Если бы h была идеальной функцией хэширования, данную систему аутентификации можно было бы считать вполне надежной. К сожалению, благодаря свойству удлинения злоумышленник Е может присоединить к сообщению m свой собственный текст и обновить код аутентичности h в соответствии с новым сообщением. Система аутентификации, которая позволяет злоумышленнику изменять сообщение, конечно же, совершенно непригодна с точки зрения безопасности.

6.3.2 Коллизия при частичном хэшировании сообщений

Второй недостаток функций хэширования связан с наличием у большинства из них итеративной структуры. Поясним это на примере конкретного различителя.

Первым шагом любого различителя является задание условий, при которых будет проводиться поиск различия между функцией хэширования и идеальной функцией хэширования. Иногда эти условия очень просты: дана функция хэширования, требуется найти коллизию. Попытаемся немного усложнить условия работы различителя. Предположим, у нас есть система, которая проводит аутентификацию сообщения m с помощью значения $h(m \parallel X)$, где X — ключ аутентификации. Злоумышленник может выбрать сообщение m, однако система позволяет аутентифицировать только одно сообщение[4].

При использовании идеальной функции хэширования с результатом размера n уровень безопасности описанной выше системы будет равен n бит.

[4]Большинство систем позволяют аутентифицировать только ограниченное количество сообщений; наш пример — это всего лишь предельный случай, когда количество сообщений равно единице. На практике многие системы отсылают вместе с сообщением его номер, что для данного типа атак равноценно возможности выбора только одного сообщения.

Злоумышленнику удастся всего лишь выбрать сообщение m, аутентифицировать его в системе с помощью значения $h(m \parallel X)$ и затем попытаться найти X путем полного перебора вариантов. С итеративной функцией хэширования, однако, перед злоумышленником открываются более радужные перспективы. Он находит строки m и m', которые при хэшировании функцией h приводят к коллизии. Используя атаку, в основе которой лежит парадокс задачи о днях рождения, это можно выполнить всего лишь примерно за $2^{n/2}$ шагов. Затем злоумышленник аутентифицирует в системе сообщение m и заменяет его сообщением m'. Напомним, что значение h вычисляется итеративно, поэтому, если при хэшировании части второго сообщения возникнет коллизия, а все оставшиеся входные значения будут такими же, как и для первого сообщения, значение хэш-кода тоже не изменится. Поскольку хэширование сообщений m и m' дает одно и то же значение, $h(m \parallel X) = h(m' \parallel X)$ для всех X.

Мы привели пример типичного различителя. Он устанавливает собственные "правила игры" (условия, при которых он пытается осуществить атаку) и затем атакует систему. Цель, как и прежде, состоит в том, чтобы найти различие между функцией хэширования и идеальной функцией хэширования, но в данном случае это очень легко. Если атака завершится удачно, значит, перед нами итеративная функция хэширования, а если неудачно — идеальная функция хэширования.

6.4 Исправление недостатков

Нам нужна функция хэширования, которая будет вести себя как случайное отображение. К сожалению, все известные функции хэширования данному требованию не соответствуют. Неужели придется выполнять проверку на наличие проблемы удлинения сообщения во всех местах, где используется функция хэширования? А как насчет коллизий при частичном хэшировании сообщений? И есть ли у функций хэширования еще какие-нибудь слабые стороны, которые нужно проверять?

Оставлять функцию хэширования такой, как она есть, со всеми недостатками — весьма неудачная идея. Поверьте нам: всегда найдется такой способ использования функции хэширования, который их раскроет. Даже если вы задокументируете все известные недостатки, их не будут проверять в реальных системах. Более того, если бы вы могли контролировать процесс разработки системы, то обязательно столкнулись бы с проблемой сложности. Предположим, функция хэширования имеет три слабых места, блочный шифр — два, схема цифровой подписи — четыре и т.д. Чтобы убедиться в этом, придется проверить сотни вариантов взаимодействия этих слабых мест, что прак-

тически невозможно. Поэтому необходимо исправить саму функцию хэширования.

Следует отметить, что это нестандартная точка зрения. Большинство криптографов вполне устраивают итеративные функции хэширования; описанные проблемы не устраняются даже в новых системах. Некоторые пользователи функций хэширования предпринимают массу усилий, чтобы избежать проблем. Большинство же удачливых пользователей так и живут, оставаясь в счастливом неведении относительно того, какие неприятности могли с ними приключиться. Между тем неприятности все же случаются и довольно часто. Каждая из этих проблем может быть исправлена путем применения к системе какого-нибудь специального метода, но нам кажется, что это неправильно и следует исправить саму функцию хэширования.

6.4.1 Полное исправление

До сих пор нам не встречалась литература, посвященная исправлению недостатков функций хэширования. Здесь лишь изложены выводы, к которым мы пришли в процессе написания этой книги. Все проблемы могут быть устранены, если использовать функцию хэширования дважды. Мы убеждены, что данное решение позволяет избавиться от всех упомянутых недостатков функций хэширования. Но разработка криптографических функций — дело сложное, поэтому дать стопроцентную гарантию пока нельзя. Наше решение еще не было проанализировано другими экспертами, что обычно занимает долгие годы (если, конечно, им кто-то заинтересуется). Такова жизнь.

Пусть h — одна из рассмотренных функций хэширования. Вместо того чтобы использовать в качестве функции хэширования $m \mapsto h(m)$, будем использовать функцию $m \mapsto h(h(m) \parallel m)$[5]. Как видите, перед хэшированием сообщения m к его началу было дописано значение $h(m)$. Благодаря этому итеративное вычисление хэш-кода будет сразу же зависеть от всех битов сообщения, что сделает невозможными атаки, осуществляемые путем частичного хэширования или путем удлинения сообщений.

Определение 6 *Пусть h — итеративная функция хэширования. Тогда функция хэширования h_{DBL} определяется выражением $h_{DBL}(m) := h(h(m) \parallel m)$.*

Если h является одной из рассмотренных ранее функций хэширования, то мы убеждены, что предложенная конструкция будет обладать уровнем безопасности в n бит, где n — размер результата функции хэширования.

[5]Запись $x \mapsto f(x)$ — еще один способ обозначения функции. Его применяют тогда, когда не хотят присваивать функции имя. Например, $x \mapsto x^2$ — это функция, которая возводит в квадрат свой аргумент.

Недостатком этого подхода является низкая скорость работы. Сообщение приходится хэшировать дважды, что занимает в два раза больше времени по сравнению с обычным хэшированием. Лично нас это не огорчает, потому что мы всегда ставим безопасность выше скорости. В мире и так достаточно быстрых небезопасных систем, так зачем же разрабатывать еще одну? Тем не менее скорость хэширования является "узким местом" производительности некоторых приложений, поэтому нужно придумать что-нибудь более эффективное.

Еще одним недостатком этого подхода является необходимость буферизации всего сообщения m. Мы больше не сможем подсчитывать хэш-код потока данных по мере их поступления. Некоторые приложения нуждаются в этой возможности, и функция h_{DBL} для них просто не подойдет.

6.4.2 Более эффективное исправление

Как же сохранить полную скорость исходной функции хэширования? Здесь можно немного схитрить. Вместо $h(m)$ можно использовать функцию $h(h(m))$ и объявить, что ее уровень безопасности составляет всего $n/2$ бит. Хитрость состоит в том, что обычно от n-битовой функции хэширования ожидают обеспечения уровня безопасности n бит в тех ситуациях, где атака на основе коллизий невозможна[6]. Все атаки на основе коллизий при частичном хэшировании сообщений используют парадокс задачи о днях рождения, поэтому, если снизить уровень безопасности до $n/2$ бит, эти атаки больше не будут подпадать под заявленный уровень безопасности.

В большинстве ситуаций подобное снижение уровня безопасности было бы неприемлемым, но нам повезло. Функции хэширования изначально разрабатывались с учетом возможности осуществления атак на основе коллизий, поэтому размеры результатов функций хэширования довольно велики. Если применить данный прием к функции SHA-256, то будет получена функция хэширования со 128-битовым уровнем безопасности, что в точности соответствует нашим требованиям.

Некоторые могут возразить, что все n-битовые функции хэширования и так обеспечивают только $n/2$-битовый уровень безопасности. Это верная точка зрения. К сожалению, если не конкретизировать реальное положение вещей, функции хэширования будут неправильно эксплуатироваться и будет предполагаться, что они должны обеспечивать n-битовый уровень безопасности. Например, многие хотят использовать SHA-256 для хэширования 256-битовых ключей алгоритма AES, полагая, что это обеспечит 256-битовый уровень безопасности. Как уже отмечалось, мы используем 256-битовые ключи

[6] Даже в документации к SHA-256 говорится, что n-битовая функция хэширования должна требовать выполнения 2^n шагов для обнаружения прообраза заданного значения.

для достижения 128-битового уровня безопасности, что в точности совпадает со сниженным уровнем безопасности "исправленной" версии SHA-256. Такое совпадение неслучайно. В обоих случаях разрыв между размером криптографического значения и заявленным уровнем безопасности вызван наличием атак на основе коллизий. Поскольку мы полагаем, что атаки на основе коллизий существуют всегда, различие между размерами результатов функций хэширования и уровнями безопасности прекрасно укладывается в нашу концепцию.

Приведем более формальное определение нашего исправления.

Определение 7 *Пусть h — итеративная функция хэширования. Тогда функция хэширования h_d определяется выражением $h_d := h(h(m))$ и имеет заявленный уровень безопасности, равный $\min(k, n/2)$, где k — уровень безопасности функции h, а n — размер результата функции хэширования.*

В большинстве случаев мы будем применять данную конструкцию к функциям семейства SHA. Для любой функции хэширования SHA-X, где X равняется 1, 256, 384 или 512, мы определим $\mathrm{SHA}_d - X$ как функцию, которая отображает m на $\mathrm{SHA} - X(\mathrm{SHA} - X(m))$. В частности, $\mathrm{SHA}_d - 256$ — это всего лишь функция $m \mapsto \mathrm{SHA} - 256(\mathrm{SHA} - 256(m))$.

Однако все эти конструкции слишком новы, а потому еще не заслужили доверия. К счастью, можно показать, что наша исправленная функция хэширования h_d обладает, как минимум, такой же стойкостью к атакам, что и исходная функция h[7]. Аналогичный подход повторного хэширования используется в алгоритме HMAC для защиты от атак с удлинением сообщения. И функция h_{DBL}, и функция h_d позволяют избежать проблемы удлинения сообщения, которая представляет наибольшую угрозу для реальных систем. Вопрос о том, действительно ли функция h_{DBL} обеспечивает n-битовый уровень безопасности, пока остается открытым. Мы предпочитаем доверять обеим функциям на $n/2$-битовом уровне безопасности, поэтому на практике рекомендовали бы использовать более эффективную функцию h_d.

6.5 Какую функцию хэширования выбрать

Альтернатив не так уж много. Мы бы рекомендовали выбрать одну из функций семейства SHA_d. Вопрос состоит лишь в том, какой размер хэш-кода и соответственно какая производительность нужны вашей системе.

[7] Здесь мы вновь немного схитрили. Повторное хэширование сокращает множество значений функции, и осуществлять атаки, в основе которых лежит парадокс задачи о днях рождения, становится немного проще. Данный эффект, однако, имеет очень небольшое влияние и хорошо укладывается в рамки приближений, используемых нами во всех других случаях.

Как уже отмечалось, криптографическое сообщество провело совсем мало работы по изучению функций хэширования. В отличие от блочных шифров, которые часто подвергались различающим атакам, на функции хэширования в большинстве случаев осуществлялись атаки на основе коллизий. Таким образом, еще никто не анализировал функции хэширования с практической точки зрения на предмет соответствия нашему определению безопасности. К счастью, фортуна улыбнулась нам, и функция h_d уже позволяет использовать сниженный, $n/2$-битный уровень безопасности, так что это не проблема. Семейство функций SHA разрабатывалось специально с учетом атак на основе коллизий, поэтому они должны обеспечивать, как минимум, такой уровень безопасности.

Приведенные аргументы, очевидно, свидетельствуют не в пользу функции SHA$_d$-1, поскольку ее 80-битового уровня безопасности явно недостаточно для долговременного применения. У нас остаются SHA$_d$-256 и SHA$_d$-512. (Использовать SHA$_d$-384 нет смысла, потому что, проделав такой же объем работы, вы можете получить и SHA$_d$-512.)

Поскольку желательный уровень безопасности наших систем — 128 бит, выбор в качестве функции хэширования SHA$_d$-256 очевиден. Для обеспечения более высоких уровней безопасности следует использовать SHA$_d$-512. Не думайте, однако, что действительно сможете достигнуть такого уровня безопасности — для этого вам понадобился бы хороший блочный шифр с ключами, размер которых превышает 256 бит. И разумеется, падение производительности, вызванное применением функции SHA$_d$-512, делает ее неприемлемой для многих систем. В подобных случаях вам придется ограничиться 128-битовым уровнем безопасности и функцией SHA$_d$-256.

6.6 Работа на будущее

Над функциями хэширования еще нужно работать и работать. В целом нам кажется, что навыки открытого криптографического сообщества относительно разработки и взлома функций хэширования находятся примерно на том же уровне, на котором находились наши навыки работы с блочными шифрами в середине 80-х годов прошлого века. Нам предстоит узнать еще очень многое о том, как создавать и взламывать функции хэширования. Хорошим началом могло бы стать осуществление атак на версии функций SHA с сокращенным количеством раундов. Пока же, к сожалению, в этой области не наблюдается никакого прогресса.

Глава 7

Коды аутентичности сообщений

Код аутентичности сообщения (message authentication code — MAC) представляет собой число, которое мешает злоумышленнику подделывать сообщения. Применение шифрования не дает злоумышленнику читать текст сообщений, но ни в коей мере не мешает ему изменять эти сообщения. На помощь приходит MAC. Как и функции шифрования, коды аутентичности сообщений используют секретный ключ K, известный пользователям А и Б, но неизвестный злоумышленнику Е. Пользователь А отсылает пользователю Б не только само сообщение m, но и значение MAC этого сообщения, вычисленное с помощью соответствующей функции. Пользователь Б проверяет, соответствует ли значение MAC, полученное вместе с сообщением, настоящему значению MAC этого сообщения. Если значения не совпадают, сообщение отбрасывается как не прошедшее аутентификацию. Теперь злоумышленник Е не сможет изменить сообщение: не зная ключа K, он не сможет подсчитать правильное значение MAC, которое следовало бы отправить вместе с измененным сообщением.

В этой главе речь идет только об аутентификации. Как комбинировать шифрование и аутентификацию, рассматривается в главе 8, “Безопасный канал общения”.

7.1 Что такое MAC

Код аутентичности сообщения, или MAC, — это функция, которая принимает на вход два аргумента (ключ K фиксированной длины и сообщение m произвольной длины) и выдает значение фиксированной длины. Функцию вычисления MAC мы будем обозначать как $\mathrm{MAC}(K, m)$. Для обеспечения аутентификации сообщения пользователь А отсылает не только сообщение m, но и код аутентичности этого сообщения $\mathrm{MAC}(K, m)$.

Вначале обсудим особенности функции вычисления MAC. Будьте осторожны: корректное использование функции вычисления MAC — процедура намного более сложная, чем простое применение к сообщению. Мы еще вернемся к этой проблеме в разделе 7.8.

7.2 Идеальная функция вычисления MAC

Вначале необходимо определить идеальную функцию вычисления MAC. Затем мы определим безопасную функцию вычисления MAC как такую, которую невозможно отличить от идеальной. Думаем, никого не удивит тот факт, что идеальная функция вычисления MAC является случайным отображением. Пусть n — это длина значения MAC в битах.

Определение 8 *Идеальная функция вычисления MAC — это случайное отображение всех возможных входных значений на множество n-битовых выходных значений.*

Идеальная функция вычисления MAC во многом схожа с идеальной функцией хэширования. И та и другая являются случайными отображениями. Основное различие между ними заключается в более "мягком" определении безопасности функции вычисления MAC, что делает их более эффективными, чем функции хэширования.

7.3 Безопасность MAC

Как уже отмечалось, функция вычисления MAC имеет два входных аргумента — ключ K и сообщение m. Ключ K неизвестен злоумышленнику или, точнее, известен не полностью. В оставшейся части системы может быть слабое место, которое обеспечит злоумышленника частичной информацией о ключе K. Пусть k — это неопределенность, которая имеется у злоумышленника относительно значения K. Под словом "неопределенность" мы понимаем количество битов значения K, которые неизвестны злоумышленнику. В более общем случае, если неопределенность равна k, то K может иметь около 2^k возможных значений (разумеется, с точки зрения злоумышленника)[1].

Определение 9 *Пусть n — длина значения MAC, а k — неопределенность в битах, которой обладает злоумышленник относительно ключа K. Атака на функцию вычисления MAC — это нетривиальный метод обнаружения*

[1] Для формализации неопределенности можно было бы воспользоваться понятием энтропии, но это отвлекло бы нас от текущей темы обсуждения. Для самых дотошных: k — это энтропия значения K (в битах) при заданном объеме информации, имеющейся у злоумышленника.

различия между функцией вычисления MAC и идеальной функцией вычисления MAC менее чем за $2^{\min(n,k)}$ *шагов.*

Другими словами, функция вычисления MAC — это случайное отображение, уровень безопасности которого ограничен k битами. Это совсем не то, что обычное случайное отображение. В обычном случайном отображении злоумышленник знает все; в нашем же случае у него есть некая неопределенность относительно значения K. (Если злоумышленник полностью уверен в значении K, тогда $k = 0$, а значит, уровень безопасности равен нулю и нам нечего обсуждать.)

Разумеется, заявленный уровень безопасности конкретной функции MAC может быть ниже, чем размер ее результата. В этом случае злоумышленник ограничен $2^{\min(s,k)}$ шагами, где s — заявленный уровень безопасности. Неопределенность злоумышленника относительно значения K может быть использована для того, чтобы сделать функции вычисления MAC более эффективными, чем функции хэширования.

Следует отметить, что никто другой, кроме нас, не определяет безопасность MAC подобным образом. Большинство исследователей подходят к определению безопасности MAC с более ограниченной точки зрения. В частности, они используют модель атак, в которой злоумышленник по своему желанию выбирает n различных сообщений и получает значение MAC для каждого из этих сообщений. В результате у злоумышленника должно оказаться $n+1$ сообщение, каждое с правильным значением MAC. Данная модель не учитывает некоторых типов атак, например атак со связанным ключом и атак, которые предполагают, что у злоумышленника имеется частичная информация о ключе. Вот почему мы предпочитаем наши определения безопасности, которые оказываются корректными даже тогда, когда функция используется неправильно или в необычном для нее окружении.

7.4 CBC-MAC

Это метод превращения блочного шифра в функцию вычисления MAC. Ключ K при этом используется в качестве ключа шифрования. Основная идея алгоритма CBC-MAC заключается в том, чтобы зашифровать сообщение m с помощью блочного шифра, используя режим CBC, и затем откинуть все блоки шифрованного текста кроме последнего. Для сообщения, состоящего из блоков $P_1, \ldots, P_k$, значение MAC вычисляется следующим образом:

$$\begin{aligned} H_0 &:= \mathrm{IV}, \\ H_i &:= E_K(P_i \oplus H_{i-1}), \\ \mathrm{MAC} &:= H_k. \end{aligned}$$

Иногда результатом функции CBC-MAC считают половину последнего блока, но это уже зависит от конкретной ситуации.

Классическое определение алгоритма CBC-MAC требует, чтобы значение вектора инициализации было фиксированным и равнялось нулю. Именно такое определение часто встречается в учебниках. Вообще говоря, функцию CBC-MAC можно использовать и с любыми другими типами вектора инициализации, которые обсуждались при рассмотрении режима CBC, но очевидных преимуществ это не дает.

Никогда не используйте один и тот же ключ для шифрования и аутентификации (разве что вы твердо уверены в необходимости этого шага). Особенно опасно использовать шифрование в режиме CBC и аутентификацию CBC-MAC с одним и тем же ключом. В этом случае значение MAC будет просто равно последнему блоку шифрованного текста.

Алгоритм CBC-MAC не слишком прост в применении. Тем не менее в общем случае он считается безопасным, если соответствующий блочный шифр также безопасен. Существует ряд атак на основе коллизий, которые сокращают уровень безопасности CBC-MAC до половины размера блока [12]. Вот простой пример одной из таких атак. Пусть M — это функция CBC-MAC. Если известно, что $M(a) = M(b)$, значит, $M(a \parallel c) = M(b \parallel c)$. Это объясняется спецификой алгоритма CBC-MAC. Проиллюстрируем данное свойство на простом примере, когда c состоит из одного блока текста. Мы знаем, что

$$M(a \parallel c) = E_K(c \oplus M(a)),$$
$$M(b \parallel c) = E_K(c \oplus M(b)).$$

Поскольку $M(a) = M(b)$, правые части этих выражений равны.

Такая атака выполняется в два этапа. На первом этапе злоумышленник собирает значения MAC для большого количества сообщений, пока не наткнется на коллизию. В этом случае у него появятся сообщения a и b, для которых $M(a) = M(b)$. Если теперь злоумышленнику удастся аутентифицировать у получателя сообщение $a \parallel c$, он может заменить его сообщением $b \parallel c$, причем значение MAC останется прежним. Получатель проверит значение MAC и примет поддельное сообщение. (Напомним, что мы работаем с параноидальной моделью. Вполне возможно, что злоумышленник может создать сообщение и аутентифицировать его у получателя. В реальной жизни такие ситуации встречаются довольно часто.) Существует много усовершенствованных вариантов такой атаки, которые работают даже при добавлении поля с длиной сообщения и применении правил дополнения [12].

Эта атака не является тривиальной, поскольку она не срабатывает для идеальной функции вычисления MAC. Найти коллизию несложно и для идеальной функции. Тем не менее, даже когда у вас появятся сообщения a и b,

для которых $M(a) = M(b)$, вы не сможете применить их, чтобы подделать значение MAC для нового сообщения, как при использовании алгоритма CBC-MAC.

Существует несколько теоретических результатов, которые показывают, что при использовании конкретной модели доказательств алгоритм CBC-MAC обеспечивает 64-битовый уровень безопасности, если размер блока равен 128 бит [4]. К сожалению, этого недостаточно для обеспечения уровня безопасности, установленного для наших систем. Мы бы смогли использовать CBC-MAC, если бы имели блочный шифр с 256-битовым блоком.

Использовать алгоритм CBC-MAC следует крайне осторожно. Функцию CBC-MAC нельзя применять к самому сообщению — это позволяет злоумышленнику осуществлять довольно простые атаки. Необходимо поступить несколько иначе.

1. Создайте строку s путем конкатенации значений l и m, где l — это длина сообщения m, представленного в формате фиксированной длины.
2. Дополните строку s, чтобы ее длина стала кратной длине блока. (Более подробно это рассматривается в разделе 5.1.)
3. Примените функцию CBC-MAC к дополненной строке s.
4. Результатом применения функции CBC-MAC считайте последний блок шифрованного текста или часть этого блока. Не используйте в качестве MAC промежуточные значения.

Алгоритм CBC-MAC имеет одно существенное преимущество: он использует тот же тип вычислений, что и режимы работы блочных шифров. К содержимому сообщения применяются только две функции — шифрования и вычисления MAC, а значит, у нас есть только две критические точки в отношении скорости работы. Использование в этих точках одних и тех же элементарных функций позволит повысить эффективность реализаций, особенно аппаратных.

Несмотря на это, мы не рекомендуем использовать CBC-MAC, поскольку сделать это правильно крайне сложно.

7.5 HMAC

Если идеальная функция вычисления MAC — это случайное отображение, а у нас уже есть функции хэширования, которые ведут себя как случайное отображение, то почему бы не вычислять значение MAC с помощью функции хэширования? Именно эта идея легла в основу алгоритма HMAC [3, 58]. Разработчики данного алгоритма, конечно же, знали о проблемах функций хэширования, которые обсуждаются в главе 6, “Функции хэширования”. В то время как функции хэширования обеспечивают только $n/2$-битовый уровень

безопасности по отношению к некоторым типам атак, функция вычисления MAC должна обеспечивать n-битовый уровень безопасности. Определять функцию $\mathrm{MAC}(K, m)$ как $h(K \parallel m)$, $h(m \parallel K)$ или даже как $h(K \parallel m \parallel K)$ при использовании одной из стандартных итеративных функций хэширования небезопасно [76]. Если присоединить ключ к началу сообщения, он станет уязвимым по отношению к атакам с удлинением сообщения. Если же присоединить ключ к концу сообщения, сообразительный злоумышленник сможет восстановить ключ примерно за $2^{n/2}$ шагов.

Разработчики алгоритма HMAC не только учли все эти моменты, но и отнеслись к этому весьма серьезно. Структура функции HMAC позволяет избежать атак с восстановлением ключа и атак, которые могут быть проведены злоумышленником в автономном режиме, без взаимодействия с системой. Несмотря на это, HMAC все еще ограничен $n/2$-битовым уровнем безопасности. Это объясняется наличием атак, построенных на парадоксе задачи о днях рождения, которые используют внутренние коллизии итеративных функций хэширования. Алгоритм HMAC гарантирует, что выполнение таких атак потребует $2^{n/2}$ взаимодействий с атакуемой системой. Это гораздо сложнее, нежели выполнить $2^{n/2}$ вычислений на собственном компьютере.

Для простоты мы не будем проводить различий между оперативными и автономными атаками. Будем лишь оценивать объем работы, который придется проделать злоумышленнику. Итак, даже несмотря на тщательно проработанную структуру, уровень безопасности HMAC ограничен $n/2$ битами для n-битовой функции хэширования.

В статье [3], посвященной алгоритму HMAC, приведено несколько удачных примеров проблем, возникающих в том случае, когда у криптографических функций (в данном случае функций хэширования) обнаруживаются неожиданные свойства. Вот почему мы так настойчиво призываем обеспечивать простые спецификации поведения криптографических функций.

Значение HMAC подсчитывается по формуле $h(K \oplus a \parallel h(K \oplus b \parallel m))$, где a и b — некоторые константы. Как видите, сам текст сообщения хэшируется только единожды, а полученный результат еще раз хэшируется вместе с ключом аутентификации. Более подробно об алгоритме HMAC можно прочитать в его спецификациях [3, 58]. Алгоритм HMAC работает с любыми итеративными функциями хэширования, которые описаны в главе 6, “Функции хэширования”. Это единственное исключение, когда можно непосредственно применять функции SHA, а не SHA_d — в HMAC уже учтены проблемы, на решение которых направлены функции SHA_d.

Нам нравится HMAC. Он красив, эффективен и прост в реализации. В качестве функций хэширования HMAC обычно применяют MD5 или SHA-1. Сегодня подобные реализации данного алгоритма содержатся во многих программных библиотеках. Несмотря на это, для достижения заявленного 128-

битового уровня безопасности мы бы рекомендовали использовать HMAC только с 256-битовой функцией хэширования, например SHA-256. Поскольку в HMAC уже учтены слабые места функции SHA-256, использовать SHA_d-256 в данном случае необязательно.

7.5.1 HMAC или SHA_d?

Как уже отмечалось, функция вычисления MAC, определенная как $h(K \parallel m)$, хороша настолько, насколько хороша соответствующая функция хэширования h. В качестве хорошей функции хэширования предлагалась SHA_d-256. Тогда функцию вычисления MAC можно переписать следующим образом:

$$(K, m) \mapsto SHA - 256(SHA - 256(K \parallel m)).$$

Между тем в предыдущем разделе мы рекомендовали воспользоваться алгоритмом HMAC с функцией хэширования SHA-256. Это можно записать так:

$$(K, m) \mapsto SHA - 256((K \oplus a) \parallel SHA - 256((K \oplus b) \parallel m)).$$

Почему же мы рекомендуем использовать HMAC, если первая конструкция проще?

Это хороший вопрос. Ответ на него достаточно тонок. Алгоритм HMAC был ориентирован на несколько иную модель затрат, требующихся для осуществления атаки. Предположим, пользователь A применяет функцию вычисления MAC в целях аутентификации. Мы считаем только количество шагов, которые необходимо проделать злоумышленнику, и не учитываем разные тонкости наподобие того, требует ли каждый шаг взаимодействия с пользователем A или нет. В отличие от нас, разработчики HMAC сочли выполнение автономных атак (т.е. не требующих взаимодействия с атакуемой системой) более простым, чем выполнение оперативных атак (т.е. основанных на взаимодействии с атакуемой системой), и приняли особые меры, чтобы в первую очередь защитить свой алгоритм от автономных атак. Вот почему в HMAC и при втором хэшировании используется ключ аутентификации.

Разработчики HMAC, безусловно, правы. Осуществлять автономные атаки намного легче, нежели оперативные. На наш взгляд, однако, это не оправдывает обеспечения различных уровней безопасности по отношению к автономным и оперативным атакам. Проблема состоит в том, что мы не хотим указывать некий фиксированный фактор затрат наподобие: “Каждый шаг оперативной атаки эквивалентен S шагам автономной атаки”. Мы не знаем, какое значение должно иметь S, особенно потому, что не представляем, как именно будет использоваться функция вычисления MAC. Таким образом, мы предпочитаем придерживаться общего 128-битового уровня безопасности. Только не воспринимайте это как критику разработчиков HMAC. Когда они

разрабатывали свой алгоритм, большинство функций хэширования выдавали 128-битовые результаты. В то время автономная атака на основе коллизий требовала осуществить лишь2^{64} шагов — то, от чего определенно следовало защититься. Сейчас результаты функций хэширования имеют бóльшие размеры, и подобные атаки для них несущественны.

Так какой же из двух алгоритмов следует выбрать? Мы рекомендуем HMAC. Затрат на его реализацию требуется ненамного больше, чем на $\mathrm{SHA}_d - 256(K \parallel m)$, зато HMAC применяется уже довольно долго и удостоился более пристального внимания криптоаналитиков. Мы всегда пытаемся быть консервативными, а HMAC, безусловно, более консервативный выбор.

7.6 UMAC

Семейство функций UMAC — хороший пример того, как использовать неопределенность в отношении K для получения функции вычисления MAC, намного более быстрой, чем функции хэширования[2]универсальной функции хэширования (universal hash function). *Это совсем не те функции хэширования, о которых мы говорим в этой книге. Не путайте их.* [8, 59]. Скорость работы алгоритма UMAC может на целый порядок превышать скорость работы HMAC. Кроме того, для UMAC существуют доказательства его безопасности.

К сожалению, функции UMAC имеют несколько недостатков, которые не дают им право называться идеальными функциями вычисления MAC.

7.6.1 Размер значения

Авторы алгоритма UMAC предлагают использовать 64-битовый результат, которого, на их взгляд, должно быть достаточно для большинства областей применения. Это стандартная практика; и еще несколько лет назад мы бы порекомендовали то же самое. Причиной, по которой значение MAC может быть столь небольшим, состоит в том, что осуществить атаку путем полного перебора значений MAC в автономном режиме невозможно. Если злоумышленник попытается подобрать значение MAC, ему нужно вступить во взаимодействие с системой, чтобы проверить, правильно ли подобранное значение. Поскольку хорошая система не позволит злоумышленнику аутентифицировать так много сообщений, для обеспечения безопасности может хватить даже значения MAC небольшой длины.

В последние годы, однако, мы изменили свою точку зрения и теперь считаем, что длины в 64 бит недостаточно. Безопасность MAC не должна зависеть от остальных частей системы. Как разработчики, мы не знаем, где будет

[2]В документации UMAC слово “хэширование” применяется для определения

использоваться — или неправильно использоваться — значение MAC[3]. Одни системы допускают большое количество попыток аутентификации поддельных сообщений. Другие неправильно используют функцию вычисления MAC, что оборачивается сплошными неприятностями. Мы не хотим усложнять свои системы многочисленными перекрестными зависимостями. Структура криптографических систем столь сложна, что дальнейшее ее усложнение было бы катастрофой. Вот почему нам нужно 128-битовое значение MAC. Вообще говоря, согласно нашему правилу проектирования 3 (см. раздел 4.5.8), следовало бы использовать 256-битовое значение MAC — это бы гарантировало обеспечение 128-битового уровня безопасности даже по отношению к атакам, в основе которых лежит парадокс задачи о днях рождения. Если это возможно, используйте 256-битовое значение MAC. К сожалению, поскольку в большинстве систем на данный момент используются 64- или 96-битовые значения MAC, доказать необходимость 256-битовых значений будет очень трудно (особенно потому, что значение MAC отсылается вместе с самим сообщением и таким образом непосредственно влияет на стоимость пересылки). Если вы используете хорошую функцию вычисления MAC и не боитесь атак на основе коллизий, вам будет вполне достаточно и 128-битового значения.

7.6.2 Выбор функции

Существует еще один, более важный вопрос: какую функцию UMAC выбрать? В исходной статье, посвященной алгоритму UMAC [8], определялось четыре различные функции: UMAC-STD-30, UMAC-STD-60, UMAC-MMX-30 и UMAC-MMX-60. В более новом черновике RFC [59] предлагается значительное изменение структуры алгоритма и еще две функции: UMAC16 и UMAC32. Говоря точнее, этот черновик содержит определение функции UMAC с шестью параметрами, а UMAC16 и UMAC32 — лишь два рекомендуемых набора значений этих параметров.

Казалось бы, нам остается только выбрать один из наборов параметров, но какой именно? Параметры оказывают значительное влияние на эффективность UMAC. Одни из них работают быстрее на машинах с прямым порядком байтов, а другие — с обратным. Одни параметры хорошо подходят для Pentium MMX, а другие — для других 32-разрядных процессоров. Одни используют команды беззнакового, а другие — знакового умножения. Этот список можно продолжать до бесконечности. Что бы вы ни выбрали, наверняка найдется кто-нибудь, кто обвинит вас в неправильном использовании особенностей оборудования, поскольку задержка в производительности может быть весьма значительной. На каждой конкретной платформе скорость

[3]Неправильное использование значения MAC встречается сплошь и рядом. Например, протокол IKE в IPSec [40] использует функцию вычисления MAC в тех ситуациях, когда сообщение является секретным, но ключ известен!

работы UMAC может различаться в 3-5 раз, в зависимости от точного набора значений параметров, а разные платформы предпочитают различные наборы параметров.

Ужасающее изобилие функций UMAC — результат непомерного стремления к оптимизации. В погоне за высокими скоростями разработчики UMAC создали версии функций на все случаи жизни, поскольку в каждой ситуации требуется выполнение конкретных оптимизаций.

Мы не согласны с этим подходом. Он имеет преимущества в краткосрочной перспективе, однако обладает существенным недостатком в аспекте долговременного пользования. В отличие от HMAC или SHA-1, у алгоритма UMAC нет единого стандарта. Разные разработчики используют разные версии UMAC. Чтобы осуществлять взаимодействие между такими системами, понадобится реализовать сразу несколько версий UMAC. Многие библиотеки позволяют выбирать набор параметров UMAC только во время компиляции, что создает целый ряд проблем, если приложению нужны две или более версии UMAC. Кроме того, на протяжении времени жизни криптографической системы — а это 20-30 лет — архитектура процессоров претерпит немало изменений. При таких условиях микрооптимизация, основанная на текущей архитектуре, просто не имеет будущего.

7.6.3 Платформенная гибкость

Как уже отмечалось, разные версии UMAC оптимизированы для конкретных платформ. Более того, каждая версия выдвигает достаточно строгие требования к своей платформе. Если у вас есть процессор с быстрой командой умножения и достаточный объем памяти, UMAC способен достичь очень высоких скоростей. Тем не менее, на наш взгляд, он не сможет развить нужной производительности при работе со смарт-картами или другими 8-разрядными устройствами с ограниченным объемом памяти. Требования UMAC к временному хранилищу данных очень высоки, а смарт-карты всегда имеют ограниченный объем памяти.

Из этого следует, что UMAC подходит для современных настольных систем и не подходит для процессоров с малой разрядностью[4]. Мы также считаем, что эффективные аппаратные реализации UMAC обходятся слишком дорого по сравнению с альтернативными алгоритмами наподобие HMAC. Ал-

[4]Некоторые уже давно твердят о том, что процессоры с малой разрядностью безнадежно устарели и в течение нескольких лет будут вытеснены новыми, более мощными процессорами. Мы с этим не согласны. Процессоры с малой разрядностью применяются уже несколько десятилетий. Они не вымирают, а просто переходят в еще более “мелкие” системы.. Появившиеся на свет в начале 1970-х годов 8-разрядные процессоры до сих пор счастливо живут и здравствуют, хотя по прогнозам должны были бы исчезнуть, как минимум, 10 лет назад.

горитм UMAC скорее напоминает бегового скакуна, чем рабочую лошадку. Он развивает хорошую скорость, но может пригодиться только в особых обстоятельствах.

7.6.4 Нехватка анализа

Еще одной проблемой алгоритма UMAC является недостаточный объем криптоанализа, которому была подвергнута его структура. Как уже отмечалось, существует доказательство безопасности UMAC, но оно не заменяет собой проведения анализа, исходя из позиций злоумышленника. Довольно много систем, имевших доказанную безопасность, впоследствии были взломаны. К сожалению, подобный недостаток присущ всем новым алгоритмам, таким, как AES, SHA-256, или некоторым конструкциям, предлагаемым в данной книге. Среди них наибольшего внимания со стороны криптоаналитиков, пожалуй, удостоился AES. (Мы это знаем, потому что сами принимали участие в его анализе.) Функция SHA-256 была разработана Управлением национальной безопасности США (NSA), предыдущие наработки которого были весьма неплохи. В NSA также работает большое количество экспертов. Некоторые из них, несомненно, анализировали функцию SHA-256 перед тем, как предоставить ее на суд широкой общественности. Новые конструкции, предложенные в этой книге, еще не были проанализированы другими экспертами на предмет возможности нападения. Тем не менее структура наших конструкций крайне консервативна, а многие из них, как минимум, не хуже стандартных конструкций, используемых на протяжении многих лет. UMAC же имеет весьма агрессивный дизайн, который напрямую зависит от доказательства безопасности. Давайте посмотрим на вещи с оптимистической точки зрения и предположим, что вероятность полноты и правильности доказательства безопасности алгоритма UMAC составляет 95%. Однако в 5% случаев можно предположить, что UMAC не имеет доказательства безопасности, и такого рода вероятность нас совершенно не привлекает.

7.6.5 Зачем тогда нужен UMAC?

“Если нам не нравится UMAC, то зачем же так много о нем говорить?” — спросите вы. Вообще-то мы думаем, что разработчики UMAC двигались в правильном направлении. Идея использовать неопределенность относительно значения K для того, чтобы ускорить вычисление MAC, крайне ценна. Нам бы хотелось проделать еще ряд исследований в этой области. Где-то обязательно должна отыскаться функция вычисления MAC, которая будет более быстрой, чем функция хэширования, и вместе с тем такой же надежной и гибкой, как современные блочные шифры.

7.7 Какую функцию вычисления MAC выбрать

Как вы, наверное, уже догадались, мы отдаем предпочтение комбинации HMAC-SHA-256, т.е. алгоритму HMAC, в котором в качестве функции хэширования используется SHA-256. Мы бы действительно хотели, чтобы код аутентичности сообщения был 256-битовым. В большинстве систем, однако, используются лишь 64- или 96-битовые значения MAC, и даже тогда многие жалуются на большие расходы. Добавление к каждому сообщению по 32 байт (256 бит) ни в коей мере не принесет популярности вашему проекту. Насколько мы знаем, для функции вычисления MAC не существует атак на основе коллизий, если использовать ее в классическом понимании, поэтому надеемся, что усечение результата алгоритма HMAC-SHA-256 до 128 бит не должно навредить безопасности системы.

Еще раз подчеркнем, что применять в качестве функции хэширования HMAC функцию SHA_d-256 нет смысла, так как в алгоритме HMAC уже учтена проблема удлинения сообщения. Тем не менее для достижения 128-битового уровня безопасности в HMAC следует использовать 256-битовую функцию хэширования, поэтому SHA-1 нам не подходит.

Нельзя сказать, что алгоритм HMAC полностью нас устраивает. Мы верим в существование более быстрых функций вычисления MAC, но подходящих вариантов пока что не появилось.

7.8 Использование MAC

Правильно использовать MAC намного сложнее, чем кажется. Давайте посмотрим, в чем же состоят основные проблемы.

Когда пользователь Б получает значение $\mathrm{MAC}(K, m)$, он знает, что кто-то, кому известен ключ K, подтвердил правильность сообщения m. Между тем это отнюдь не обеспечивает действительной корректности сообщения. Злоумышленник Е мог записать сообщение, отправленное пользователем А пользователю Б, и затем послать копию этого сообщения пользователю Б когда-нибудь позднее. Если система последнего не имеет специальной защиты от такого типа атак, пользователь Б примет эту копию как корректное сообщение от пользователя А. Аналогичные проблемы возникают, если пользователи А и Б применяют один и тот же ключ K для аутентификации трафика сообщений в обоих направлениях. Злоумышленник Е может послать записанное им сообщение обратно пользователю А, и тот поверит, что сообщение было отправлено пользователем Б.

Довольно часто пользователям А и Б нужно аутентифицировать не только сообщение m, но и некоторые дополнительные данные d. Это может быть

номер сообщения, используемый для защиты от атак воспроизведения (replay attacks), адрес источника и назначения сообщения и т.п. Большинство подобных полей являются частью заголовка аутентифицируемого (и часто зашифрованного) сообщения. Значение MAC должно обеспечивать аутентификацию не только m, но и d. Как правило, для решения этой проблемы функцию вычисления MAC применяют не к m, а к $d \parallel m$.

Следующую проблему лучше всего определить с помощью правила проектирования.

Правило проектирования 4. *Принцип Хортона: аутентифицируйте не то, что сказано, а то, что имеется в виду.*

Значение MAC аутентифицирует только строку байтов, в то время как пользователи А и Б хотят обеспечить аутентификацию сообщения с конкретным смыслом. В действительности отличие того, что сказано (например, последовательность байтов, отосланная пользователем А пользователю Б), от того, что имеется в виду (например, интерпретация сообщения), играет очень важную роль.

Предположим, пользователь А применяет значение MAC для аутентификации сообщения $m := a \parallel b \parallel c$, где a, b и c — некоторые поля данных. Пользователь Б получает сообщение m и разбивает его на поля a, b и c. Но правильно ли он это делает? Разбивка на поля должна проводиться в соответствии с некоторыми правилами. Если эти правила не совпадают с теми, которые применяются пользователем А для построения сообщения, пользователь Б получит неправильные значения полей. Подобные ситуации крайне нежелательны, поскольку они означают возможность аутентификации фальсифицированных данных. Поэтому очень важно, чтобы пользователь Б разбивал сообщение m именно на те поля, из которых оно было составлено пользователем А.

Этого несложно добиться в простых системах. Обычно поля данных имеют фиксированный размер. Тем не менее рано или поздно многие из нас оказываются в ситуации, когда поля данных должны иметь переменный размер или новая версия программного обеспечения будет использовать поля большего размера. Разумеется, новой версии программного обеспечения понадобится обратная совместимость с более старой версией. Тут-то и начинаются проблемы. Когда длина поля перестанет быть постоянной, пользователь Б будет определять ее из некоторого контекста, и этот контекст может стать объектом манипуляций злоумышленника. Предположим, что пользователь А применяет старую версию протокола обмена данными и старые, более короткие поля. Пользователь Б работает с новой версией протокола. Злоумышленник Е манипулирует каналом общения между пользователями А и Б таким образом, чтобы пользователь Б думал, что пользователь А применяет новую

версию протокола. (Детали того, как именно злоумышленник Е осуществляет подобные манипуляции, не имеют значения. Чтобы подсистема вычисления MAC была безопасной, она не должна зависеть от других частей системы.) В результате этого пользователь Б разбивает сообщение на поля большего размера, чем нужно, и получает некорректные данные.

Именно здесь на помощь приходит принцип Хортона[5]. Аутентифицировать нужно не само сообщение, а его смысл. Другими словами, значение MAC должно аутентифицировать не только сообщение m, но и всю информацию, которая применяется пользователем Б для интерпретации этого сообщения. Обычно подобная информация включает в себя идентификатор протокола, номер версии протокола, идентификатор сообщения, образованный в соответствии с данным протоколом, размеры полей и т.п. Одним из частичных решений проблемы может стать отказ от простой конкатенации полей и использование вместо этого структуры данных наподобие XML, которая может быть проанализирована без необходимости предоставления какой-либо дополнительной информации.

Принцип Хортона — одна из причин того, почему аутентификация для протоколов нижних уровней не обеспечивает необходимой аутентификации для протоколов верхних уровней. Система аутентификации на уровне IP-пакетов не может знать о том, как почтовый клиент будет интерпретировать сообщения. Следовательно, она не сможет проверить, действительно ли контекст, в котором интерпретируется сообщение, соответствует контексту, в котором это сообщение было отослано. Единственным решением данной проблемы является создание для почтового клиента собственной системы аутентификации данных (разумеется, в дополнение к системе аутентификации на нижних уровнях).

В завершение необходимо отметить следующее: тщательно обдумайте, какие данные включать в процесс аутентификации. Все эти данные вместе с самим сообщением должны быть представлены в виде строки байтов так, чтобы последнюю можно было уникальным образом разбить на исходные поля. Не забывайте применять это к результату конкатенации дополнительных данных и сообщения, которая упоминалась в начале этого раздела. Если вы собираетесь аутентифицировать строку $d \parallel m$, постарайтесь сформулировать фиксированное правило того, как разбивать полученную строку на поля d и m.

[5] Для тех, чье счастливое детство прошло не в США, поясняем: Хортон — один из персонажей д-ра Сьюсса (Dr. Seuss), автора популярных детских книг [89].

Глава 8

Безопасный канал общения

Пришло время заняться решением первой настоящей проблемы, присущей реальным системам. Пожалуй, наиболее распространенной практической проблемой является создание безопасного канала общения.

8.1 Формулировка проблемы

Неформально нашу проблему можно определить следующим образом: создание безопасного соединения между пользователями А и Б. Чтобы четко сформулировать, о чем пойдет речь в этой главе, описание проблемы придется немного формализовать.

8.1.1 Роли

Большинство соединений являются двунаправленными. Пользователь А отсылает сообщения пользователю Б, а пользователь Б отсылает сообщения пользователю А. Чтобы не перепутать эти потоки данных, протокол обмена данными должен обладать некоторой "асимметричностью". В реальных системах одним из участников общения может быть клиент, а другим — сервер. Иногда для удобства говорят об инициаторе (участнике общения, инициировавшем безопасное соединение) и ответчике. В любом случае участникам общения нужно назначить роли пользователя А и пользователя Б, чтобы каждый четко знал, в какой роли он выступает.

Разумеется, у нас всегда есть злоумышленник Е, который пытается атаковать безопасный канал общения всеми возможными способами. Злоумышленник Е может читать все сообщения, которыми обмениваются пользователи А и Б, и как угодно манипулировать этими сообщениями. В частности, злоумышленник Е может удалять, вставлять или изменять данные, передаваемые по каналу общения.

Когда мы говорим о передаче сообщений от пользователя А к пользователю Б, то в большинстве случаев имеем в виду два физических компьютера, отсылающих друг другу сообщения по некоторой сети. Еще одной интересной областью применения этой модели является безопасное хранение данных. Если представить процесс хранения данных как передачу данных из настоящего времени в будущее, все приведенные ниже сообщения будут справедливы и для хранения данных. В этом случае пользователем А и пользователем Б может быть одно и то же лицо, а носителем, применяемым для передачи данных, — магнитная лента. Как и обычный канал общения, магнитную ленту нужно защищать от внешних вторжений и манипуляций ее содержимым. Разумеется, отсылая данные в будущее, мы не можем применять двусторонний протокол, потому что из будущего невозможно отправить сообщение в прошлое.

8.1.2 Ключ

Чтобы реализовать безопасный канал общения, нам нужен какой-нибудь общий секрет. В данном случае предположим, что пользователи А и Б применяют общий секретный ключ K, не известный никому, кроме них самих. Это очень важное свойство. Криптографические функции никогда не могут идентифицировать пользователя А как физическое лицо. В большинстве случаев они идентифицируют ключ. Алгоритм верификации, применяемый пользователем Б, сообщает ему примерно следующее: “Это сообщение было послано кем-то, кто знает ключ K и выступает в роли пользователя А”. Данное утверждение может пригодиться пользователю Б только в том случае, если он знает, что ключ K известен ограниченному числу лиц, например только ему самому и пользователю А.

На данном этапе нас не волнует конкретный способ выбора и распространения ключа. Мы просто предполагаем, что ключ уже есть. Об управлении ключами речь идет в главе 15, “Протокол согласования ключей”. К ключу K предъявляются следующие требования:

- он должен быть известен только пользователям А и Б;
- каждый раз при инициализации безопасного канала общения следует генерировать новое значение ключа K.

Второе требование не менее важно, чем первое. Если один и тот же ключ будет использоваться на протяжении долгого времени, злоумышленник Е сможет воспроизвести старые сообщения и отправить их пользователю А или Б, внеся полную неразбериху в поток сообщений. Таким образом, даже если в качестве основного ключа используется фиксированный пароль, пользователи А и Б должны применять протокол согласования ключей для

установки некоторого уникального ключа K и перезапускать этот протокол при каждой новой инициализации безопасного канала общения. Ключ K, используемый на протяжении только одного сеанса общения, называется *ключом сеанса (session key)*. Как генерировать ключи сеанса, описано в главе 15, "Протокол согласования ключей".

Безопасный канал общения проектируется таким образом, чтобы достичь 128-битового уровня безопасности. Следуя правилу проектирования 3 (см. раздел 4.5.8), мы будем использовать 256-битовый ключ. Таким образом, длина значения K составляет 256 бит.

8.1.3 Сообщения или поток

Теперь нужно решить, как будут передаваться данные между пользователями А и Б: в виде дискретной последовательности сообщений (например, писем электронной почты) или же непрерывного потока байтов (например, потока мультимедийных данных). Мы будем рассматривать только те системы, которые обрабатывают дискретную последовательность сообщений. При необходимости такие системы можно легко приспособить к обработке потока данных, "нарезая" поток данных на отдельные сообщения и собирая их в единое целое на стороне получателя. В реальной жизни большинство систем на криптографическом уровне работают с дискретными сообщениями.

Мы также предполагаем, что транспортная система, которая занимается доставкой сообщений от пользователя А к пользователю Б и наоборот, не является надежной. С криптографической точки зрения, даже надежный коммуникационный протокол наподобие TCP/IP не в состоянии сформировать надежный канал общения. В конце концов, злоумышленник может легко менять, удалять или вставлять данные в TCP-поток, не прерывая процесс передачи данных. Протокол TCP устойчив по отношению только к случайным событиям наподобие потери пакета. Он не защищен от активных атак. С нашей, "злодейской", точки зрения, надежного коммуникационного протокола нет и быть не может. (Это хороший пример того, насколько разнятся взгляды криптографов на окружающий мир.)

8.1.4 Свойства безопасности

Теперь можно сформулировать свойства безопасности канала общения. Пользователь А отсылает последовательность сообщений $m_1, m_2, \ldots$, которые обрабатываются алгоритмами безопасного канала общения и затем передаются пользователю Б. Пользователь Б обрабатывает полученные сообщения с помощью алгоритмов безопасного канала общения, в результате чего получает последовательность сообщений $m'_1, m'_2, \ldots$.

Безопасный канал общения должен обладать следующими свойствами:

- злоумышленник Е ничего не знает о сообщениях m_i кроме времени их отправки и размера;
- даже если злоумышленник Е атакует канал общения, манипулируя данными, отправленными пользователем А пользователю Б, последовательность сообщений $m'_1, m'_2, \ldots$, полученная пользователем Б, является подпоследовательностью последовательности сообщений $m_1, m_2, \ldots$, причем пользователь Б точно знает, какую подпоследовательность сообщений он получил. (Подпоследовательность может быть получена из исходной последовательности путем отбрасывания нуля или более элементов.)

Первое свойство можно также назвать секретностью. В идеале злоумышленник Е не должен знать о сообщениях *ничего*. В реальной жизни, однако, это практически недостижимо. Скрыть информацию о времени отправки или размере сообщения крайне сложно. Существующие решения этой проблемы предполагают отправку пользователем А непрерывного потока сообщений с максимально возможной пропускной способностью. Даже если пользователю А нечего отослать в данный момент, он должен составить несколько искусственных сообщений и отправить хотя бы их. Подобное решение может быть приемлемо для военных систем, но никак не для простых граждан. Если же злоумышленник Е видит размер и время отправки сообщений, он может узнать, кто с кем, когда и в каком объеме обменивается данными. Это называется *анализом потока данных (traffic analysis)*. Такой анализ снабжает злоумышленника многочисленной информацией, а помешать его проведению очень трудно. Мы не будем решать эту проблему, поэтому предположим, что злоумышленник Е может проводить анализ потока данных, передаваемых по каналу общения.

Второе свойство гарантирует, что пользователь Б будет получать только корректные сообщения и только в правильном порядке. В идеале пользователь Б должен получать в точности ту же последовательность сообщений, которая была отправлена пользователем А. К сожалению, ни один из существующих коммуникационных протоколов не является надежным в криптографическом смысле. Злоумышленник Е всегда может удалить передаваемое сообщение. Поскольку мы не в состоянии предотвратить потерю сообщений, пользователю Б нужно смириться с тем, что он получит только подпоследовательность сообщений. Обратите внимание, что оставшиеся сообщения, которые получит пользователь Б, будут находиться в правильном порядке. Среди них не будет дубликатов, измененных сообщений или же поддельных сообщений, посланных кем-то, отличным от пользователя А. Более того, пользователь Б будет точно знать, каких сообщений он не получил. Это может

быть важно в ситуациях, когда интерпретация сообщений зависит от порядка, в котором они получены.

Зачастую пользователь А хочет быть уверенным в том, что пользователь Б получит всю переданную ему информацию. Во многих системах для этого реализуют схему уведомлений, при которой пользователь Б отправляет пользователю А уведомление (явное или неявное) о получении сообщения. Пользователь А повторно отправляет пользователю Б все сообщения, о доставке которых не было получено уведомлений. Обратите внимание, что наш безопасный канал общения никогда не инициирует повторную отправку сообщений. Все это приходится выполнять самому пользователю А или, как минимум, протоколу, который использует канал общения.

"А почему бы не реализовать механизм повторной отправки в рамках самого канала общения?" — спросите вы. Потому что это усложнило бы его описание. Мы хотим, чтобы модули, ответственные за обеспечение безопасности, оставались простыми. Уведомления и повторная отправка — это стандартные механизмы коммуникационных протоколов, и они могут быть реализованы поверх нашего безопасного канала общения. В конце концов, эта книга посвящена криптографии, а не основным механизмам коммуникационных протоколов.

8.2 Порядок аутентификации и шифрования

Очевидно, к нашим сообщениям будет применяться и шифрование и аутентификация. Это можно сделать двумя способами: вначале зашифровать сообщение, а затем провести аутентификацию шифрованного текста или же вначале обеспечить аутентификацию сообщения, а затем зашифровать и сообщение, и значение MAC.

Существует два основных аргумента в пользу первого подхода. Теоретические результаты показывают, что при использовании конкретных определений безопасности шифрования и аутентификации подход, при котором вначале применяется шифрование, а затем аутентификация, является безопасным, а подход, при котором вначале применяется аутентификация, а затем шифрование, — небезопасным. Следует, однако, отметить, что второй подход оказывается небезопасным только тогда, когда схема шифрования имеет определенное слабое место. На практике мы никогда не используем схемы шифрования с подобными недостатками. Между тем такие слабые схемы шифрования удовлетворяют конкретному формальному определению безопасности. (Это хороший пример расхождения между доказательствами безопасности и практической безопасностью.) Применение функции вычисления MAC к шифрованному тексту, полученному с помощью подобной схемы

шифрования, исправляет недостатки последней и делает ее безопасной. Как видите, для реальных систем подобные теоретические результаты не имеют большого значения. В действительности существуют аналогичные доказательства того, что такая проблема вообще не возникает для всех поточных шифров (таких, как режим CTR) и для режима CBC.

Необходимо также отметить, что подход "сначала шифрование, затем аутентификация" более эффективен при отбрасывании фальсифицированных сообщений. Если с сообщением все в порядке, пользователь Б должен и расшифровать сообщение, и проверить его значение MAC независимо от того, в каком порядке применялись шифрование и аутентификация. Если же сообщение является поддельным (т.е. имеет неправильное значение MAC), пользователь Б его отбросит. Если к сообщению вначале применялось шифрование, а затем аутентификация, на стороне получателя операция дешифрования будет проводиться после операции аутентификации. В этом случае пользователю Б не придется расшифровывать поддельные сообщения, поскольку он обнаружит и отбросит их еще до расшифровки. Если же к сообщению сначала применялась аутентификация, а затем шифрование, пользователю Б придется расшифровывать абсолютно все сообщения — только так он сможет проверить их значения MAC. Таким образом, во втором случае пользователю Б придется проделывать лишнюю работу по расшифровке поддельных сообщений. Данная проблема становится актуальной тогда, когда злоумышленник Е отсылает пользователю Б большое количество фальсифицированных сообщений. Отбрасывая эти сообщения еще до расшифровки, пользователь Б снижает нагрузку на процессор. Кроме того, в некоторых (надо сказать, весьма редких) случаях применение данного подхода несколько затрудняет проведение атак типа "отказ в обслуживании" (denial-of-service — DOS). На практике большинство DOS-атак направлены на переполнение канала общения, а вовсе не на загрузку фальсифицированными сообщениями процессора пользователя Б. Лично нам этот аргумент не кажется убедительным, так как мы всегда готовы пожертвовать производительностью ради безопасности.

Существует два основных аргумента и в пользу второго подхода, при котором вначале применяется аутентификация, а затем шифрование. Если шифрование выполняется перед аутентификацией, злоумышленник Е видит и текст, для которого вычислено значение MAC, и само значение MAC. Если же шифрование выполняется после аутентификации, злоумышленник видит только шифрованный текст и зашифрованное значение MAC; сам текст, для которого вычислено значение MAC (т.е. открытый текст), и настоящее значение MAC от него скрыты. Это существенно затрудняет нападение на функцию вычисления MAC по сравнению с первым подходом. Вообще говоря, суть спора о порядке шифрования и аутентификации состоит в том,

какую функцию применять последней. Если последним применяется шифрование, злоумышленник сможет беспрепятственно нападать на функцию шифрования. Если же последней применяется аутентификация, злоумышленник будет нападать на функцию аутентификации. В общем случае аутентификация важнее шифрования. Поэтому мы предпочитаем подвергнуть опасности функцию шифрования, но защитить значение MAC насколько это возможно.

Вы не ослышались: вопреки мнению большинства, аутентификация действительно важнее шифрования. Для большей наглядности попробуйте представить себе использование безопасного канала общения. Подумайте, какой ущерб нанесет злоумышленник Е, если он сможет читать все сообщения. Затем представьте, каков будет ущерб, если злоумышленник Е сможет изменять все эти сообщения. В большинстве случаев изменение данных оказывает воистину разрушающее воздействие на систему, а потому наносит намного больше ущерба, чем просто чтение этих данных кем-нибудь посторонним.

Второй аргумент в пользу подхода "сначала аутентификация, затем шифрование" касается принципа Хортона. Аутентифицировать нужно не то, что сказано, а то, что имеется в виду. Аутентификация шифрованного текста нарушает это правило и создает еще одно потенциальное слабое место. Пользователь Б может успешно аутентифицировать шифрованный текст, но затем расшифровать его с помощью неправильного ключа, отличного от того, который применялся пользователем А для шифрования. В этом случае, даже несмотря на корректное прохождение аутентификации, пользователь Б получит не тот открытый текст, который был послан ему пользователем А. Данная проблема, в частности, присуща одной из конкретных (нестандартных) конфигураций протокола IPSec [32]. Для исправления этой ошибки ключ шифрования можно было бы включить в дополнительные данные, которые подвергаются аутентификации вместе с самим сообщением. Нам, однако, не хотелось бы применять этот ключ в каких-либо других целях помимо его стандартного назначения. Это связано с дополнительным риском; использование не очень надежной функции вычисления MAC может привести к утечке информации о ключе шифрования. Более удачным решением является генерация ключа шифрования и ключа аутентификации на основе одного и того же ключа безопасного канала общения (см. раздел 8.4.1). Данное решение позволяет устранить слабое место, но порождает перекрестную зависимость: система аутентификации приобретает зависимость от системы генерации ключей.

Можно до хрипоты спорить о том, в каком порядке лучше выполнять шифрование и аутентификацию. Оба решения могут служить основой как для хороших, так и для плохих систем. Каждый подход имеет свои преимущества и недостатки, поэтому окончательный выбор зависит от того, какую из операций вы считаете более важной. Мы предпочитаем сначала проводить

аутентификацию, а затем шифрование, поскольку простота и безопасность кажутся нам важнее экономии процессорного времени.

8.3 Структура решения

Наше решение состоит из трех компонентов: нумерации сообщений, шифрования и аутентификации.

8.3.1 Номера сообщений

Номера сообщений очень важны по ряду причин. Они могут применяться для генерации векторов инициализации, используемых в некоторых алгоритмах шифрования. Они позволяют получателю отбрасывать сообщения, воспроизведенные злоумышленником, не требуя наличия большой базы данных сообщений. С их помощью можно определить, какие сообщения были потеряны в процессе передачи по каналу общения. И наконец, они гарантируют, что пользователь Б будет получать сообщения в правильном порядке. По этим причинам номера сообщений должны монотонно возрастать (т.е. более поздние сообщения должны иметь бóльшие номера, чем более ранние) и быть уникальными (не должно быть двух разных сообщений, имеющих одинаковые номера).

Назначать сообщениям номера очень просто. Пользователь А нумерует первое сообщение как 1, второе как 2 и т.п. Пользователь Б запоминает номер сообщения, которое было получено последним. Номер каждого нового сообщения должен быть больше номера предыдущего принятого сообщения. Принимая только сообщения с возрастающими номерами, пользователь Б гарантирует, что злоумышленник Е не сможет воспроизвести и отослать ему какое-нибудь старое сообщение.

Для разработки безопасного канала общения мы будем использовать 32-битовые номера сообщений. Первому сообщению присваивается номер 1. Общее количество сообщений ограничено числом $2^{32} - 1$. Если количество сообщений превысит это число, пользователю А придется прекратить применение текущего ключа и перезапустить протокол согласования ключей, чтобы сгенерировать новый ключ. Номер сообщения *должен* быть уникальным, поэтому мы не можем позволить, чтобы он вернулся к нулю.

Разумеется, мы бы могли использовать и 64-битовые номера сообщений, но это потребовало бы слишком больших расходов. (К каждому сообщению пришлось бы добавлять не четыре лишних байта, а восемь.) Для большинства систем вполне достаточно и 32-битовых номеров сообщений. Кроме того,

ключ все равно следует периодически изменять[1]. При необходимости вы, конечно, можете использовать 40- или 48-битовые номера сообщений; особого значения это не имеет.

Почему мы начинаем нумеровать сообщения с единицы, если в языке C нумерацию принято начинать с нуля? Это небольшая хитрость реализации. Если сообщениям могут быть присвоены N номеров, пользователям A и Б придется отслеживать $N + 1$ состояний. Действительно, ведь тогда количество отосланных сообщений может принимать любое значение из множества $\{0, \ldots, N\}$. Если ограничить количество возможных сообщений до $2^{32} - 1$, это состояние может быть представлено с помощью одного 32-битового числа. Если бы нумерация сообщений начиналась с нуля, нам бы пришлось ввести дополнительный флаг, указывающий, что пользователь A еще не отослал ни одного сообщения или же что пространство номеров сообщений было исчерпано. Наличие таких флагов требует реализации дополнительного и достаточно сложного кода, который практически никогда не будет использоваться. Если же этот код практически не используется, то и практически не тестируется, а значит, может отказать в нужный момент. По сути, введение дополнительного флага — лишь еще одно потенциальное слабое место, которого так легко избежать, начиная нумерацию сообщений с единицы.

В следующих разделах главы номер сообщения будет обозначаться как i.

8.3.2 Аутентификация

Для аутентификации сообщений мы будем использовать функцию вычисления MAC. Как вы уже, наверное, догадались, это будет функция HMAC-SHA-256 с полным 256-битовым результатом. Входное значение функции вычисления MAC будет состоять из сообщения m_i и дополнительных данных аутентификации x_i. Как отмечалось в главе 7, "Коды аутентичности сообщений", вместе с сообщением зачастую необходимо аутентифицировать и данные о контексте, в котором оно было отослано. Эти данные применяются пользователем Б для правильной интерпретации сообщения. Обычно они включают в себя номер версии протокола, размеры полей и подтверждение того, что это третье сообщение из последовательности сообщений, отправленных по каналу. Здесь мы просто определяем безопасный канал общения, поэтому реальное значение x_i должно предоставляться оставшейся частью приложения. С нашей точки зрения, каждый элемент x_i — это строка, значение которой одинаково для пользователей A и Б.

[1] Все ключи следует обновлять через разумные промежутки времени. Интенсивно используемые ключи необходимо обновлять почаще. Ограничение количества сообщений, которые могут быть зашифрованы или аутентифицированы с помощью одного и того же ключа, до $2^{32} - 1$ является вполне приемлемым.

Пусть $\ell(\cdot)$ — это функция, которая возвращает длину строки данных (в байтах). Значение MAC a вычисляется по формуле

$$a_i := \mathrm{MAC}(i \parallel \ell(x_i) \parallel x_i \parallel m_i),$$

где i и $\ell(x_i)$ — 32-битовые беззнаковые целые числа, представленные в формате, в котором наименее значимый байт записывается первым. Функция $\ell(x_i)$ гарантирует, что строка $i \parallel \ell(x_i) \parallel x_i \parallel m_i$ уникальным образом разбивается на нужные части. При отсутствии $\ell(x_i)$ разбивка соответствующей строки на i, x_i и m_i, а следовательно, и аутентификация были бы неоднозначными. Разумеется, данные x_i должны быть закодированы таким образом, чтобы их можно было разбить на разные поля без какой-либо дополнительной контекстной информации, однако гарантировать это на уровне безопасного канала общения мы не можем. За это должно отвечать приложение, использующее наш канал общения.

8.3.3 Шифрование

Для шифрования сообщений мы будем применять блочный шифр AES в режиме CTR. В роли уникального значения оказии, необходимого для шифрования в режиме CTR, будет выступать номер сообщения.

Размер каждого сообщения имеет ограничение в $16 \cdot 2^{32}$ байт, что ограничивает размер счетчика блоков до 32 бит. Разумеется, мы могли бы использовать и 64-битовый счетчик, однако 32-битовые значения легче обрабатывать на многих платформах. Кроме того, большинству приложений вообще не нужны такие громадные сообщения.

Ключевой поток состоит из байтов $k_0, k_1, \ldots$. Для сообщения с оказией i ключевой поток определяется как

$$k_0, \ldots, k_{2^{36}-1} := E(K, 0 \parallel i \parallel 0) \parallel E(K, 1 \parallel i \parallel 0) \parallel \ldots \parallel E(K, 2^{32} - 1 \parallel i \parallel 0),$$

где каждый блок открытого текста представляет собой конкатенацию 32-битового номера блока, 32-битового номера сообщения и 64 бит нулей. Полученный ключевой поток будет очень длинной строкой. Мы будем использовать только первые $\ell(m_i) + 32$ байт ключевого потока. (Надеемся, не стоит упоминать, что оставшуюся часть ключевого потока подсчитывать не нужно...) Мы выполним конкатенацию значений m_i и a_i и сложим эти байты с помощью операции XOR с последовательностью байтов $k_0, \ldots, k_{l(m_i)+31}$.

8.3.4 Формат пакета

Мы не можем просто отослать зашифрованное значение $m_i \parallel a_i$, так как пользователю Б нужно знать номер сообщения. Поэтому перед зашифрованным сообщением $m_i \parallel a_i$ будет добавлен номер сообщения i, представленный

в виде 32-битового целого числа, в котором наименее значимый байт записывается первым.

8.4 Детали реализации

Пришло время перейти к обсуждению деталей реализации безопасного канала общения. Для удобства наш канал общения определен как двунаправленный, чтобы в обоих направлениях можно было использовать один и тот же ключ. Если определить канал общения как однонаправленный, обязательно найдется кто-нибудь, кто использует в обоих направлениях один и тот же ключ и полностью разрушит безопасность системы. Определение канала общения как двунаправленного снижает риск возникновения подобной ситуации.

Для описания наших алгоритмов будем использовать псевдокод, понятный всем, кто хоть немного знаком с соглашениями, принятыми в языках программирования. Блоки операторов будут выделяться с помощью отступов и парных ключевых слов наподобие `if`/`fi` или `do`/`od`.

8.4.1 Инициализация

Первый алгоритм, представленный в этой главе, — это инициализация параметров канала общения. Она состоит из двух основных функций: установка ключей и установка номеров сообщений. Ключ канала общения будет выступать в качестве источника для генерации четырех дочерних ключей: ключа шифрования и ключа аутентификации, которые будут применяться для отправки сообщений пользователем А пользователю Б, а также ключа шифрования и ключа аутентификации, которые будут применяться для отправки сообщений пользователем Б пользователю А.

функция InitializeSecureChannel
вход: K Ключ канала общения, 256 бит.
R Роль. Указывает, кем является участник — пользователем А или Б.
выход: $\mathcal{S}$ Состояние безопасного канала общения.

Вначале построим четыре дочерних ключа. Это будут четыре строки ASCII без фиксированной длины и нуль-терминаторов.
KeySendEnc$\leftarrow \mathrm{SHA}_d - 256(K \parallel$ “Шифрование от А к Б”)
KeyRecEnc$\leftarrow \mathrm{SHA}_d - 256(K \parallel$ “Шифрование от Б к А”)
KeySendAuth$\leftarrow \mathrm{SHA}_d - 256(K \parallel$ “Аутентификация от А к Б”)
KeyRecAuth$\leftarrow \mathrm{SHA}_d - 256(K \parallel$ “Аутентификация от Б к А”)

Поменяем местами ключи шифрования и дешифрования, если участник — пользователь Б.
if R = "Пользователь Б" **then**
  swap(KeySendEnc, KeyRecEnc)
  swap(KeySendAuth, KeyRecAuth)
fi
Установим значения счетчиков отправленных и полученных сообщений равными 0. Счетчик отправленных сообщений — это номер последнего отправленного сообщения. Счетчик полученных сообщений — это номер последнего полученного сообщения.
(MsgCntSend, MsgCntRec) $\leftarrow (0, 0)$
Зафиксируем состояние.
$\mathcal{S} \leftarrow$ (KeySendEnc,
  KeyRecEnc,
  KeySendAuth,
  KeyRecAuth,
  MsgCntSend,
  MsgCntRec)
return $\mathcal{S}$

У нас есть еще одна функция, которая уничтожает значение состояния $\mathcal{S}$. Мы не будем рассматривать ее реализацию более подробно. Все, что требуется от данной функции, — это очистить и высвободить память, в которой хранится переменная $\mathcal{S}$. Обратите внимание на слово "очистить" — это очень важный момент. Во многих системах функция высвобождения памяти не обязательно очищает ее, поэтому по окончании использования состояния $\mathcal{S}$ его необходимо удалить явно.

8.4.2 Отправка сообщения

Рассмотрим, какие действия необходимо выполнить, чтобы отправить сообщение. Алгоритм отправки сообщения принимает на вход состояние сеанса, сообщение, которое должно быть отправлено, и дополнительные данные, которые должны быть аутентифицированы, а выдает сообщение, зашифрованное вместе со своим аутентификатором и готовое к отправке. Для аутентификации сообщения у получателя должны быть те же дополнительные данные, что и у отправителя.

функция SendMessage
вход: $\mathcal{S}$ Состояние безопасного сеанса.
  m Сообщение, которое должно быть отправлено.

x Дополнительные данные, которые должны подвергнуться аутентификации.

выход: t Данные, которые должны быть переданы получателю.

Вначале проверим номер сообщения и обновим его.

assert MsgCntSend$< 2^{32} - 1$

MsgCntSend $\leftarrow$ MsgCntSend $+ 1$

$i \leftarrow$MsgCntSend

Подсчитаем значение MAC. Каждое значение $\ell(x)$ и i представлено в виде 4-байтового числа, у которого наименее значимый байт записывается первым.

$a \leftarrow$HMAC-SHA-256(KeySendAuth, $i \parallel \ell(x) \parallel x \parallel m$)

$t \leftarrow m \parallel a$

Сгенерируем ключевой поток. Каждый блок открытого текста, подающийся на вход блочного шифра, состоит из 4-байтового счетчика блоков, четырех байтов значения i и восьми байтов нулей. Целые числа представлены в формате, в котором наименее значимый байт записывается первым. E — это функция шифрования AES с 256-битовым ключом.

$K \leftarrow$KeySendEnc

$k \leftarrow E_K(0 \parallel i \parallel 0) \parallel E_K(1 \parallel i \parallel 0) \parallel \ldots$

Сформируем окончательный вариант сообщения. И вновь значение i будет представлено 4-байтовым числом, в котором наименее значимый байт записывается первым.

$t \leftarrow i \parallel (t \oplus \text{First} - \ell(t) - \text{bytes}(k))$

return t

Данный алгоритм выглядит довольно просто (см. раздел 8.3). Вначале мы проверяем, не исчерпаны ли значения счетчика сообщений. Переоценить важность такой проверки невозможно. Если счетчик сообщений *когда-нибудь* вернется к нулю, это полностью разрушит безопасность системы. К сожалению, подобные ошибки встречаются слишком часто. Аутентификация и шифрование выполняются так, как было описано ранее в главе. И наконец, к зашифрованному и аутентифицированному сообщению добавляется значение i, чтобы получатель знал номер сообщения.

Обратите внимание: мы обновляем состояние сеанса, потому что количество отправленных сообщений изменяется. Это крайне важно, поскольку номер сообщения должен быть уникальным. Впрочем, практически каждая составляющая наших алгоритмов критически важна для безопасности системы.

8.4.3 Получение сообщения

Алгоритм получения сообщений принимает на вход зашифрованное сообщение с аутентификатором, полученное в результате применения функции SendMessage, и те же самые дополнительные данные x, которые должны быть аутентифицированы.

функция ReceiveMessage

вход: $\mathcal{S}$ Состояние безопасного сеанса.
t Текст, переданный отправителем.
x Дополнительные данные, которые должны подвергнуться аутентификации.

выход: m Сообщение, которое было отправлено.

Полученное сообщение должно, как минимум, содержать 4-байтовый номер сообщения и 32-байтовое значение MAC. Следующая проверка гарантирует, что все последующие операции разбивки сообщения на поля могут быть выполнены корректно.

assert $\ell(t) \geq 36$

Разобьем t на две части: i и зашифрованное сообщение плюс аутентификатор. Такая разбивка будет однозначной, поскольку длина i всегда равна 4 байт.

$i \parallel t \leftarrow t$

Сгенерируем ключевой поток точно так же, как это делал отправитель.

$K \leftarrow$KeyRecEnc

$k \leftarrow E_K(0 \parallel i \parallel 0) \parallel E_K(1 \parallel i \parallel 0) \parallel \ldots$

Расшифруем t и разобьем полученный результат на сообщение и значение MAC. Такая разбивка будет однозначной, поскольку длина a всегда равна 32 байт.

$m \parallel a \leftarrow t \oplus \textsc{First} - \ell(t) - \textsc{bytes}(k)$

Пересчитаем значение аутентификатора. Каждое значение $\ell(x)$ и i представлено в виде 4-байтового числа, у которого наименее значимый байт записывается первым.

$a' \leftarrow$ HMAC-SHA-256(KeyRecAuth, $i \parallel \ell(x) \parallel x \parallel m$)

if $a \neq a'$ **then**
  destroy k, m
  return AuthenticationFailure
else if $i \leq$ MsgCntRec **then**
  destroy k, m
  return MessageOrderError
fi

MsgCntRec← i
return m

В этом алгоритме использован канонический порядок операций. Вообще-то номер сообщения можно было бы проверять и перед расшифровкой, однако, если бы он был искажен злоумышленником, наша функция сгенерировала бы не ту ошибку, которую нужно. Вместо уведомления о том, что сообщение было искажено, пользователь был бы оповещен о неправильном порядке сообщений. Поскольку подобные ситуации зачастую требуется обрабатывать поразному, данная функция не должна предоставлять неверную информацию. Проверку номера сообщения ставят перед расшифровкой для того, чтобы быстрее отбрасывать ложные сообщения. Нам этот аспект не кажется очень важным; если вы получаете так много ложных сообщений, что скорость их отклонения начинает иметь значение, следовательно, вы столкнулись с гораздо более серьезными проблемами.

Нельзя не отметить еще один важный момент, касающийся получателя сообщений. Функция ReceiveMessage не должна раскрывать никакой информации о ключе или открытом тексте сообщения до тех пор, пока сообщение не прошло аутентификацию. Если аутентификация окончится неудачей, функция должна уведомить получателя об ошибке, но не возвращать никакой информации о ключевом потоке или открытом тексте. Вместо этого она должна очистить области памяти, в которых те хранились. Почему это так важно? Наличие открытого текста сообщения позволяет вычислить ключевой поток, поскольку мы исходим из предположения, что злоумышленник знает шифрованный текст. Опасность состоит в том, что злоумышленник отошлет ложное сообщение (с некорректным значением MAC). Сообщение будет отброшено, но злоумышленник все равно узнает ключевой поток по данным, выданным функцией получателя. Как видите, в дело вновь вступает параноидальная модель, согласно которой любая информация, преждевременно раскрытая или упущенная функцией ReceiveMessage, автоматически попадает в руки злоумышленника. Уничтожая данные, хранящиеся в переменных k и m, еще до того как функция ReceiveMessage возвратит ошибку, мы предотвращаем какую-либо утечку этих данных.

8.4.4 Порядок сообщений

Как и отправитель, получатель обновляет состояние $\mathcal{S}$, изменяя значение переменной MsgCntRec. Получатель гарантирует, что номера принятых им сообщений строго возрастают. Это исключает возможность повторного принятия одного и того же сообщения, однако, если в процессе передачи по сети поток сообщений будет переупорядочен, получателю придется отбросить целый ряд совершенно корректных (во всем остальном) сообщений.

Это упущение довольно легко исправить (хотя и за счет некоторых расходов). Если разрешить получателю принимать сообщения вне зависимости от их порядка, тогда приложение, использующее безопасный канал общения, должно обладать возможностью обработки сообщений, номера которых "выбиваются" из общей последовательности. Многие приложения не способны справиться с подобной ситуацией. Некоторые системы могут обрабатывать неупорядоченные сообщения, однако обладают небольшими изъянами (зачастую в области безопасности). В большинстве ситуаций мы предпочитаем исправлять ошибки транспортного уровня и гарантировать, что случайное перемешивание сообщений невозможно, а значит, безопасному каналу общения не придется сталкиваться с подобной проблемой.

Порой возникает ситуация, когда получатель допускает прием сообщений, прибывших не по порядку; и тому есть очень важная причина. Речь идет об IPSec — протоколе безопасности IP (IP Security) [50], который выполняет шифрование и аутентификацию IP-пакетов. Поскольку IP-пакеты могут быть переупорядочены в процессе передачи по сети и все приложения, использующие протокол IP, знают об этом свойстве, IPSec не просто запоминает номер последнего полученного сообщения, а отслеживает целый диапазон номеров сообщений. Если c — это номер последнего полученного сообщения, IPSec организует битовую карту для номеров сообщений $c-31, c-30, c-29, \ldots, c-1, c$. Каждый бит этой карты указывает на то, было ли получено сообщение с соответствующим номером. Сообщения с номерами, меньшими чем $c-31$, всегда отклоняются. Сообщения с номерами в диапазоне от $c-31$ до $c-1$ принимаются только тогда, когда значение соответствующего бита равно 0 (разумеется, после этого оно изменяется на 1). Если же номер нового сообщения больше c, значение c обновляется, а битовая карта соответствующим образом сдвигается. Подобная схема допускает ограниченный прием переупорядоченных сообщений и при этом не требует хранить слишком много сведений о состоянии.

8.5 Альтернативы

Данное нами определение безопасного канала общения не всегда практично. В частности, реализация безопасного канала общения во встроенном аппаратном обеспечении с использованием функции SHA-256 обходится слишком дорого. А нет ли каких-нибудь альтернативных решений?

Когда мы писали эту книгу, Нильс работал с клиентом, который столкнулся с аналогичной проблемой. Вначале в качестве режима работы блочного шифра предполагалось использовать OCB [82, 83]. Преимущество данного режима состоит в том, что и аутентификация и шифрование выполняются с помощью блочного шифра, а потому не требуют реализации дополнитель-

ных криптографических функций. К сожалению, проблемы с патентованием OCB сделали его использование дорогостоящим и весьма рискованным. Существуют и другие относительно новые режимы, которые обеспечивают схожую функциональность "аутентификация плюс шифрование", однако они обладают аналогичными проблемами с получением патентов.

Для решения этой проблемы Дуг Уайтинг (Doug Whiting), Расс Хаусли (Russ Housley) и Нильс скомбинировали режим CTR с аутентификацией CBC-MAC. Полученный режим работы блочного шифра получил название CCM (по первым буквам аббревиатур CTR, CBC и MAC) [93]. По сравнению с OCB режим CCM требует выполнения вдвое большего числа операций для шифрования и аутентификации сообщения, однако, насколько мы знаем, проблем с его патентованием нет вообще. Разработчикам CCM не известно ни одного патента, защищающего этот режим, а сами они патентовать его не собираются. Якоб Йонссон (Jakob Jonsson) получил доказательство безопасности для CCM [14], а NIST рассматривает вопрос стандартизации CCM в качестве режима работы блочного шифра.

Итак, что лучше использовать: OCB или CCM? Оба режима достаточно новы (CCM совсем новый), что заставляет относиться к ним с некоторым подозрением. Оба режима обладают доказательствами безопасности, которым мы не доверяем до конца. CCM представляет собой комбинацию хорошо проверенных методов, в то время как OCB использует совершенно новую технологию. OCB в два раза эффективнее, зато защищен всевозможными патентами.

Одно из существенных различий между режимами CCM и OCB состоит в их поведении при осуществлении атак на основе коллизий. В CCM применяется режим шифрования CTR, при использовании которого после шифрования более 2^{64} блоков данных (при условии, что размер блока составляет 128 бит) начинается утечка информации. Поэтому спецификации CCM ограничивают количество блоков, которые могут быть зашифрованы с помощью одного и того же ключа, до 2^{60}. При этом объем утечки информации опускается до несущественного (порядка доли бита). На часть режима CCM, ответственную за выполнение аутентификации, известных атак на основе коллизий пока что не существует. Таким образом, хотя CCM не позволяет полностью достигнуть 128-битового уровня безопасности, он практически приближается к этому уровню и работает так же хорошо, как другие режимы работы блочных шифров.

В отличие от CCM, для режима OCB существует атака на основе коллизий, направленная на функцию аутентификации. При возникновении коллизии свойства аутентификации будут полностью утеряны [34]. Это означает, что OCB не может обеспечить 128-битовый уровень безопасности. Если ограничить количество блоков, которые могут быть зашифрованы с помощью

одного и того же ключа, до 2^{48}, уровень безопасности OCB достигнет лишь 81 бит или около того. Нам кажется, что в долговременной перспективе этого недостаточно, хотя аналогичные уровни безопасности используются многими современными системами.

Думаем, вы нисколько не удивитесь, узнав, что мы предпочитаем CCM. Данный режим представляется нам вполне разумной альтернативой безопасному каналу общения, определенному в этой главе. Но, конечно же, мы не можем быть полностью объективными, поскольку сами принимали участие в разработке этого режима.

Еще один момент: CCM, OCB и аналогичные режимы не гарантируют полной безопасности канала общения. Они обеспечивают функциональность шифрования и аутентификации и требуют наличия ключа и уникальной оказии для каждого пакета. При необходимости наши алгоритмы безопасного канала общения легко адаптировать таким образом, чтобы они использовали подобные режимы, а не отдельные функции шифрования и вычисления MAC. Вместо четырех дочерних ключей, которые генерировались функцией InitializeSecureChannel, вам понадобится два ключа — по одному на каждое направление трафика. Значение оказии может быть получено путем дополнения номера сообщения до нужного размера.

8.6 Заключение

Безопасный канал общения — один из самых полезных компонентов, который применяется почти во всех криптографических системах. При наличии хороших функций шифрования и аутентификации построить безопасный канал общения не так сложно. Существует множество мелких деталей, требующих особого внимания и, конечно же, корректной реализации, но в криптографии это обычное дело. В реальной жизни главная сложность состоит не в реализации безопасного канала общения, а в том, чтобы убедить людей в его необходимости, а также осуществить безопасный обмен ключами.

Глава 9

Проблемы реализации. Часть I

После всего того, что уже было сказано в этой книге, можно перейти к обсуждению проблем реализации. Реализация криптографических систем настолько отличается от реализации остального программного обеспечения, что определенно заслуживает особого внимания.

Одна из основных проблем, как всегда, касается правила слабого звена (см. раздел 2.2). Нет ничего проще, чем пошатнуть безопасность системы на уровне реализации. В действительности ошибки реализации (в основном в виде переполнения буфера) представляют собой самую серьезную угрозу безопасности реальных систем. Если вы следили за проблемами компьютерной безопасности на протяжении последних нескольких лет, то, безусловно, понимаете, что мы имеем в виду. На практике криптографические системы взламываются крайне редко. Это происходит отнюдь не потому, что они такие надежные; мы уже проанализировали достаточно подобных систем, чтобы с уверенностью заявить, что это не так. Все дело в том, что найти ошибку реализации гораздо легче, чем искать уязвимое место в криптографических алгоритмах. Зачем же обременять себя проблемами криптоанализа при наличии более легкого пути?

До сих пор тематика наших обсуждений была ограничена вопросами криптографии. В этой главе, однако, мы решили сконцентрировать внимание на окружении, в котором функционируют криптографические алгоритмы. Каждая часть системы влияет на безопасность. Чтобы получить действительно надежную систему, все ее компоненты с самого начала должны разрабатываться не просто с оглядкой на безопасность; безопасность должна стать одной из главных целей. “Система”, о которой мы говорим, может быть очень большой. Она включает в себя все компоненты окружения, неправильное поведение которых может негативно отразиться на свойствах безопасности.

Одной из главных частей окружения, конечно же, является операционная система. К сожалению, ни одна из современных операционных систем не была

ориентирована в первую очередь на обеспечение безопасности. Из этого следует, что реализовать безопасную систему невозможно. Мы не знаем, как это сделать, и пока не видим никого, кто бы это знал. Реальные системы включают в себя массу компонентов, которые никогда не были ориентированы на обеспечение безопасности, поэтому достигнуть того уровня безопасности, который нам нужен, просто невозможно. Так, может быть, стоит махнуть на все рукой? Вовсе нет. Когда мы разрабатываем криптографическую систему, то прилагаем все усилия, чтобы гарантировать безопасность хотя бы нашей части системы. На первый взгляд это может напомнить психологию эгоиста, которого беспокоит только его маленький мирок. Это совсем не так: нас *действительно* беспокоят другие части системы; мы просто не в состоянии контролировать их разработку. Потому-то мы и решили написать эту книгу, чтобы другие тоже осознали коварную природу безопасности и поняли, насколько важно правильно выполнять свою работу.

Еще одна причина того, что криптографическая часть системы должна быть реализована правильно, уже рассматривалась в одной из предыдущих глав: атаки на криптографические алгоритмы особенно опасны потому, что могут остаться незамеченными. Если злоумышленнику удастся взломать криптографический алгоритм, вы вряд ли это заметите. Это можно сравнить с вором, у которого есть все ключи от вашего дома. Если вор предпримет необходимые меры предосторожности, как вы узнаете, что он у вас побывал?

Наша долгосрочная цель состоит в том, чтобы разрабатывать безопасные компьютерные системы. Для достижения этой цели каждый должен выполнять то, что от него требуется. Наша работа, которой и посвящена эта книга, — обеспечить безопасность компонентов, отвечающих за криптографию. Кто-то другой должен обеспечить безопасность остальных частей системы. Мы не знаем, как это делать, но, вероятно, это знают другие разработчики (а, может быть, когда-нибудь узнаем и мы). Пока этого не случится, безопасность системы в целом будет ограничена надежностью ее самого слабого звена, и мы сделаем все возможное, чтобы этим звеном никогда не стала криптография.

Нельзя не упомянуть и еще одну причину, по которой так важно правильно проектировать криптографические системы. Операционная система функционирует на отдельном компьютере. Криптографические системы, в свою очередь, часто применяются в коммуникационных протоколах, предназначенных для обмена данными между множеством компьютеров. Обновление операционной системы, установленной на отдельном компьютере, является вполне осуществимой задачей и на практике выполняется относительно часто. Изменение коммуникационных протоколов в сети — это сущий кошмар. Не зря же многие сети все еще работают с технологиями 70-х и 80-х годов прошлого века! Мы должны учитывать, что любая разработанная крипто-

графическая система в случае ее широкого распространения будет использоваться на протяжении еще 30-50 лет. Надеемся, что к тому времени другие части системы достигнут гораздо более высокого уровня безопасности.

9.1 Создание правильных программ

Источник всех проблем реализации кроется в том, что мы, IT-специалисты, не знаем, как написать правильную программу или модуль. ("Правильная" программа — это программа, которая ведет себя именно так, как описано в ее спецификациях.) Существует несколько причин того, почему нам так трудно написать правильную программу.

9.1.1 Спецификации

Первая причина касается отсутствия спецификаций. Для большинства программ не существует четкого описания того, что именно они должны делать. Если у программы нет спецификаций, мы даже не можем проверить, правильная она или нет. Сама концепция правильности для таких программ теряет свою определенность.

Многие программные проекты сопровождаются документом, называемым функциональной спецификацией. Теоретически это и есть спецификация программы. На практике, однако, этот документ либо не существует, либо является неполным, либо описывает вещи, не относящиеся к поведению программы. Не имея четкой спецификации, нечего и надеяться получить правильную программу.

Процесс создания спецификаций можно разделить на три этапа.

- **Требования.** Это неформальное описание того, для чего предназначена программа. Этот документ описывает скорее "*что* я могу сделать с помощью этой программы", нежели то, "*как* я могу это сделать". В большинстве случаев требования несколько расплывчаты и не содержат мелких деталей. Они сконцентрированы на программе в целом.
- **Функциональная спецификация.** Это подробное и исчерпывающее описание поведения программы. В функциональной спецификации должны содержаться только те элементы программы, поведение которых может быть оценено "из внешнего мира".
- Прежде чем помещать какой-либо элемент в функциональную спецификацию, подумайте, можно ли разработать тест для готовой программы, который будет определять, функционирует ли этот элемент должным образом. Тест должен использовать только внешнее поведение программы, а не ее внутренние механизмы. Если такой тест создать нельзя,

соответствующий элемент не должен быть указан в функциональной спецификации.

- Функциональная спецификация должна быть полной. Другими словами, она должна содержать описание абсолютно всех функциональных элементов программы. Все, что не было занесено в функциональную спецификацию, не должно быть реализовано.
- Для большей наглядности функциональную спецификацию можно представить себе в виде основы для тестирования готовой программы. Все элементы, указанные в функциональной спецификации, могут и должны быть протестированы.
- **План реализации.** У этого документа много названий. План реализации описывает внутреннее поведение программы. В нем указаны все те элементы, которые не могут быть протестированы извне. Хороший план реализации зачастую разбивает программу на несколько модулей и описывает функциональность каждого из них. Описание модуля, в свою очередь, можно рассматривать как составление требований к модулю с последующим созданием функциональной спецификации и плана реализации.

Из всех перечисленных документов наиболее важной, несомненно, является функциональная спецификация. Именно она служит основой для тестирования готовой программы. Иногда для разработки программы может вполне хватить неформальных требований или плана реализации, представляющего собой лишь несколько заметок на клочке бумаги. Но при отсутствии функциональной спецификации мы даже не сможем описать, чего мы достигли по окончании разработки программы.

9.1.2 Тестирование и исправление

Вторая проблема написания правильных программ касается повсеместно распространенного метода разработки “протестировать и исправить”. Программисты пишут программу и затем тестируют ее на предмет того, правильно ли она работает. Если она не работает или работает неправильно, программисты исправляют найденные ошибки и снова тестируют программу. Всем известно, что такой подход отнюдь не приводит к появлению правильной программы. Он позволяет лишь получить программу, которая будет работать (а точнее, возможно, будет работать) в наиболее стандартных ситуациях.

В 1972 году Эдсгер Дейкстра (Edsger Dijkstra) в одной из своих речей отметил, что тестирование может показать только наличие ошибок, а не их отсутствие [22]. Подмечено как нельзя точно. В идеале мы хотели бы создавать такие программы, правильность которых могла бы быть доказана.

К сожалению, современные методы доказательства правильности программ не способны справиться и с повседневными задачами программирования, не говоря уже о целых проектах.

Пока что специалисты по компьютерным технологиям не знают, как решить эту проблему. Может быть, когда-нибудь мы научимся доказывать правильность программы. Возможно, нам просто нужна более обширная и тщательная инфраструктура и методология тестирования. Но, даже не имея на руках полного решения, мы определенно способны сделать все возможное с помощью тех средств, которые у нас уже есть.

Существует несколько простых правил отслеживания ошибок, которые содержатся в каждом хорошем учебнике по программной инженерии.

- Если вы нашли ошибку, реализуйте тест, который будет обнаруживать эту ошибку. Убедитесь, что он действительно ее обнаруживает. Затем исправьте ошибку и убедитесь, что тест больше не обнаруживает ее. Впоследствии применяйте этот тест ко всем будущим версиям программы, чтобы проверить, не появится ли указанная ошибка снова.
- Найдя ошибку, подумайте, чем она вызвана. Нет ли в программе других мест, в которых может находиться аналогичная ошибка? Проверьте все эти места.
- Ведите учет всех найденных ошибок. Простой статистический анализ обнаруженных ошибок может показать, какая часть программы является самой проблемной, ошибки какого типа встречаются наиболее часто и т.п. Это необходимо для системы контроля качества.

Перечисленные правила даже нельзя назвать необходимым минимумом, однако выбирать все равно не из чего. Книг, посвященных качеству программного обеспечения, слишком мало, и все они в чем-то не согласуются друг с другом. Многие из них представляют конкретную методологию разработки программного обеспечения как *единственное* решение, а мы всегда с подозрением относимся к схемам, которые "лечат от всех болезней". Истина почти всегда находится где-то посередине.

9.1.3 Халатное отношение

Третья проблема — это невероятно халатное отношение к программному обеспечению большинства людей, работающих в компьютерной индустрии. Ошибки в программах воспринимаются как нечто само собой разумеющееся. Если ваш текстовый процессор неожиданно "упадет" и уничтожит все, что вы успели сделать за день, это будет воспринято как вполне обыденная ситуация. Зачастую всю вину пытаются переложить на пользователя: "*Вы* должны

были почаще сохранять документ". Производители программного обеспечения постоянно выпускают продукты, содержащие массу известных ошибок. Было бы полбеды, если бы они продавали только компьютерные игры, но сегодня от программного обеспечения зависит наша работа, наша экономика и — все больше и больше — наша жизнь. Если производитель машин обнаружит дефект в модели машины после ее поступления в продажу, он отзовет все экземпляры этой модели и исправит найденный дефект. Компании, занимающиеся разработкой программного обеспечения, поступают по-другому, всячески заявляя об ограничении своей ответственности во всевозможных лицензиях. В любой другой отрасли промышленности этого бы не спустили с рук. Такое халатное отношение означает отсутствие каких-либо серьезных попыток разработать правильное программное обеспечение.

9.1.4 Так что же нам делать?

Даже и не думайте, что вам будет достаточно лишь хорошего программиста, нескольких обзоров кода сторонними аналитиками, процесса разработки, сертифицированного по ISO 9001, обширного тестирования или даже комбинации всего этого. Действительность более сурова. Программное обеспечение слишком строптиво, чтобы его можно было приручить с помощью нескольких правил и процедур.

В качестве поучительного примера можно привести самую лучшую систему контроля качества в мире — авиационную промышленность. Каждый человек, работающий в этой отрасли, связан с обеспечением безопасности. Практически каждая операция выполняется согласно строгим правилам и процедурам. На случай отказа элементов системы предусмотрены многочисленные резервные компоненты. Каждый болт и каждая гайка самолета проходят тщательные технические испытания на пригодность, прежде чем их вообще разрешат использовать. Всякий раз, когда механик подходит к самолету хотя бы с самой безобидной отверткой, его работа проверяется и утверждается руководителем инженерно-авиационной службы. Любое изменение фиксируется самым тщательным образом. Каждое незначительное происшествие тщательно исследуется на предмет выявления всех возможных причин, которые затем исправляются. Эта беспрецедентная погоня за качеством оборачивается огромными расходами. Производство самолета обходится на порядок дороже, чем если бы вы просто передали его чертежи в обычную конструкторскую фирму. Но в то же время эта погоня за качеством оказывается исключительно эффективной.

Сегодня летать самолетом — совершенно привычное дело, а ведь даже незначительная ошибка в подобной машине была бы равносильна смерти. В машине, в которой нельзя просто нажать на тормоза и остановиться, если

что-нибудь пойдет не так. В машине, которая сможет вернуться на землю и при этом не разбиться, только совершив точную и мастерскую посадку на один из специально подготовленных участков, которых так немного на нашей планете. Авиационная индустрия удивительно преуспела в обеспечении безопасности полетов. Было бы хорошо, если бы у нее поучились и все остальные.

Возможно, написание правильного программного обеспечения *действительно* сто́ит на порядок дороже, чем на него привыкли тратить сейчас. Тем не менее, учитывая расходы, которыми оборачиваются ошибки программного обеспечения для современного общества, такое решение показало бы себя весьма эффективным в долгосрочной перспективе.

9.2 Создание безопасного программного обеспечения

До сих пор речь шла только о создании правильных программ. К сожалению, одного лишь правильного программного обеспечения недостаточно для построения системы безопасности. Программное обеспечение должно быть еще и безопасным.

В чем же различие между правильным и безопасным программным обеспечением? Правильное программное обеспечение обладает заявленной функциональностью: если вы щелкнете на кнопке *А*, случится событие *Б*. К безопасному программному обеспечению выдвигается еще одно требование: *недостаточность* функциональности. Что бы ни делал злоумышленник, он не должен выполнить операцию X. Это различие поистине можно назвать фундаментальным; можно протестировать программу на предмет функциональности, но никак не на предмет недостаточности функциональности. Аспекты безопасности программного обеспечения не могут быть протестированы каким-либо эффективным образом. Все это делает разработку безопасного программного обеспечения гораздо более сложной, нежели создание правильного программного обеспечения. Из этого следует вывод:

Стандартные методы реализации совершенно не подходят для написания безопасного кода.

В действительности мы не знаем, как создавать безопасный код. Проблема качества программного обеспечения настолько обширна, что на ее рассмотрение понадобилось бы несколько отдельных книг. Мы не разбираемся в этой теме настолько хорошо, чтобы написать их, но *знаем* наиболее распространенные проблемы реализации, касающиеся криптографии. Именно о них и пойдет речь в оставшейся части главы.

Прежде чем переходить к обсуждению этих проблем, давайте проясним нашу точку зрения: если вы не собираетесь вложить все свои силы в разработку безопасной системы, не стоит и браться за криптографию. Разработка криптографических систем может быть интересным занятием, однако, небрежно относясь к их реализации, вы лишь впустую потратите время.

9.3 Как сохранить секреты

Все, кто работает с криптографией, имеют дело с секретами. А секреты для того и нужны, чтобы сохранять их в тайне. Это значит, что программное обеспечение, связанное с секретной информацией, должно гарантировать, что ее утечка невозможна.

Работая с безопасным каналом общения, мы имеем дело с двумя типами секретов: ключи и данные. И те и другие являются временными; мы не хотим хранить их слишком долго. Данные хранятся только на протяжении того времени, пока мы обрабатываем каждое сообщение. Ключи хранятся только на протяжении времени существования безопасного канала общения. В этой главе обсуждаются только временные секреты. Как обеспечить хранение долгосрочных секретов, рассматривается в главе 22, “Хранение секретов”.

Временные секреты хранятся в оперативной памяти компьютера. К сожалению, память большинства современных компьютеров не обладает нужной безопасностью. Ниже описываются типичные проблемы, сопровождающие хранение секретов.

9.3.1 Уничтожение состояния

Основное правило написания систем безопасности гласит: уничтожайте всю информацию, как только она вам больше не понадобится. Чем дольше она будет храниться в памяти компьютера, тем вероятнее, что до нее доберется кто-нибудь посторонний. Более того, не забывайте уничтожать данные прежде, чем вы потеряете контроль над носителем, используемым для их хранения. Касательно временных секретов это означает очищение соответствующих областей памяти.

На первый взгляд кажется, что соблюсти это правило проще простого. Между тем оно приводит к возникновению огромного числа проблем. Если вы самостоятельно пишете весь текст программы на языке C, то сможете позаботиться об уничтожении данных и сами. Если же вы пишете библиотеку для использования другими людьми, то попадаете в зависимость от главной программы, которая должна сообщать вам о том, что то или иное состояние больше не понадобится. Например, при закрытии соединения криптографическая библиотека должна получить уведомление о том, что она может

уничтожить состояние сеанса безопасного канала общения. Библиотека даже может содержать специальную функцию для уничтожения состояния, но что, если разработчик приложения не станет утруждать себя вызовом еще одной функции? В конце концов, программа будет прекрасно работать и без обращения к этой функции.

В некоторых объектно-ориентированных языках программирования решить эту проблему несколько проще. В C++ у каждого объекта есть деструктор, который может автоматически уничтожать его состояние. Это стандартный прием, применяемый при написании систем безопасности на C++. Если главная программа ведет себя правильно и уничтожает все ненужные объекты, деструкторы автоматически очистят соответствующие области памяти. Язык C++ гарантирует, что при обработке исключения будут корректно уничтожены все объекты стека, однако уничтожением объектов, хранящихся в динамической памяти ("куче"), должна заниматься сама программа. Если для завершения работы программы вызывается функция операционной системы, она может не удосужиться даже пройти по стеку вызовов. Вам придется самим обеспечивать уничтожение данных, даже если программа вскоре должна завершить работу. В конце концов, операционная система не дает никаких гарантий того, что данные будут стерты из памяти в самое ближайшее время. Некоторые операционные системы вообще не утруждают себя очисткой освободившейся памяти, когда передают ее следующему приложению.

Даже если вы проделаете все необходимые операции по уничтожению объектов, ваши усилия могут быть сведены на нет. Некоторые компиляторы слишком усердствуют, стараясь все и всегда оптимизировать. Типичная функция, имеющая отношение к системе безопасности, выполняет несколько вычислений в локальных переменных и затем пытается уничтожить их, очистив соответствующие области памяти. В языке C для этого часто используют функцию `memset`. Хорошие компиляторы оптимизируют функцию `memset`, заменяя при трансляции вызов этой функции ее телом, что повышает скорость работы программы. Но некоторые компиляторы в своем стремлении к оптимизации заходят чересчур далеко. Они определяют, что уничтожаемая переменная или массив больше не будут использоваться в программе, и вообще пропускают функцию `memset`. Это еще больше повышает скорость работы программы, однако способно изменить ее поведение самым неожиданным образом. Нередко код программы выдает данные, случайно обнаруженные им в оперативной памяти. Если высвобожденная, но не очищенная память передается какой-нибудь библиотеке, использование последней может привести к утечке данных, которые тут же попадут в руки злоумышленника. Поэтому не забывайте проверять код, который генерирует ваш компилятор, и убеждаться в том, что секретные данные действительно удаляются из памяти компьютера.

Ситуация еще более усложняется в языках наподобие Java. Здесь все объекты находятся в динамической памяти, которая периодически подвергается очистке посредством сборщика мусора (garbage collector). Это означает, что метод `Finalize` (аналог деструктора в C++) не вызывается до тех пор, пока сборщик мусора не обнаружит, что объект больше не используется. Никаких спецификаций относительно того, как часто запускается сборщик мусора, не существует. Вполне вероятно, что секретные данные остаются в памяти на протяжении достаточно долгого времени. Использование обработки исключений значительно затрудняет очищение памяти вручную. Если программа выдает исключение, она проходит по стеку вызовов, не давая программисту никакой возможности вставить свой собственный код. Единственное, что можно было бы сделать в данной ситуации, — это представить *каждую* функцию в виде большого блока `try`. Разумеется, данное решение слишком уродливо и непрактично. Оно также должно было бы применяться на протяжении абсолютно всей программы, что сделало бы невозможным создание хорошей библиотеки безопасности для Java. В процессе обработки исключений Java спокойно проходит по стеку вызовов, выбрасывая ссылки на объекты и не уничтожая при этом самих объектов. В этом отношении Java действительно оказывается не на высоте. Самое лучшее решение, которое мы смогли придумать на данный момент, — это гарантировать запуск методов `Finalize` по крайней мере при завершении работы программы. Для этого метод `main` должен содержать операторы `try-finally`. Код, содержащийся в блоке `finally`, должен инициировать принудительную очистку памяти, дав указания сборщику мусора, чтобы тот попытался завершить все методы `Finalize`. (См. документацию к функциям `System.gc()` и `System.runInitialization()`.) Данный прием тоже не гарантирует, что методы `Finalize` действительно будут запущены, но это лучшее, что можно сделать.

Чего нам действительно не хватает — так это поддержки от самого языка программирования. В C++ есть хотя бы теоретическая возможность написать программу, уничтожающую содержимое всех объектов, которые уже не нужны. К сожалению, другие особенности этого языка делают его выбор крайне неудачным для написания систем безопасности. В Java уничтожить содержимое объекта явно практически невозможно. Было бы хорошо, если бы мы могли объявлять такие переменные, как “sensitive” (“требующие особого обращения”), что гарантировало бы уничтожение их содержимого. Еще лучше, если бы у нас был язык программирования, который бы всегда уничтожал все ненужные данные. Это бы позволило избежать массы ошибок без существенного снижения производительности.

Секретная информация может очутиться еще в нескольких местах. Все данные в конце концов попадают в регистры процессора. Большинство языков программирования не имеют средств для очистки регистров. Впрочем,

в процессорах наподобие Pentium, которые страдают от извечного "дефицита" регистров, вероятность того, что данные продержатся в регистрах на протяжении хоть сколько-нибудь долгого времени, крайне мала.

Во время переключения контекста (когда операционная система переключается с одной работающей программы на другую) значения, находящиеся в регистрах процессора, перемещаются в оперативную память, где могут оставаться на протяжении долгого времени. Насколько мы знаем, эта проблема не имеет иного решения, кроме внесения исправлений в саму операционную систему, что будет гарантировать конфиденциальность этих данных.

9.3.2 Файл подкачки

Большинство операционных систем (включая все текущие версии Windows и все версии UNIX) используют принцип виртуальной памяти для увеличения числа программ, которые могут быть запущены параллельно. В процессе работы программы не все ее данные хранятся в физической оперативной памяти. Некоторые из них переносятся в файл подкачки, находящийся на жестком диске. Когда программа пытается осуществить доступ к данным, которых нет в оперативной памяти, ее выполнение прерывается. Система виртуальной памяти извлекает необходимые данные из файла подкачки и переносит их обратно в оперативную память, после чего выполнение программы может быть продолжено. Если же системе виртуальной памяти понадобится больше свободной памяти, она извлекает из оперативной памяти произвольный сегмент данных и переносит его в файл подкачки.

Разумеется, большинство систем виртуальной памяти не предпринимают каких-либо серьезных попыток, чтобы сохранить данные в секрете или зашифровать их перед перенесением на диск. Практически все программные продукты ориентированы на коллективную работу единомышленников, а не на противоборствующее окружение, с которым имеют дело криптографы. Проблема состоит в том, что система виртуальной памяти может взять какие-нибудь данные нашей программы и записать их на жесткий диск в файл подкачки. Программа об этом даже и не узнает. Предположим, что в файл подкачки были перенесены секретные ключи. Если компьютер даст сбой или неожиданно будет выключен, данные останутся на диске. Следует отметить, что большинство операционных систем оставляют данные на диске и при корректном выключении компьютера. Механизма очищения файла подкачки, как правило, не существует, поэтому данные могут оставаться на диске до бесконечности. А что, если файл подкачки когда-нибудь попадет в руки

злоумышленника? Нет, мы определенно не можем допустить, чтобы наши секреты записывались в файл подкачки[1].

Как же помешать системе виртуальной памяти записывать наши данные на диск? В некоторых операционных системах есть специальные системные вызовы. С их помощью можно уведомить систему виртуальной памяти о том, что содержимое указанных областей памяти не должно подвергаться перемещению в файл подкачки. Иногда (хотя, к сожалению, крайне редко) нам все-таки попадаются операционные системы, которые поддерживают безопасную подкачку. В этом случае данные, перемещаемые в файл подкачки, подвергаются криптографической защите. Если ваша операционная система не обладает ни одной из этих возможностей, считайте, что вам не повезло. Громко пожалуйтесь на нее окружающим и сделайте все, что в ваших силах.

Предположим, вы можете заблокировать память и предотвратить перемещение ее содержимого в файл подкачки. Но какую же память следует блокировать? Разумеется, всю ту память, в которой могут храниться наши секретные данные. Тут возникает еще одна проблема. Во многих средах программирования практически невозможно определить, где именно хранятся те или иные данные. Объекты часто содержатся в "куче", константы — в статической памяти, а большинство локальных переменных оказываются в стеке. Определить более точное расположение данных при этом может быть затруднительно, да и не исключено появление ошибок. Вероятно, более удачным решением будет блокирование всей памяти, занятой приложением. Конечно же, это не так просто, как кажется на первый взгляд, поскольку может привести к потере целого ряда служб операционной системы, таких, как автоматически выделяемый стек. Кроме того, блокирование всей памяти приложения делает систему виртуальной памяти неэффективной.

Вообще говоря, правильное решение не должно быть таким сложным. В действительности нам нужна система виртуальной памяти, которая защищает конфиденциальность данных. Это подразумевает серьезное изменение операционной системы и, разумеется, не составляет прерогативы. Даже если следующая версия используемой вами операционной системы будет обладать подобной возможностью, придется тщательно проверить, действительно ли система виртуальной памяти способна сохранить данные в секрете.

9.3.3 Кэш

Современные компьютеры не ограничены каким-то одним типом памяти. Они обладают целой иерархией типов. В самом низу иерархии находится

[1] В действительности никогда не следует записывать секретные данные на какой-либо постоянный носитель, предварительно не зашифровав их, но об этом мы поговорим позже.

главная память, объем которой зачастую достигает сотен мегабайт. Но главная память функционирует сравнительно медленно, поэтому помимо нее существует и кэш. Кэш — это небольшая, зато более быстрая память. В кэше содержится копия недавно использованных данных из главной памяти. Когда процессору нужно осуществить доступ к каким-либо данным, он вначале проверяет свой кэш. Если в кэше имеются нужные данные, доступ процессора к ним осуществляется сравнительно быстро. Если же нужных данных в кэше нет, они считываются (относительно медленно) из главной памяти, после чего копия этих данных помещается в кэш для будущего использования. Чтобы освободить в кэше место для новых данных, из него выбрасывается копия каких-нибудь других данных.

Почему мы упомянули о кэше? Потому что в нем могут храниться копии данных, включая и копии наших секретных данных. Проблема состоит в том, что, когда мы пытаемся удалить из памяти наши секреты, они не всегда удаляются там, где нужно. В некоторых системах изменения данных вносятся только в кэш, а не в главную память. Они будут записаны в главную память только тогда, когда кэшу понадобится свободное место для размещения новых данных. Мы не знаем всех тонкостей работы этих систем, а они изменяются от процессора к процессору. Невозможно узнать, существует ли такое взаимодействие между единицей выделения памяти и системой кэширования, которое может привести к удалению данных из кэша без их записи в главную память, когда она высвобождается еще до сброса кэша. Производители никогда не объясняют, как следует поступить, чтобы гарантированно удалить из памяти конкретные данные. По крайней мере мы никогда не видели подобных спецификаций, а поскольку такие методы не задокументированы, доверять им не стоит.

Вторая опасность, связанная с наличием кэша, состоит в следующем. Иногда кэш узнает, что содержимое конкретной области памяти было изменено (возможно, другим процессором в многопроцессорной системе). В этом случае кэш помечает свои данные, которые когда-то находились в этой области памяти, как “недостоверные”, но не всегда уничтожает сами данные. Это может привести к появлению копии наших секретных данных, которая не будет уничтожена.

Мы мало что можем сделать в этой ситуации. Вообще-то опасность утечки данных из кэша невелика, так как в большинстве систем непосредственный доступ к кэшу может получать только код операционной системы. Кроме того, нам так или иначе приходится доверять операционной системе, поэтому мы можем довериться и этому механизму. Тем не менее нас беспокоит текущий принцип работы кэша, поскольку он никак не обеспечивает функциональности, которая требуется для реализации хороших систем безопасности.

9.3.4 Удерживание данных в памяти

Очень многих удивляет тот факт, что простое перезаписывание данных не обязательно приводит к их удалению. Детали этого процесса в некоторой степени зависят от типа используемой памяти, однако в общем случае, если данные помещаются в какую-либо область памяти, эта область начинает потихоньку "заучивать" или "удерживать" данные. Когда вы перезаписываете данные или выключаете компьютер, старые данные теряются не полностью. В зависимости от обстоятельств, простое выключение и включение питания может восстановить некоторые или все старые данные. Другие типы памяти могут "вспомнить" старые данные, если доступ к ним осуществляется в тестовых режимах (часто незадокументированных) [38].

Причина этого феномена кроется в принципах работы нескольких механизмов. Если одни и те же данные на протяжении достаточно долгого времени хранятся в одной и той же области статической оперативной памяти (Static RAM — SRAM), они будут рассматриваться как предпочтительное состояние загрузки этой памяти. Наш друг столкнулся с этой проблемой много лет назад при сборке компьютера в домашних условиях [9]. Он написал BIOS, которая использовала магическое число (magic value), расположенное в конкретной области памяти, чтобы определить, какая выполняется перезагрузка: "холодная" (cold reboot) или "горячая" (warm reboot)[2]. Через некоторое время машина отказалась загружаться после выключения и повторного включения компьютера, потому что оперативная память "выучила" магическое число, в результате чего каждая перезагрузка воспринималась системой как горячая. Поскольку в процессе горячей перезагрузки не инициализировались некоторые нужные переменные, система не могла загрузиться. Для устранения этой проблемы пришлось поменять местами некоторые модули памяти, чтобы SRAM не могла "вспомнить" магическое число, которое она "выучила". Для нас это послужило уроком: оперативная память сохраняет больше данных, чем мы думаем.

Аналогичные, хотя и несколько более сложные процессы происходят с динамической оперативной памятью (Dynamic RAM — DRAM). Кодирование битов в памяти DRAM выполняется путем помещения небольшого электрического заряда в крошечный конденсатор. Изоляционный материал вокруг пластин конденсатора подвергается напряжению, что приводит к измене-

[2] В те времена компьютеры, собранные в домашних условиях, программировались путем непосредственного введения двоичной формы машинного языка, что, конечно же, приводило к появлению массы ошибок. Один из гарантированных способов восстановить машину после "падения" программы заключался в перезагрузке компьютера. Холодная перезагрузка — это выключение и повторное включение компьютера. Горячая перезагрузка — это перезагрузка наподобие той, что выполняется при нажатии кнопки Reset. В процессе горячей перезагрузки система не выполняет повторную инициализацию всего состояния, а значит, не уничтожает настройки, сделанные пользователем.

нию структуры материала, в частности к перемещению в нем примесей [38]. Как следствие этого, злоумышленник, имеющий физический контроль над памятью, обладает потенциальной возможностью восстановления секретных данных.

Можно долго спорить о том, действительно ли описанная ситуация представляет сколько-нибудь значительную угрозу секретности данных, однако нам этот вопрос кажется важным. Если ваш компьютер когда-нибудь подвергнется опасности (например, будет украден), вы, конечно же, не захотите, чтобы злоумышленник добрался и до ваших данных, которые находились в памяти компьютера и затем были стерты. Для достижения этой цели нужно заставить компьютер забыть информацию.

Пока что нами придумано лишь частичное решение этой проблемы, которое срабатывает только при наличии определенных (и достаточно разумных) предположений об оперативной памяти. Это решение, которое мы назвали Boojum[3], подходит для сравнительно небольших объемов данных, таких, как ключи. Пусть m — это данные, которые нужно сохранить в оперативной памяти. Вместо того чтобы сохранять m, мы генерируем случайную строку R и сохраняем значения R и $R \oplus m$. Эти значения сохраняются в различных областях памяти, по возможности расположенных не слишком близко друг от друга. Хитрость состоит в том, чтобы регулярно изменять значение R. Через одинаковые промежутки времени (скажем, 100 мс) мы генерируем новое случайное значениеR' и обновляем содержимое памяти, сохраняя значения $R \oplus R'$ и $R \oplus R' \oplus m$. Это гарантирует, что поверх каждого бита памяти будет перезаписана случайная последовательность битов. Чтобы очистить память, мы просто создаем новое m, значение которого равно нулю. В результате этого в обеих областях памяти окажутся два равных случайных значения.

Чтобы считать необходимую информацию из упомянутых областей памяти, нужно считать оба значения и сложить их с помощью операции XOR. Результатом этого сложения и будет значение m. Чтобы перезаписать m, следует прибавить новое значение к R и записать полученный результат во вторую область памяти.

Необходимо тщательно следить за тем, чтобы биты значений R и $R \oplus m$ не оказались смежными на микросхеме памяти. Если не известно, как функционирует микросхема памяти, это может быть непростой задачей. В большинстве типов памяти, однако, биты данных хранятся в узлах прямоугольной решетки, причем одни биты адреса области памяти указывают на строку, а другие — на столбец. Если адреса двух элементов данных различаются на 0x5555, тогда с очень высокой долей вероятности эти данные не являются смежными. (Разумеется, это предполагает, что микросхема памяти не использует четные биты адреса в качестве номера строки, а нечетные — в качестве

[3]По имени Буджума — героя книги Льюиса Кэрролла “Охота на снарка” [15].

номера столбца, но подобная архитектура нам никогда не попадалась.) Более удачным решением является случайный выбор двух адресов в очень большом адресном пространстве. В этом случае вероятность того, что две выбранные области памяти окажутся смежными, чрезвычайно мала вне зависимости от реальной структуры микросхемы памяти.

Как уже отмечалось, данное решение является частичным и к тому же достаточно громоздким. Оно применимо только к небольшим объемам данных — в противном случае функция обновления оказалась бы слишком дорогостоящей. Тем не менее использование этого решения гарантирует, что на микросхеме памяти не будет физической точки, которая длительное время будет находиться под напряжением или без напряжения (в зависимости от значений секретных данных).

Наше решение все еще не позволяет гарантировать полную очистку памяти. Если вы прочитаете документацию к микросхеме памяти, то не найдете там спецификаций, описывающих, как предотвратить удержание данных, когда-либо в ней хранившихся. Разумеется, ни одна микросхема не в состоянии удержать абсолютно все предыдущие данные, тем не менее это показывает, что мы можем достичь не более чем эвристической безопасности.

До сих пор, описывая Boojum, мы говорили только о главной памяти. Это же решение применимо и к кэш-памяти за исключением того, что вы не сможете контролировать, в каком узле микросхемы будут храниться данные. К сожалению, Boojum не подходит для регистров процессора, однако они применяются так часто для хранения огромного количества самой разной информации, что возникновение эффекта удержания весьма маловероятно. С другой стороны, регистры расширения, такие, как регистры с плавающей запятой или дополнительные регистры MMX, используются гораздо реже, а значит, представляют собой потенциальную проблему.

Если у вас есть большие объемы данных, которые нужно сохранить в секрете, тогда сохранение в оперативной памяти двух значений и регулярное прибавление к ним новых случайных строк может оказаться слишком дорогостоящим решением. В этом случае предпочтительнее зашифровать большой блок данных и сохранить в памяти шифрованный текст. Избежать “удержания” должен только ключ, для сохранения которого можно воспользоваться алгоритмом Boojum. Более подробно это рассматривается в статье [23].

9.3.5 Доступ других программ

Сохранение секретных данных на компьютере сопряжено еще с одной проблемой — возможностью доступа к данным других программ, установленных на том же компьютере. Некоторые операционные системы разрешают совместное использование памяти различными программами. Если другая программа может считывать ваши секретные ключи, вы в опасности. Зачастую

использование общей памяти должно подтверждаться обеими программами, что несколько снижает риск. Но иногда общая память может быть предоставлена автоматически как результат загрузки общей библиотеки.

Огромную опасность в этом отношении представляют программы-отладчики. Современные операционные системы часто содержат средства, предназначенные для использования отладчиками. Различные версии Windows позволяют подключить отладчик к уже запущенному процессу. Отладчику разрешено делать многое, в том числе и читать память. Кроме того, в UNIX иногда возможно создать дамп оперативной памяти для конкретной программы. Дамп оперативной памяти (core dump) — это файл, содержащий образ памяти, в которой хранятся данные программы, включая и все секретные данные.

Еще одна угроза исходит от особо могущественных пользователей. Такие люди, называемые *суперпользователями (superusers)* или *администраторами (administrators)*, могут осуществлять доступ к содержимому компьютера, недоступному для обычных пользователей. Например, в операционной системе UNIX суперпользователь может читать любую часть памяти.

В общем случае ваша система не сможет эффективно защитить себя от подобных типов атак. Если вы достаточно осторожны, то, вероятно, сможете избежать некоторых из перечисленных проблем, но обычно добиться удается немногого. Тем не менее все эти вопросы следует тщательно рассмотреть для конкретной платформы, с которой вы работаете.

9.3.6 Целостность данных

Помимо сохранения секретов, мы должны защищать и целостность данных. Для защиты целостности данных в процессе их передачи по каналу общения используется код аутентичности сообщения (MAC), но как же быть, если данные будут изменены в памяти компьютера?

В этой главе мы предполагаем, что аппаратное обеспечение является надежным. Если оно ненадежно, вам вряд ли удастся сделать что-нибудь стоящее. Если вы не уверены в надежности своего оборудования, потратьте немного времени и памяти на то, чтобы это проверить, хотя в действительности такой работой должна заниматься операционная система. Лично мы всегда стараемся гарантировать, что главная память компьютера поддерживает технологию кода коррекции ошибок (error correcting code — ECC)[4]. Если в каком-нибудь одном разряде памяти произойдет ошибка, ECC обнаружит и исправит ее. При отсутствии ECC ошибка в любом разряде приведет к тому, что процессор считает неправильные данные.

[4]Убедитесь, что все компоненты компьютера поддерживают ECC. Остерегайтесь более дешевых модулей памяти, которые не сохраняют дополнительную информацию, а пересчитывают ее “на лету”. Это сводит на нет само назначение ECC.

Почему это так важно? Память современных компьютеров содержит гигантское количество разрядов. Предположим, что модули памяти обладают высоким качеством сборки, а значит, вероятность сбоя каждого разряда в конкретную секунду времени составляет только 10^{-15}. Если в компьютере установлены 128 Мбайт памяти, у нас есть около 10^9 разрядов, а следовательно, сбоя одного разряда можно ожидать каждые 11 дней, что, разумеется, не слишком хорошо для наших систем. Частота ошибок возрастает с увеличением объема памяти, поэтому в компьютере с 1 Гбайт памяти появления ошибки можно ожидать каждые 32 часа. Серверы обычно используют память с поддержкой ECC, поскольку обладают бóльшим количеством памяти и вынуждены беспрерывно работать на протяжении длительных периодов времени. Хотелось бы, чтобы подобной стабильностью обладали и все другие машины.

Разумеется, данная проблема касается исключительно аппаратного обеспечения. В большинстве случаев вы не сможете проконтролировать тип памяти компьютера, на котором будет запущена готовая система.

Некоторые угрозы, касающиеся нарушения конфиденциальности данных, могут привести и к нарушению их целостности. Отладчики иногда способны изменить содержимое памяти, используемой программой. Ничем не лучше суперпользователи, которые могут изменить содержимое памяти напрямую. И хотя от вас в этой ситуации ничего не зависит, знать о потенциальной опасности все же необходимо.

9.3.7 Что делать

Сохранить в секрете данные на современном компьютере отнюдь не так просто, как кажется. Существует множество путей утечки информации. Для получения действительно эффективной системы безопасности их нужно перекрыть. К сожалению, современные операционные системы и языки программирования не обеспечивают поддержки, необходимой для полного прекращения утечки. Вам придется сделать все, что в ваших силах. Это потребует выполнения большого объема работы, специфической для используемого окружения.

Помимо всего прочего, описанные проблемы значительно затрудняют создание библиотеки криптографических функций. Сохранение данных в секрете часто требует внесения изменений в главную программу. И разумеется, главная программа также работает с данными, которые должны быть сохранены в секрете; в противном случае ей бы вообще не понадобилась криптографическая библиотека. Это одна из распространенных проблем безопасности, которая влияет на все части системы.

9.4 Качество кода

Реализуя криптографическую систему, необходимо тщательно заботиться о качестве кода. В нашей книге речь идет не о программировании. Тем не менее, поскольку большинство учебников по программированию зачастую оставляют качество кода за пределами обсуждения, мы решили посвятить этому вопросу несколько разделов.

9.4.1 Простота

Сложность — главный враг безопасности. Каждый разработчик системы безопасности должен стремиться к простоте. Нужно сказать, что в этом отношении мы довольно безжалостны (правда, это не добавляет нам популярности). Уберите из системы все параметры и настройки, какие только возможно. Избавьтесь от всех тех "наворотов", которые практически никому не нужны. Избегайте разработки структуры системы на заседании комитета, поскольку попытки достигнуть компромисса всегда приводят к появлению в системе дополнительных средств или параметров. В плане безопасности главной должна быть простота.

В качестве типичного примера простой системы можно привести наш безопасный канал общения. У него нет настроек. Он не позволяет зашифровать данные без аутентификации или применить аутентификацию без шифрования. Многие клиенты просят реализовать подобные возможности, но в большинстве случаев они просто не представляют себе последствий частичного использования средств безопасности. Многие пользователи знают о безопасности отнюдь не достаточно, чтобы самостоятельно выбирать правильные параметры безопасности. Лучшее, что можно предложить в подобной ситуации, — это вообще лишить систему настраиваемых параметров и сделать ее безопасной по умолчанию. Если же без настроек все-таки не обойтись, оставьте один параметр — безопасная или небезопасная.

Многие криптографические системы также оснащены несколькими пакетами шифров. В таких системах пользователь (или кто-нибудь другой) может самостоятельно выбирать шифр и функцию аутентификации. На наш взгляд, этого делать не рекомендуется. Оставьте один режим, который будет достаточно безопасным для всех возможных областей применения. Разница в объеме вычислений для разных режимов шифрования не так уж велика, а криптография редко является узким местом в производительности современных компьютеров. Помимо того что это сделает систему менее сложной, вы застрахуете пользователя и от потенциальной опасности, связанной с выбором заведомо слабых шифров. Ведь если правильный выбор функций шифрования и аутентификации сложен даже для разработчика, неужели с ним справится обычный пользователь?

9.4.2 Модуляризация

Даже после удаления многочисленных параметров и настроек система все равно останется сложной. Существует один основной прием, позволяющий сделать эту сложность хоть немного более управляемой: модуляризация. В этом случае система разбивается на отдельные модули, после чего каждый из них разрабатывается, анализируется и реализуется.

Безусловно, вы уже не раз встречались с модуляризацией. В криптографии же правильная разбивка на модули играет еще более важную роль. Ранее мы уже рассматривали криптографические объекты и функции в качестве отдельных модулей. Интерфейс модуля должен быть простым и понятным. Его поведение должно соответствовать разумным ожиданиям пользователя этого модуля. Приглядитесь к интерфейсам своих модулей. Зачастую они включают в себя средства или настройки, предназначенные для решения задач какого-нибудь другого модуля. Если это возможно, выбросите такие средства. Каждый модуль должен заниматься только собственными проблемами. По своему опыту мы знаем, что наличие в интерфейсе модуля "странных" средств практически всегда является результатом неадекватного проектирования. Такое программное обеспечение определенно требует переделки.

Правильная разбивка на модули играет особенно важную роль, поскольку это единственный имеющийся у нас эффективный способ борьбы со сложностью. Если действие какой-нибудь настройки ограничивается одним модулем, она может быть проанализирована в контексте этого модуля. Тем не менее, если настройка изменяет внешнее поведение одного модуля, она может повлиять и на другие. Если у нас есть 20 модулей, каждый из которых обладает двоичным параметром, изменяющим поведение этого модуля, у системы появляется более миллиона различных конфигураций. Для проверки безопасности системы нам придется проанализировать каждую из этих конфигураций, что, разумеется, невыполнимо на практике.

Мы обнаружили, что большинство параметров и настроек добавляются к системе для повышения ее эффективности. Это одна из распространенных проблем программной инженерии. Многие системы содержат так называемые оптимизации, которые на деле оказываются бесполезными, непродуктивными или же незначительными, поскольку они оптимизируют совсем не те части системы, которые образуют узкое место. Поэтому мы довольно скептически относимся к оптимизациям, обычно не уделяя им внимания. Мы тщательно прорабатываем структуру системы и пытаемся убедиться, что она может выполнять сразу большие "порции" работы. В качестве типичного примера можно привести старую IBM PC BIOS. Ее процедура, выполняющая печать символа на экране, принимала в качестве аргумента один символ. Практически все время работы этой процедуры шло на выполнение массы служебных

операций, и лишь небольшая его часть непосредственно касалась вывода символа на экран. Если бы интерфейс процедуры позволял указывать в качестве аргумента не символ, а строку символов, тогда печать целой строки выполнялась бы лишь немного медленнее, чем печать одного символа. Результатом такого неудачного проектирования стала ужасающе медленная скорость вывода на экран всех компьютеров, работающих под управлением DOS. Этот же принцип касается и проектирования криптографических систем. Убедитесь, что работа может выполняться сразу большими порциями. Затем оптимизируйте только те части программы, влияние скорости работы которых на общую производительность системы может быть *измерено* и является существенным.

9.4.3 Утверждения

Утверждения (assertions) — это хорошее средство, с помощью которого можно повысить качество кода[5].

Реализуя криптографический код, необходимо подходить к нему с точки зрения "профессиональной паранойи". Каждый модуль не доверяет другим модулям и всегда проверяет достоверность параметров, накладывает ограничения на вызывающую последовательность и отказывается выполнять небезопасные операции. В большинстве случаев это описывается явными утверждениями. Если спецификации модуля утверждают, что перед использованием объекта его необходимо инициализировать, тогда попытка использования объекта до его инициализации приведет к возникновению ошибки утверждения. Ошибки утверждения всегда должны приводить к аварийному завершению программы с выдачей подробного сообщения о том, какое из утверждений было нарушено и почему.

Общее правило выглядит следующим образом: каждый раз, когда вы выполняете какую-либо значимую проверку внутренней согласованности системы, добавьте к ней утверждение. Постарайтесь перехватить как можно больше ошибок (и собственных, и допущенных другими программистами). Ошибка, перехваченная с помощью утверждения, не приведет к появлению "бреши" в системе безопасности.

Некоторые программисты реализуют проверку утверждений в процессе разработки программного обеспечения, однако отключают ее в готовом продукте. Хотелось бы знать, кто это выдумал. Что бы вы сказали об атомной электростанции, операторы которой тренируются работать с реакторами при включенных системах безопасности, а затем отключают их на настоящем

[5]Мы знаем, что наша книга потихоньку превращается в урок программирования, но что поделать? Мы постоянно вынуждены повторять эти, казалось бы, очевидные вещи программистам, с которыми работаем.

реакторе? Или о парашютисте, который одевает запасной парашют на тренировках, но снимает его, когда выпрыгивает из самолета? Зачем кому-то вообще понадобилось отключать проверку утверждений в готовом продукте — ведь это, по сути, единственное место, где она нужна? Если в процессе функционирования реальной системы произойдет нарушение утверждения, мы всего-навсего получим ошибку программирования. Игнорирование ошибки, в свою очередь, может привести (и, скорее всего, приведет) к выдаче неверного ответа, потому что по крайней мере одно предположение, сделанное кодом, будет неправильным. Выдача неверных ответов — это, пожалуй, худшее, что может сделать программа. Гораздо лучше проинформировать пользователя о возникновении ошибки программирования, чтобы он не доверял ошибочным результатам, выданным программой. Никогда не отключайте проверку ошибок!

9.4.4 Переполнение буфера

Современной IT-индустрии должно быть стыдно за появление в нашей книге раздела с таким заголовком. Проблемы переполнения буферов преследуют нас на протяжении уже 40 лет. Все это время в мире существовали и решения для борьбы с ними. Некоторые из ранних языков программирования высокого уровня, например Algol 60, полностью решали эту проблему путем введения обязательной проверки границ массива. К сожалению, даже несмотря на наличие массы поистине замечательных решений, переполнение буферов до сих пор является причиной более половины всех проблем безопасности, возникающих в Internet. Устранять же эти проблемы никто не собирается. Мы считаем это преступной халатностью. Что бы мы подумали, если бы производитель автомобилей сделал бензобак из оберточной бумаги? Конечно же, при определенной доле везения машина могла бы прекрасно ездить и с таким бензобаком, но руководство компании-производителя все равно угодило бы за решетку. Между тем целые отрасли IT-индустрии ведут себя так, будто не являются ответственными за последствия своих действий. (Возможно, наши юристы разрешают им свободно отказываться от своих обязательств, что было бы неприемлемо в любой другой области.) Наблюдая подобное отношение к программному обеспечению, мы часто задумываемся: а стоит ли вообще пытаться внедрять что-нибудь такое сложное, как криптография?

К сожалению, мы не в состоянии изменить текущее положение дел. Мы можем лишь посоветовать вам, как написать хороший криптографический код. Откажитесь от языков программирования, которые допускают переполнение буфера. В частности, не используйте C или C++. И никогда не отключайте проверку границ массива, какой бы язык вы не использовали. Это,

казалось бы, очень простое правило может решить половину проблем безопасности вашей системы.

9.4.5 Тестирование

Развернутое тестирование — обязательная часть любого хорошего процесса разработки программного обеспечения. Тестирование помогает найти ошибки в коде программы, однако оно бессильно в поиске "дыр" безопасности. Никогда не путайте тестирование с анализом безопасности. Эти процессы прекрасно дополняют друг друга, но между ними нет ничего общего.

Существует два типа тестов, которые должны быть реализованы при разработке каждого программного продукта. Первый из них — это общий набор тестов, разработанных на основе функциональных спецификаций модуля. В идеале один программист должен реализовать модуль, а другой — тесты для этого модуля. И тот и другой работают на основе функциональных спецификаций. Любое непонимание между ними означает, что спецификации нуждаются в прояснении. Общие тесты должны по возможности охватывать весь спектр операций, выполняемых модулем. Для одних модулей это просто, для других же тестовая программа должна полностью симулировать все окружение. В большинстве наших проектов код тестов имеет практически такой же размер, что и код самой системы, и как-нибудь улучшить эту ситуацию не представляется возможным.

Второй набор тестов разрабатывается самим автором модуля. Эти тесты предназначены для проверки каких-либо ограничений реализации. Например, если внутренние механизмы модуля используют буфер размером 4 Кбайт, то дополнительная проверка границ в начале и в конце буфера поможет выявить любые ошибки, связанные с управлением буфером. Иногда для разработки подобных конкретных тестов требуется хорошо знать внутреннюю структуру модуля.

Мы часто пишем последовательности тестов, основанные на применении генератора случайных чисел. Более подробно генераторы псевдослучайных чисел (pseudorandom number generator — PRNG) рассматриваются в главе 10, "Генерация случайных чисел". Использование генератора псевдослучайных чисел позволяет легко получать большое количество тестов. Если мы сохраним начальное число, которое использовалось генератором псевдослучайных чисел, то сможем повторить ту же последовательность тестов, что очень удобно для тестирования и отладки. Дальнейшие детали этого процесса зависят от специфики тестируемого модуля.

И наконец, чрезвычайно полезно создать какой-нибудь "быстрый тест", который будет выполняться при каждом запуске программы. В одном из своих последних проектов Нильс должен был реализовать шифр AES. При запуске

этой системы код инициализации "прогоняет" AES на нескольких тестовых событиях и проверяет, соответствуют ли полученные результаты заранее известным правильным ответам. Если в процессе дальнейшей разработки приложения код AES начнет работать нестабильно, быстрый тест Нильса сразу же выявит проблемы.

9.5 Атаки с использованием побочных каналов

Существует целый класс атак, называемых *атаками с использованием побочных каналов (side-channel attacks)* [48]. Такие атаки становятся возможными, когда у злоумышленника появляется дополнительный канал информации об интересующей его системе. Например, злоумышленник может измерить, сколько времени уходит у системы на то, чтобы зашифровать сообщение. Если криптографический алгоритм встроен в смарт-карту, злоумышленник может узнать, сколько электричества потребляет смарт-карта в течение определенного времени. В качестве побочных каналов могут выступать магнитные поля, радиоизлучение, потребление энергии, время выполнения операции и даже помехи, вызываемые на других каналах данных.

Неудивительно, что атаки с использованием побочных каналов часто оказываются удачными по отношению к системам, которые проектировались без учета таких атак. Особенно успешными в этой области оказались *атаки измерения энергии (power attacks)*, применявшиеся для взлома смарт-карт [56].

Защитить систему от всех видов атак с использованием побочных каналов очень сложно, а то и вообще невозможно. Тем не менее существует несколько простых мер предосторожности, которые позволяют значительно снизить риск нападения. Когда-то давным-давно Нильс работал над внедрением криптографических механизмов в смарт-карты. В то время одно из правил проектирования требовало, чтобы последовательность, в которой процессор выполняет инструкции, зависела только от той информации, которая и так известна злоумышленнику. Это сводило на нет назначение *тайминг-атак* (*timing attacks* — атаки, основанные на сравнительных измерениях времени) и значительно усложняло проведение атак измерения энергии, потому что анализ последовательности выполняемых инструкций больше не позволял получить никаких сведений о неизвестной информации. Разумеется, это не полное решение проблемы, и современные приемы измерения энергии легко справляются со взломом тогдашних смарт-карт. Тем не менее это было лучшее, что мы могли сделать для защиты смарт-карт в те дни. Спасение от атак с использованием побочных каналов всегда подразумевает применение некоторой комбинации контрмер. Одни из них осуществляются посредством

программного обеспечения, которое реализует криптографическую систему, а другие — с помощью самого аппаратного обеспечения.

Попытки предотвратить атаки с использованием побочных каналов напоминают мышиную возню. Мы пытаемся защитить систему от известных побочных каналов, а в это время какой-нибудь хитроумный злоумышленник находит новый побочный канал, после чего нам приходится вернуться назад и учесть еще один тип атаки. В реальной жизни, к счастью, все обстоит не так плохо, поскольку в большинстве случаев атаки с использованием побочных каналов выполнить весьма сложно. Побочные каналы представляют огромную опасность для смарт-карт, потому что последние находятся под полным контролем злоумышленника, однако большинство других компьютерных систем гораздо меньше страдают от подобных нападений. На практике наиболее важными побочными каналами являются время выполнения операции и радиоизлучение. (Смарт-карты, в свою очередь, особенно чувствительны к измерению энергопотребления.)

9.6 Заключение

Надеемся, что эта глава донесла до вас одну важную вещь: безопасность системы не начинается с криптографии и не заканчивается ею. В обеспечении безопасности должны принимать участие все компоненты системы. Вот почему все так не любят людей, занимающихся безопасностью: они суют свой нос абсолютно во все дела, советуют всем и каждому, как нужно работать, да еще и запрещают использовать кучу полезных вещей только потому, что они, видите ли, небезопасны.

Реализация криптографических систем уже сама по себе является искусством. Наиболее важным аспектом этого процесса служит качество кода. Низкое качество кода — одна из наиболее распространенных причин осуществления атак на реальные системы, а ведь избежать этого очень легко. По своему опыту мы знаем, что написание высококачественного кода занимает практически столько же времени, как и написание кода плохого качества (разумеется, если считать от начала работы до получения законченного продукта, а не первой версии программы с кучей ошибок). Будьте безжалостны по отношению к качеству своего кода. Оно может и должно быть достигнуто!

В идеале мы бы переделали все окружение, с которым приходится работать, включая язык программирования и операционную систему. Основной целью нашей грандиозной переделки стала бы безопасность, безопасность и еще раз безопасность. Мы бы с удовольствием поработали над этим проектом, поэтому свяжитесь с нами, если захотите потратить несколько миллионов долларов на компьютер, которому *действительно* можно доверять.

Часть II

Согласование ключей

Глава 10

Генерация случайных чисел

Чтобы сгенерировать ключ, нам нужен *генератор случайных чисел (random number generator — RNG)*. Генерация действительно хорошей случайности — неотъемлемая и к тому же наиболее сложная часть многих криптографических операций.

Мы не будем вдаваться в подробное обсуждение того, что же в действительности представляет собой случайность. Существует масса красивых математических определений данного термина, но все они слишком сложны для рассмотрения в этой книге. С неформальной же точки зрения случайность можно определить как непредсказуемость значений данных для злоумышленника, даже если тот предпримет активные шаги для борьбы со случайностью.

Многим криптографическим функциям требуются хорошие генераторы случайных чисел. В части 1, "Наша философия проектирования", рассмотрен безопасный канал общения и его компоненты. Мы предположили, что у нас существует ключ, известный пользователям А и Б. Этот ключ должен быть где-нибудь сгенерирован. Для выбора ключей системы управления ключами используют генераторы случайных чисел. Если генератор не слишком хорош, будет получен слабый ключ. Именно это произошло с одной из ранних версий обозревателя Netscape [37].

Мера случайности называется *энтропией (entropy)* [90]. Мы не будем приводить здесь математические детали этого вопроса, просто попытаемся сделать все, чтобы вы получили представление о том, что же такое энтропия. Если у нас есть 32-битовое слово, которое выбирается совершенно случайным образом, значит, оно имеет 32 бита энтропии. Если же 32-битовое слово может принимать только четыре значения, вероятность появления каждого из которых составляет 25%, тогда это слово обладает лишь двумя битами энтропии. Энтропия показывает не количество битов в значении, а то, насколько вы *не уверены* в этом значении. Для большей наглядности энтропию можно представить в качестве среднего числа битов, необходимых для того, чтобы задать

значение при использовании идеального алгоритма сжатия. Обратите внимание, что энтропия значения зависит от того, сколько вы знаете об этом значении. Случайное 32-битовое слово обладает 32 битами энтропии. Теперь предположим, что о значении этого слова точно известно следующее: 18 бит слова равны 0, а 14 бит — 1. Таким требованиям удовлетворяют около $2^{28,8}$ значений, а следовательно, энтропия слова будет составлять лишь 28,8 бит. Другими словами, чем больше известно о значении, тем меньше его энтропия.

Несколько сложнее подсчитать энтропию для значений, не имеющих равномерного вероятностного распределения. Наиболее распространенное определение энтропии переменной X формулируется следующим образом:

$$H(X) := -\sum_{x} P(X = x) \log_2 P(X = x),$$

где $P(X = x)$ — вероятность того, что переменная X принимает значение x. Мы не будем пользоваться этой формулой, поэтому запоминать ее нет необходимости. Именно на это определение ссылаются математики, когда говорят об энтропии. В математике существует еще несколько определений энтропии; выбор того или иного определения зависит от того, чем занимается конкретный ученый. И не путайте нашу энтропию с понятием энтропии в физике, где этот термин касается термодинамики.

10.1 Истинно случайные числа

В идеальном мире следовало бы использовать “истинно случайные” числа. К сожалению, наш мир не идеален и найти в нем действительно случайные данные весьма сложно.

Обычные компьютеры имеют несколько источников энтропии. В качестве примеров таких источников часто приводят точное время нажатия клавиши и точное время перемещения мыши. В свое время некоторыми учеными даже проводились исследования по поводу случайности колебаний времени доступа к жесткому диску, вызванных турбулентностью внутри его корпуса [19]. Все эти источники энтропии несколько сомнительны, так как в определенных ситуациях злоумышленник может выполнить измерения над источником случайности или повлиять на этот источник.

Многие разработчики весьма оптимистично относятся к количеству энтропии, которое может быть извлечено из разных источников. Мы видели программное обеспечение, которое генерировало 1-2 байта случайных данных на основе времени одного нажатия клавиши. К сожалению, мы не разделяем этого оптимизма. Время нажатия клавиш профессиональной машинисткой можно предсказать с точностью до нескольких миллисекунд. А частота сканирования клавиатуры ограничивает точность, с которой можно измерить

время нажатия клавиши. Печатаемые данные также нельзя назвать случайными, даже если пользователь просто нажимает первые попавшиеся клавиши. Что еще хуже, всегда существует угроза того, что у злоумышленника имеется дополнительная информация о "случайных" событиях. Звуки, издаваемые при нажатии клавиш, можно записать на микрофон, что позволит определить точное время нажатия каждой клавиши. Будьте очень осторожны, оценивая количество энтропии, которым обладают те или иные данные. В конце концов, мы имеем дело с очень умным и активным злоумышленником.

Существует множество физических процессов, которые ведут себя случайным образом. Например, согласно законам квантовой физики поведение некоторых частиц является полностью случайным. Было бы очень хорошо измерить это случайное поведение, чтобы затем использовать его в своих системах. Технически это, конечно же, возможно. К сожалению, у злоумышленника и здесь найдутся свои подходы. Во-первых, он может попытаться повлиять на поведение квантовых частиц, чтобы оно стало предсказуемым. Во-вторых, он может попросту украсть наши измерения. Если злоумышленнику удастся заполучить измерения, тогда данные все равно останутся случайными, но, с его точки зрения, не будут обладать какой-либо энтропией. (Если злоумышленник знает точное значение, то для него это значение не обладает энтропией.) Вероятно, злоумышленник способен создать сильное радиоизлучение, которое будет смещать показания наших приборов. Мы даже можем описать несколько атак, использующих законы квантовой физики. Например, для нарушения случайности, которую мы пытаемся измерить, может быть использован парадокс Эйнштейна–Подольского–Розена [5, 11]. Аналогичные соображения применимы и к другим источникам энтропии, таким, как тепловые помехи резистора или туннельный эффект в полупроводниковом стабилитроне.

Некоторые современные компьютеры обладают встроенными генераторами истинно случайных чисел [41]. Это, несомненно, дает им значительные преимущества перед использованием отдельных генераторов, поскольку существенно затрудняет проведение некоторых типов атак. Тем не менее такой генератор случайных чисел все еще будет доступен операционной системе, поэтому для получения безопасных данных приложение должно ей полностью доверять.

10.1.1 Проблемы использования истинно случайных чисел

Помимо сложности получения истинно случайных данных, существует еще несколько проблем их использования на практике. Во-первых, они не всегда доступны. Если случайные числа генерируются на основе времени на-

жатия клавиш, вы не получите никаких данных, когда пользователь вдруг прекратит печатать. Это может стать настоящей проблемой, если ваше приложение является Web-сервером, установленным на компьютере, у которого нет клавиатуры. Еще одна схожая проблема связана с тем, что объем истинно случайных данных всегда ограничен. Если вам понадобится большое количество случайных данных, придется подождать, пока они будут сгенерированы, что совершенно неприемлемо для многих приложений.

Вторая проблема состоит в том, что источники истинно случайных данных, такие, как физические генераторы случайных чисел, могут сломаться или отказать. Иногда поведение генератора может стать предсказуемым. Поскольку генераторы истинно случайных чисел довольно замысловаты, они могут выходить из строя гораздо чаще, чем классические компоненты компьютера. Если ваша система напрямую зависит от генератора истинно случайных чисел, вам крупно не повезет в случае отказа последнего.

И наконец, третья проблема — это сложность оценки энтропии, которая может быть извлечена из конкретного физического события. Если только в качестве генератора случайных чисел не используется выделенное, специально спроектированное для этого оборудование, определить количество получаемой энтропии будет крайне сложно. Мы еще вернемся к этому вопросу немного позднее.

10.1.2 Псевдослучайные числа

Альтернативой истинно случайным числам являются так называемые псевдослучайные числа. В действительности псевдослучайные числа вообще не являются случайными. Они вычисляются с помощью детерминированного алгоритма на основе некоторого *начального числа (seed)*. Зная начальное число, можно предсказать все последующие псевдослучайные числа. Классические *генераторы псевдослучайных чисел (pseudorandom number generator — PRNG)* никак не защищены от умного злоумышленника. Они разрабатывались для устранения статистических искажений, а вовсе не для того, чтобы отражать атаки. Второй том книги Дональда Кнута (Donald Knuth) *The Art of Computer Programming* [54] содержит подробное описание генераторов случайных чисел, но все они анализируются только на предмет статистической случайности. Мы должны исходить из предположения, что наш противник знает алгоритм, используемый для генерации случайных чисел. Может ли злоумышленник, зная несколько псевдослучайных чисел, сгенерированных алгоритмом, предугадать некоторые биты будущих (или прошлых) случайных значений? Для большинства классических генераторов псевдослучайных чисел ответ может быть положительным. Для криптографических генераторов псевдослучайных чисел это недопустимо.

В контексте криптографической системы к генераторам псевдослучайных чисел предъявляются гораздо более строгие требования. Даже если злоумышленник знает целый ряд случайных чисел, созданных генератором, у него не должно быть возможности предугадать какую-либо информацию об остальных случайных числах. Мы называем такой генератор псевдослучайных чисел криптографически сильным. Поскольку нам не нужны классические генераторы псевдослучайных чисел, в дальнейшем будем говорить только о криптографически сильных генераторах.

Забудьте об обычных случайных функциях, имеющихся в ваших программных библиотеках. Практически ни одна из них не подходит для применения в криптографических системах. Большинство программных библиотек поставляются с генераторами псевдослучайных чисел, которые проваливают даже простые статистические тесты. Никогда не используйте библиотечные генераторы, если в документации к ним явно не сказано, что они являются криптографически сильными.

10.1.3 Истинно случайные числа и генераторы псевдослучайных чисел

Истинно случайные числа будут использоваться нами только для того, чтобы получить начальное число, подающееся на вход генератора псевдослучайных чисел. После того как у нас появится начальное число, генератор сможет произвести на свет любое нужное количество случайных (а точнее, псевдослучайных) чисел. При необходимости мы можем прибавлять истинно случайные числа к начальному числу генератора псевдослучайных чисел. Это гарантирует, что выходные данные последнего никогда не станут полностью предсказуемыми, даже если злоумышленник каким-либо образом узнает начальное число.

Существует теоретическое мнение, что истинно случайные числа лучше, чем псевдослучайные. Для некоторых криптографических протоколов можно доказать, что при использовании истинно случайных чисел определенные типы атак становятся невозможными. Такой протокол называется *безусловно защищенным (unconditionally secure)*. Если же использовать генератор псевдослучайных чисел, протокол будет безопасным только при условии, что злоумышленник не сможет взломать этот генератор. Такой протокол называется *защищенным по вычислениям (computationally secure)*. Это различие, однако, имеет значение только для закостенелых теоретиков. Все криптографические протоколы почти всегда основаны на вычислительных приближениях. Устранение такого приближения для одного конкретного типа атак не принесет существенного улучшения. Более того, генерация истинно случайных чисел, необходимых для обеспечения безусловной защищенности, настолько

сложна, что попытка использовать такие числа может только ухудшить безопасность системы. Любое слабое место генератора истинно случайных чисел моментально приведет к потере безопасности. С другой стороны, если использовать истинно случайные числа только для того, чтобы получить начальное число для генератора псевдослучайных чисел, можно позволить себе гораздо более придирчиво выбирать источники энтропии, что, безусловно, повышает наши шансы на создание действительно безопасной системы.

10.2 Модели атак на генератор псевдослучайных чисел

Генераторы псевдослучайных чисел исследованы сравнительно мало. Сама задача генерации случайных (псевдослучайных) чисел на основе некоторого начального числа довольно проста. Проблема состоит в том, где взять случайное начальное число и как сохранить его в секрете [47]. Наилучшее решение этой проблемы, известное нам на данный момент, называется Yarrow [46]. Этот генератор псевдослучайных чисел, который пытается защитить систему от всех известных типов атак, был разработан нами несколько лет назад вместе с Джоном Келси (John Kelsey).

В каждый момент времени у генератора псевдослучайных чисел есть внутреннее состояние. Запросы на получение случайных чисел обслуживаются криптографическим алгоритмом, который генерирует псевдослучайные числа. Этот алгоритм также обновляет внутреннее состояние генератора, чтобы гарантировать, что в следующий раз генератор не возвратит те же самые случайные числа. Этот процесс не слишком сложен; с выполнением данной задачи вполне справится любая функция хэширования или блочный шифр.

Существуют различные формы атак на генераторы псевдослучайных чисел. Прежде всего следует отметить прямую атаку, где злоумышленник пытается воссоздать внутреннее состояние генератора на основе его выходных данных. Это классический тип криптоатаки, которому довольно легко противостоять, используя современные приемы криптографии.

Ситуация усложняется, если злоумышленник на каком-то этапе сможет заполучить внутреннее состояние генератора. Пока что нас не интересует, как это произойдет. Возможно, утечка информации была вызвана каким-то изъяном в программном обеспечении, либо компьютер загружался в первый раз и еще не имел случайного начального числа, либо, скажем, злоумышленнику удалось считать файл начального числа с диска. Во всех этих случаях с генератором начнут происходить довольно неприятные вещи. Если злоумышленнику удастся узнать внутреннее состояние классического генератора псевдослучайных чисел, он сможет воспроизвести значения всех последующих вы-

ходных данных и всех обновлений внутреннего состояния. Из этого следует, что, если атака на генератор псевдослучайных чисел когда-нибудь окончится успехом, этот генератор никогда не сможет вернуться в безопасное состояние.

Восстановить безопасность генератора, внутреннее состояние которого стало известно злоумышленнику, весьма сложно. Для этого понадобится некоторый источник энтропии, с помощью которого можно генерировать истинно случайные числа. Для простоты будем предполагать, что у нас есть один или несколько источников, предоставляющих определенное количество энтропии (обычно в виде небольших порций, которые мы будем называть событиями) в непредсказуемые моменты времени.

Даже если мы перемешаем внутреннее состояние генератора с небольшими количествами энтропии, полученными в результате возникновения события, у злоумышленника все еще останутся пути нападения. Он может посылать генератору частые запросы на получение случайных данных. Поскольку общее количество энтропии, добавляемое к внутреннему состоянию генератора в промежуток времени между двумя подобными запросами, будет ограничено, скажем, 30 битами, злоумышленник сможет просто перебрать все возможные варианты случайных входных данных и определить новое внутреннее состояние, полученное после перемешивания. Это потребует около 2^{30} шагов, что вполне достижимо с помощью современных технологий[1]. Обнаружив правильное решение, злоумышленник сможет легко проверить его с помощью все тех же случайных данных, выдаваемых генератором.

Лучшее, что можно сделать для борьбы с описанным типом атак, — это организовать пул входящих событий, которые несут в себе энтропию. Мы будем собирать энтропию до тех пор, пока ее не окажется достаточно для того, чтобы при перемешивании энтропии с внутренним состоянием генератора злоумышленник не мог угадать собранные случайные данные. Сколько же энтропии для этого потребуется? Согласно нашим требованиям к уровню безопасности, злоумышленник должен потратить на осуществление атаки минимум 2^{128} шагов, для чего понадобится 128 бит энтропии. Здесь, однако, мы сталкиваемся с серьезной проблемой: получить какую-либо оценку количества энтропии невероятно сложно, если вообще возможно. Оценка энтропии весьма существенно зависит от того, сколько знает или может знать злоумышленник, а эта информация еще не доступна разработчикам системы на стадии проектирования. В этом и состоит основная проблема генератора Yarrow. Он пытается измерить энтропию источника, используя оценку энтропии, а получить правильную оценку для всех ситуаций практически невозможно.

[1] Многие, наверное, будут возмущены небрежностью наших вычислений. В этом примере нам следовало использовать оценочную энтропию (guessing entropy), а не стандартную шенноновскую энтропию. Более подробно о мерах энтропии можно прочитать в [14].

10.3 Проект Fortuna

Во время работы над этой главой мы значительно усовершенствовали генератор Yarrow. Новое решение получило название Fortuna в честь Фортуны — древнеримской богини слепого случая и удачи[2]. Fortuna решает проблему необходимости определения оценок энтропии, просто отказавшись от них. Оставшаяся часть этой главы в основном посвящена рассмотрению деталей ГПСЧ Fortuna[3].

Fortuna состоит из трех частей. Генератор принимает на вход начальное число фиксированной длины и выдает произвольное количество псевдослучайных данных. Аккумулятор собирает и накапливает энтропию из различных источников, а также время от времени изменяет начальное число генератора. И наконец, система управления файлом начального числа гарантирует, что генератор сможет производить случайные данные даже непосредственно после перезагрузки компьютера.

10.4 Генератор

Генератор преобразует некоторое состояние фиксированной длины в выходные данные произвольной длины. В качестве генератора мы будем использовать AES-подобный блочный шифр. Выберите тот, что вам больше нравится, — AES (Rijndael), Serpent или Twofish (см. раздел 4.5.7). Внутреннее состояние генератора включает в себя 256-битовый ключ блочного шифра и 128-битовый счетчик.

По своей сути генератор — это всего лишь блочный шифр, работающий в режиме счетчика. Напомним, что режим счетчика или CTR генерирует случайный поток данных, которые мы будем использовать в качестве выходных. Для повышения безопасности добавим к режиму работы блочного шифра еще несколько усовершенствований.

Если пользователь или приложение запрашивает случайные данные, генератор запускает свой алгоритм и производит псевдослучайные данные. Теперь предположим, что злоумышленнику удается раскрыть состояние генератора после выполнения очередного запроса. Было бы хорошо, если бы это не дискредитировало предыдущие результаты, выданные генератором. Для этого после выполнения каждого запроса будем генерировать еще 256 бит

[2] Вначале мы собирались назвать его по имени древнегреческой богини счастья и благоденствия Тюхэ, но затем поняли, что многие не смогут правильно прочитать слово “Tyche”.

[3] Чтобы не путать генератор, являющийся одной из трех составных частей проекта Fortuna, и сам проект Fortuna (который, по сути, представляет собой реализацию криптографического генератора псевдослучайных чисел), будем обозначать последний аббревиатурой ГПСЧ (от “генератор псевдослучайных чисел”). — Прим. перев.

псевдослучайных данных и использовать их в качестве нового ключа шифрования. Старый ключ при этом уничтожается, чтобы исключить любую возможность утечки информации о предыдущих запросах.

Чтобы сгенерированные данные обладали статистической случайностью, не следует генерировать слишком много данных одновременно. Действительно, в истинно случайных данных значения блоков могут периодически повторяться, в то время как выходные данные режима счетчика никогда не содержат повторяющихся блоков. (Более подробно это описано в разделе 5.8.2.) Существует несколько способов решения этой проблемы. Например, можно использовать только половину каждого блока шифрованного текста, что практически устранит статистическое отклонение от истинно случайной последовательности блоков. В качестве альтернативы вместо блочного шифра можно было бы использовать так называемую *псевдослучайную функцию (pseudorandom function)*, но нам пока что не попадалось хорошо исследованных и эффективных предложений. Самое простое, что можно сделать в данной ситуации, — это ограничить количество байтов случайных данных, которые могут выдаваться в ответ на один запрос. Это значительно затруднит выявление статистического отклонения от истинно случайной последовательности.

Если бы нам понадобилось сгенерировать 2^{64} блока данных с помощью одного и того же ключа, мы бы могли ожидать появления одной коллизии среди значений блоков. Несколько повторяющихся запросов такого размера быстро бы показали, что выходные данные генератора не являются истинно случайными; им не хватает ожидаемого числа коллизий. Поэтому мы ограничиваем максимальный размер данных, которые могут быть выданы в ответ на один запрос, 2^{16} блоками (т.е. 2^{20} байт). Для идеального генератора случайных чисел вероятность обнаружения коллизии среди 2^{16} блоков данных составляет около 2^{-97}, поэтому полное отсутствие коллизий может быть обнаружено только после выполнения около 2^{97} запросов. Общее количество работы, которое придется проделать злоумышленнику, составит примерно 2^{113} шагов. Это, конечно же, меньше запланированных нами 2^{128} шагов, но все равно довольно неплохо.

Мы знаем, что изменяем своим принципам, соглашаясь на более низкий (правда, лишь немного более низкий) уровень безопасности. К сожалению, хорошей альтернативы этому решению, кажется, нет. У нас нет подходящих криптографических функций, на основе которых можно было бы построить генератор псевдослучайных чисел с полным 128-битовым уровнем безопасности. Мы бы могли применить функцию SHA-256, но она слишком медленная. Люди всегда будут спорить о том, нужно ли использовать хороший криптографический генератор псевдослучайных чисел, и одним из аргументов против использования последнего является скорость. Заметное снижение скорости работы генератора для выигрыша еще нескольких бит безопасности

совершенно не устроит основную массу пользователей. Слишком многие из них просто предпочтут другой, действительно плохой генератор, чем напрочь испортят уровень безопасности всей системы.

Если бы у нас был блочный шифр с 256-битовым размером блока, тогда вопрос коллизий не стоял бы вообще. Впрочем, на практике описанная атака не представляет собой какой-либо серьезной угрозы. Для ее осуществления недостаточно тех самых 2^{113} шагов, которые должен выполнить злоумышленник. Атакуемому компьютеру тоже придется провести 2^{113} операций шифрования. Как видите, успешность подобной атаки зависит не столько от скорости работы компьютера злоумышленника, сколько от скорости работы компьютера самого пользователя. Большинство пользователей не собираются увеличивать вычислительную мощность своего компьютера только для того, чтобы помочь злоумышленнику. Честно говоря, нам не нравятся подобные аргументы в пользу безопасности решения. Они слишком неоднозначны, и если генератор псевдослучайных чисел когда-нибудь будет использован в непривычном окружении, эти аргументы могут и не сработать. Все же, учитывая ситуацию, наше решение представляется наилучшим компромиссом, к которому мы могли бы прийти.

Когда по окончании выполнения каждого запроса мы пересчитываем ключ блочного шифра, то не сбрасываем счетчик. Это второстепенный момент, однако он позволяет избежать проблемы коротких циклов. Представим, что нам необходимо сбрасывать счетчик после выполнения каждого запроса. Если значение ключа когда-нибудь повторится во второй раз, а все запросы будут получать фиксированный объем данных, тогда следующее значение ключа также повторится во второй раз. В результате можно получить короткий цикл значений ключа. Вообще-то подобная ситуация маловероятна, но, не сбрасывая счетчик, этой проблемы можно избежать вообще. Поскольку длина счетчика составляет 128 бит, мы никогда не повторим значение счетчика (генерация 2^{128} блоков выходит за пределы вычислительных возможностей наших компьютеров), что автоматически не допустит образования каких-либо циклов. Вдобавок ко всему мы можем использовать значение счетчика 0 для указания того, что генератор еще не имеет ключа и, следовательно, не может выдавать данные.

Обратите внимание: ограничение максимального объема данных, который может быть выдан в ответ на один запрос, до 1 Мбайт, вовсе не является непреодолимым. Если вам нужно больше мегабайта случайных данных, просто повторите запрос. В действительности реализация системы может содержать интерфейс, который будет автоматически выполнять подобные повторяющиеся запросы.

Сам по себе генератор является очень полезным модулем. Думаем, в большинстве реализаций он будет доступен не только как компонент ГПСЧ For-

tuna, но и как часть его интерфейса. Взять хотя бы программу, которая выполняет моделирование методом Монте-Карло[4]. Мы хотим, чтобы это моделирование было случайным. Вместе с тем желательно, чтобы при необходимости мы могли воспроизвести нужные вычисления (например, для отладки или проверки). Для этого в начале программы можно единоразово вызвать встроенный генератор случайных чисел операционной системы. С его помощью мы получим случайное начальное число. Это число может быть зафиксировано как часть выходных данных программы, а затем использовано нашим генератором для вычисления всех оставшихся случайных данных, необходимых для моделирования. Зная исходное начальное число генератора, можно проконтролировать правильность всех вычислений. Для этого достаточно еще раз запустить программу с теми же входными данными и начальным числом. А для отладки один и тот же процесс моделирования может запускаться снова и снова, причем его поведение всегда будет одинаковым — лишь бы исходное начальное число оставалось неизменным.

Теперь рассмотрим функционирование генератора подробнее.

10.4.1 Инициализация

Она довольно проста. Мы устанавливаем значения ключа и счетчика равными нулю, чтобы показать, что генератору еще не было передано начальное число.

функция InitializeGenerator
выход: $\mathcal{G}$ Состояние генератора.

Установим значения ключа K и счетчика C равными нулю.
$(K, C) \leftarrow (0, 0)$
Сформируем состояние.
$\mathcal{G} \leftarrow (K, C)$
return $\mathcal{G}$

10.4.2 Изменение начального числа

Функция Reseed обновляет состояние генератора с помощью входной строки произвольной длины. На этом уровне нас не волнует, что будет содержать входная строка. Чтобы гарантировать тщательное перемешивание входной строки с существующим ключом, мы применим функцию хэширования.

функция Reseed
вход: $\mathcal{G}$ Состояние генератора; изменяется этой функцией.

[4]Моделирование методом Монте–Карло — это моделирование, управляемое случайным выбором некоторых величин.

s Новое или дополнительное начальное число.

Вычислим новый ключ с помощью функции хэширования.
$K \leftarrow \mathrm{SHA}_d - 256(K \parallel s)$

Увеличим на единицу значение счетчика, чтобы оно стало ненулевым, и пометим, что генератору было передано начальное число. В данном случае C рассматривается как 16-байтовое целое число, представленное в формате, в котором наименее значимый байт записывается первым.
$C \leftarrow C + 1$

Здесь значение счетчика C рассматривается как целое число. Позднее он будет выступать в качестве блока открытого текста. Для преобразования одного значения в другое будем использовать соглашение о формате записи целых чисел, при котором наименее значимый байт записывается первым. Блок открытого текста — это блок, состоящий из 16 байт: $p_0, \ldots, p_{15}$. Он соответствует целочисленному значению

$$\sum_{t=0}^{15} p_i 2^{8i}.$$

Используя это соглашение, мы можем рассматривать C и как 16-байтовую строку, и как целое число.

10.4.3 Генерация блоков

Следующая функция генерирует заданное количество блоков случайных данных. Это внутренняя функция, которая будет использоваться только самим генератором псевдослучайных чисел. Любой объект за пределами генератора не должен иметь доступа к этой функции.

функция GenerateBlocks
вход: $\mathcal{G}$ Состояние генератора; изменяется этой функцией.
k Количество блоков, которое необходимо сгенерировать.
выход: r Псевдослучайная строка длиной $16k$ байт.

assert $C \neq 0$

Начнем с пустой строки.
$r \leftarrow \epsilon$

Присоединим к ней нужное количество блоков.
for $i = 1, \ldots, k$ **do**
$r \leftarrow r \parallel E(K, C)$
$C \leftarrow C + 1$

od
return r

Как вы, наверное, уже догадались, $E(K, C)$ — это функция шифрования блочного шифра, на вход которой подаются ключ K и блок открытого текста C. Вначале функция GenerateBlocks проверяет, не равно ли значение C нулю (это бы означало, что на вход данного генератора еще никогда не подавалось начальное число). Перед началом цикла переменной r присваивается пустая строка, после чего к ней поочередно присоединяется каждый следующий подсчитанный блок. Строка, сформированная таким образом, и будет выходным значением функции.

10.4.4 Генерация случайных данных

Функция PseudoRandomData генерирует случайные данные по запросу пользователя генератора. Она позволяет получить псевдослучайную строку длиной до 2^{20} байт и гарантирует, что после выполнения запроса любая информация о сгенерированных данных будет уничтожена.

функция PseudoRandomData
вход: $\mathcal{G}$ Состояние генератора; изменяется этой функцией.
n Количество байт случайных данных, которое необходимо сгенерировать.
выход: r Псевдослучайная строка длиной n байт.

Ограничим длину выходной строки, чтобы уменьшить статистическое отклонение от истинно случайных данных. Убедимся также, что заданная длина строки не является отрицательной.
assert $0 \le n \le 2^{20}$

Сгенерируем выходные данные.
$r \leftarrow$ *первые* $- n -$ *байт*(GenerateBlocks($\mathcal{G}, \lceil n/16 \rceil$))

Изменим значение ключа, чтобы избежать возможной дискредитации этих данных в будущем.
$K \leftarrow$ GenerateBlocks($\mathcal{G}, 2$)
return r

Как видите, результат функции PseudoRandomData генерируется путем вызова функции GenerateBlocks. Единственное различие между ними состоит в "усечении" результата функции PseudoRandomData до нужного количества байт. (Оператор $\lceil \ldots \rceil$ — это оператор округления сверху.) После этого мы генерируем еще два блока данных, чтобы сформировать новый ключ. Когда старый ключ K будет уничтожен, никто не сможет повторно

вычислить значение r. Если функция PseudoRandomData не сохранит копию значения r и очистит область памяти, в которой хранилось это значение, то по завершении работы функции у генератора просто не останется путей для утечки какой-либо информации об r. Именно поэтому даже последующий взлом генератора, если таковой когда-нибудь произойдет, никак не повлияет на секретность предыдущих выходных данных. Он будет угрожать лишь безопасности будущих выходных данных — проблема, решением которой должен заниматься аккумулятор.

Как уже отмечалось, объем данных, которые может возвратить функция PseudoRandomData, ограничен. Мы не будем описывать оболочку, которая сможет возвращать случайные строки большего размера путем многократного вызова функции PseudoRandomData. Обратите внимание: увеличивать максимальный размер выходных данных, получаемых за один вызов функции, не следует, поскольку это увеличит статистическое отклонение от истинно случайной последовательности. Выполнение повторяющихся вызовов функции PseudoRandomData не приведет к снижению эффективности. Дополнительные расходы состоят лишь в том, что на каждый мегабайт сгенерированных случайных данных система генерирует еще 32 байта случайных данных (для формирования нового ключа) и выполняет пересчет подключей блочного шифра. Для всех предложенных нами блочных шифров такие расходы являются незначительными.

10.4.5 Скорость работы генератора

Генератор Fortuna, описанный в предыдущих разделах, является криптографически сильным генератором псевдослучайных чисел в том смысле, что он преобразует начальное число в псевдослучайную строку произвольной длины. Скорость работы генератора Fortuna практически равна скорости работы используемого блочного шифра. В системах с PC-совместимыми процессорами генерация одного байта псевдослучайных данных для больших запросов занимает менее 20 циклов работы процессора. Таким образом, наш генератор может легко применяться в качестве замены большинства библиотечных функций генерации псевдослучайных чисел.

10.5 Аккумулятор

Аккумулятор собирает истинно случайные данные из различных источников энтропии и применяет их для обновления начального числа генератора.

10.5.1 Источники энтропии

Будем исходить из предположения, что в нашем окружении находится несколько источников энтропии. Каждый источник может генерировать события, содержащие энтропию, в любой момент времени. Нас не интересует то, что конкретно будет использоваться в качестве источников энтропии. Достаточно, чтобы как минимум один источник генерировал данные, которые не будут предсказуемыми для злоумышленника. Поскольку мы не знаем, как именно будет действовать злоумышленник, имеет смысл превращать в источники энтропии все данные, которые будто бы не являются предсказуемыми. В частности, в качестве неплохих источников случайных данных можно порекомендовать время нажатия клавиш и время перемещения мыши. Постарайтесь найти как можно больше временны́х источников энтропии. Вы можете использовать (желательно одновременно) точное время нажатия клавиш, перемещений и щелчков кнопкой мыши, а также откликов жестких дисков и принтеров. Не страшно, если злоумышленник сможет предсказать или скопировать данные из некоторых источников; достаточно, чтобы он не мог этого сделать для всех источников сразу.

Реализация источников энтропии может потребовать от разработчика много времени и усилий. Как правило, источники энтропии должны быть встроены в драйверы аппаратного обеспечения операционной системы. Это практически невозможно осуществить на уровне пользователя.

Идентифицировать каждый источник мы будем с помощью его уникального номера, находящегося в диапазоне от 0 до 255. Разработчики могут сами решать, как выделять номера источников — статически или динамически. Данные каждого события представляют собой короткую последовательность байтов. Источники энтропии должны включать в себя только те данные событий, которые невозможно предсказать. Например, информация о времени может быть представлена двумя или четырьмя наименее значимыми байтами точного таймера. Включать в эти данные год, месяц или число не имеет смысла — злоумышленник их и так знает.

Мы будем выполнять конкатенацию событий, собранных из различных источников. Чтобы гарантировать, что строка, полученная в результате подобной конкатенации, будет кодировать события уникальным образом, ее нужно жестко структурировать. Каждое событие кодируется тремя или более байтами данных. Первый содержит номер источника случайных данных; второй — количество дополнительных байтов данных; следующие байты содержат данные, полученные от источника.

Разумеется, злоумышленник будет получать информацию о событиях, сгенерированных некоторыми источниками энтропии. Чтобы смоделировать подобную ситуацию, предположим, что некоторые источники находятся в

полной власти злоумышленника. Последний выбирает, когда и какие события будут генерироваться этими источниками. Кроме того, как и все остальные пользователи, злоумышленник может в любой момент запросить случайные данные у генератора псевдослучайных чисел.

10.5.2 Пулы

Чтобы обновить начальное число генератора, события нужно накапливать в пуле. Последний должен быть достаточно большим для того, чтобы злоумышленник не смог перебрать возможные значения содержащихся в пуле событий. Обновление начального числа с помощью "достаточно большого" пула случайных событий сделает бессмысленной всю ту информацию о состоянии генератора, которой мог обладать злоумышленник. К сожалению, мы не знаем, сколько событий нужно накопить в пуле, прежде чем обновлять начальное число генератора. В Yarrow мы пытались решить эту проблему путем использования оценок энтропии и различных эвристических правил. Fortuna же поступает гораздо более удачным способом.

У нас есть 32 пула: $P_0, \ldots, P_{31}$. Теоретически каждый пул содержит строку байтов неограниченной длины. На практике же все обстоит немного по-другому. Полученная строка будет использоваться лишь в качестве входных данных для функции хэширования. По этой причине реализациям ГПСЧ Fortuna не нужно сохранять строку неограниченной длины — они могут подсчитывать хэш-код строки по мере накопления данных в пуле.

Каждый источник энтропии распределяет свои случайные события между пулами по циклическому принципу. Это гарантирует, что энтропия, полученная от каждого источника, будет распределена между пулами более или менее равномерно. Каждое случайное событие, попавшее в тот или иной пул, присоединяется к строке, которая уже содержится в этом пуле.

Мы будем обновлять начальное число генератора каждый раз, когда объем содержимого пула P_0 станет достаточно большим. Процедуры обновления начального числа будут пронумерованы как $1, 2, 3, \ldots$. В зависимости от номера обновления r в процесс обновления включаются один или более пулов. Пул P_i участвует в обновлении, если 2^i является делителем r. Таким образом, пул P_0 будет участвовать в каждом обновлении, пул P_1 — в каждом втором обновлении, пул P_2 — в каждом четвертом обновлении и т.п. После того как пул принял участие в обновлении, его содержимое сбрасывается и заменяется пустой строкой.

Описанная система способна автоматически адаптироваться к конкретной ситуации. Если злоумышленник практически ничего не знает об источниках случайных данных, он не сможет предсказать содержимое пула P_0 при следующем обновлении. Но что, если злоумышленник имеет много информации об

источниках энтропии или же сам периодически генерирует ложные события? В этом случае он, вероятно, знает о содержимом пула P_0 достаточно, для того чтобы предсказать новое состояние генератора на основе его старого состояния и выходных данных. Тем не менее, когда в обновлении будет участвовать пул P_1, он будет содержать в два раза больше данных, не предсказуемых для злоумышленника, пул P_2 — в четыре раза больше непредсказуемых данных и т.п. Таким образом, если хотя бы один источник случайных событий не является предсказуемым для злоумышленника, то независимо от количества событий, которые генерирует сам злоумышленник или о которых у него есть информация, у нас всегда будет пул, содержащий достаточно энтропии для отражения атаки.

Скорость восстановления системы после раскрытия злоумышленником информации о внутреннем состоянии генератора зависит от интенсивности, с которой энтропия (являющаяся таковой и для злоумышленника) поступает в пулы. Если предположить, что скорость поступления энтропии является фиксированной и равна ρ, тогда через t секунд у нас будет ρt бит энтропии. За этот период времени каждый пул получит около $\rho t/32$ бит энтропии. Злоумышленник больше не сможет отслеживать внутреннее состояние генератора, если оно будет обновлено с помощью пула, содержащего более 128 бит энтропии. Здесь возможны два случая. Если перед следующей операцией обновления в пуле P_0 наберется более 128 бит энтропии, безопасность генератора будет восстановлена. В этом случае быстрота восстановления безопасности зависит от того, сколько данных удастся собрать в пуле P_0, прежде чем произойдет обновление. Второй случай — это когда сбрасывание пула P_0 происходит слишком быстро из-за возникновения случайных событий, известных злоумышленнику (или сгенерированных им самим). Пусть t — это время между двумя обновлениями состояния генератора. За это время пул P_i соберет $2^i\rho t/32$ бит энтропии. Отметим также, что этот пул будет участвовать в процедуре обновления каждые $2^i t$ секунд. Восстановление безопасности генератора произойдет при следующем обновлении его состояния с помощью первого пула P_i, для которого будет справедливо неравенство $128 \le 2^i\rho t/32 < 256$. (Наличие верхней границы неравенства объясняется следующим фактом: если бы число $2^i\rho t/32$ было больше или равно 256, это бы означало, что 128 бит энтропии накопилось еще в пуле P_{i-1}, а это противоречит тому, что первым таким пулом является P_i.) Из приведенного выше неравенства следует, что

$$\frac{2^i\rho t}{32} < 256,$$

а значит,

$$2^i t < \frac{8192}{\rho}.$$

Другими словами, интервал времени между моментами восстановления ($2^i t$) ограничен временем, необходимым для накопления 2^{13} бит энтропии ($8192/\rho$). На первый взгляд число 2^{13} выглядит довольно большим, однако его появление можно объяснить следующим образом. Чтобы восстановить безопасность генератора, требуется по крайней мере 2^7 бит энтропии. Если нам не повезет и обновление системы произойдет как раз перед тем, как мы набрали 2^7 бит в некотором пуле, придется воспользоваться следующим пулом, который ко времени обновления соберет около 2^8 бит энтропии. И наконец. мы разбиваем данные на 32 пула, что добавляет еще один множитель 2^5.

Это очень хороший результат. Наше решение отличается от идеального не более чем на множитель 64 (нам понадобится максимум в 64 раза больше случайности, чем требует идеальное решение). Это постоянный множитель, который гарантирует, что ситуация никогда не станет слишком плохой и безопасность генератора в конце концов будет восстановлена. Более того, нам не обязательно знать, сколько энтропии содержат наши события или сколько информации есть у злоумышленника. Именно этим Fortuna выгодно отличается от Yarrow. Отсутствие оценок энтропии, которые практически невозможно построить правильно, пошло новому решению только на пользу. Механизм обновления полностью автоматизирован; если случайные данные поступают с хорошей интенсивностью, генератор быстро восстановит свою безопасность. Если же случайные данные поступают медленно, процесс восстановления будет гораздо длительнее.

До сих пор мы никак не учитывали тот факт, что у нас есть только 32 пула. А что, если даже в пуле P_{31} между двумя обновлениями не наберется достаточно энтропии для того, чтобы восстановить безопасность генератора? Это может произойти в том случае, если злоумышленник инициирует так много случайных событий, что генератор испытает 2^{32} обновлений еще до того, как источники, не находящиеся под влиянием злоумышленника, успеют сгенерировать 2^{13} бит энтропии. Вообще говоря, это маловероятно, но, чтобы полностью предотвратить подобную ситуацию, можно ограничить частоту обновлений. Каждое новое обновление будет выполняться не ранее чем через 100 мс после предыдущего. Это ограничит частоту обновлений до 10 в секунду, а значит, пул P_{32}, если бы таковой существовал, был бы впервые использован не ранее чем через 13 лет после начала работы генератора! Учитывая, что экономический и технологический срок жизни большинства современных компьютеров значительно меньше 10 лет, вполне достаточно и 32 пулов.

10.5.3 Вопросы реализации

Ниже приводится несколько соображений по поводу реализации аккумулятора.

Распределение событий между пулами

Поступающие события должны каким-то образом распределяться между пулами. Заниматься распределением событий мог бы и сам аккумулятор, но это опасно по следующей причине. Нам понадобится реализовать функцию, которая будет передавать события аккумулятору. Вполне вероятно, что к этой же функции сможет обращаться и злоумышленник. Последний может выполнять дополнительные вызовы функции каждый раз при генерации "настоящего" события, тем самым влияя на выбор пула, в который должно было бы поступить следующее "настоящее" событие. Если злоумышленнику удастся собрать все "настоящие" события в пуле P_0, то система пулов станет неэффективной и злоумышленник сможет осуществлять атаки на один пул. Если же все настоящие события окажутся собранными в пуле P_{31}, то они вообще никогда не будут использованы.

Наше решение состоит в том, чтобы каждый источник энтропии передавал вместе с событием и номер пула, в который следует поместить это событие. В этом случае, чтобы повлиять на распределение событий, злоумышленнику потребовался бы доступ к памяти программы, которая генерирует событие. Если же злоумышленник действительно обладает доступом такого высокого уровня, тогда, вероятно, дискредитированным следует считать и сам источник энтропии.

Аккумулятор мог бы осуществлять проверку того, в правильном ли порядке каждый источник энтропии распределяет свои события между пулами. Хорошие функции часто проверяют, правильно ли сформированы их входные параметры, поэтому на первый взгляд такая идея кажется довольно удачной. Тем не менее в нашей ситуации не совсем понятно, как должен вести себя аккумулятор, если такая проверка выявит нарушения. Если весь ГПСЧ выполняется как пользовательский процесс, он мог бы сгенерировать неисправимую ошибку и завершить выполнение программы. Но зачем же лишать систему генератора псевдослучайных чисел только из-за неправильного поведения одного источника энтропии? Если же ГПСЧ является частью ядра операционной системы, ситуация становится еще сложнее. Представим себе, что какой-нибудь драйвер генерирует случайные события, но не в состоянии отслеживать даже простенький 5-битовый циклический счетчик событий. Что должен делать аккумулятор? Возвратить код ошибки? Но если программист делает такие простые ошибки, он, скорее всего, не будет проверять коды возврата. Остановить работу ядра? Это слишком радикальная мера, которая

приведет к полному отказу системы из-за какого-то одного неправильного драйвера. Лучшее, что мы смогли придумать на данный момент, — это "наказать" драйвер задержкой в предоставлении процессорного времени. Если проверка распределения событий выявит нарушения, аккумулятор может задержать выполнение соответствующего драйвера примерно на одну секунду.

Подобную идею тоже нельзя назвать полезной. Напомним: мы разрешили источнику событий самому определять номера пулов, так как предположили, что злоумышленник может осуществлять ложные обращения к аккумулятору, используя поддельные события. Если это случится и аккумулятор выполнит проверку порядка распределения событий, за плохое поведение злоумышленника будет наказан ни в чем не повинный генератор настоящих событий. Наше заключение таково: аккумулятору не стоит проверять порядок распределения событий, поскольку даже в случае обнаружения нарушений он не сможет сделать что-нибудь полезное. Каждый источник энтропии отвечает за распределение своих событий между пулами по циклическому принципу. Если же работа какого-нибудь источника будет нарушена, мы потеряем энтропию, предоставляемую этим источником (чего мы, собственно, и ожидали), но этим все и ограничится.

Время обработки события

Желательно ограничить количество вычислений, которые могут выполняться при передаче события аккумулятору. Большинство событий являются временны́ми и генерируются драйверами реального времени. Драйверам не нужен аккумулятор, который слишком долго обрабатывает события.

Существует определенный минимальный объем вычислений, необходимый для обработки каждого события. В частности, нам нужно присоединить данные события к содержимому выбранного пула. Конечно, мы не собираемся хранить в памяти всю строку пула, так как ее длина потенциально не ограничена. Вместо этого мы выделим каждому пулу небольшой буфер памяти и будем подсчитывать частичный хэш-код строки по мере заполнения буфера. Это и есть минимальный объем вычислений, которые необходимо выполнить для обработки события.

Не хотелось бы лишний раз выполнять обновление начального числа, в котором задействованы один или несколько пулов. Каждая из таких операций занимает на порядок больше времени, нежели простое добавление события к пулу. Поэтому мы отложим обновление начального числа до тех пор, пока пользователь не запросит случайные данные. Как только это произойдет, аккумулятор выполнит обновление начального числа непосредственно перед генерацией случайных данных. Применение подобной схемы позволяет хотя бы немного сместить вычислительную нагрузку с генераторов событий на

пользователей случайных данных. И это вполне справедливо — ведь случайные данные нужны именно пользователям, которые ради такой цели могут и подождать. В конце концов, большинство генераторов событий ничего не выигрывают от производства случайных данных, в котором участвуют.

Чтобы обновление начального числа выполнялось непосредственно перед обработкой запроса на получение случайных данных, необходимо инкапсулировать генератор. Другими словами, генератор будет скрыт, чтобы его нельзя было вызвать напрямую. Для этого в аккумулятор будет включена функция RandomData с таким же интерфейсом, как и у PseudoRandomData. Это защитит систему от пользователей, пытающихся вызвать генератор напрямую и тем самым пропустить процесс обновления, над реализацией которого мы так долго трудились. Разумеется, пользователи все еще могут создавать собственные экземпляры генератора для каких-то своих целей.

Типичная функция хэширования наподобие SHA_d-256 обрабатывает входящие сообщения, разбивая их на блоки фиксированного размера. Если мы будем обрабатывать строку пула по мере накопления в буфере каждого следующего блока, обработка события ограничится не более чем хэшированием одного блока. К сожалению, данный подход имеет свои недостатки. Для повышения эффективности работы процессоров в современных компьютерах применяется система кэшей. Одна из особенностей кэширования состоит в том, что скорость работы процессора повышается, когда он на протяжении некоторого времени работает с одними и теми же данными. Если обрабатывать блоки случайных данных по одному, процессору придется каждый раз извлекать из памяти предыдущий промежуточный хэш-код строки и перемещать его в наиболее быстрый кэш. Если же обрабатывать несколько блоков сразу, перемещение хэш-кода в наиболее быстрый кэш будет выполняться только для первого блока строки. Остальные блоки будут обрабатываться намного быстрее из-за того, что хэш-код уже находится в кэше процессора. В целом производительность современных процессоров значительно повышается, если процессор работает в рамках небольшого цикла, а не тогда, когда он вынужден постоянно переключаться между различными частями кода.

Учитывая сказанное выше, для повышения эффективности работы можно увеличить размер буфера пула. В этом случае перед применением операции хэширования в буфере будет накапливаться больше данных. Это позволит сократить общее количество необходимого процессорного времени, но вместе с тем увеличит максимальное время добавления нового события к пулу. К сожалению, мы не можем предложить вам какого-либо разумного компромисса для решения этой проблемы. Все зависит от специфики конкретного окружения.

10.5.4 Инициализация

Процедура инициализации, как всегда, очень проста. До сих пор речь шла только о генераторе и аккумуляторе, но функции, которые мы собираемся определять ниже, являются частью внешнего интерфейса всего ГПСЧ Fortuna. Слово `PRNG` в именах этих функций означает, что они применимы ко всему проекту в целом.

функция InitializePRNG
выход: $\mathcal{R}$ Состояние ГПСЧ

Поместим в каждый из 32 пулов пустую строку.
for $i = 0, \ldots, 31$ **do**
  $P_i \leftarrow \epsilon$
od
Установим значение счетчика обновлений равным 0.
ReseedCNT$\leftarrow 0$
Затем инициализируем генератор.
$\mathcal{G} \leftarrow$InitializeGenerator()
Упакуем состояние.
$\mathcal{R} \leftarrow (\mathcal{G}, \text{ReseedCNT}, P_0, \ldots, P_{31})$
return $\mathcal{R}$

10.5.5 Получение случайных данных

Как уже отмечалось, функция RandomData представляет собой оболочку для компонента-генератора. Структура этой функции будет не слишком простой, так как последняя должна обрабатывать обновление начального числа генератора.

функция RandomData
вход: $\mathcal{R}$ Состояние ГПСЧ; изменяется этой функцией.
  n Количество байт случайных данных, которое необходимо сгенерировать.
выход: r Псевдослучайная строка байтов.

if *length*$(P_0) \geq$ MinPoolSize $\wedge$ *последнее обновление* $>$ 100 мс назад **then**
  Нам нужно выполнить обновление.
  ReseedCNT $\leftarrow$ ReseedCNT $+ 1$
  Объединим в одну строку хэш-коды всех пулов, которые следует использовать в этой операции обновления.
  $s \leftarrow \epsilon$

```
    for i = 0, ..., 31 do
        if 2^i | ReseedCNT then
            s ← s || SHA_d − 256(P_i)
            P_i ← ϵ
        fi
    od
    Получив данные, мы можем выполнить обновление.
    Reseed(G, s)
fi
Обновление, если таковое требовалось, выполнено. Теперь в дело может вступать генератор, состояние которого является частью R.
return PseudoRandomData(G, n)
```

Вначале функция RandomData сравнивает размер пула P_0 с параметром MinPoolSize, чтобы узнать, нужно ли выполнять обновление. Мы можем использовать оптимистическую оценку того, каким должен быть размер пула, чтобы он мог содержать 128 бит энтропии. Предполагая, что каждое событие содержит 8 бит энтропии и занимает 4 байта в пуле (как вы помните, это соответствует 2 байтам данных события), разумным значением параметра MinPoolSize можно считать 64 байт. Вообще-то точное значение этого параметра не играет особой роли, хотя использовать значение, меньшее 32 байт, все же не рекомендуется. Не следует выбирать и слишком большое значение, потому что тогда обновление будет отложено на слишком долгий срок даже при наличии очень хороших источников энтропии.

Следующий шаг состоит в увеличении счетчика обновлений. В процессе инициализации счетчику ReseedCNT было присвоено значение 0, поэтому первое обновление будет пронумеровано как 1. Это автоматически гарантирует, что в первой операции обновления, как мы и планировали, будет участвовать только пул P_0.

Цикл `for ... do` выполняет конкатенацию хэш-кодов строк, содержащихся в пулах. Мы могли бы выполнять конкатенацию и самих строк пулов, однако в этом случае программе вместо простого хэширования содержимого каждого пула пришлось бы сохранять все строки полностью. Запись $2^i | \text{ReseedCNT}$ означает проверку делимости. Это выражение истинно, если 2^i является делителем значения ReseedCNT. Дотошные читатели, конечно же, обратят внимание на следующий факт: как только условие делимости не будет выполняться для какого-то i, оно не будет выполняться и для всех следующих i, поэтому все оставшиеся итерации цикла заведомо не нужны. Как видите, наша программа определенно требует оптимизации.

10.5.6 Добавление события

Источники энтропии вызывают функцию AddRandomEvent, когда у них появляется очередное случайное событие. Обратите внимание, что каждый источник энтропии идентифицируется уникальным номером. Мы не будем описывать, как система выделяет номера источников, так как это зависит от конкретной ситуации.

функция AddRandomEvent

вход:
- $\mathcal{R}$ Состояние ГПСЧ; изменяется этой функцией.
- s Номер источника в диапазоне $0, \ldots, 255$.
- i Номер пула в диапазоне $0, \ldots, 31$. Каждый источник должен распределять свои события между пулами по циклическому принципу.
- e Данные события. Строка байтов, длина которой находится в диапазоне $1, \ldots, 32$.

Вначале выполним проверку параметров.
assert $1 \leq length(e) \leq 32 \wedge 0 \leq s \leq 255 \wedge 0 \leq i \leq 31$

Добавим данные в пул.
$P_i \leftarrow P_i \parallel s \parallel length(e) \parallel e$

Напомним, что каждое событие кодируется с помощью $2 + length(e)$ байт, причем на каждую из величин s и $length(e)$ отводится по одному байту. После этого полученная конкатенация присоединяется к содержимому пула. Обратите внимание, что в наших примерах мы просто присоединяем данные к содержимому пула и не выполняем никакого хэширования. Хэширование содержимого пула происходит только тогда, когда пользователь запрашивает случайные данные. В реальной системе хэширование данных должно выполняться “на лету” по мере их поступления. Это функционально эквивалентно нашему решению, к тому же проще в реализации, однако непосредственно описывать этот процесс в книге было бы гораздо сложнее.

Мы ограничили длину данных события 32 байтами. События большей длины довольно бесполезны. Источники энтропии должны передавать аккумулятору не все свои данные, а лишь те несколько байтов, которые и содержат непредсказуемые случайные данные. Если же энтропия разбросана по большому объему данных, источник должен провести предварительное хэширование передаваемых данных. Функция AddRandomEvent должна выполняться как можно быстрее. Это особенно важно, поскольку многие источники энтропии в силу своей специфики работают в режиме реального времени. Такие источники не могут тратить слишком много драгоценного времени на вызов функции AddRandomEvent. Даже если источник выдает небольшие события, он не должен ждать, пока функция AddRandomEvent

разберется с другими источниками, события которых имеют больший размер. Большинству реализаций понадобится также сериализовать вызовы функции AddRandomEvent с помощью мьютекса (mutex object), чтобы гарантировать, что в один и тот же момент времени к аккумулятору будет добавлено только одно событие[5].

У некоторых источников энтропии может совсем не оказаться времени на вызов функции AddRandomEvent. В этом случае события можно сохранять в буфере и реализовать отдельный процесс, который будет извлекать события из буфера и передавать их аккумулятору.

В качестве альтернативной архитектуры можно предложить, чтобы источники энтропии просто передавали события процессу аккумулятора, а все вычисления хэш-кодов выполнялись отдельным потоком аккумулятора. Эта структура более сложна, но имеет значительные преимущества для источников энтропии. Выбор той или иной архитектуры во многом зависит от конкретной ситуации.

10.6 Управление файлом начального числа

Наш ГПСЧ умеет собирать энтропию и генерировать случайные данные после первого обновления случайного числа. Тем не менее, если мы перезагрузим компьютер, придется подождать, пока источники энтропии сгенерируют достаточно случайных событий для выполнения первого обновления. Кроме того, у нас нет гарантий, что состояние генератора после первого обновления не будет предсказуемым для злоумышленника.

Для решения этой проблемы можно использовать файл начального числа. Он представляет собой отдельный файл, содержащий достаточное количество энтропии и не доступный никому, кроме ГПСЧ. После перезагрузки компьютера ГПСЧ считывает файл начального числа и использует его содержимое в качестве энтропии для перевода генератора в состояние, не предсказуемое для злоумышленника. Конечно же, сразу после этого содержимое файла начального числа должно быть заменено новыми данными.

В следующих разделах главы описывается управление файлом начального числа. Вначале будем исходить из предположения, что операционная система поддерживает атомарные операции, а затем обсудим проблемы, связанные с управлением файлом начального числа в реальных системах.

[5] В многопоточном окружении необходимо всегда тщательно следить за тем, чтобы разные потоки не пересекались друг с другом.

10.6.1 Запись в файл начального числа

Вначале необходимо сгенерировать файл начального числа. Это делается с помощью простой функции.

функция WriteSeedFile
вход: $\mathcal{R}$ Состояние ГПСЧ; изменяется этой функцией.
f Файл, в который нужно записать энтропию.

$write(f, \text{RandomData}(R, 64))$

Эта функция просто генерирует 64 байт случайных данных и записывает их в файл. Вообще-то нам никогда не понадобится так много данных, но экономить здесь совершенно нет смысла.

10.6.2 Обновление файла начального числа

Очевидно, нам нужна функция и для считывания файла начального числа. По причинам, которые мы объясняли выше, после считывания файла начального числа будет сразу же выполняться его обновление.

функция UpdateSeedFile
вход: $\mathcal{R}$ Состояние ГПСЧ; изменяется этой функцией.
f Файл, который нужно обновить.

$s \leftarrow read(f)$
assert $length(s) = 64$
$\text{Reseed}(\mathcal{G}, s)$
$write(f, \text{RandomData}(\mathcal{R}, 64))$

Эта функция считывает файл начального числа, проверяет его длину и выполняет обновление состояния генератора. Затем содержимое файла начального числа заменяется новыми случайными данными.

Функция UpdateSeedFile должна следить за тем, чтобы никто не смог использовать ГПСЧ в промежутке между обновлением состояния генератора и записью в файл начального числа новых данных. Проблема состоит в том, что после перезагрузки компьютера файл начального числа считывается функцией UpdateSeedFile, которая применяет содержимое файла для обновления состояния генератора. Предположим, злоумышленник запрашивает у генератора случайные данные до того, как функция UpdateSeedFile успеет перезаписать файл начального числа. Получив от генератора затребованные случайные данные, злоумышленник тут же перезагружает компьютер, так и не дав функции UpdateSeedFile перезаписать файл. В результате при следующей перезагрузке компьютера функция считает из файла начального числа те же самые данные, что и в прошлый раз, и обновит с их

помощью состояние генератора. Теперь предположим, что случайные данные понадобятся какому-нибудь пользователю, который тоже запросит их еще до перезаписи файла начального числа. Ничего не подозревающий пользователь получит от генератора те же самые случайные данные, что и злоумышленник! Это нарушает секретность случайных данных. Поскольку случайные данные часто используются для генерации криптографических ключей, описанная проблема далеко не безобидна.

Разработчик должен гарантировать, что файл начального числа будет сохраняться в секрете. Кроме того, все операции обновления файла начального числа должны быть атомарными (см. раздел 10.6.5).

10.6.3 Когда нужно считывать и перезаписывать файл начального числа?

После перезагрузки компьютера у генератора еще нет энтропии, на основе которой он мог бы генерировать случайные данные. Вот для чего нам нужен файл начального числа. Таким образом, файл начального числа должен считываться и обновляться после каждой перезагрузки компьютера.

Когда компьютер функционирует, он собирает энтропию из различных источников. Мы хотим, чтобы эта энтропия каким-то образом влияла и на файл начального числа. Одно из очевидных решений — перезаписывать файл начального числа во время выключения компьютера. Поскольку некоторые компьютеры никогда не выключаются в корректном порядке, файл начального числа следует периодически перезаписывать и во время работы компьютера. Мы не будем обсуждать детали этого процесса, потому что они совсем неинтересны и часто зависят от конкретной платформы. Важно гарантировать, чтобы ГПСЧ регулярно обновлял файл начального числа после того, как он соберет достаточное количество энтропии. Вполне разумно обновлять файл начального числа при каждом выключении компьютера, а также, например, каждые 10 минут.

10.6.4 Архивирование

Пытаясь обновить состояние генератора, мы сталкиваемся с массой проблем. Мы не можем допустить, чтобы одно и то же состояние генератора повторилось дважды. Для этого мы используем файл начального числа, который хранится в файловой системе. К сожалению, разработчики большинства файловых систем никак не учитывали того, что мы не должны допустить повторения одного и того же состояния. Отсюда автоматически следуют многочисленные неприятности.

Как известно, на свете существуют архивы. Если мы заархивируем всю файловую систему и затем перезагрузим компьютер, состояние генератора будет обновлено с помощью файла начального числа. Если же впоследствии мы восстановим всю файловую систему из архива и выполним перезагрузку компьютера, состояние генератора будет обновлено на основе того же самого файла начального числа. Другими словами, пока аккумулятор не соберет достаточное количество энтропии, выходные данные генератора после обеих перезагрузок будут одинаковыми. Это очень серьезная проблема. Используя эти действия, злоумышленник сможет воссоздать случайные данные, которые до этого получил другой пользователь.

Прямой защиты от атак подобного рода не существует. Если система архивирования может полностью восстановить состояние компьютера, мы ничем не сможем помешать восстановлению состояния генератора. В идеале нам следовало бы приспособить систему архивирования к особенностям работы генератора, но, это, пожалуй, было бы уж слишком. Помочь в данной ситуации могло бы хэширование файла начального числа вместе с текущим временем, если только злоумышленник не сбрасывает часы. Это же решение можно применять и тогда, когда система архивирования ведет счетчик того, сколько операций восстановления она провела. В этом случае мы бы могли хэшировать файл начального числа вместе со счетчиком восстановлений. Этот вопрос определенно заслуживает дальнейшего изучения. К сожалению, он существенно зависит от специфики конкретной платформы, поэтому дать какой-либо общий совет мы не можем.

10.6.5 Атомарность операций обновления файловой системы

Еще одной важной проблемой, связанной с файлом начального числа, является атомарность операций обновления файловой системы. В большинстве операционных систем запись в файл начального числа в действительности приводит лишь к обновлению нескольких буферов памяти. На жесткий диск эти данные будут перенесены намного позднее. Даже если мы принудительно заставим операционную систему перенести данные на жесткий диск (если такое вообще возможно), на свете существует так много жестких дисков, которые просто накапливают данные в своих буферах без всякой гарантии того, когда же эти данные действительно будут записаны на магнитный носитель.

Обновив состояние генератора с помощью файла начального числа, мы должны перезаписать этот файл прежде, чем какой-либо пользователь сможет запросить случайные данные. Другими словами, мы должны быть абсолютно уверены в том, что данные на магнитном носителе действительно были перезаписаны. Ситуация еще более усложняется, если учесть, что многие файловые системы обрабатывают административную информацию о файле

отдельно от данных этого файла. В этом случае перезапись файла начального числа может привести к временной несогласованности административной информации о файле. Если в этот момент произойдет внезапное отключение электропитания, файл начального числа может быть поврежден или вообще утерян. Не слишком хорошее начало для реализации системы безопасности!

Некоторые файловые системы для решения подобных проблем применяют журналы. Изначально эта методика была разработана для больших систем управления базами данных. Журнал — это последовательный список всех обновлений, которые выполнялись в файловой системе. При корректном использовании журнал гарантирует, что все обновления файловой системы будут согласованными. С точки зрения надежности всегда предпочтительнее использовать именно такую файловую систему. К сожалению, ни одна из известных нам популярных файловых систем не дает полных гарантий относительно того, когда обновление данных действительно будет зафиксировано в постоянном хранилище.

Поскольку оборудование и операционная система не поддерживают полностью атомарных операций обновления файлов на постоянных носителях, мы не можем предложить идеальное решение, использующее файл начального числа. Вам понадобится тщательно исследовать конкретную платформу, с которой вы работаете, и приложить все усилия, чтобы обеспечить надежность обновления файла начального числа.

10.6.6 Первая загрузка

Когда генератор псевдослучайных чисел запускается в первый раз, ему неоткуда взять данные для обновления своего состояния — ведь файла начального числа еще не существует! Возьмем, например, только что купленный компьютер с предустановленной операционной системой. Современные операционные системы после установки генерируют несколько административных криптографических ключей. Для этого им нужен генератор псевдослучайных чисел. Для облегчения производственного процесса все компьютеры выпускаются одинаковыми и загружаются одинаковыми данными. На новых компьютерах еще нет файла начального числа, поэтому воспользоваться им мы не можем. Мы бы могли подождать, пока источники энтропии не сгенерируют достаточно случайных событий для проведения одного или нескольких обновлений, но это заняло бы слишком много времени. Кроме того, мы просто не сможем определить, когда в системе наберется достаточно энтропии для генерации хороших криптографических ключей.

Было бы хорошо, если бы программа установки генерировала файл начального числа прямо в процессе конфигурации операционной системы. Например, она бы могла использовать генератор, установленный на отдельном

компьютере, чтобы сгенерировать новый файл начального числа для каждой машины, на которой проводится установка. Вместо этого программа установки могла бы попросить пользователя подвигать мышью, чтобы собрать немного начальной энтропии. Выбор решения зависит от специфики конкретного окружения, но без предоставления начальной энтропии в любом случае не обойтись. Весьма вероятно, что некоторые очень важные криптографические ключи будут сгенерированы сразу же после установки операционной системы, когда аккумулятор энтропии еще не успеет набрать достаточного количества случайных событий, чтобы корректно обновить состояние генератора.

Не забывайте, что аккумулятор Fortuna произведет обновление состояния генератора тогда, когда у него *может* оказаться достаточно энтропии для обеспечения хорошей случайности. В зависимости от того, сколько энтропии будет поступать от источников на самом деле (то, о чем Fortuna не знает), сбор достаточного количества энтропии для действительно правильного обновления может занять некоторое время. Пожалуй, наилучшим решением в данной ситуации было бы наличие внешнего источника энтропии, который мог бы создать первый файл начального числа.

10.7 Так что же делать?

Прежде всего, никогда не доверяйте генераторам случайных чисел, которые поставляются вместе с программными библиотеками или операционной системой. Большинство из них не соответствуют никаким требованиям, выдвигаемым к криптографически сильному генератору псевдослучайных чисел. Генератор Fortuna, описанный в этой главе, определенно представляет собой значительное улучшение текущего положения дел, поэтому мы бы, не колеблясь, отдали предпочтение именно ему. Но, как обычно, наше мнение нельзя назвать объективным по очевидным причинам.

Собрать нужное количество энтропии всегда трудно. Используйте как можно больше источников энтропии. Огромным преимуществом генератора Fortuna является тот факт, что источники с низким качеством случайных данных не смогут навредить системе, поэтому отказываться от них вряд ли стоит. Несмотря на это, подключение к аккумулятору источников энтропии может потребовать изменения драйверов и других служебных программ операционной системы, что далеко не просто.

Концепция файла начального числа очень проста в понимании, а вот правильно реализовать ее на практике невероятно сложно. В этой главе мы привели несколько практических советов по реализации управления файлом начального числа, но большинство моментов реализации зависит от конкрет-

ных обстоятельств. Возможно, вам придется внести изменения в компоненты операционной системы или даже в само оборудование. Это еще один пример влияния безопасности на другие части системы. Сделайте все, что только в ваших силах. Удачи!

10.8 Выбор случайных элементов

Наш генератор выдает последовательности случайных байтов. Иногда это именно то, что нужно. Существуют, однако, ситуации, когда нам требуется выбрать случайный элемент из множества элементов. Правильное выполнение этой задачи требует определенного внимания.

Когда мы выбираем случайный элемент, то неявно предполагаем, что этот элемент выбирается равновероятным случайным образом из заданного множества элементов (если только не указано другого вероятностного распределения). Это означает, что возможность выбора каждого элемента должна обладать в точности одной и той же вероятностью[6]. Реализовать данное условие в программном обеспечении намного сложнее, чем кажется на первый взгляд.

Пусть n — это число элементов множества, из которого требуется выбрать случайный элемент. В дальнейшем будем говорить только о том, как выбрать случайный элемент из диапазона $0, 1, \ldots, n-1$. Когда вы научитесь решать эту задачу, то сможете выбирать элементы из любого множества размером n.

Если $n = 0$, тогда у нас вообще нет элементов и мы получаем простую ошибку. Если $n = 1$, у нас нет вариантов выбора и мы вновь получаем простой случай. Если $n = 2^k$, мы можем просто получить от генератора k бит случайных данных и интерпретировать их как простое число в диапазоне $0, \ldots, n-1$. Полученное число будет случайным и выбранным равновероятным образом. (Возможно, вам придется получить от генератора целое количество байт и отбросить несколько бит последнего байта, чтобы оставшееся число бит равнялось k, но реализовать такую схему очень просто.)

Что делать, если n не является степенью двух? Некоторые программы выбирают случайное 32-битовое число и подсчитывают его значение по модулю n. К сожалению, использование данного алгоритма приводит к смещению результирующего распределения вероятностей. В качестве примера рассмотрим $n = 5$ и определим $m := \lfloor 2^{32}/5 \rfloor$. Если мы равновероятным образом выберем случайное 32-битовое число и подсчитаем его значение по модулю 5, то каждое из чисел 1, 2, 3 и 4 будет встречаться с вероятностью $m/2^{32}$, а число 0 —

[6] Если мы разрабатываем систему со 128-битовым уровнем безопасности, то можем позволить себе отклонение в 2^{-128} от равномерного распределения, но реализовать точное равномерное распределение все-таки проще.

с вероятностью $(m+1)/2^{32}$. В данном случае отклонение от равномерного распределения вероятностей невелико, однако оно может быть и значительным. Умному злоумышленнику не составит труда распознать отклонение за те 2^{128} шагов, которые мы отводим ему на осуществление атаки.

Чтобы правильно выбрать случайное число из произвольного диапазона, следует воспользоваться методом проб и ошибок. Например, чтобы сгенерировать случайное число в диапазоне $0,\ldots,4$, мы вначале генерируем случайное число в диапазоне $0,\ldots,7$. Последняя операция возможна, поскольку 8 является степенью двух. Если полученное число окажется больше или равно 5, мы отбросим его и выберем новое число в диапазоне $0,\ldots,7$. Так будет продолжаться до тех пор, пока полученное случайное число не попадет в желаемый диапазон $0,\ldots,4$. Другими словами, мы генерируем случайное число, состоящее из нужного количества бит и отбрасываем все неподходящие числа.

Ниже приведено более формальное описание того, как следует выбирать случайное число из диапазона $0,\ldots,n-1$ для $n \geq 2$.

1. Пусть k — наименьшее целое число, для которого $2^k \geq n$.
2. Воспользуйтесь генератором псевдослучайных чисел, чтобы получить k-битовое случайное число K. Это число будет находиться в диапазоне $0,\ldots,2^k-1$. Возможно, вам придется сгенерировать целое число байт и отбросить часть последнего байта, но это несложно реализовать.
3. Если $K \geq n$, вернитесь к шагу 2.
4. Число K является требующимся результатом.

Описанный алгоритм может оказаться не совсем эффективным. В худшем случае нам придется отбросить примерно половину попыток, поэтому попробуем немного улучшить предложенное решение. Вернемся к примеру с $n=5$. Поскольку $2^{32}-1$ делится на 5, мы можем выбрать случайное число из диапазона $0,\ldots,2^{32}-2$ и подсчитать значение полученного числа по модулю 5. Чтобы выбрать число из диапазона $0,\ldots,2^{32}-2$, мы снова воспользуемся "неэффективным" методом проб и ошибок, однако на сей раз вероятность того, что полученное случайное число придется отбросить, очень мала.

Общее правило состоит в том, чтобы выбрать удобное k, для которого $2^k \geq n$. Определим $q := \lfloor 2^k/n \rfloor$. Вначале выберем случайное число r в диапазоне $0,\ldots,nq-1$, используя метод проб и ошибок. Когда подходящее r будет найдено, окончательный результат подсчитывается как $(r \bmod n)$.

Мы не знаем другого способа сгенерировать равномерно распределенные случайные числа, размер которых в битах не является степенью двух, кроме отбрасывания от полученного результата нескольких случайных битов. Это не проблема. При наличии современного генератора псевдослучайных чисел проблем с нехваткой случайных битов быть не должно.

Глава 11

Простые числа

В следующих двух главах книги речь идет о криптографических системах с открытым ключом. К сожалению, изучение этого материала требует определенного знания математики. Не секрет, что порой хочется пропустить подробные объяснения и ограничиться лишь уравнениями и формулами, но мы хорошо осознаем, что этого делать не следует. Чтобы использовать какой-нибудь инструмент, вы должны понимать его свойства. Это очень легко, когда речь идет о чем-нибудь наподобие функции хэширования. У нас есть "идеальная" модель функции хэширования, и мы требуем, чтобы реальная функция хэширования вела себя точно так же, как эта модель. Для систем с открытым ключом все гораздо сложнее. У нас нет идеальной модели криптографической системы с открытым ключом. На практике вам придется иметь дело с математическими свойствами подобных систем и, чтобы чувствовать себя уверенно, необходимо понимать эти свойства. Короткого пути здесь нет; вы должны понимать математику. Это не так сложно, как кажется; единственное, что от вас требуется, — это знание школьной программы математики на уровне старших классов (а точнее, математики, которую изучали авторы этой книги, когда были в старших классах).

Эта глава посвящена простым числам. Простые числа играют в математике очень важную роль, но нас они интересуют только потому, что большинство существующих криптосистем с открытым ключом основаны на применении простых чисел.

11.1 Делимость и простые числа

Число a является делителем числа b (это записывается как $a|b$ и читается "a делит b"), если b делится на a без остатка. Например, число 7 является делителем числа 35, поэтому можно записать, что $7|35$. Число называется

простым (prime), если у него есть только два делителя: единица и само это число. Например, число 13 является простым, так как у него есть только два делителя: 1 и 13. Перечислить первые несколько простых чисел несложно — это 2, 3, 5, 7, 11, 13 и т.д. Любое число, большее единицы, которое не является простым, называется *составным (composite)*. Число 1 не является ни простым, ни составным.

В следующих главах книги используется терминология и обозначения, принятые в математике. Это намного облегчит чтение другой литературы по данному вопросу. Вначале математические обозначения могут показаться вам сложными и запутанными, но не стоит беспокоиться — этот раздел математики действительно очень прост.

Ниже приведена простая лемма о делимости.

Лемма 1. *Если $a|b$ и $b|c$, тогда $a|c$.*

Доказательство. Если $a|b$, тогда существует целое число s, такое, что $as = b$. (Действительно, если b делится на a, тогда оно должно быть кратным a.) Если $b|c$, тогда существует целое число t, такое, что $bt = c$. Но из этого следует, что $c = bt = (as)t = a(st)$, а значит, a является делителем c. (Чтобы понять ход последнего рассуждения, просто убедитесь в том, что каждый из знаков равенства использован корректно. Логическое умозаключение состоит в том, что первый элемент c должен быть равен последнему элементу $a(st)$.) □

Лемма является констатацией факта. Доказательство объясняет, почему эта лемма верна. Маленький квадратик справа обозначает конец доказательства. Математики вообще любят использовать всевозможные символы[1]. Это очень простая лемма, и доказательство должно быть понятно всем, кто помнит, что означает запись $a|b$.

Простые числа исследовались математиками еще тысячи лет назад. Даже сегодня, если требуется определить все простые числа до миллиона, мы используем алгоритм, разработанный более 2000 лет назад Эратосфеном, другом Архимеда. (Эратосфен также был первым человеком, измерившим точный диаметр Земли. Спустя 1700 лет Колумб будто бы использовал намного меньшую — и неверную — оценку диаметра земного шара, когда собирался достичь Индии западным путем.) Евклид, еще один великий древнегреческий математик, привел гениальное доказательство того, что количество простых чисел бесконечно. Это доказательство настолько замечательно, что мы решили включить его в нашу книгу. Оно поможет быстро вернуть ваши мысли в лоно математики.

[1] Использование символов имеет свои преимущества и недостатки. Мы будем применять их только тогда, когда посчитаем, что это нужно для данной книги.

Прежде чем переходить к доказательству теоремы Евклида, приведем еще одну простую лемму.

Лемма 2. *Пусть n — положительное число, большее единицы. Пусть d — наименьший делитель числа n, больший единицы. Тогда d — простое число.*

Доказательство. Прежде всего следует убедиться в том, что определение числа d корректно. (Если существует такое положительное число n, у которого нет наименьшего делителя, то определение d дано некорректно и лемма теряет смысл.) Мы знаем, что n является делителем n и $n > 1$. Таким образом, у числа n должен существовать и наименьший делитель, больший 1.

Чтобы доказать, что d является простым числом, используем стандартный математический прием — *reductio ad absurdum*, или *доказательство от противного (proof by contradiction)*. Чтобы доказать утверждение X, предположим, что оно неверно, и покажем, что это предположение в конце концов приводит к противоречию. Если предположение о том, что X неверно, приводит к противоречию, это, очевидно, означает, что X верно.

В нашем случае предположим, что d не является простым числом. Тогда у него должен быть делитель e, такой, что $1 < e < d$. Но мы знаем из леммы 1, что если $e|d$ и $d|n$, то $e|n$, поэтому e является делителем n и $e < d$. Но это противоречит тому, что d является наименьшим делителем n. Поскольку мы пришли к противоречию, наше предположение должно быть неверным, а значит, d является простым числом. □

Пусть вас не беспокоит несколько запутанный метод доказательства — к нему нужно привыкнуть.

Теперь можно доказать, что количество простых чисел бесконечно.

Теорема 1 (Евклида). *Существует бесконечное количество простых чисел.*

Доказательство. Еще раз воспользуемся доказательством от противного. Предположим, что количество простых чисел конечно, а значит, их можно записать в виде конечного списка. Обозначим эти числа как $p_1, p_2, p_3, \ldots, p_k$, где k — количество простых чисел. Введем число $n := p_1 p_2 p_3 \ldots p_k + 1$ (произведение всех простых чисел плюс 1).

Возьмем наименьший делитель числа n, больший единицы, и снова обозначим его как d. Мы знаем, что d — это простое число (согласно лемме 2) и $d|n$. Но ни одно простое число из нашего конечного списка простых чисел не является делителем n. Действительно, все числа p_i являются делителями числа $(n-1)$, а n при делении на любое из этих чисел дает остаток 1. Мы получили, что d является простым числом, но его нет в списке всех простых чисел. Это противоречит тому, что список $p_1, p_2, p_3, \ldots, p_k$ содержит все существующие простые числа, а значит, наше предположение о конечности этого

списка неверно. Таким образом, существует бесконечное количество простых чисел. □

Примерно такое доказательство и было дано Евклидом более 2000 лет назад.

На протяжении многих веков было получено множество результатов, касающихся распределения простых чисел. Что интересно, ученые так и не вывели универсальной формулы того, как найти все простые числа из конкретного интервала, поскольку они располагаются в случайном порядке. Многие простые гипотезы так и не были доказаны. Например, гипотеза Гольдбаха утверждает, что каждое четное число, большее 2, является суммой двух простых чисел. Это утверждение легко проверить с помощью компьютера для относительно малых четных чисел, но математики все еще не знают, является ли это утверждение верным для всех четных чисел.

Для общего развития нелишне ознакомиться и с *фундаментальной теоремой арифметики*: любое целое число, большее единицы, может быть представлено в виде произведения простых чисел, и такое представление является однозначным (если не учитывать порядок, в котором записываются эти простые числа). Например: $15 = 3 \cdot 5$, $255 = 3 \cdot 5 \cdot 17$, а $60 = 2 \cdot 2 \cdot 3 \cdot 5$. Мы не будем приводить доказательство этой теоремы в данной книге. Интересующиеся могут найти его в любом учебнике по теории чисел.

11.2 Генерация малых простых чисел

Иногда под рукой полезно иметь список малых простых чисел. Для получения последних воспользуемся так называемым “решетом Эратосфена”; этот алгоритм генерации малых простых чисел все еще считается лучшим.

функция SmallPrimeList

вход: n Максимально возможное значение простого числа. Должно удовлетворять условию $2 \le n \le 2^{20}$.

выход: P Список всех простых чисел $\le n$.

Ограничим значение n. Если оно слишком большое, нам не хватит памяти.

assert $2 \le n \le 2^{20}$

Инициализируем список флажков. Каждому из них будет присвоено значение 1.

$(b_2, b_3, \ldots, b_n) \leftarrow (1, 1, \ldots, 1)$

$i \leftarrow 2$

while $i^2 \le n$ **do**

```
    Мы нашли простое число i. Пометим все следующие числа, кратные i, как составные.
    for j ∈ 2i, 3i, 4i, ..., ⌊n/i⌋ i do
        b_j ← 0
    od
    Поищем следующее простое число в нашем списке. Можно показать, что этот цикл никогда не приведет к условию i > n, по достижении которого функция выполняла бы доступ к несуществующему b_i.
    repeat
        i ← i + 1
    until b_i = 1
od
Теперь все простые числа помечены флажками 1. Соберем их в список.
P ← [ ]
for k ∈ 2, 3, 4, ..., n do
    if b_k = 1 then
        P ← P ‖ k
    fi
od
return P
```

В основе этого алгоритма лежит очень простая идея. Каждое составное число c делится на простое число, меньшее c. Каждому из чисел от 2 до n ставится в соответствие флажок, который указывает, может ли данное число быть простым. Сначала все числа помечаются как потенциально простые. Для этого их флажки устанавливаются равными 1. Переменной i присваивается значение первого простого числа, т.е. 2. Разумеется, ни одно из чисел, кратных 2, не может быть простым, поэтому мы помечаем числа $2i, 3i, 4i$ и т.д. как составные, изменяя их флажки на 0. Затем увеличиваем i, пока не доберемся до следующего кандидата на звание простого числа. Отметим, что этот "кандидат" не делится ни на одно из простых чисел, меньших его самого, — в противном случае он был бы уже помечен как составное число. Следовательно, новое значение i является простым числом. Аналогичным образом будем помечать составные числа и искать новое простое число, пока i^2 не станет бóльшим, чем n.

Очевидно, в результате применения данного алгоритма ни одно простое число не будет ошибочно помечено как составное, поскольку это делается лишь тогда, когда найден его делитель. (Цикл, который помечает числа как составные, последовательно перебирает значения $2i, 3i, \ldots$. Каждое из этих чисел имеет множитель i, а потому не может считаться простым.)

Почему можно остановиться, когда $i^2 > n$? Предположим, число k является составным. Пусть p — его наименьший делитель, больший единицы. Мы уже знаем, что p — это простое число (согласно лемме 2). Пусть $q := k/p$. Можно утверждать, что $p \leq q$; в противном случае q было бы делителем k, меньшим p, что противоречит определению p. Теперь покажем, что $p \leq \sqrt{k}$. Если бы p было больше чем $\sqrt{k}$, мы бы получили, что $k = p \cdot q > \sqrt{k} \cdot q \geq \sqrt{k} \cdot p > \sqrt{k} \cdot \sqrt{k} = k$. Из этого неравенства следует, что $k > k$, а это явное противоречие. Таким образом, $p \leq \sqrt{k}$.

Мы показали, что любое составное число k делится на простое число, которое меньше или равно $\sqrt{k}$. Из этого следует, что любое составное число $\leq n$ обязательно делится на простое число $\leq \sqrt{n}$. Если $i^2 > n$, то $i > \sqrt{n}$. Но мы уже пометили все числа, кратные каким-либо простым числам, меньшим i, как составные, поэтому больше составных чисел в списке быть не должно. Оставшиеся числа списка, помеченные как простые, действительно являются таковыми.

Последняя часть алгоритма организует все найденные простые числа в список и возвращает их как результат выполнения функции.

Описанный алгоритм можно было бы несколько оптимизировать, но мы решили отказаться от оптимизации в пользу простоты понимания. При правильной реализации этот алгоритм работает очень быстро.

Возможно, вас интересует, зачем нам нужны малые простые числа. Оказывается, что их можно использовать для генерации больших простых чисел, которые нам вскоре понадобятся.

11.3 Арифметика по модулю простого числа

Простые числа играют такую важную роль в криптографии прежде всего потому, что могут применяться для выполнения арифметических операций по модулю.

Пусть p — простое число. Когда мы вычисляем результат арифметической операции по модулю p, то используем только числа $0, 1, \ldots, p-1$. Основное правило выполнения операций по модулю простого числа состоит в следующем: мы используем целые числа так, как обычно, но каждый раз, получая результат r, берем его значение по модулю p. Операция взятия по модулю очень проста: мы делим r на p и отбрасываем целую часть полученного результата. Остаток от деления r на p и будет искомым ответом. Например, чтобы вычислить значение 25 по модулю 7, мы делим 25 на 7, в результате чего получаем частное 3 и остаток 4, который и будет ответом, а значит, $(25 \bmod 7) = 4$. Запись $(a \bmod b)$ применяется для обозначения непосредственной операции взятия числа a по модулю b, но, поскольку вычисления

по модулю используются слишком часто, а математики довольно ленивы, существует еще несколько общепринятых способов обозначения. Зачастую все выражение записывается без каких-либо символов модуля, и только в самом конце выражения добавляется оператор $(\mathrm{mod} p)$, чтобы напомнить, что все действия в этом выражении выполняются по модулю p. Более того, если смысл выражения понятен из контекста, оператор $(\mathrm{mod} p)$ могут вообще опустить, и тогда вам придется постоянно помнить о том, что все вычисления следует выполнять по модулю p.

Вообще-то заключать операцию взятия по модулю в скобки необязательно. Вполне допустимо записать ее как $a \bmod b$. Однако те, кто не привык к использованию оператора mod, могут поначалу путать его с обычным текстом. Чтобы избежать путаницы, мы будем заключать выражение $(a \bmod b)$ в скобки.

Обратите внимание: число, взятое по модулю p, всегда должно находиться в диапазоне $0, \ldots, p-1$, даже если исходное целое число было отрицательным. Некоторые языки программирования допускают применение операций по модулю к отрицательным числам (что очень раздражает математиков). Например, если мы возьмем –1 по модулю p, то получим $p-1$. Общее правило формулируется таким образом: чтобы подсчитать $(a \bmod p)$, необходимо найти числа q и r, такие, что $a = qp + r$ и $0 \leq r < p$. Результатом выражения $(a \bmod p)$ будет r. Если $a = -1$, то $q = -1$ и $r = p - 1$.

11.3.1 Сложение и вычитание

Складывать по модулю p очень легко. Просто сложите два числа и вычтите из полученного результата p, если он окажется бóльшим или равным p. Поскольку оба слагаемых находятся в диапазоне $0, \ldots, p-1$, их сумма не может превышать $2p-2$, поэтому, чтобы получить результат в нужном диапазоне, достаточно вычесть p не более одного раза.

Вычитание выполняется аналогично сложению. Вычтите из одного числа другое и, если полученный результат окажется отрицательным, добавьте к нему p.

Эти правила применимы только к числам, уже взятым по модулю p. Если же они находятся за пределами диапазона $0, \ldots, p-1$, вам придется выполнять полноценное взятие их суммы по модулю p.

Чтобы привыкнуть к вычислениям по модулю, нужно некоторое время. Вначале вас будут смущать выражения наподобие $5 + 3 = 1(\mathrm{mod} 7)$. Вы хорошо знаете, что 5 плюс 3 равняется никак не 1, а 8. В то же время, работая по модулю 7, мы имеем, что $8 \bmod 7 = 1$, а значит, $5 + 3 = 1(\mathrm{mod} 7)$.

Довольно часто, сами того не понимая, мы используем арифметические операции по модулю и в реальной жизни. Подсчитывая время суток, мы скла-

дываем часы по модулю 12 (или 24). В расписании автобусов может быть написано, что автобус отправляется в 55 минут такого-то часа, а его время в пути составляет 15 минут. Чтобы узнать, когда автобус приезжает в место назначения, мы складываем $55 + 15 = 10(\mathrm{mod}60)$ и определяем, что автобус приезжает в 10 минут следующего часа. Мы пока ограничимся вычислениями по модулю простого числа, однако вы можете выполнять операции по модулю любого числа, которое вам понравится.

11.3.2 Умножение

Выполнять умножение, как правило, немного труднее, чем сложение. Чтобы вычислить $(ab \bmod p)$, мы вначале подсчитываем произведение ab как обычное целое число и затем берем полученный результат по модулю p. Если оба числа находятся в диапазоне $0, \ldots, p-1$, их произведение может достигать числа $(p-1)^2 = p^2 - 2p + 1$. В этом случае вам придется выполнить полноценное деление с остатком, чтобы найти пару (q, r), такую, что $ab = qp + r$ и $0 \leq r < p$. Отбросьте q; искомым результатом является r.

Для большей наглядности рассмотрим пример. Пусть $p = 5$. Результатом выражения $3 \cdot 4(\mathrm{mod}p)$ будет 2. Действительно, $3 \cdot 4 = 12$ и $(12 \bmod 5) = 2$. Таким образом, $3 \cdot 4 = 2(\mathrm{mod}5)$.

11.3.3 Группы и конечные поля

Математики называют множество чисел по модулю простого числа $p(0, 1, \ldots, p-1)$ *конечным полем (finite field)* и часто именуют его "полем $\bmod p$" или же просто "mod$p$". Ниже приведено несколько полезных замечаний касательно вычислений в поле mod$p$.

- Если к элементу поля mod$p$ прибавить любое кратное числа p или вычесть из него любое кратное числа p, исходный элемент не изменится.
- Все результаты арифметических операций над полем mod$p$ будут находиться в диапазоне $0, 1, \ldots, p-1$.
- Все вычисления можно проводить в обыкновенных целых числах и применять операцию взятия по модулю только в последний момент. Таким образом, к элементам поля mod$p$ применимы все обычные алгебраические законы относительно сложения и умножения целых чисел (такие, как $a(b + c) = ab + ac$).

В литературе применяются разные символы для обозначения конечного поля целых чисел по модулю p. Мы будем обозначать его как $\mathbb{Z}_p$. В других источниках можно встретить обозначения $\mathrm{GF}(p)$ или даже $\mathbb{Z}/p\mathbb{Z}$.

Теперь необходимо познакомиться с понятием *группы (group)* — еще одним математическим термином, но на сей раз совсем простым. Группа — это множество чисел, для которых определена одна операция, например сложение или умножение[2]. Сумма двух элементов группы должна равняться третьему элементу этой же группы. В группе, для которой определена операция умножения, не может находиться число 0. (Во-первых, умножать на 0 не очень-то интересно, а во-вторых, делить на 0 нельзя.) Тем не менее числа $1, \ldots, p-1$ вместе с операцией умножения по модулю p образуют группу. Эта группа называется *мультипликативной группой по модулю p (multiplicative group modulo p)* и обозначается по-разному; мы будем использовать обозначение $\mathbb{Z}_p^*$. Конечное поле состоит из двух групп: аддитивной (группы сложения) и мультипликативной. Таким образом, конечное поле $\mathbb{Z}_p$ состоит из аддитивной группы, определенной с помощью операции сложения по модулю p, и мультипликативной группы $\mathbb{Z}_p^*$.

Группа может включать в себя *подгруппу (subgroup)*. Подгруппа содержит несколько элементов группы. Если применить операцию группы к двум элементам подгруппы, мы должны снова получить элемент подгруппы. На первый взгляд это звучит довольно сложно, поэтому рассмотрим небольшой пример. Числа по модулю 8 вместе с операцией сложения (по модулю 8) образуют группу. Числа $\{0, 2, 4, 6\}$ образуют подгруппу. Сложив два любых числа этой подгруппы по модулю 8, мы снова получим элемент этой же подгруппы. Аналогичное утверждение справедливо и для мультипликативных групп. Мультипликативная группа по модулю 7 состоит из чисел $1, \ldots, 6$ и операции умножения по модулю 7. Множество $\{1, 6\}$ образует подгруппу, как, впрочем, и множество $\{1, 2, 4\}$. Попробуйте перемножить два любых элемента одной и той же подгруппы по модулю 7, и вы сами убедитесь в том, что получите элемент этой же подгруппы.

Подгруппы применяются для ускорения некоторых криптографических операций. Они также часто используются злоумышленниками для нападения на системы. Вот почему понятие подгруппы так важно для нас.

До сих пор речь шла только о сложении, вычитании и умножении по модулю p. Чтобы полностью определить мультипликативную группу, нужно ввести операцию, обратную умножению, т.е. деление. Оказывается, что мы можем определить и операцию деления по модулю p. Самое простое определение этой операции выглядит следующим образом: $a/b(\mathrm{mod}p)$ — это такое число c, что $c \cdot b = a(\mathrm{mod}p)$. Делить на 0 нельзя, но оказывается, что выражение $a/b(\mathrm{mod}p)$ определено для всех чисел при $b \neq 0$.

[2] В действительности к группе выдвигается еще несколько требований, но все группы, о которых идет речь в этой книге, им соответствуют.

Как же вычислить частное двух чисел по модулю p на практике? Объяснение того, как это делается, займет несколько страниц. А для начала вновь вернемся на 2000 лет назад — к Евклиду и его алгоритму поиска наибольшего общего делителя.

11.3.4 Алгоритм поиска НОД

Еще раз вернемся к курсу математики старших классов: *наибольшим общим делителем (greatest common divisor — GCD)*, или *НОД*, двух чисел a и b называют наибольшее число k, такое, что $k|a$ и $k|b$. Другими словами, НОД(a, b) — это самое большое число, которое делит и a и b.

Алгоритм вычисления НОД двух чисел, который мы используем сегодня, был разработан еще Евклидом. Более подробное описание этого алгоритма содержится в книге Дональда Кнута [54].

функция GCD
вход: a Положительное число.
b Положительное число.
выход: k Наибольший общий делитель a и b.

```
assert a ≥ 0 ∧ b ≥ 0
while a ≠ 0 do
    (a, b) ← (b mod a, a)
od
return b
```

Почему этот алгоритм приводит к нужному результату? Вначале покажем, что операция присвоения, используемая в цикле, не изменяет множество общих делителей чисел a и b. Действительно, $(b \bmod a)$ — это всего лишь $b - sa$ для некоторого целого числа s. Любое число k, которое делит a и b, будет также делить a и $(b \bmod a)$. (Между прочим, обратное утверждение тоже верно.) А когда $a = 0$, b будет общим делителем чисел a и b, причем наибольшим делителем. Можете сами убедиться в том, что цикл рано или поздно придет к завершению, потому что значения a и b все время уменьшаются, пока одно из них не достигнет нуля.

В качестве примера подсчитаем НОД чисел 21 и 30. Начнем с пары $(a, b) = (21, 30)$. На первой итерации мы вычисляем $(30 \bmod 21) = 9$, а значит, получаем $(a, b) = (9, 21)$. На второй итерации вычисляем $(21 \bmod 9) = 3$ и получаем $(a, b) = (3, 9)$. И наконец, на последней итерации вычисляем $(9 \bmod 3) = 0$ и получаем $(a, b) = (0, 3)$. Алгоритм возвращает число 3, которое действительно является наибольшим общим делителем чисел 21 и 30.

У НОД есть еще один "родственник": *наименьшее общее кратное (least common multiple — LCM)*, или *НОК*. Наименьшее общее кратное чисел a и b —

это наименьшее число, которое делится и на a и на b. Например, НОК$(6, 8) = 24$. НОД и НОК тесно связаны между собой следующим соотношением:

$$\text{НОК}(a, b) = \frac{ab}{\text{НОД}(a, b)}.$$

Данное утверждение приводится просто как констатация факта, доказывать его мы не будем.

11.3.5 Расширенный алгоритм Евклида

Несмотря на всю свою прелесть, приведенный выше алгоритм не помог нам вычислить частное по модулю p. Для выполнения этой операции воспользуемся другим алгоритмом, известным как расширенный алгоритм Евклида. Его идея такова: вычисляя НОД(a, b), мы можем найти два числа u и v, такие, что НОД$(a, b) = ua + vb$. Это позволит подсчитать значение выражения $a/b (\mathrm{mod} p)$.

функция ExtendedGCD
вход: a Положительное число.
b Положительное число.
выход: k Наибольший общий делитель a и b.
(u, v) Целые числа такие, что $ua + vb = k$.

assert $a \geq 0 \wedge b \geq 0$
$(c, d) \leftarrow (a, b)$
$(u_c, v_c, u_d, v_d) \leftarrow (1, 0, 0, 1)$
while $c \neq 0$ **do**
Инвариант: $u_c a + v_c b = c \wedge u_d a + v_d b = d$
$q \leftarrow \lfloor d/c \rfloor$ $(c, d) \leftarrow (d - qc, c)$ $(u_c, v_c, u_d, v_d) \leftarrow (u_d - qu_c, v_d - qv_c, u_c, v_c)$
od
return d, (u_d, v_d)

Данный алгоритм во многом аналогичен алгоритму поиска НОД. Мы заменили переменные a и b новыми переменными c и d, чтобы сохранить исходные значения a и b, так как они нужны для инварианта. Мысленно заменив c и d переменными a и b, мы получим в точности алгоритм поиска НОД. (Мы записали формулу $d \bmod c$ в несколько ином виде, однако результат остается тем же.) Помимо этого, мы добавили четыре переменные, в которых хранятся значения коэффициентов инварианта; для каждого сгенерированного значения c или d мы отслеживаем, как представить это значение в виде линейной комбинации a и b. Инициализировать значения этих переменных легко, потому что в начале алгоритма значению c присваивается значение a, а значению

d — значение b. Когда мы изменяем значения c и d в ходе цикла, обновить значения переменных u и v также несложно.

Зачем нам нужен расширенный алгоритм Евклида? Пусть необходимо вычислить $1/b \bmod p$, где $1 \leq b < p$. Мы используем расширенный алгоритм Евклида, чтобы определить ExtendedGCD(b, p). Мы знаем, что НОД b и p равен единице, поскольку p — простое число и других делителей кроме единицы и самого себя у него нет. Но в результате применения функции ExtendedGCD мы получаем числа u и v, такие, что $ub+vp = \text{НОД}(b,p) = 1$. Другими словами, $ub = 1 - vp$ или, что то же самое, $ub = 1(\text{mod}p)$. Это значит, что $u = 1/b(\text{mod}p)$, т.е. u является числом, обратным b по модулю p. Теперь частное a/b может быть подсчитано путем умножения a на u. Мы получили формулу $a/b = au(\text{mod}p)$, а подсчитать значение такого выражения не составляет труда.

Расширенный алгоритм Евклида позволяет находить обратное число по модулю p, что, в свою очередь, может применяться для поиска частного по модулю p. Теперь вместе со сложением, вычитанием и умножением по модулю p мы можем выполнять все четыре элементарных операции над конечным полем по модулю p.

Обратите внимание: значение u может быть отрицательным, поэтому, прежде чем использовать его в качестве обратного числа b, рекомендуется подсчитать $u \bmod p$.

Внимательно посмотрев на алгоритм ExtendedGCD, вы заметите следующее: если в качестве выходных данных нас интересует только значение u, можно не вычислять значения переменных v_c и v_d, поскольку они никак не влияют на вычисление u. Это позволяет немного сократить объем работы, необходимый для подсчета частного по модулю p.

11.3.6 Вычисления по модулю 2

Особым случаем арифметики по модулю простого числа является арифметика по модулю 2. Действительно, поскольку 2 — это простое число, мы можем выполнять любые операции по модулю 2. Вычисления по модулю 2 знакомы всем, кто когда-нибудь занимался программированием. На рис. 11.1 представлены таблицы сложения и умножения по модулю 2. Сложение по модулю 2 — это не что иное, как операция “исключающее ИЛИ” (XOR), которая встречается во всех языках программирования. Умножение по модулю 2 — это всего лишь операция AND. В поле по модулю 2 существует только одна пара взаимно обратных чисел ($1/1 = 1$), поэтому деление совпадает с умножением. Думаем, вас нисколько не удивит, что поле $\mathbb{Z}_2$ является важным средством анализа многих алгоритмов, используемых компьютерами.

+	0	1
0	0	1
1	1	0

·	0	1
0	0	0
1	0	1

Рис. 11.1. Сложение и умножение по модулю 2

11.4 Большие простые числа

Некоторые криптографические алгоритмы используют очень большие простые числа. Говоря об "очень больших числах", мы имеем в виду числа с многими сотнями разрядов. Но не стоит беспокоиться — вам не придется работать с такими числами вручную. Для этого существуют компьютеры.

Чтобы компьютер смог оперировать такими большими числами, требуется библиотека арифметических операций многократной точности (multiprecision library). Мы не можем использовать числа с плавающей запятой, потому что у них нет нескольких сотен разрядов точности. Мы не можем использовать и обычные целые числа, поскольку в большинстве языков программирования значения целочисленных типов ограничены примерно десятком разрядов. Лишь некоторые языки программирования обладают встроенной поддержкой целых чисел с произвольной точностью. Написание функций для выполнения вычислений над большими целыми числами — весьма интересное занятие. Более подробно об этом можно прочитать в книге Кнута [54, раздел 4.3]. Тем не менее реализация библиотеки арифметических операций многократной точности требует намного больше работы, чем кажется на первый взгляд. Нам нужно не только получить правильный ответ, но и вычислить его настолько быстро, насколько это возможно. Существует довольно много особых случаев, рассмотрение которых требует крайне тщательного подхода. Поэтому рекомендуем оставить время для более важных вещей и загрузить из Internet одну из многочисленных бесплатных библиотек или же воспользоваться языком наподобие Python, который обладает встроенной поддержкой больших целых чисел.

Для реализации криптографических систем с открытым ключом нам понадобятся простые числа в 2000-4000 бит длиной. Базовый метод генерации таких больших простых чисел удивительно прост: взять случайное число и проверить, не является ли оно простым. Существует много хороших алгоритмов, позволяющих определить, является число простым или нет. Существует также масса простых чисел. В окружении числа n приблизительно одно из каждых $\ln n$ чисел является простым. (Натуральный логарифм n, или $\ln n$ — одна из стандартных функций, встречающихся на любом инже-

нерном калькуляторе. Проиллюстрировать, насколько медленно возрастает значение натурального логарифма больших чисел, можно на следующем примере: натуральный логарифм числа 2^k немного меньше, чем $0{,}7 \cdot k$.) Число длиной в 2000 бит находится в диапазоне от 2^{1999} до 2^{2000}. В этом диапазоне примерно одно из каждых 1386 чисел является простым. Кроме того, бóльшую часть чисел этого диапазона сразу же можно откинуть как составные, например четные числа.

Ниже приведен пример генерации больших простых чисел.

функция GenerateLargePrime
вход: l Нижняя граница диапазона, в котором должно находиться простое число.
u Верхняя граница диапазона, в котором должно находиться простое число.
выход: p Простое число в диапазоне $l, \ldots, u$.

Проверим корректность задания диапазона.
assert $2 < l \leq u$
Подсчитаем максимальное количество попыток.
$r \leftarrow 100\,(\lfloor \log_2 u \rfloor + 1)$
repeat
$r \leftarrow r - 1$
assert $r > 0$
Выберем в заданном интервале случайное число n.
$n \in_R l, \ldots, u$
Продолжим попытки до тех пор, пока не найдем простое число.
until isPrime(n)
return n

Оператор $\in_R$ используется для обозначения случайного выбора числа из заданного множества. Разумеется, для выполнения этой операции понадобится генератор псевдослучайных чисел.

Описанный алгоритм достаточно прост. Вначале мы проверяем, что интервал задан корректно. Случаи $l \leq 2$ и $l \geq u$ не представляют для нас интереса и влекут за собой массу проблем. Обратите внимание на граничное условие: случай $l = 2$ не допускается[3]. Затем мы подсчитываем, какое количество попыток нужно совершить, чтобы найти простое число. Существуют интервалы, которые не содержат простых чисел. Например, в интервале

[3] Алгоритм Рабина–Миллера, который мы будем применять позднее, не слишком хорошо работает, когда на его вход подается число 2. Это не страшно — мы и так знаем, что 2 является простым числом, поэтому генерировать его с помощью функции GenerateLargePrime нет никакой нужды.

$90, \ldots, 96$ все числа являются составными. Программа никогда не должна "зависнуть", невзирая на то, что было подано ей на вход, поэтому ограничим количество попыток и сгенерируем ошибку в случае, если это количество будет превышено. Как уже отмечалось, в окружении числа u примерно одно из каждых $0{,}7 \cdot \log_2 u$ чисел является простым. (Функция $\log_2$ — это логарифм по основанию 2. Для простоты ее можно определить следующим образом: $\log_2(x) := \ln x / \ln 2$). Подсчитать число $\log_2 u$ довольно сложно, а вот $\lfloor \log_2 u \rfloor + 1$ намного проще: это количество бит, необходимое для того, чтобы представить u в виде двоичного числа. Таким образом, если длина числа u равна 2017 бит, тогда $\lfloor \log_2 u \rfloor + 1 = 2017$. Множитель 100 гарантирует, что мы с большой вероятностью найдем простое число. Для достаточно больших интервалов вероятность того, что простое число не будет найдено, составляет менее 2^{-128}, поэтому ею можно пренебречь. В то же время наличие данного ограничения гарантирует, что выполнение функции GenerateLargePrime когда-нибудь завершится. В этом примере мы весьма небрежно использовали утверждения `assert` для генерации ошибки; реальная программа должна выдать ошибку с объяснениями того, что было сделано неправильно.

Основной цикл алгоритма очень прост. Проверив, не исчерпано ли число попыток, мы выбираем случайное число и с помощью функции isPrime выясняем, не является ли оно простым.

Убедитесь, что полученное число n действительно выбрано равновероятным случайным образом из диапазона $l, \ldots, u$. Кроме того, если простое число необходимо сохранить в секрете, интервал $l, \ldots, u$ должен быть достаточно большим. Если злоумышленник знает, какой интервал вы используете, и этот интервал содержит менее 2^{128} простых чисел, он потенциально сможет перебрать их все.

При желании вы можете гарантировать, что сгенерированное случайное число является нечетным, установив наименее значимый бит только что сгенерированного числа n. Поскольку число 2 не находится в заданном интервале, изменение бита никак не повлияет на вероятностное распределение выбираемых вами простых чисел, зато наполовину сократит число попыток, которые вам пришлось бы осуществить для обнаружения простого числа. К сожалению, данная операция допустима только в том случае, если u является нечетным. В противном случае установка наименее значимого бита может выбросить n за пределы допустимого диапазона.

Функция isPrime — это фильтр, состоящий из двух этапов. Первый этап — это простой тест, в котором мы пытаемся проверить делимость n на все малые простые числа. Это позволит быстро откинуть множество составных чисел, которые делятся на малые простые числа. Если же такие делители не будут найдены, в ход пойдет "тяжелая артиллерия" — тест Рабина–Миллера (Rabin–Miller).

функция isPrime
вход: n Целое число ≥ 3.
выход: b Булево значение, указывающее, является ли n простым числом.

```
assert n ≥ 3
for p ∈ {все простые числа ≤ 1000} do
    if p является делителем n then
        return p = n
    fi
od
return Rabin-Miller(n)
```

Если у вас нет желания генерировать малые простые числа, можете немного схитрить. Вместо того чтобы проверять делимость на все простые числа, вы можете проверять делимость на 2 и все нечетные числа $3, 5, 7, \ldots, 999$ в порядке возрастания последних. Эта последовательность содержит все простые числа до 1000, а также множество не интересующих нас составных чисел. Порядок использования чисел по возрастанию важен для того, чтобы малое составное число наподобие 9 было правильно распознано как составное. Граница 1000 выбрана произвольным образом и может быть изменена в целях оптимизации производительности.

Все, что нам остается, — это рассмотреть тот самый таинственный тест Рабина–Миллера, который выполняет всю тяжелую работу.

11.4.1 Проверка того, является ли число простым

Оказывается, проверить, является ли число простым, невероятно легко (по крайней мере по сравнению с разложением числа на множители и поиском его простых делителей). К сожалению, подобные "легкие" алгоритмы далеко не совершенны. Все они определяют простые числа с некоторой долей вероятности. Существует определенный риск получить неверный ответ. Повторяя одну и ту же проверку несколько раз, мы можем сократить вероятность ошибки до приемлемого уровня.

Для проверки того, является ли число простым, будем использовать тест Рабина–Миллера. Математическое обоснование этого теста выходит за рамки нашей книги, хотя его структура довольно проста. Цель теста — определить, является ли нечетное число n простым. Мы выбираем случайное число a, меньшее n (число a называется *базисом (basis)*), и проверяем наличие определенного свойства a по модулю n, которое выполняется всегда, когда n является простым числом. Тем не менее мы можем доказать, что, даже если n не является простым числом, это свойство будет выполняться максимум

для 25% всех возможных значений базиса. Повторяя этот тест для различных случайных значений a, можно убедиться в правильности полученного ответа. Если n действительно является простым числом, оно всегда будет проходить тест как простое число. Если не является — это смогут выявить по крайней мере 75% возможных значений a, и вероятность того, что составное число n успешно пройдет несколько тестов как простое, можно снизить до любого уровня. Мы ограничим вероятность получения неверного результата до 2^{-128}, чтобы достичь запланированного уровня безопасности.

Приведем тест Рабина–Миллера.

функция RABIN-MILLER
вход: n Нечетное число ≥ 3.
выход: b Булево значение, указывающее, является ли n простым числом.

```
assert n ≥ 3 ∧ n mod 2 = 1
Вначале найдем такую пару (s, t), что s — нечетное и 2^t s = n − 1.
(s, t) ← (n − 1, 0)
while s mod 2 = 0 do
    (s, t) ← (s/2, t + 1)
od
Вероятность получения неверного результата будет отслеживаться с помощью переменной k. Эта вероятность не превышает 2^{−k}.
Будем прокручивать цикл до тех пор, пока вероятность получения неверного результата не станет достаточно низкой.
k ← 0
while k < 128 do
    Выберем случайное a, такое, что 2 ≤ a ≤ n − 1.
    a ∈_r 2, ..., n − 1
    Ресурсоемкая операция: возведение в степень по модулю.
    v ← a^s mod n
    Если v = 1, n успешно проходит тест для базиса a.
    if v ≠ 1 then
        Если n — простое число, тогда последовательность v, v^2, ..., v^{2^t}
            должна завершиться числом 1, а последним числом, не равным единице, должно быть n − 1.
        i ← 0
        while v ≠ n − 1 do
            if i = t − 1 then
                return false
            else
```

```
                (v, i) ← (v^2 mod n, i + 1)
            fi
        od
    fi
```

Если мы добрались до этого этапа, значит, число n успешно прошло проверку на простоту для базиса a. Таким образом, мы сократили вероятность получения неверного результата в 2^2 раза, поэтому значение k можно увеличить на 2.

```
    k ← k + 2
od
return true
```

Данный алгоритм работает только для нечетных n, больших или равных 3, поэтому вначале мы проверяем соответствующее условие. Функция isPrime должна вызывать функцию Rabin-Miller только с корректным аргументом, однако каждая функция сама отвечает за проверку собственных входных и выходных значений. Никто не знает, как программное обеспечение может измениться в будущем.

В основе теста Рабина–Миллера лежит идея, известная как малая теорема Ферма[4]. Для любого простого числа n и для всех $1 \leq a < n$ справедливо отношение $a^{n-1} \bmod n = 1$. Чтобы понять доказательство этого утверждения, требуется более глубокое знание математики, чем то, на которое ориентирована эта книга. Небольшой тест (называемый также алгоритмом проверки на простоту Ферма) позволяет доказать это утверждение для ряда случайно выбранных значений a. К сожалению, существует несколько "неприятных" чисел, называемых числами Кармайкла (Carmichael). Эти числа являются составными, но проходят тест Ферма для всех (почти) базисов a.

Тест Рабина–Миллера является разновидностью теста Ферма. Вначале мы записываем $n-1$ как $2^t s$, где s — нечетное число. Если требуется вычислить a^{n-1}, можно вначале подсчитать a^s и затем возвести полученный результат в квадрат t раз, чтобы получить $a^{s \cdot 2^t} = a^{n-1}$. Если $a^s = 1(\mathrm{mod} n)$, тогда многократное возведение в квадрат не изменит результата, поэтому получаем, что $a^{n-1} = 1(\mathrm{mod} n)$. Если же $a^s \neq 1(\mathrm{mod} n)$, тогда мы анализируем числа $a^s, a^{s \cdot 2}, a^{s \cdot 2^2}, a^{s \cdot 2^3}, \ldots, a^{s \cdot 2^t}$ (разумеется, взятые по модулю n). Если n является простым числом, тогда, согласно теореме Ферма, последним элементом приведенной выше последовательности должна быть 1. Отметим также, что если n — простое, тогда единственными числами, удовлетворяющими уравнению $x^2 = 1(\mathrm{mod} n)$, являются 1 и $n-1$. (Легко проверить, что

[4]Существует несколько теорем, названных именем Ферма (Fermat). Наибольшую известность приобрела последняя теорема Ферма, касающаяся уравнения $a^n + b^n = c^n$. Ее доказательство слишком велико, чтобы приводить его здесь.

$(n-1)^2 = 1(\mathrm{mod} n)$.) Таким образом, если n — простое, тогда одно из чисел в этой последовательности должно равняться $n-1$, так как в противном случае последнее число никогда не будет равно единице. В действительности это и все, что проверяет тест Рабина–Миллера. Если хотя бы одно случайно выбранное a показывает, что число n является составным, мы сразу же возвращаем результат `false`. Если же n продолжает успешно проходить тест как простое число, мы повторяем проверку для другого значения a до тех пор, пока вероятность получения неверного результата не составит менее чем 2^{-128}.

Если применить этот тест к случайно выбранному числу, вероятность ошибки будет во много раз меньше установленного нами предела. Почти все составные числа n могут быть распознаны тестом Рабина–Миллера при использовании практически любых базисов. Создатели многих программных библиотек исходили именно из этого и ограничивались выполнением проверки примерно для 5-10 базисов. Идея в принципе неплоха, хотя нам нужно было исследовать, сколько попыток требуется для достижения уровня ошибки 2^{-128} или менее. Однако обратите внимание, что данное утверждение справедливо только при применении функции isPrime к *случайно* выбранным числам. Позднее мы столкнемся с ситуациями, когда тест Рабина–Миллера будет применяться к числам, полученным от кого-нибудь другого. Эти числа могут быть выбраны злоумышленником, поэтому функция isPrime должна выполнить все необходимые действия для гарантированного достижения уровня ошибки 2^{-128}.

Выполнение всех 64 тестов Рабина–Миллера необходимо тогда, когда число, проверяемое на простоту, получено от кого-нибудь другого. Это совершенно излишне, когда мы пытаемся сгенерировать простое число случайным образом. С другой стороны, пытаясь сгенерировать простое число, мы потратим бóльшую часть времени на отбрасывание составных чисел. (Практически все составные числа будут распознаны в ходе первого же теста Рабина–Миллера.) Поскольку для получения простого числа нам, вероятно, придется отбросить сотни составных чисел, выполнение 64 тестов для простого числа, когда оно в конце концов будет найдено, займет ненамного больше времени, чем выполнение 10 аналогичных тестов.

В более ранней версии этой главы у функции Rabin-Miller был еще один аргумент, который мог применяться для выбора максимальной вероятности ошибки. Впоследствии он оказался чудесным примером бесполезного параметра, а потому был удален. Гораздо проще (да к тому же и безопаснее применительно к возможности неправильного использования) всегда проводить проверку до тех пор, пока уровень ошибки не снизится до 2^{-128}.

У нас все еще остается вероятность 2^{-128} того, что функция isPrime выдаст неверный ответ. Чтобы понять, насколько в действительности мала эта вероятность, представьте себе, что вас убьет случайно залетевший метеорит

в то время, когда вы читаете это предложение. Так вот, вероятность этого печального события значительно превышает 2^{-128}. Все еще живы? Тогда не беспокойтесь о функции isPrime.

11.4.2 Оценивание степеней

Бóльшая часть времени выполнения теста Рабина–Миллера уходит на вычисление $a^s \bmod n$. Мы не можем вначале подсчитать значение a^s и затем взять его по модулю n. Ни один компьютер в мире не обладает достаточным количеством памяти даже для того, чтобы просто разместить в ней значение a^s, не говоря уже о процессорной мощности, необходимой для вычисления последнего. И a и s могут быть в тысячи бит длиной. Вместе с тем нам нужно только $a^s \bmod n$. Поэтому мы можем применять операцию $\bmod n$ ко всем промежуточным результатам, что не позволит числам разрастись до гигантских размеров.

Существует несколько способов вычисления $a^s \bmod n$, но мы приведем лишь несколько простых соображений. Чтобы подсчитать $a^s \bmod n$, используйте приведенные ниже правила.

- Если $s = 0$, тогда ответом будет 1.
- Если $s > 0$ и является четным числом, тогда вначале подсчитайте $y := a^{s/2} \bmod n$, используя эти же правила. Окончательный результат будет выглядеть следующим образом: $a^s \bmod n = y^2 \bmod n$.
- Если $s > 0$ и является нечетным числом, тогда вначале подсчитайте $y := a^{(s-1)/2} \bmod n$, используя эти же правила. Окончательный результат будет выглядеть следующим образом: $a^s \bmod n = a \cdot y^2 \bmod n$.

Это рекурсивная формулировка так называемого двоичного алгоритма. Если вы проанализируете описанные выше операции, то увидите, что необходимый показатель степени формируется бит за битом от наиболее значимой части двоичного представления показателя к наименее значимой его части. При желании данный рекурсивный алгоритм можно переписать в виде цикла.

Сколько операций умножения потребуется, чтобы вычислить $a^s \bmod n$? Пусть k — это число бит значения s; другими словами, $2^{k-1} \leq s < 2^k$. Тогда данный алгоритм требует выполнения не более чем $2k$ умножений по модулю n. Это совсем неплохо. Если мы проверяем, является ли простым 2000-битовое число, тогда длина s тоже будет составлять около 2000 бит и нам понадобится лишь 4000 вычислений. Этот объем работы, хотя и достаточно большой, определенно доступен вычислительным возможностям большинства настольных компьютеров.

Многие криптографические системы с открытым ключом используют операции возведения в степень по модулю наподобие рассмотренной выше. Хоро-

шая библиотека арифметических операций многократной точности должна иметь оптимизированную функцию для оценки таких операций. С выполнением этой задачи неплохо справляется особый тип умножения, называемый умножением Монтгомери (Montgomery). Существуют также способы вычисления a^s с использованием меньшего количества умножений [10, глава 4]. Каждый из этих приемов может сэкономить от 10 до 30% времени, необходимого для возведения в степень по модулю, поэтому в комбинации друг с другом они могут принести немалую пользу.

Прямые реализации операций возведения в степень по модулю зачастую бывают чувствительны к тайминг-атакам. Более подробно тайминг-атаки и способы борьбы с ними рассматриваются в главе 16.

Глава 12

Алгоритм Диффи–Хеллмана

Обсуждая вопросы криптографии с открытым ключом, мы собираемся проследить исторический путь ее развития. Впервые понятие криптографии с открытым ключом было введено в 1976 году Уитфилдом Диффи (Whitfield Diffie) и Мартином Хеллманом (Martin Hellman) в опубликованной ими статье *New Directions in Cryptography* (Новые направления в криптографии) [21].

До сих пор речь шла только о шифровании и аутентификации с общими секретными ключами. Но где же взять эти общие секретные ключи? Если вы хотите общаться, скажем, с 10 друзьями, то можете встретиться и обменяться секретными ключами друг с другом для дальнейшего использования в общении. Но, как и все секретные ключи, эти ключи должны подвергаться регулярному обновлению, поэтому вам придется вновь и вновь встречаться и обмениваться ключами. Для группы из 10 друзей нужно 45 секретных ключей. По мере увеличения этой группы количество необходимых ключей возрастает квадратически. Для ста человек, желающих общаться друг с другом, вам понадобится уже 4950 ключей! Ситуация быстро выходит из-под контроля.

Диффи и Хеллман поставили вопрос о том, нельзя ли проводить обмен ключами более эффективно. Предположим, у нас есть алгоритм шифрования, в котором для шифрования и дешифрования применяются разные ключи. В этом случае мы можем спокойно опубликовать ключ шифрования, а ключ дешифрования сохранить в секрете. Теперь каждый человек может послать нам зашифрованное сообщение, но расшифровать его сможем только мы. Это бы решило проблему необходимости распространения такого большого количества разных ключей.

Диффи и Хеллман сформулировали этот вопрос, но смогли дать на него лишь частичный ответ. Полученное ими решение известно сегодня как *протокол обмена ключами Диффи–Хеллмана (Diffie-Hellman key exchange protocol)*, сокращенно DH [21].

Идея протокола DH поистине остроумна. Оказывается, что два человека, взаимодействующие друг с другом по небезопасному каналу общения, могут договориться об использовании секретного ключа таким образом, что оба получат один и тот же ключ, не раскрывая его злоумышленнику, который прослушивает канал общения.

12.1 Группы

Если вы читали предыдущую главу, то не удивитесь, узнав, что в алгоритме Диффи–Хеллмана задействованы простые числа. В этой главе буквой p будет обозначаться большое простое число. Длина p составляет примерно 2000-4000 бит. Большинство вычислений в этой главе выполнены по модулю p, что не всегда будет указано явно. Протокол DH использует $\mathbb{Z}_p^*$ — мультипликативную группу по модулю p, которая описана в разделе 11.3.3.

Выберем любой элемент группы g и рассмотрим числа $1, g, g^2, g^3, \ldots$ (разумеется, взятые по модулю p). Это бесконечная последовательность чисел. С другой стороны, количество элементов группы $\mathbb{Z}_p^*$ конечно. (Напомним, что группа $\mathbb{Z}_p^*$ состоит из чисел $1, \ldots, p-1$, для которых определена операция умножения по модулю p.) По этой причине на каком-то этапе элементы последовательности должны начать повторяться. Предположим, что $g^i = g^j$, где $i < j$. Поскольку мы можем выполнять деление по модулю p, значит, можем поделить каждую сторону этого равенства на g^i, в результате чего получим: $1 = g^{j-i}$. Другими словами, существует число $q := j - i$, такое, что $g^q = 1(\mathrm{mod} p)$. Назовем наименьшее положительное число q, для которого выполняется равенство $g^q = 1(\mathrm{mod} p)$, *порядком (order)* числа g. (К сожалению, данная тема включает в себя большое количество терминов. Мы предпочитаем использовать стандартную терминологию, а не изобретать собственные слова. В противном случае читатели нашей книги могут запутаться, обратившись к другой литературе.)

Последовательно возводя g в степень, мы можем получить числа $1, g, g^2, \ldots, g^{q-1}$. После этого элементы последовательности начнут повторяться, так как $g^q = 1$. Число g называют *образующим элементом (generator)* и говорят, что оно порождает множество $1, g, g^2, \ldots, g^{q-1}$. Количество элементов, которое может быть представлено в виде степеней g, в точности равно q, т.е. порядку числа g.

Одно из свойств умножения по модулю p таково: существует по крайней мере одно число g, которое может породить все элементы группы $\mathbb{Z}_p^*$. Другими словами, существует по крайней мере одно g, для которого $q = p - 1$. Таким образом, вместо того чтобы рассматривать группу $\mathbb{Z}_p^*$ как множество чисел $1, \ldots, p-1$, мы можем представить ее и как множество $1, g, g^2, \ldots, g^{p-2}$. Число

g, порождающее все элементы группы, называется *примитивным элементом (primitive element)* группы.

Другие значения g могут порождать множества меньших размеров. Обратите внимание: если мы перемножим два числа из множества, порожденного элементом g, то получим еще одну степень g, а следовательно, еще один элемент этого же множества. Проверив все остальные признаки группы, мы получим, что множество, порожденное элементом g, тоже является группой. Таким образом, мы можем выполнять умножение и деление в этой группе точно так же, как и в большой группе по модулю p. Такие группы, входящие в состав других групп, называются подгруппами (см. раздел 11.3.3). Понятие подгруппы очень важно для осуществления атак на криптосистемы.

Осталось рассмотреть еще одно важное утверждение. Для любого элемента g порядок g является делителем числа $p-1$. Это легко показать. Пусть g — это примитивный элемент группы, а h — какой-нибудь другой элемент. Поскольку g порождает всю группу, существует такое x, что $h = g^x$. Теперь рассмотрим элементы множества, порожденного h. Это элементы $1, h, h^2, h^3, \ldots$, которые равны элементам $1, g^x, g^{2x}, g^{3x}, \ldots$ соответственно. (Напомним, что все наши вычисления проводятся по модулю p.) Порядок h — это наименьшее число q, для которого $h^q = 1$. Другими словами, это наименьшее число q, для которого $g^{xq} = 1$. Для любого t значение $g^t = 1$ тогда и только тогда, когда $t = 0(\mathrm{mod} p-1)$. Таким образом, порядок h — это наименьшее число q, для которого $xq = 0(\mathrm{mod} p-1)$. Это выполняется тогда, когда $q = (p-1)/\mathrm{НОД}(x, p-1)$. Отсюда очевидно, что q является делителем $p-1$.

Вот небольшой пример. Пусть $p = 7$. Число $g = 3$ является примитивным элементом, потому что $1, g, g^2, \ldots, g^5 = 1, 3, 2, 6, 4, 5$. (Напомним, что все вычисления выполняются по модулю p.) Элемент $h = 2$ порождает подгруппу $1, h, h^2 = 1, 2, 4$, так как $h^3 = 2^3 \bmod 7 = 1$. Элемент $h = 6$ порождает подгруппу $1, 6$. Размеры этих подгрупп равны 3 и 2 соответственно. Очевидно, оба этих числа являются делителями $p-1$.

Приведенное утверждение частично объясняет тест Ферма, который упоминался в разделе 11.4.1. Тест Ферма основан на том, что для любого a справедливо отношение $a^{p-1} = 1$. Это легко проверить. Пусть g — это примитивный элемент группы $\mathbb{Z}_p^*$. Возьмем x, такое, что $g^x = a$. Поскольку g порождает всю группу, такое x будет существовать для любого a. Но тогда $a^{p-1} = g^{x(p-1)} = (g^{p-1})^x = 1^x = 1$.

12.2 Базовый алгоритм Диффи–Хеллмана

Первоначальная версия протокола DH выглядела следующим образом. Мы выбираем большое простое число p и примитивный элемент g, который

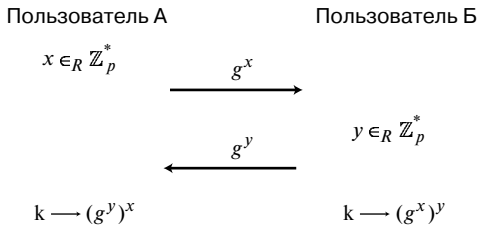

Рис. 12.1. Первоначальная версия протокола Диффи–Хеллмана

порождает всю группу $\mathbb{Z}_p^*$. И p и g являются открытыми константами этого протокола, поэтому предполагается, что они известны всем участникам общения, включая и злоумышленников. Схема обмена ключами приведена на рис. 12.1. Это один из общепринятых способов схематического изображения криптографических протоколов. У нас есть два участника: пользователь А и пользователь Б. Шкала времени направлена сверху вниз. Вначале пользователь А выбирает случайное число x из множества $\mathbb{Z}_p^*$, что эквивалентно выбору случайного числа из диапазона $1, \ldots, p-1$. Пользователь А вычисляет $g^x \bmod p$ и отсылает результат пользователю Б, который, в свою очередь, выбирает из множества $\mathbb{Z}_p^*$ случайное число y. Он вычисляет $g^y \bmod p$ и отсылает результат пользователю А. Окончательное значение секретного ключа выглядит как g^{xy}. Пользователь А может вычислить это значение, возведя число g^y, полученное от пользователя Б, в известную ему степень x. (Еще раз вспомним школьный курс математики: $(g^y)^x = g^{xy}$.) Аналогичным образом пользователь Б может вычислить значение ключа k как $(g^x)^y$. Оба пользователя получат одно и то же значение k, которое может применяться в качестве секретного ключа.

А что же наш злоумышленник? Он может перехватить значения g^x и g^y, но не знает x или y. Задача вычисления g^{xy} по заданным g^x и g^y известна как проблема Диффи–Хеллмана (сокращенно DH). Если числа p и g выбраны должным образом, тогда эффективного алгоритма вычисления g^{xy} не существует, по крайней мере нам таковой не известен. Наилучший из существующих методов состоит в том, чтобы вначале вычислить x по известному g^x, после чего подсчитать значение k как $(g^y)^x$ — именно так, как это делает пользователь А. На множестве действительных чисел вычисление x по заданному g^x называется функцией логарифма. Ее можно найти на любом инженерном калькуляторе. В конечном поле $\mathbb{Z}_p^*$ эта функция называется *дискретным логарифмом (discrete logarithm)*. В общем случае задача вычисления x по заданному g^x над конечной группой известна как проблема дискретного логарифма, или DL.

Первоначальная версия протокола DH имеет множество способов применения. Мы рассмотрели ее на примере обмена сообщениями между двумя участниками. Еще один возможный вариант — обратиться к каждому потенциальному участнику общения с просьбой выбрать случайное x и опубликовать свое значение $g^x \bmod p$ в цифровом эквиваленте телефонной книги. Теперь, если пользователь А захочет пообщаться с пользователем Б, он находит в телефонной книге значение g^y и подсчитывает g^{xy} для своего x. Пользователь Б может точно так же подсчитать значение g^{xy} без какого-либо предварительного взаимодействия с пользователем А. Такую схему удобно использовать в окружениях наподобие системы электронной почты, где участники общения не взаимодействуют напрямую.

12.3 Атака посредника

К сожалению, базовый протокол Диффи–Хеллмана не защищен от так называемой *атаки посредника (man-in-the-middle attack)*. Давайте еще раз взглянем на протокол. Пользователь А знает, что он с кем-то общается, но не знает, с кем именно. Злоумышленник Е может вклиниться в протокол, имитируя пользователя Б при общении с пользователем А, а также имитируя пользователя А при общении с пользователем Б. Схема этого процесса приведена на рис. 12.2. Для пользователя А данный протокол выглядит точно так же, как настоящий протокол Диффи–Хеллмана. Пользователь А никак не сможет определить, что он общается со злоумышленником Е, а не с пользователем Б. Это же справедливо и по отношению к пользователю Б. Злоумышленник Е сможет поддерживать видимость такого общения столько, сколько захочет. Представим себе, что пользователи А и Б начали общаться друг с другом, используя секретный ключ, который, как им кажется, они выбрали вдвоем. Все, что требуется от злоумышленника Е, — это послушно пересылать сообщения, отправленные пользователем А пользователю Б и наоборот. Разумеется, злоумышленник Е должен расшифровывать сообщения, полученные от пользователя А, который зашифровал их с помощью ключа k, а затем заново зашифровывать эти сообщения с помощью ключа k', чтобы отослать пользователю Б. Аналогичные операции нужно проделывать и над трафиком, идущим в другом направлении, однако больших усилий это не потребует.

Осуществить атаку посредника на цифровую телефонную книгу гораздо сложнее. Если лицо, опубликовавшее телефонную книгу, проверяет личность каждого человека, который присылает свое значение g^x, пользователь А может быть уверен в том, что он использует значение g^x пользователя Б. Позд-

Рис. 12.2. Атака посредника на протокол Диффи–Хеллмана

нее, когда речь пойдет о цифровых подписях и инфраструктуре открытого ключа, будут описаны и другие способы защиты от атаки посредника.

Существует одно окружение, которое позволяет противостоять атаке посредника без внедрения какой-либо дополнительной инфраструктуры. Если ключ k применяется для шифрования телефонных разговоров или видеосвязи, пользователь А может поговорить с пользователем Б и узнать его по голосу. Пусть h — это некоторая функция хэширования. Если пользователь Б прочитает пользователю А несколько первых разрядов значения $h(k)$, пользователь А сможет проверить, действительно ли он использует тот же секретный ключ, что и пользователь Б. Данное решение приносит хорошие результаты, однако только в тех ситуациях, когда знание ключа k можно связать с реальным человеком, находящимся “по ту сторону общения”. Однако большинство компьютерных коммуникаций не предоставляют такой возможности. Кроме того, злоумышленник Е может создать синтезатор речи, имитирующий голос пользователя Б. И наконец, самый большой недостаток этого решения — необходимость соблюдения пользователями строгой дисциплины. Но, как известно, пользователи сплошь и рядом игнорируют правила безопасности.

12.4 “Подводные камни” реализации

Реализация протокола DH может таить в себе массу подвохов. Например, если злоумышленник Е вмешивается в общение и заменяет оба числа g^x и g^y единицей, у пользователей А и Б оказывается ключ $k = 1$. В результате мы получаем протокол согласования ключей, который безупречен во всем, кроме одного: злоумышленник Е знает секретный ключ. Это, конечно же, очень плохо, а значит, нам нужно каким-либо образом предотвратить подобную атаку.

Еще одна распространенная проблема возникает тогда, когда образующий элемент g не является примитивным элементом группы $\mathbb{Z}_p^*$, а порождает лишь небольшую подгруппу. Пусть порядок g равен одному миллиону. В этом случае множество $\left\{1, g, g^2, \ldots, g^{q-1}\right\}$ содержит лишь миллион элементов. Поскольку в этом множестве оказывается и ключ k, злоумышленник Е сможет легко перебрать все варианты в поисках подходящего ключа. Очевидно, одно из требований состоит в том, чтобы у числа g был более высокий порядок. Но кто выбирает p и g? Все пользователи работают с одними и теми же значениями p и g, поэтому большинство из них получают эти значения от кого-нибудь другого. Для обеспечения безопасности общения они должны убедиться в том, что значения p и g выбраны правильно. Каждый из пользователей А и Б должен проверить, что p является простым числом и что g — это примитивный элемент группы по модулю p.

Отдельную проблему образуют подгруппы по модулю p. Пользователь Б может легко распознать атаку злоумышленника Е, заменившего g^x на 1; для этого достаточно лишь проверить соответствующее число. Но злоумышленник Е может заменить g^x числом h, имеющим низкий порядок. В этом случае ключ, сгенерированный пользователем Б, будет принадлежать множеству, порожденному элементом h. Поскольку количество элементов этого множества невелико, злоумышленник Е сможет легко перебрать все значения, чтобы найти k. (Разумеется, эта же атака может быть осуществлена и по отношению к пользователю А.) По этой причине пользователям А и Б необходимо проверять, не порождают ли полученные ими числа подгрупп с малым количеством элементов.

Давайте еще раз взглянем на подгруппы. При работе по модулю простого числа все (мультипликативные) подгруппы могут быть порождены одним элементом. Вся группа $\mathbb{Z}_p^*$ состоит из элементов $1, \ldots, p-1$. Каждая подгруппа может быть представлена в виде $1, h, h^2, h^3, \ldots, h^{q-1}$, где h — некоторый элемент группы $\mathbb{Z}_p^*$, а q — порядок h. Как уже отмечалось, оказывается, что q должно быть делителем $p - 1$. Другими словами, размер любой подгруппы должен быть делителем $p - 1$. Справедливо и обратное утверждение: для любого числа d, являющегося делителем $p - 1$, существует одна подгруппа

размера d. Если мы хотим, чтобы в группе не было подгрупп малых размеров, следует избегать чисел $p-1$, имеющих малые делители.

Проблема заключается вот в чем. Если p — это большое простое число, тогда $p-1$ всегда будет четным, а значит, будет делиться на 2. Таким образом, у нас всегда будет присутствовать подгруппа, состоящая из двух элементов: 1 и $p-1$. Тем не менее, за исключением этой подгруппы, можно избежать наличия других подгрупп малых размеров, требуя, чтобы число $p-1$ не имело других малых делителей.

12.5 Надежные простые числа

Наше решение состоит в том, чтобы использовать в качестве p *надежное простое число (safe prime)*. Это достаточно большое простое число вида $2q+1$, где q — также простое число. В этом случае мультипликативная группа $\mathbb{Z}_p^*$ будет иметь следующие подгруппы:

- тривиальная подгруппа, состоящая только из числа 1;
- подгруппа размером 2, состоящая из элементов 1 и $p-1$;
- подгруппа размером q;
- подгруппа размером $2q$.

Первые две подгруппы тривиальны, и справиться с ними совсем несложно. Третья — это и есть та группа, которую мы хотим использовать в схеме Диффи–Хеллмана. Последняя подгруппа, которая и является полной группой $\mathbb{Z}_p^*$, обладает одним существенным недостатком. Рассмотрим множество всех чисел по модулю p, которые являются квадратом какого-нибудь другого числа (разумеется, по модулю p). Оказывается, что половина всех чисел $1,\ldots,p-1$ являются квадратами, а половина — не являются. Любой примитивный элемент, порождающий всю группу, не может быть квадратом. (Если бы он был квадратом, тогда возведение этого числа в степень всегда давало бы квадрат, а значит, мы бы никогда не смогли породить элементы группы, которые не являются квадратами.)

В математике существует функция, называемая символом Лежандра (Legendre symbol). Она позволяет определить, является ли число по модулю p квадратом, не находя самого квадратного корня. Существуют эффективные алгоритмы для вычисления символа Лежандра. Таким образом, если образующий элемент g не является квадратом и мы посылаем g^x, тогда любой наблюдатель (например, злоумышленник E) сразу же сможет определить, что за число x — четное или нечетное. Если x четное, тогда g^x является квадратом. Если x нечетное, тогда g^x не является квадратом. Поскольку злоумышленник E может использовать символ Лежандра, чтобы определить,

является ли число квадратом, он может узнать, что за число x — четное или нечетное. Перед нами случай исключительного поведения: злоумышленник Е не может узнать значение x, за исключением наименее значимого бита. Чтобы избежать этой проблемы, необходимо использовать только квадраты по модулю p. Последние и образуют ту самую подгруппу порядка q. Еще одно замечательное свойство состоит в том, что q — простое число, а значит, вложенных подгрупп у нас не будет.

Вот как использовать надежное простое число. Выберите пару (p, q), такую, что $p = 2q + 1$ и оба числа p и q являются простыми. (Для этого можно воспользоваться функцией isPrime и методом проб и ошибок.) Выберите случайное число α в диапазоне $2, \ldots, p-2$ и установите $g = \alpha^2 (\mathrm{mod} p)$. Убедитесь, что $g \neq 1$ и $g \neq p-1$. (Если g будет равно одному из этих запрещенных значений, выберите другое α и повторите попытку.) Полученный набор параметров (p, q, g) вполне годится для использования в протоколе Диффи–Хеллмана.

Каждый раз, когда пользователь А (или Б) получает значение, которое, как предполагается, является степенью g, он должен проверить, действительно ли полученное значение попадает в подгруппу, порожденную числом g. Если вы используете надежное простое число так, как было описано выше, для проверки членства в подгруппе можно воспользоваться символом Лежандра. Существует также более простой, хотя и более медленный метод. Число r является квадратом тогда и только тогда, когда $r^q = 1 (\mathrm{mod} p)$. Мы также рекомендуем запретить использование числа 1, потому что оно всегда приводит к проблемам. Тогда окончательный вариант условия будет выглядеть следующим образом: $r \neq 1 \wedge r^q \bmod p = 1$.

12.6 Использование подгрупп меньшего размера

Недостаток использования метода надежных простых чисел — низкая эффективность. Если длина простого числа p равна n бит, тогда длина q равна $n-1$ бит, а значит, все показатели степеней будут иметь длину $n-1$ бит. Средняя операция возведения в степень будет включать в себя около $3n/2$ умножений по модулю p. Для больших чисел p такой объем работы выглядит достаточно громоздким.

Стандартным решением этой проблемы является использование подгрупп меньшего размера. Вот как это делается. Вначале в качестве q выбирается 256-битовое простое число. (Другими словами, $2^{255} < q < 2^{256}$.) Затем необходимо найти намного большее простое число p, такое, что $p = Nq + 1$ для некоторого произвольного N. Для этого выберем случайное число N из подходящего диапазона, вычислим значение p как $Nq + 1$ и проверим, является

ли p простым. Поскольку p должно быть нечетным, очевидно, что N должно быть четным. Длина простого числа p будет составлять несколько тысяч бит.

Теперь нужно найти элемент порядка q. Это делается точно так же, как и в методе надежных простых чисел. Выберем случайное α из группы $\mathbb{Z}_p^*$ и установим $g = \alpha^N$. Теперь убедимся, что $g \neq 1$ и $g^q = 1$. (Случай $g = p - 1$ охватывается вторым условием, потому что q — нечетное.) Если полученное значение g не удовлетворяет этим условиям, выберите другое α и повторите попытку. Полученный набор параметров (p, q, g) пригоден для использования в протоколе Диффи–Хеллмана.

Когда используется подгруппа меньшего размера, значения, которыми обмениваются пользователи А и Б, всегда попадают в подгруппу, порожденную g. Между тем злоумышленник Е может вмешаться в процесс общения и заменить g^x абсолютно другим значением. Таким образом, каждый раз, когда пользователь А или Б получает значение, которое, как предполагается, должно находиться в подгруппе, порожденной g, он должен проверить, действительно ли это так. Выполнение проверки происходит точно так же, как и в методе надежных простых чисел. Число r находится в нужной подгруппе, если $r \neq 1 \wedge r^q \bmod p = 1$. Разумеется, следует проверить и то, не попадает ли r за пределы диапазона чисел по модулю p, поэтому полное условие проверки должно выглядеть как $1 < r < p \wedge r^q = 1$.

Для всех элементов r подгруппы, порожденной g, справедливо равенство $r^q = 1$. Таким образом, чтобы возвести число r в некоторую степень e, достаточно подсчитать $r^{e \bmod q}$. Это может значительно сократить объем работы, особенно если e намного больше, чем q.

Насколько эффективен метод подгрупп? Длина большого простого числа p составляет, как минимум, 2000 бит. В методе надежных простых чисел вычисление g^x требует около 3000 умножений. При использовании подгрупп это число сокращается примерно до 384, так как значение x можно взять по модулю q, в результате чего длина x составит лишь 256 бит. Как видите, во втором случае для вычисления g^x выполняется почти в 8 раз меньше действий, чем в первом. Чем больше значение p, тем больше экономия усилий. Вот почему подгруппы так широко применяются в криптографии.

12.7 Размер p

Правильно выбрать размер параметров схемы Диффи–Хеллмана довольно сложно. До сих пор мы опирались на требование, что злоумышленнику необходимо осуществить не менее 2^{128} шагов, чтобы напасть на систему. Этого было сравнительно легко достичь для алгоритмов симметричного шифрования. Операции же с открытым ключом наподобие алгоритма Диффи–

Хеллмана обходятся гораздо дороже, причем с увеличением необходимого уровня безопасности их стоимость возрастает намного быстрее.

Если придерживаться требования относительно 2^{128} шагов, тогда длина p должна составлять около 6800 бит. На практике такой размер p представляет собой реальную проблему с точки зрения производительности.

Существует огромная разница между размерами ключей алгоритмов симметричного шифрования и алгоритмов шифрования с открытым ключом наподобие схемы Диффи–Хеллмана. Не стоит впадать в заблуждение, сравнивая размеры ключей симметричного шифрования (обычно это 128 или 256 бит) с размерами открытых ключей, которые могут достигать тысяч бит. Размеры открытых ключей всегда во много раз больше, чем размеры ключей симметричного шифрования[1].

Операции с открытым ключом выполняются намного медленнее, чем функции шифрования и аутентификации, которые рассматривались ранее. В большинстве систем операции симметричного шифрования не оказывают значительного влияния на производительность, в то время как операции с открытым ключом могут существенно ее снизить. Поэтому следует уделить особое внимание аспектам производительности операций с открытым ключом.

Размеры ключей симметричного шифрования обычно являются фиксированными. Ориентируя систему на использование конкретного блочного шифра и функции хэширования, мы также фиксируем в ней и размер ключа. Это означает, что размер ключа симметричного шифрования остается неизменным на протяжении всего времени жизни системы. В отличие от этого, размеры открытых ключей практически всегда являются переменными, что позволяет легко изменять размер ключа в системе. Мы разрабатываем систему, которая будет использоваться в течение 30 лет, причем данные должны сохраняться в секрете на протяжении 20 лет после того, как они были впервые обработаны этой системой. Размер ключа симметричного шифрования изначально должен быть достаточно большим, чтобы он мог защитить систему на протяжении следующих 50 лет. Но открытые ключи, имеющие переменный размер, должны защищать данные только в течение 20 лет. В конце концов, все ключи имеют ограниченное время жизни. Открытый ключ может быть действителен на протяжении одного года и должен защищать данные еще 20 лет. Таким образом, открытый ключ должен защищать данные только на протяжении 21 года, а не 50 лет, как требуется от ключа симметричного шифрования. Каждый год мы генерируем новый открытый ключ, поэтому по

[1]Это касается схем шифрования с открытым ключом, которые рассматриваются в данной книге. Другие схемы шифрования с открытым ключом, например основанные на использовании эллиптических кривых, имеют совершенно другие размеры ключей.

мере развития компьютерных технологий можем выбирать ключи все большего и большего размера.

Лучшие оценки того, какой длины должно быть число p, содержатся в работе [62]. Согласно предположениям ее авторов, простое число длиной 2048 бит сможет обеспечить защиту данных до 2022 года, число длиной 3072 бит — до 2038 года, а число длиной 4096 бит — до 2050 года. Упоминавшееся нами число 6800 было получено с использованием все тех же формул, приведенных в [62]. Именно этот размер p гарантирует, что для осуществления атаки злоумышленнику придется выполнить 2^{128} шагов.

Будьте крайне осторожны с прогнозами подобного рода. Разумеется, все они имеют под собой серьезные основания, однако попытки предсказания будущего всегда небезопасны. Мы можем сделать некоторые осмысленные предположения относительно размеров ключей на протяжении следующих 10 лет, однако всякие рассуждения о том, как будут обстоять дела через 50 лет, достаточно наивны. Сравните хотя бы текущее положение дел в компьютерной индустрии и криптографии с тем, что наблюдалось 50 лет назад. Прогнозы, приведенные в [62], — это лучшие оценки на сегодняшний момент, однако безоговорочно доверять им тоже не стоит.

Так что же нам делать? Как разработчики криптографических систем, мы должны выбрать размер ключа, который сможет обеспечить безопасность системы на протяжении по крайней мере следующих 20 лет. Очевидно, 2048 бит — это нижняя граница. Чем больше, тем лучше; но использование ключей большего размера значительно повышает затраты. Учитывая такую неопределенность ситуации, мы предпочитаем скептическую точку зрения. Наш совет выглядит следующим образом: используйте 2048 бит в качестве абсолютного минимума. (И не забывайте, что с течением времени этот минимум будет повышаться.) Если позволяет производительность системы, используйте 4096 бит или около того. И еще: обязательно убедитесь, что ваша система способна работать с параметрами размером до 8192 бит. Это спасет положение в случае неожиданного усовершенствования атак на системы с открытым ключом. Новые достижения криптоанализа практически неминуемо приведут к осуществлению успешных атак на ключи меньших размеров. Переход на ключи бóльших размеров можно проводить прямо во время работы системы. Разумеется, это определенным образом скажется на производительности, однако основная функциональность системы будет сохранена. Это намного лучше, чем полная потеря безопасности и необходимость переделывать всю систему, что потребуется, если система не поддерживает ключи большего размера.

Некоторые приложения требуют, чтобы секретность их данных сохранялась намного дольше 20 лет. В этом случае большие ключи необходимо использовать уже сейчас.

12.8 Практические правила

Приведем несколько практических правил относительно того, как задать подгруппу, которая будет применяться в протоколе DH.

Выберите в качестве q 256-битовое простое число. (Показатели степеней в алгоритме DH могут подвергнуться атакам на основе коллизий, поэтому длина всех показателей должна достигать минимум 256 бит, чтобы злоумышленнику пришлось проделать не менее 2^{128} операций.) Выберите в качестве p большое простое число вида $Nq+1$ для некоторого целого N. (О том, какого размера должно быть число p, см. в разделе 12.7. Вычислить соответствующий интервал для N не составляет никакого труда.) Выберите случайное g, такое, что $g \neq 1$ и $g^q = 1$. (Для этого проще всего выбрать случайное α, установить $g = \alpha^N$ и проверить, подходит ли полученное значение g. В случае неудачи выберите другое α и повторите попытку.)

Любой участник общения, который получает описание подгруппы (p, q, g), должен убедиться в следующем:

- и p и q являются простыми числами, длина q составляет 256 бит, а длина p достаточно велика (не доверяйте ключам слишком малого размера);
- число q является делителем числа $(p-1)$;
- $g \neq 1$ и $g^q = 1$.

Это необходимо проверить даже тогда, когда описание подгруппы приходит из надежного источника. Вы бы наверняка удивились, узнав, как часто системы отказывают самым непредсказуемым образом, особенно когда на них нападают. Проверка параметров (p, q, g) занимает определенное время. Впрочем, поскольку в большинстве систем одна и та же подгруппа используется на протяжении долго времени, проводить проверку придется только единожды.

Каждый раз, когда участник общения получает число r, которое должно находиться в заданной подгруппе, он должен удостовериться, что $1 < r < p$ и $r^q = 1$. Обратите внимание: значение $r = 1$ *не* допускается.

Используя эти правила, мы получаем версию протокола Диффи–Хеллмана, изображенную на рис. 12.3. Оба участника начинают общение с проверки параметров подгруппы. Каждый участник должен проделать эту процедуру только единожды при запуске системы. Выполнять ее при каждом запуске протокола DH не нужно. (Несмотря на это, проверку следует повторять после каждой перезагрузки или повторной инициализации, так как в ходе последних значения параметров могут измениться.)

Оставшаяся часть протокола DH во многом идентична первоначальной версии, схема которой представлена на рис. 12.1. Теперь пользователи А и Б применяют подгруппу, поэтому показатели x и y будут находиться в диапазоне $1, \ldots, q-1$. Оба пользователя должны проверить, действительно ли

Рис. 12.3. Протокол Диффи–Хеллмана в рамках подгруппы

полученное ими число принадлежит к нужной подгруппе, чтобы избежать атак, связанных с использованием подгруппы малого размера.

На рис. 12.3 проверка обозначена операторами сравнения (например, $=$ или $<$), над которыми стоят знаки вопроса. Это означает, что пользователь А (или Б) должен проверить, действительно ли между значениями сохраняется указанное отношение. Если оно сохраняется, тогда все в порядке. Если же отношение неверно, тогда пользователь А должен предположить, что на него осуществляется нападение. В этом случае необходимо остановить выполнение протокола, прекратить отправку сообщений и уничтожить все данные, касающиеся протокола. Например, если проверка последней группы условий покажет, что они неверны, пользователь А должен уничтожить значения x и Y. Более подробно о том, как бороться с такими нарушениями, речь идет в разделе 14.5.5.

Наш протокол представляет собой безопасный вариант схемы Диффи–Хеллмана, но он не должен применяться в точности так, как здесь описано. Результат k должен пройти хэширование, прежде чем будет использоваться оставшейся частью системы. Более подробно это рассматривается в разделе 15.6.

12.9 Что может пойти не так

Лишь некоторые книги и статьи обращают внимание читателей на важность проверки того, действительно ли значения, полученные от собеседника, принадлежат необходимой подгруппе. Нильс впервые обнаружил это проблему в IKE (Internet Key Exchange — обмен ключами в Internet), одном из протоколов IPSec. Некоторые протоколы IKE используют обмен ключами по схе-

ме Диффи–Хеллмана. Поскольку IKE функционирует в реальном мире, ему приходится иметь дело с потерянными сообщениями. Поэтому IKE требует от пользователя Б, чтобы тот в случае отсутствия ответа повторно отослал свое последнее сообщение. При этом IKE не указывает, как именно пользователь А должен обрабатывать сообщение, повторно отосланное ему пользователем Б. Между тем пользователю А легко совершить серьезную ошибку.

Для простоты предположим, что пользователи А и Б применяют протокол DH в рамках подгруппы (см. рис. 12.3), но не проверяют правильность значений X и Y. Более того, после обмена данными пользователь А применяет ключ k, чтобы отослать пользователю Б зашифрованное и аутентифицированное сообщение, содержащее еще некоторые параметры протокола. (Это стандартная ситуация, аналогичные которой могут случаться и в IKE.)

Пользователь А может совершить следующую ошибку. Получив копию повторно отправленного сообщения, содержащего значение Y, он просто пересчитывает значение ключа k и отсылает соответствующий ответ пользователю Б. Звучит довольно безобидно, не так ли? Но тут в дело вступает злоумышленник Е. Пусть d — это малый делитель числа $(p-1)$. Злоумышленник Е может заменить Y элементом порядка d. Теперь ключ k пользователя А может иметь лишь d возможных значений и полностью определяется значениями Y и $(x \bmod d)$. Злоумышленник Е перебирает все возможные значения $(x \bmod d)$, вычисляет ключ k, который должен был получить пользователь А, и пытается расшифровать следующее сообщение, отправленное пользователем А. Если значение $(x \bmod d)$ было угадано верно, сообщение будет успешно расшифровано, а злоумышленник узнает точное значение $(x \bmod d)$.

Но что, если у числа $p-1$ есть целый ряд малых делителей $(d_1, d_2, \ldots, d_k)$? Тогда злоумышленник Е сможет повторить атаку для каждого из этих делителей и узнать значения $(x \bmod d_1), \ldots, (x \bmod d_k)$. Используя общую форму китайской теоремы об остатках (см. раздел 13.2), злоумышленник сможет определить значение $(x \bmod d_1 d_2 d_3 \ldots d_k)$. Таким образом, если произведение всех малых делителей числа $p-1$ достаточно велико, злоумышленник получит значительный объем информации об x. Поскольку x предполагается сохранять в секрете, дальнейшее развитие событий будет далеко не самым приятным. В нашем случае злоумышленник Е может завершить атаку, переслав пользователю А исходное значение Y и подождав, пока пользователи А и Б выполнят все действия, требуемые протоколом. Но теперь у злоумышленника накопилось достаточно информации об x, чтобы найти ключ k, который применяют пользователи А и Б.

Немного уточним: описанная атака — это вовсе не атака на IKE. Это атака на реализацию IKE, которая допускается стандартом. Несмотря на это, на наш взгляд, протокол должен содержать достаточно информации для того, чтобы компетентный разработчик мог создать безопасную реализацию.

Пренебрежение этой информацией крайне опасно, поскольку практика показывает, что рано или поздно кто-нибудь реализует данный протокол не так, как нужно.

Злоумышленнику E должно повезти, чтобы у числа $p-1$ оказалось достаточно малых делителей. Мы проектируем систему в расчете на атаку, состоящую из 2^{128} шагов. Это позволит злоумышленнику воспользоваться всеми делителями $p-1$ величиной до 2^{128} или около того. Нам никогда не попадался хороший анализ вероятности того, сколько информации может получить злоумышленник, но уже быстрая оценка показывает, что в среднем злоумышленник сможет получить около 128 бит информации об x, используя делители, меньшие 2^{128}. Затем злоумышленник сможет напасть на неизвестную часть x, используя атаку на основе коллизий. Поскольку длина x составляет всего 256 бит, это может привести к реальной атаке. По крайней мере могло бы привести, если бы мы не проверяли принадлежность X и Y к нужной подгруппе.

Осуществить атаку становится еще проще, если злоумышленник является тем самым человеком, который выбирает подгруппу (p, q, g). Он может сам перемножить малые делители, чтобы получить число $p-1$ и таким образом выбрать p раньше, чем q. Он также может присутствовать на заседании комитета, который рекомендует определенные параметры в качестве стандарта. Это не так невероятно, как кажется. Правительство США в лице NIST заботливо предлагает простые числа, которые могут быть использованы в DSA (Digital Signature Algorithm — алгоритм цифровой подписи), схеме цифровой подписи, работающей с подгруппами наподобие этой. Другие органы того же правительства США (NSA, ЦРУ, ФБР и т.п.) весьма заинтересованы в том, чтобы иметь возможность вмешиваться в частное общение. Мы, конечно, не хотим сказать, что эти простые числа плохи, однако подобные вещи нужно тщательно проверять перед использованием. Это совсем несложно; более того, NIST опубликовала алгоритм выбора параметров, не имеющих малых делителей, и вы можете проверить, действительно ли параметры вашей подгруппы были выбраны в соответствии с этим алгоритмом. К сожалению, его практически никто не придерживается.

Подводя итог сказанному выше, можно сделать следующий вывод: самое простое решение — проверять, действительно ли каждое полученное вами значение принадлежит к нужной подгруппе. Все остальные методы борьбы с атаками, основанными на использовании подгрупп малого размера, намного сложнее. Вы можете попытаться непосредственно найти все малые делители числа $p-1$, но это слишком трудно. Вы можете потребовать, чтобы человек, генерирующий набор параметров, предоставил вам разложение на множители числа $p-1$, но это существенно повышает сложность всей системы. Проверка полученных значений на принадлежность к нужной подгруппе занимает определенное время, зато это наиболее простое и самое надежное решение.

Глава 13

Алгоритм RSA

Схема Райвеста–Шамира–Адлемана (Rivest–Shamir–Adleman — RSA), пожалуй, наиболее популярная криптосистема с открытым ключом (во всяком случае это самая известная криптосистема с открытым ключом). Алгоритм RSA обеспечивает как цифровое подписывание, так и шифрование, что делает его весьма универсальным средством. Кроме того, в основу RSA положена проблема выделения множителей больших чисел, которая на протяжении последних тысячелетий занимала лучшие умы человечества, а потому была подвергнута самому тщательному исследованию.

13.1 Введение

Алгоритм RSA во многом схож с алгоритмом Диффи–Хеллмана (см. главу 12, "Алгоритм Диффи–Хеллмана"), хотя между ними существует принципиальное различие. Алгоритм Диффи–Хеллмана (или сокращенно DH) основан на использовании односторонней функции: предполагая, что p и g публично известны, мы можем найти $g^x (\mathrm{mod} p)$ по заданному x, но не можем найти x по заданному $g^x (\mathrm{mod} p)$. Алгоритм RSA, в свою очередь, основан на использовании *односторонней функции с лазейкой (trapdoor one-way function)*. Зная публично известные n и e, мы можем найти $m^e (\mathrm{mod} n)$ по заданному m, но не наоборот. При этом, зная, как n раскладывается на множители, выполнить обратную операцию очень легко. Разложение числа n на множители и есть той самой "лазейкой". Если мы знаем эту информацию, то можем легко выполнить обратное действие, а если не знаем, то не можем. Наличие лазейки позволяет применять RSA как для шифрования, так и в схеме цифровой подписи. Алгоритм RSA изобретен Рональдом Райвестом (Ronald Rivest), Ади Шамиром (Adi Shamir) и Леонардом Адлеманом (Leonard Adleman) и впервые опубликован в 1978 году [80].

В материале этой главы используются значения p, q и n. Значения p и q — это два разных больших простых числа, длина каждого из которых составляет порядка тысячи бит или более. Значение n определяется как $n := pq$. (На сей раз это обычное произведение, а не взятое по модулю какого-нибудь числа.)

13.2 Китайская теорема об остатках

Вместо того чтобы проводить вычисления по модулю простого числа p, как это было в схеме Диффи–Хеллмана, мы будем работать по модулю составного числа n. Чтобы лучше понять изложенное, вы должны познакомиться еще с некоторыми аспектами теории чисел. Пожалуй, наиболее полезным из них является *китайская теорема об остатках (Chinese Remainder Theorem — CRT)*. Таким названием данная теорема обязана тому, что в своей базовой форме впервые была доказана китайским математиком Сунь Цзе (Sun Tsu), жившим в первом веке до нашей эры. (Большинство математических понятий, которые нужно знать для работы с алгоритмами Диффи–Хеллмана и RSA, были сформулированы еще тысячи лет назад, поэтому не могут быть слишком сложными, не так ли?)

Числа по модулю n — это $0, 1, \ldots, n-1$. Они не образуют конечное поле, как в том случае, если бы n было простым числом. Математики обозначают это множество чисел как $\mathbb{Z}_n$ и называют его *кольцом (ring)*, но нам этот термин не понадобится. Для каждого x из $\mathbb{Z}_n$ можно вычислить пару $(x \bmod p, x \bmod q)$. Китайская теорема об остатках утверждает, что можно выполнить и обратную операцию: зная $(x \bmod p, x \bmod q)$, восстановить исходное значение x.

Для упрощения записи введем обозначение $(a, b) := (x \bmod p, x \bmod q)$.

Вначале покажем, что восстановление исходного значения x вообще возможно, а затем приведем алгоритм его вычисления. Чтобы вычислить x по заданным (a, b), следует убедиться, что в $\mathbb{Z}_n$ не существует второго такого числа x', для которого $x' \bmod p = a$ и $x' \bmod q = b$. В противном случае и x и x' привели бы к появлению одной и той же пары (a, b), и ни один алгоритм не смог бы распознать, какое из этих значений является исходным.

Пусть $d := x - x'$ — это разность чисел, которым соответствует одна и та же пара (a, b). Имеем $(d \bmod p) = (x - x') \bmod p = (x \bmod p) - (x' \bmod p) = a - a = 0$, а следовательно, d кратно p. Аналогичным образом получаем, что d кратно q. Отсюда следует, что d является кратным НОК(p, q), так как НОК — это *наименьшее* общее кратное. Поскольку p и q — это неодинаковые простые числа, НОК$(p, q) = pq = n$, а значит, $x - x'$ кратно n. Но и x и x' лежат в диапазоне $0, 1, \ldots, n-1$, поэтому разность $x - x'$, кратная n, находится

в диапазоне $-n+1, \ldots, n-1$. Единственным возможным значением такой разности, кратным n, является $x - x' = 0$, или $x = x'$. Это доказывает, что для любой заданной пары (a, b) существует не более одного x, удовлетворяющего условиям теоремы. Остается лишь найти значение x.

13.2.1 Формула Гарнера

Самым удобным способом вычисления x является так называемая *формула Гарнера (Garner's formula)*:

$$x = (((a - b)(q^{-1} \bmod p)) \bmod p) \cdot q + b.$$

Здесь множитель $(q^{-1} \bmod p)$ — это константа, которая зависит только от p и q. Не забывайте, что мы можем выполнять деление по модулю p, а значит, и вычислять $(1/q \bmod p)$, что является всего лишь другой формой записи выражения $(q^{-1} \bmod p)$.

Нам не нужно понимать, откуда взялась формула Гарнера; требуется лишь доказать, что результат x правилен.

Вначале покажем, что x находится в диапазоне $0, \ldots, n-1$. Очевидно, $x \geq 0$. Значение $t := (((a-b)(q^{-1} \bmod p)) \bmod p)$ должно попадать в диапазон $0, \ldots, p-1$, так как является результатом вычисления по модулю p. Если $t \leq p-1$, тогда $tq \leq (p-1)q$ и $x = tq + b \leq (p-1)q + (q-1) = pq - 1 = n - 1$. Как видите, значение x действительно находится в диапазоне $0, \ldots, n-1$.

Теперь покажем, что найденное значение x дает правильный результат и по модулю p, и по модулю q.

$$\begin{aligned} x \bmod q &= ((((a - b)(q^{-1} \bmod p)) \bmod p) \cdot q + b) \bmod q \\ &= (K \cdot q + b) \bmod q && \text{для некоторого } K \\ &= b \bmod q \\ &= b \end{aligned}$$

Выражение $(((a - b)(q^{-1} \bmod p)) \bmod p)$, которое умножается на q, — это некоторое целое число K, но при выполнении операций по модулю q любое кратное q можно отбросить. Вычислить $x \bmod p$ немного сложнее.

$$\begin{aligned} x \bmod p &= ((((a - b)(q^{-1} \bmod p)) \bmod p) \cdot q + b) \bmod p \\ &= (((a - b)q^{-1}) \cdot q + b) \bmod p \\ &= ((a - b)(q^{-1}q) + b) \bmod p \\ &= (a - b) + b) \bmod p \\ &= a \bmod p \\ &= a \end{aligned}$$

В первой строке мы просто расписываем формулу ($x \bmod p$). Во второй избавляемся от нескольких лишних операторов mod p. Затем изменяем порядок умножения, что никак не сказывается на результате. (Возможно, вы еще помните из школьного курса математики, что операция умножения обладает ассоциативностью, т.е. $(ab)c = a(bc)$.) На следующем шаге мы замечаем, что $(q^{-1}q) = 1(\mathrm{mod}p)$, а потому можем избавиться и от этого множителя. Оставшаяся часть преобразований тривиальна.

Это выведение намного сложнее, тех, которые встречались до сих пор, особенно потому, что в нем используется больше алгебраических законов. Если вы не совсем его поняли, не стоит беспокоиться.

Из всего сказанного выше следует, что формула Гарнера дает результат x, который лежит в нужном диапазоне и для которого $(a, b) = (x \bmod p, x \bmod q)$. Поскольку мы уже знаем, что такое решение может быть только одно, результат формулы Гарнера полностью решает китайскую теорему об остатках.

В реальных системах значение $q^{-1} \bmod p$ обычно подсчитывают предварительно, поэтому применение формулы Гарнера требует выполнения одного вычитания по модулю p, одного умножения по модулю p, одного полного умножения и одного сложения.

13.2.2 Обобщение

Китайская теорема об остатках справедлива и тогда, когда n является произведением нескольких разных простых чисел[1]. Формулу Гарнера можно обобщить, чтобы она распространялась и на подобные ситуации, но мы этого делать не будем.

13.2.3 Использование

Так где же может пригодиться китайская теорема об остатках? Если вам когда-нибудь понадобится выполнить множество вычислений по модулю n, использование этой теоремы позволит сэкономить массу времени. Для числа $0 \le x < n$ назовем пару $(x \bmod p, x \bmod q)$ представлением x по китайской теореме об остатках или CRT-представлением x. Если мы представим x и y по китайской теореме об остатках, тогда CRT-представление суммы $(x+y)$ будет иметь вид $((x + y) \bmod p, (x + y) \bmod q)$, что очень легко подсчитать, имея CRT-представления x и y. Первый элемент этой пары $(x + y) \bmod p$ может быть вычислен как $((x \bmod p)+(y \bmod p) \bmod p)$. Это просто сумма (по моду-

[1] Существуют варианты китайской теоремы об остатках для n, которое делится на квадрат или более высокую степень некоторых простых чисел, но они еще более сложны.

лю p) первой половины каждого из CRT-представлений x и y. Аналогичным образом вычисляется и второй элемент.

Точно так же можно выполнять и умножение. CRT-представлением числа xy будет пара $(xy \bmod p, xy \bmod q)$, элементы которой легко подсчитать, зная CRT-представления x и y. Первый элемент $(xy \bmod p)$ вычисляется путем перемножения значений $(x \bmod p)$ и $(y \bmod p)$ и последующего взятия результата по модулю p. Аналогичным образом вычисляется и второй элемент, только все операции выполняются по модулю q.

Пусть k — количество бит числа n. Каждое из простых чисел p и q имеет длину примерно $k/2$ бит. Выполнение одной операции сложения по модулю n требует одного k-битового сложения, за которым может последовать одно k-битовое вычитание, если полученная сумма окажется больше n. При использовании CRT-представления, в свою очередь, требуется проделать две операции сложения по модулю над числами, длина которых примерно в два раза меньше. Это практически тот же объем работы, что и в первом случае.

Значительный выигрыш от использования CRT-представлений можно получить при выполнении умножения. Умножение двух k-битовых чисел требует намного больше работы, чем два умножения двух $k/2$-битовых чисел. В большинстве реализаций умножение с применением CRT-представлений будет выполняться в два раза быстрее, чем полное умножение. Это весьма ощутимая экономия времени.

Еще более существенный результат может быть достигнут при возведении в степень. Предположим, необходимо вычислить $x^s \bmod n$. Показатель степени s может быть до k бит длиной. Непосредственное возведение в степень требует выполнения около $3k/2$ умножений по модулю n. При использовании CRT-представлений каждая операция умножения выполняется быстрее, но это еще не все. Мы хотим вычислить $(x^s \bmod p, x^s \bmod q)$. Работая по модулю p, мы можем сократить показатель s по модулю $(p-1)$. Аналогичная операция может быть выполнена и по модулю $(q-1)$. Таким образом, останется вычислить лишь $(x^{s \bmod (p-1)} \bmod p, x^{s \bmod (q-1)} \bmod q)$. Каждый из показателей степени будет иметь длину лишь $k/2$ бит, а потому каждое возведение в степень потребует только $3k/4$ операции умножения. Вместо выполнения $3k/2$ умножений по модулю n нам придется выполнить $2 \cdot 3k/4 = 3k/2$ умножений по модулю одного из простых чисел p и q. В стандартной реализации это позволит сократить объем работы в 3-4 раза.

Издержками использования CRT-представлений являются лишь дополнительная сложность программного обеспечения и необходимость выполнения преобразований. Если одно вычисление включает в себя более нескольких операций умножения, тогда затраты на выполнение преобразований окупятся сполна. В большинстве учебников по криптографии китайская теорема об остатках упоминается лишь как один из приемов реализации алгоритма

RSA. Однако, по нашему мнению, CRT-представление значительно облегчает понимание всей схемы RSA. Вот почему мы решили вначале познакомить вас с китайской теоремой об остатках. Вскоре мы воспользуемся ею, чтобы объяснить поведение RSA.

13.2.4 Заключение

Подведем итоги: число x по модулю n может быть представлено в виде пары $(x \bmod p, x \bmod q)$, где $n = pq$. Преобразовать число из одного представления в другое довольно просто. CRT-представление может пригодиться в случае, если вам необходимо выполнить целый ряд умножений по модулю составного числа, для которого известно его разложение на множители. (Мы не можем использовать CRT-представление для ускорения вычислений, если не знаем, как n раскладывается на простые множители.)

13.3 Умножение по модулю n

Прежде чем вникать в тонкости алгоритма RSA, необходимо выяснить, как числа по модулю n ведут себя при умножении. Последнее несколько отличается от описанного ранее умножения по модулю p.

Для любого простого числа p и для всех $0 < x < p$ справедливо равенство $x^{p-1} = 1(\mathrm{mod}p)$. Это не выполняется для умножения по модулю составного числа n. Чтобы применить алгоритм RSA, нужно найти такой показатель степени t, чтобы $x^t = 1 \bmod n$ для всех (а вернее, почти всех) x. Большинство учебников просто приводят правильный ответ, что никак не помогает понять, откуда он взялся. В действительности получить правильный ответ довольно легко с помощью все той же китайской теоремы об остатках.

Нам нужно найти такое t, чтобы практически для всех x выполнялось $x^t = 1(\mathrm{mod}n)$. Из последнего уравнения следует, что $x^t = 1(\mathrm{mod}p)$ и $x^t = 1(\bmod q)$. Поскольку и p и q — простые числа, это может выполняться только в том случае, если и $p-1$ и $q-1$ являются делителями t. Наименьшее t, которое обладает данным свойством, — это НОК$(p-1, q-1) = (p-1)(q-1)/$НОД$(p-1, q-1)$. В оставшейся части главы мы будем использовать обозначение $t =$ НОК$(p-1, q-1)$.

Обозначения p, q и n используются во всех книгах, хотя некоторые авторы предпочитают прописные буквы. Вместо t чаще всего применяется функция Эйлера (Euler) $\phi(n)$. Для составного числа n вида $n = pq$ значение функции Эйлера вычисляется как $\phi(n) = (p-1)(q-1)$, что кратно нашему t. Очевидно, что $x^{\phi(n)} = 1$ и что использование$\phi(n)$ вместо t также приводит к правильному ответу, однако точнее все-таки использовать t.

Обсуждая, каким должно быть значение t, мы пропустили один небольшой момент: $x^t \bmod p$ не может быть равно единице, если $x \bmod p = 0$. Поэтому равенство $x^t \bmod n = 1$ не может выполняться для *всех* значений x. К счастью, исключений из этого правила немного: q чисел, для которых $x \bmod p = 0$, и p чисел, для которых $x \bmod q = 0$. Всего таких чисел $p + q$, точнее, $p + q - 1$, поскольку 0 мы посчитали дважды. Это лишь небольшая доля всех значений, общее количество которых равно $n = pq$. Следует отметить, что на самом деле алгоритм RSA использует свойство несколько иного вида: $x^{t+1} = x(\mathrm{mod} n)$. Последнее выполняется даже для упомянутых выше исключений. Это также легко показать с помощью CRT-представлений. Если $x = 0(\mathrm{mod} p)$, тогда $x^{t+1} = 0 = x(\mathrm{mod} p)$. Аналогичное справедливо и для q. Таким образом, фундаментальное свойство $x^{t+1} = x(\mathrm{mod} n)$ сохранено и выполняется для всех элементов $\mathbb{Z}_n$.

13.4 Определение RSA

Теперь можно определить систему RSA. Вначале случайным образом выберем два разных простых числа p и q и вычислим $n = pq$. Простые числа p и q должны быть примерно одного размера, в результате чего число n будет в два раза длиннее.

Мы также используем два разных показателя степени, которые принято обозначать как e и d. Они должны удовлетворять требованию $ed = 1(\mathrm{mod} t)$, где, как и раньше, $t = \mathrm{НОК}(p - 1, q - 1)$. В качестве открытого показателя e мы выбираем малое нечетное значение, после чего используем функцию ExtendedGCD (см. раздел 11.3.5), чтобы вычислить d как обратное e по модулю t. Это гарантирует, что $ed = 1(\mathrm{mod} t)$.

Чтобы зашифровать сообщение m, отправитель генерирует шифрованный текст $c := m^e(\mathrm{mod} n)$. Чтобы расшифровать шифрованный текст c, получатель вычисляет $c^d(\mathrm{mod} n)$. Это эквивалентно значению $(m^e)^d = m^{ed} = m^{kt+1} = (m^t)^k \cdot m = (1)^k \cdot m = m(\mathrm{mod} n)$, где k — некое целое число, которое всегда существует. Таким образом, получатель может расшифровать шифрованный текст m^e, чтобы получить открытый текст m.

Пара (n, e) образует открытый ключ. Последний обычно распространяется среди многих участников общения. Значения (p, q, t, d) образуют *закрытый ключ (private key)* и сохраняются в секрете человеком, который сгенерировал ключ RSA.

Для удобства вместо $c^d \bmod n$ часто пишут $c^{1/e} \bmod n$. Следует помнить, что все показатели вычислений по модулю n взяты по модулю t, так как $x^t = 1(\mathrm{mod} n)$, а значит, кратные t в показателе на результат не влияют. Поскольку мы определили d как обратное e по модулю t, запись d как $1/e$ вполне

естественна. Запись $c^{1/e}$ зачастую оказывается легче для восприятия, особенно при использовании сразу большого количества ключей RSA. Вот почему мы также говорим о взятии корня степени e из числа c. Просто запомните, что вычисления любых корней по модулю n требуют знания закрытого ключа.

13.4.1 Создание цифровой подписи с помощью RSA

До сих пор речь шла только о шифровании сообщений с помощью RSA. Между тем одним из больших преимуществ данного алгоритма является возможность его применения как для шифрования, так и для создания цифровых подписей. Эти операции используют одни и те же вычисления. Чтобы подписать сообщение m, владелец закрытого ключа вычисляет $s := m^{1/e} \bmod n$. Пара (m, s) образует подписанное сообщение. Чтобы удостовериться в подлинности подписи, каждый, кто знает открытый ключ, может проверить, действительно ли $s^e = m (\mathrm{mod} n)$.

Как и для шифрования, безопасность цифровой подписи основывается на том, что корень степени e из числа m может быть вычислен только тем человеком, который знает закрытый ключ.

13.4.2 Открытые показатели степеней

Описанная выше процедура имеет один недостаток. Если e имеет общий делитель с числом $t = \mathrm{НОК}(p-1, q-1)$, тогда нужного d не существует. Поэтому параметры p, q и e следует выбирать таким образом, чтобы не допустить подобной ситуации. Это скорее нюанс, нежели проблема, однако упускать его из виду тоже нельзя.

Использование открытого показателя малой длины значительно повышает эффективность RSA, поскольку для возведения числа в степень e потребуется меньше вычислений. Вот почему мы будем стараться выбрать в качестве e малое значение. Здесь мы устанавливаем фиксированное значение e, а затем подбираем p и q, чтобы они удовлетворяли упомянутым условиям.

Необходимо тщательно следить за тем, чтобы функции шифрования и цифровой подписи не взаимодействовали между собой каким-либо нежелательным образом. Нельзя допустить, чтобы злоумышленник мог расшифровать сообщение c, убедив владельца частного ключа подписать это сообщение. В конце концов, подписывание "сообщения" c — это то же самое, что и расшифровка шифрованного текста c. Функции шифрования, которые рассматриваются немного позднее, помогают не допустить подобной ситуации, однако использовать одну и ту же операцию RSA в качестве обеих функций все же не стоит. Мы могли бы применять для аутентификации и шифрова-

ния разные ключи RSA, однако это бы повысило сложность и удвоило объем данных ключа.

В качестве возможного решения можно порекомендовать использование одного и того же n с двумя разными показателями степеней. Мы будем использовать $e = 3$ для цифровых подписей и $e = 5$ для шифрования. Это разъединит системы, потому что кубические корни и корни пятой степени по модулю n не зависят друг от друга. Знание одного из этих корней не поможет злоумышленнику вычислить другой [28].

Выбор фиксированных значений для e упрощает систему и позволяет оценить ее производительность. С другой стороны, он накладывает ограничения на используемые простые числа, поскольку $p-1$ и $q-1$ не могут быть кратными 3 или 5. Впрочем, наличие этого свойства легко проверить еще при генерации p и q.

Аргументы в пользу значений 3 и 5 очень просты. Они являются наименьшими подходящими значениями[2]. Меньший показатель степени будет применяться для подписывания, потому что цифровые подписи зачастую проверяются по многу раз, в то время как любой фрагмент данных шифруется только единожды. Поэтому имеет смысл сделать процедуру верификации цифровых подписей более эффективной.

Другими распространенными значениями, которые принято использовать в качестве e, являются 17 и 65 537. Мы предпочитаем меньшие значения, поскольку они повышают эффективность системы. Разумеется, применение малых открытых показателей может быть связано с некоторыми второстепенными проблемами, однако функции кодирования, которые обсуждаются немного позднее, позволяют этого избежать.

Было бы неплохо использовать малое значение и в качестве d, но здесь мы вынуждены вас разочаровать. Хотя найти пару (e, d) с малым значением d в принципе возможно, использование малого d небезопасно [94]. Поэтому не стоит поддаваться соблазну, пытаясь найти удобное значение d.

13.4.3 Закрытый ключ

Злоумышленнику крайне сложно найти параметр закрытого ключа p, q, t или d, если он знает только открытый ключ (n, e). При достаточно большом значении n не существует известного алгоритма, который мог бы определить закрытый ключ за приемлемое время. Наилучшее решение, которое нам известно, — это разложить n на множители p и q и затем попытаться на их основе вычислить d. Вот почему разложение на множители играет такую важную роль в криптографии.

[2]Теоретически мы могли бы использовать $e = 2$, однако это привнесло бы в систему целый ряд дополнительных сложностей.

Речь идет о закрытом ключе, состоящем из параметров p, q, t и d. Оказывается, достаточно знать любое из этих значений, чтобы вычислить три остальных. Доказательство этого факта крайне поучительно.

Мы предполагаем, что злоумышленник знает открытый ключ (n, e), поскольку эта информация обычно широко известна. Если злоумышленник знает p или q, вычислить остальные значения не составляет труда. Зная p, он может найти $q = n/p$, а затем вычислить t и d точно так же, как это делали мы.

Что, если злоумышленнику известны значения (n, e, t)? Во-первых, $t = (p-1)(q-1)/\text{НОД}(p-1, q-1)$, но, поскольку произведение $(p-1)(q-1)$ очень близко к n, найти $\text{НОД}(p-1, q-1)$ нетрудно — это будет ближайшее к n/t целое число. (Значение $\text{НОД}(p-1, q-1)$ никогда не бывает слишком большим, так как маловероятно, что два случайных числа имеют один и тот же большой делитель.) Это позволяет злоумышленнику вычислить $(p-1)(q-1)$. Он также может вычислить $n-(p-1)(q-1)+1 = pq-(pq-p-q+1)+1 = p+q$. Теперь у злоумышленника есть $n = pq$ и $s := p+q$. Исходя из этого, он может вывести следующее:

$$\begin{aligned} s &= p + q; \\ s &= p + n/p; \\ ps &= p^2 + n; \\ 0 &= p^2 - ps + n. \end{aligned}$$

Последнее выражение — не что иное, как квадратное уравнение относительно p, которое может решить любой школьник. Разумеется, зная p, злоумышленник сможет вычислить и остальные параметры закрытого ключа.

Нечто аналогичное происходит и тогда, когда злоумышленник знает d. Во всех наших системах значение e будет очень малым. Поскольку $d < t$, значение $ed-1$ будет лишь в несколько раз больше t. Злоумышленник может просто подобрать соответствующий множитель, вычислить t и затем попытаться найти p и q описанным способом. В случае неудачи он просто попытается перебрать остальные варианты. (Существуют и более быстрые приемы, однако этот наиболее легок для понимания.)

Как видите, зная одно из значений — p, q, t или d, злоумышленник может вычислить остальные значения. Поэтому мы можем предположить, что владелец закрытого ключа знает все четыре значения. Программной реализации RSA достаточно хранить в памяти лишь одно из этих значений. Впрочем, большинство реализаций хранят несколько значений, которые нужны для выполнения операции дешифрования RSA. Выбор того или иного решения зависит от конкретной реализации и в контексте криптографии существенной роли не играет.

Если пользователь A хочет расшифровать или подписать сообщение, ему, очевидно, должно быть известно значение d. Поскольку это эквивалентно знанию p или q, можно с полной уверенностью предположить, что пользователь A знает множители n, а потому может проводить вычисления с помощью CRT-представлений. Это очень удобно, так как в алгоритме RSA возведение числа в степень d является наиболее ресурсоемкой операцией, а использование CRT-представления позволяет сократить объем работы в 3-4 раза.

13.4.4 Размер n

Число n должно быть такого же размера, как число p, которое применялось в алгоритме Диффи–Хеллмана. Более подробно это рассматривается в разделе 12.7. Напомним: чтобы обеспечить защиту данных на протяжении 20 лет, минимальный размер числа n должен составлять 2048 бит или около того. По мере увеличения мощности современных компьютеров этот минимум будет постепенно возрастать. Если ресурсы приложения позволяют, то используйте n длиной 4096 бит или наиболее близкой к этому. Более того, убедитесь, что ваше программное обеспечение поддерживает значения n длиной до 8192 бит. Никто не знает, чего можно ожидать в будущем, поэтому возможность перехода на использование ключей бóльших размеров без замены программного или аппаратного обеспечения может оказаться поистине спасительной.

Простые числа p и q должны быть одного и того же размера. Чтобы получить k-битовое число n, можно просто сгенерировать два случайных $k/2$-битовых простых числа и перемножить их. Иногда в результате такого умножения мы можем получить $(k-1)$-битовое число n, но это несущественно.

13.4.5 Генерация ключей RSA

Чтобы подытожить сказанное выше, рассмотрим две функции, которые генерируют ключи RSA, обладающие необходимыми свойствами. Первая из них представляет собой модификацию функции GenerateLargePrime, описанной в разделе 11.4. Единственное функциональное изменение — это появление условий $p \bmod 3 \neq 1$ и $p \bmod 5 \neq 1$, которые гарантируют, что мы можем использовать открытые показатели степеней 3 и 5. Разумеется, если в качестве e будут применяться другие значения, указанные условия необходимо подкорректировать.

функция GenerateRSAPrime

вход:	k	Размер требующегося простого числа в битах.
выход:	p	Случайное простое число из интервала $2^{k-1}, \ldots, 2^k - 1$, удовлетворяющее условию $p \bmod 3 \neq 1 \wedge p \bmod 5 \neq 1$.

Проверим, правильно ли задан интервал.
assert $1024 \leq k \leq 4096$
Подсчитаем максимальное количество попыток.
$r \leftarrow 100k$
repeat
  $r \leftarrow r - 1$
  assert $r > 0$
  Выберем в заданном интервале случайное число n.
  $n \in_R 2^{k-1}, \ldots, 2^k - 1$
  Продолжим попытки до тех пор, пока не найдем простое число.
until $n \bmod 3 \neq 1 \wedge n \bmod 5 \neq 1 \wedge$ isPrime(n)
return n

Вместо того чтобы указывать полный диапазон, в котором должно находиться искомое простое число, мы задаем лишь размер последнего. Назвать это определение слишком гибким нельзя, зато применять его для RSA гораздо проще и эффективнее. Дополнительные требования к простому числу фигурируют в качестве условия цикла. При более умелой реализации этого алгоритма можно позволить себе обойтись без вызова функции isPrime(n), если n дает нежелательный остаток по модулю 3 или 5, поскольку выполнение функции isPrime(n) включает в себя огромное количество вычислений.

Для чего же мы снова вставили в функцию счетчик цикла с условием ошибки? Неужели при наличии такого большого интервала мы не найдем подходящего простого числа? Хотелось бы верить, но иногда в этом мире происходят странные вещи. Нас беспокоит отнюдь не возможность получения интервала, в котором не окажется простых чисел, а испорченный генератор псевдослучайных чисел, который всегда будет возвращать одно и то же составное число. Это, к сожалению, один из самых распространенных сбоев генераторов случайных чисел. Выполнение простой проверки защитит функцию GenerateRSAPrime от неправильного поведения генераторов. Еще одной распространенной проблемой является сбой функции isPrime, в результате которого каждое сгенерированное число объявляется составным.

Приведенная ниже функция генерирует все параметры закрытого ключа.

функция GenerateRSAKey
вход: k Размер модуля в битах.
выход: p, q Множители модуля.
n Модуль длиной приблизительно k бит.
d_3 Показатель степени для цифрового подписывания.
d_5 Показатель степени для дешифрования.

Проверим, правильно ли задан интервал.

assert $2048 \leq k \leq 8192$

Сгенерируем простые числа.

$p \leftarrow$ GENERATERSAPRIME $(\lfloor k/2 \rfloor)$

$q \leftarrow$ GENERATERSAPRIME $(\lfloor k/2 \rfloor)$

Небольшая проверка на случай, если генератор испорчен...

assert $p \neq q$

Вычислим t как НОК$(p-1, q-1)$.

$t \leftarrow (p-1)(q-1)/\mathrm{GCD}(p-1, q-1)$

Вычислим секретные показатели степеней. Для этого найдем обратное число по модулю с помощью расширенного алгоритма Евклида.

$g, (u, v) \leftarrow$ EXTENDEDGCD$(3, t)$

Проверим правильность НОД. В противном случае обратного числа не существует.

assert $g = 1$

Возъмем u по модулю t, потому что u может быть отрицательным, а d_3 должно быть положительным.

$d_3 \leftarrow u \bmod t$

Повторим описанные выше действия для d_5.

$g, (u, v) \leftarrow$ EXTENDEDGCD$(5, t)$

assert $g = 1$

$d_5 \leftarrow u \bmod t$

return p,q, pq, d_3, d_5

Обратите внимание, что в качестве открытых показателей степеней применяются фиксированные значения и что мы сгенерировали ключ, который может применяться как для подписывания ($e = 3$), так и для шифрования ($e = 5$).

13.5 “Подводные камни” использования RSA

Использовать алгоритм RSA описанным выше способом очень опасно. Проблема заключается в том, что RSA имеет четкую математическую структуру. Например, если пользователь А подпишет цифровой подписью два сообщения — m_1 и m_2, пользователь Б сможет вычислить, какой должна быть подпись пользователя А для сообщения $m_3 := m_1 m_2 \bmod n$. Действительно, подписывая сообщения, пользователь А вычислил значения $m_1^{1/e}$ и $m_2^{1/e}$, а пользователь Б может перемножить эти значения, чтобы получить $(m_1 m_2)^{1/e}$.

Еще одна проблема возникает тогда, когда пользователь Б зашифровывает с помощью открытого ключа пользователя А сообщение небольшого размера. Если $e = 5$ и $m < \sqrt[5]{n}$, тогда $m^e = m^5 < n$, поэтому взятия числа по

модулю не требуется. Злоумышленник Е сможет восстановить m, просто извлекая корень пятой степени из m^5. Это будет очень легко, так как операции взятия по модулю в данном случае не задействованы. Подобные ситуации часто возникают тогда, когда пользователь Б пытается отослать пользователю А ключ AES. Если рассматривать 256-битовое значение ключа как целое число, тогда значение зашифрованного ключа будет меньше $2^{256 \cdot 5} = 2^{1280}$, т.е. намного меньше нашего n. К такому числу не будут применяться операции взятия по модулю, поэтому злоумышленник Е сможет вычислить ключ, просто извлекая корень пятой степени из значения зашифрованного ключа.

Одна из причин такого подробного объяснения теории, лежащей в основе RSA, состоит в том, что мы хотим немного познакомить вас с обнаруженной нами математической структурой. Такая структура допускает осуществление сразу нескольких типов атак. Наиболее простые из них уже упоминались выше. Но существуют и более изощренные атаки, основанные на методах решения полиномиальных уравнений по модулю n. Все они сводятся к одному: подчинение чисел, которыми оперирует алгоритм RSA, *какой бы то ни было* структуре крайне нежелательно.

Для решения этой проблемы используют функцию, которая разрушает любую имеющуюся структуру. Иногда ее называют функцией дополнения, но это неправильно. Термином *дополнение (padding)* обычно обозначают добавление лишних битов, которое применяют, чтобы получить строку нужной длины. Различные виды дополнения часто использовались для шифрования и цифрового подписывания RSA, и в большинстве случаев это приводило к осуществлению атак на такие системы. Нам же нужна функция, которая уничтожает всякую имеющуюся структуру настолько, насколько это возможно. Мы называем такую функцию *функцией кодирования (encoding function)*.

Наиболее примечательным из всех стандартов функций RSA является, пожалуй, PKCS #1 v.2.1 [84]. Как обычно, это не единственный стандарт. Существует две схемы шифрования и две схемы цифровых подписей RSA, причем каждая из этих схем может работать с целым рядом функций хэширования. Нельзя сказать, что это плохо, но нам не нравится дополнительная сложность. Поэтому представим вам несколько более простых методов, хотя они могут и не обладать всеми свойствами методов PKCS.

Стандарт PKCS #1 v.2.1 также демонстрирует распространенную проблему технической документации: он смешивает спецификацию с реализацией. Функция дешифрования RSA описывается в нем дважды: один раз с помощью уравнения $m = c^d \bmod n$, а второй раз с помощью CRT-представлений. И то и другое выражение дает одинаковый результат: одно из них является лишь оптимизированной реализацией второго. Подобные описания реализации не должны быть частью стандарта, потому что они не задают разное поведение. Каждый из них должен рассматриваться отдельно. Впрочем, мы

не собираемся критиковать именно этот стандарт PKCS. Данная проблема весьма широко распространена во всех сферах компьютерной индустрии.

13.6 Шифрование

Шифрование сообщений является канонической областью применения алгоритма RSA. Тем не менее оно редко используется на практике. Причина проста: размер сообщения, которое может быть зашифровано с помощью RSA, ограничивается размером числа n. В реальных системах мы даже не можем использовать все биты сообщения, так как функции кодирования требуются дополнительные биты. Подобное ограничение на размер сообщения для большинства систем крайне непрактично, а поскольку в терминах вычислений каждая операция RSA обходится довольно дорого, вряд ли кому-нибудь захочется разбивать сообщение на блоки меньшего размера и шифровать каждый из них с помощью отдельной операции RSA.

В большинстве случаев эту проблему решают следующим образом: выбирают случайный секретный ключ K и зашифровывают его с помощью ключей RSA. После этого фактическое сообщение m зашифровывается с помощью обыкновенного блочного или поточного шифра, где в качестве ключа шифрования используется K. Таким образом, вместо того чтобы отсылать нечто наподобие $E_{\mathrm{RSA}}(m)$, мы отсылаем $E_{\mathrm{RSA}}(K)$ и $E_K(m)$. Размер сообщения больше не ограничен, а для шифрования даже больших сообщений требуется только одна операция RSA. Конечно, нам приходится передавать немного дополнительной информации, но затраты на это несоизмеримо малы по сравнению с преимуществами данного подхода.

Наш метод шифрования еще проще. Вместо того чтобы выбирать и зашифровывать ключ K, мы выбираем случайное $r \in \mathbb{Z}_n$ и определяем ключ группового шифрования как $K := h(r)$ для некоторой функции хэширования h. Шифрование числа r выполняется путем простого возведения r в пятую степень по модулю n. (Напомним, что для шифрования применяется $e = 5$.) Это решение простое и безопасное. Поскольку значение r выбирается случайным образом, оно не подчиняется какой-либо структуре, которая могла бы использоваться для атаки на шифрование RSA. Функция хэширования, в свою очередь, гарантирует, что ни одна зависимость между различными значениями r не перейдет на значения K, за исключением очевидного требования, что одинаковым r должны соответствовать одинаковые K.

Для простоты реализации будем выбирать r в диапазоне $0, \ldots, 2^{k-1}$, где k — наибольшее число, для которого выполняется $2^k < n$. Гораздо легче сгенерировать случайное k-битовое число, чем случайное число из $\mathbb{Z}_n$, а по-

лученное небольшое отклонение от равномерного распределения в данном случае не принесет никакого вреда.

Приведем формализованное описание данного алгоритма.

функция EncryptRandomKeyWithRSA
вход: (n, e) Открытый ключ RSA, в нашем случае $e = 5$.
выход: K Секретный ключ, который был зашифрован.
c Шифрованный текст RSA.

Вычислим k.
$k \leftarrow \lfloor \log_2 n \rfloor$
Выберем случайное r, такое, что $0 \le r \le 2^k - 1$.
$r \in_{\mathcal{R}} \{0, \ldots, 2^k - 1\}$
$K \leftarrow \mathrm{SHA}_d - 256(r)$
$c \leftarrow r^e \bmod n$
return (K, c)

Получатель вычисляет $K = h(c^{1/e} \bmod n)$ и получает то же значение K.

функция DecryptRandomKeyWithRSA
вход: (n, d) Закрытый ключ RSA для $e = 5$.
c Шифрованный текст.
выход: K Секретный ключ, который был зашифрован.

assert $0 \le c \le n$
Остальное выполняется тривиально.
$K \leftarrow \mathrm{SHA}_d - 256(c^{1/e} \bmod n)$
return K

Итак, мы подробно рассмотрели, как вычислить $c^{1/e}$ по известному закрытому ключу, поэтому дальнейшее обсуждение не требуется. Напомним только, что использование CRT-представлений позволяет ускорить выполнение алгоритма в 3-4 раза.

Теперь продемонстрируем безопасность данного решения. Предположим, что пользователь Б зашифровывает ключ K и отсылает его пользователю А. Злоумышленник Е хочет получить больше информации об этом ключе. Сообщение пользователя Б зависит только от некоторых случайных данных и от открытого ключа пользователя А. Поэтому в самом худшем случае сообщение может раскрыть злоумышленнику Е данные о ключе K, но никак не о других секретах (например, о закрытом ключе пользователя А). Ключ K вычисляется с помощью функции хэширования, и мы можем рассматривать ее как случайное отображение. (Если рассматривать функцию хэширования как случайное отображение невозможно, значит, она не удовлетворяет сформулированным нами требованиям безопасности к функциям хэширования.)

Единственный способ получить информацию о результате функции хэширования — это узнать бóльшую часть ее входных данных. Для этого злоумышленнику нужна информация об r. Но если алгоритм RSA является безопасным (а мы должны исходить из предположения, что он безопасен, раз выбрали его для шифрования), тогда получить какой-либо значительный объем информации о случайно выбранном r, зная лишь $r^e \bmod n$, невозможно. Поэтому у злоумышленника остается большая неопределенность относительно r и, как следствие, никакой информации о K.

Предположим, через некоторое время злоумышленнику E стал известен ключ K (возможно, из-за сбоя какого-нибудь другого компонента системы). Приведет ли это к утечке информации о закрытом ключе пользователя A? Нет. Ключ K — это результат функции хэширования, а злоумышленник не может получить какую-либо информацию о входных данных функции хэширования на основе ее выходных данных. Следовательно, даже если злоумышленнику E удастся особым образом подобрать шифрованный текст c, полученный им ключ K не предоставит никакой информации об r. Закрытый ключ пользователя A применялся только для вычисления r, поэтому злоумышленнику E снова ничего не удастся узнать о закрытом ключе пользователя A.

Это одно из преимуществ применения функции хэширования в функции DecryptRandomKeyWithRSA. Предположим, что последняя возвращала бы лишь значение $c^{1/e} \bmod n$. В этом случае данная функция могла бы быть использована злоумышленником для осуществления всевозможных атак. Предположим, что другая часть системы имеет слабое место, в результате чего злоумышленник E может узнать наименее значимый бит выходных данных. В этом случае злоумышленник может отослать пользователю A специально подобранные значения $c_1, c_2, c_3, \ldots$ и получить наименее значимые биты значений $c_1^{1/e}, c_2^{1/e}, c_3^{1/e}, \ldots$. Полученные результаты будут обладать всеми видами алгебраических свойств. Вполне вероятно, что злоумышленник E сможет извлечь из подобной ситуации что-нибудь полезное. Функция хэширования h, примененная в DecryptRandomKeyWithRSA, полностью разрушает математическую структуру. Знание одного бита значения K не предоставит злоумышленнику E практически никакой информации о $c^{1/e}$. Более того, даже знание всего значения K не принесет слишком большой пользы, так как функция хэширования не является обратимой. Таким образом, применение функции хэширования делает алгоритм RSA более безопасным по отношению к сбоям остальных компонентов системы.

Описанное поведение, помимо всего прочего, является также причиной того, почему мы *не* проверяем, попадает ли значение r, вычисленное по заданному c, в интервал $0, \ldots, 2^k - 1$. Если бы мы реализовали подобную про-

верку, то должны были бы обрабатывать ошибки в случае их возникновения. Поскольку поведение системы при обработке ошибок всегда отличается от поведения в нормальной ситуации, вполне вероятно, что злоумышленник сможет обнаружить факт возникновения ошибки. В результате у него появится функция, которая раскрывает информацию о закрытом ключе: злоумышленник E может выбрать любое c и проверить, выполняется ли отношение $c^{1/e} \bmod n < 2^k$. Злоумышленник E не может проверить данное свойство без помощи пользователя A, но зачем же лишний раз помогать злоумышленнику, если этого можно избежать? Отказавшись от проверки условия, мы в худшем случае получим лишь бессмысленные выходные данные, а это может произойти и при наличии проверки, поскольку c может быть повреждено таким образом, что r все равно будет попадать в нужный диапазон[3].

Небольшое замечание: существует огромная разница между раскрытием информации о случайной паре $(c, c^{1/e})$ и вычислением $c^{1/e}$ для c, выбранного кем-то другим. Генерировать пары вида $(c, c^{1/e})$ может каждый. Для этого достаточно выбрать случайное r, вычислить пару (r^e, r) и затем обозначить $c := r^e$. Ничего секретного в таких парах нет. Но, если пользователь A будет так добр, что подсчитает значение $c^{1/e}$ для c, полученного от злоумышленника, последний сможет выбирать значения c некоторым специальным образом, что было невозможно для случайных пар $(c, c^{1/e})$, сгенерированных самим злоумышленником. Отсюда вывод: не следует лишний раз помогать злоумышленнику.

13.7 Подписи

Реализовать схему цифровых подписей несколько сложнее. Проблема состоит в том, что сообщение m, которое мы хотим подписать, имеет довольно четкую структуру. Допускать, чтобы эта структура перешла на числа, для которых подсчитываются корни RSA, нежелательно. Поэтому структуру сообщения необходимо уничтожить.

Первый шаг к уничтожению структуры состоит в хэшировании сообщения. Таким образом, вместо сообщения произвольной длины m будем работать со значением фиксированной длины $h(m)$, где h — функция хэширования. Если мы используем функцию SHA$_d$-256, то получим 256-битовый результат. Но значение n намного больше, поэтому мы не можем применять $h(m)$ непосредственно.

[3] Введение каких бы то ни было ограничений на r не решает проблему бессмысленных выходных данных. Злоумышленник E всегда может использовать открытый ключ пользователя A и измененную функцию EncryptRandomKeyWithRSA, чтобы отсылать пользователю A бессмысленные ключи в зашифрованном виде.

Самое простое решение — использовать псевдослучайное отображение, которое будет "растягивать" $h(m)$, отображая его на случайное число s из диапазона $0, \ldots, n-1$. Затем подпись сообщения m будет вычисляться как $s^{1/e} (\mathrm{mod} n)$. Отображение $h(m)$ на число по модулю n выполнить довольно сложно (см. раздел 10.8). В нашем случае можно упростить проблему, отображая $h(m)$ на случайный элемент из диапазона $0, \ldots, 2^k - 1$, где k — наибольшее число, такое, что $2^k < n$. Числа из диапазона $0, \ldots, 2^k - 1$ генерировать гораздо проще, поскольку для этого достаточно сгенерировать лишь k случайных бит. В нашем конкретном случае подобное решение можно использовать без опаски, однако не применяйте его повсеместно. Существует множество ситуаций, в которых подобное упрощение разрушит безопасность всей системы.

Мы будем использовать генератор псевдослучайных чисел Fortuna, описанный в главе 10, "Генерация случайных чисел". Многие разработчики используют функцию хэширования h, чтобы построить небольшой генератор случайных чисел, предназначенный специально для этой цели, но у нас уже есть хороший генератор. Кроме того, нам все равно понадобится генератор псевдослучайных чисел, чтобы выбирать простые числа для генерации ключей RSA.

Для реализации схемы цифровых подписей понадобятся три функции: одна, чтобы отобразить сообщение m на s, вторая, чтобы подписать сообщение, и третья, чтобы проверить подпись.

функция MsgToRSANumber

вход: n Открытый ключ RSA, модуль, по которому выполняются вычисления.

m Сообщение, которое должно быть преобразовано в число по модулю n.

выход: s Число по модулю n.

Создадим новый генератор псевдослучайных чисел.

$\mathcal{G} \leftarrow$ InitializeGenerator()

Обновим начальное число генератора с помощью хэш-кода сообщения.

ReSeed($\mathcal{G}, \mathrm{SHA}_d - 256(m)$)

Вычислим k.

$k \leftarrow \lfloor \log_2 n \rfloor$

$x \leftarrow$ PseudoRandomData($\mathcal{G}, \lceil k/8 \rceil$)

Как обычно, мы рассматриваем строку байтов x как целое число, представленное в формате, в котором наименее значимый байт записывается первым. Операция взятия по модулю будет выполняться путем простого применения операции AND к последнему байту x.

$s \leftarrow x \bmod 2^k$

return s

функция SignWithRSA
вход: (n, d) Закрытый ключ RSA для $e = 3$.
m Сообщение, которое должно быть подписано.
выход: σ Подпись сообщения m.

$s \leftarrow$ MsgToRSANumber(n, m)
$\sigma \leftarrow s^{1/e} \bmod n$
return σ

Буква σ (сигма) часто применяется для обозначения цифровой подписи, поскольку это греческий эквивалент английской буквы s (signature — подпись). Как вычислить $s^{1/e} \bmod n$ по заданному закрытому ключу, было описано ранее.

функция VerifyRSASignature
вход: (n, e) Открытый ключ RSA для $e = 3$.
m Сообщение, которое было подписано.
σ Подпись сообщения m.

$s \leftarrow$ MsgToRSANumber(n, m)
assert $s = \sigma^e \bmod n$

Разумеется, в реальных приложениях необходимо предпринимать определенные меры, если проверка подписи покажет, что та не является подлинной. В этом примере мы просто вставили утверждение `assert`, чтобы указать на то, что выполнение каких-либо нормальных действий должно быть прервано. Ошибка подписи должна обрабатываться так же, как и всякая другая ошибка в криптографических протоколах: это верный признак того, что система находится под нападением. Не посылайте никаких ответов (разве что без этого никак не обойтись) и уничтожьте все данные, с которыми работаете. Чем больше информации будет отослано, тем больше сведений появится у злоумышленника.

Аргументы в пользу безопасности цифровых подписей RSA аналогичны тем, что были высказаны относительно шифрования RSA. Если мы попросим пользователя A подписать ряд сообщений $m_1, m_2, \ldots, m_i$, то получим пары вида $(s, s^{1/e})$, однако значения s в них будут случайными. Если функция хэширования безопасна, мы можем лишь подобрать $h(m)$ методом проб и ошибок. Генератор случайных чисел — это тоже случайное отображение. Каждый может создавать пары $(s, s^{1/e})$ для случайных значений s, поэтому даже наличие такой информации не поможет злоумышленнику фальсифицировать подпись. С другой стороны, для любого конкретного сообщения m вычислить соответствующую пару $(s, s^{1/e})$ может только тот, кто знает закрытый ключ, так как на основе $h(m)$ вычисляется s, а затем $s^{1/e}$. Последняя

операция требует знания закрытого ключа. Таким образом, каждый, кто проверит подпись и обнаружит, что она подлинна, будет знать, что ее должен был сгенерировать именно пользователь A.

На этом обсуждение алгоритма RSA, а также знакомство с математическими понятиями и теоремами завершается. Впоследствии мы будем использовать алгоритмы RSA и Диффи–Хеллмана для реализации протокола согласования ключей и инфраструктуры открытого ключа (PKI), но нам понадобятся лишь те понятия и формулы, которые уже рассматривались в этой книге. Никаких новых математических выкладок больше не будет.

Глава 14

Введение в криптографические протоколы

Криптографические протоколы описывают обмен сообщениями между двумя или несколькими участниками. Мы уже встречались с простым криптографическим протоколом в главе 12, “Алгоритм Диффи–Хеллмана”.

Протоколы — это, пожалуй, наиболее сложная часть криптографии. Основная проблема реализации протоколов состоит в том, что разработчики протоколов не могут контролировать их применение. До сих пор мы разрабатывали систему и имели полную власть над поведением ее компонентов. Теперь же, взаимодействуя с другими участниками, мы не сможем контролировать их поведение. Наш собеседник имеет другой набор интересов и вполне может отклониться от правил, чтобы получить для себя бóльшую выгоду. Поэтому, разрабатывая протоколы, приходится исходить из предположения, что мы имеем дело с врагом.

14.1 Роли

Обычно протоколы описывают взаимодействие между пользователем А и пользователем Б или, скажем, между покупателем и продавцом. При этом названия наподобие “пользователь А”, “пользователь Б”, “покупатель” или “продавец” не обозначают конкретного человека или организацию. Они определяют роль этого человека в протоколе. Если мистер Смит захочет пообщаться с мистером Джонсом, он может запустить протокол согласования ключей. При этом мистер Смит может исполнять роль пользователя А, а мистер Джонс — пользователя Б, затем они могут поменяться ролями. Следует

помнить, что одна и та же сущность может исполнять любую роль[1]. Это особенно важно при проведении анализа безопасности протокола. Мы уже рассматривали возможность осуществления атаки посредника на протокол DH. В атаке такого типа злоумышленник Е исполняет роль как пользователя А, так и пользователя Б. (Разумеется, "злоумышленник Е" — это всего лишь еще одна роль.)

14.2 Доверие

Доверие — это фундаментальная основа, на которой строятся наши взаимоотношения с другими людьми. Если мы никому ни в чем не доверяем, зачем тогда вообще иметь с ними дело? Например, чтобы купить шоколадный батончик, нам нужен некий базовый уровень доверия. Покупатель должен верить, что продавец даст ему шоколадку и правильно посчитает сдачу. Продавец, в свою очередь, должен верить, что покупатель заплатит за шоколадку. Если одна из заинтересованных сторон поведет себя не так, как положено, вторая предпримет соответствующие меры. Воров берут под стражу. Нечестные продавцы рискуют заработать плохую репутацию, получить повестку в суд или же остаться с синяком под глазом.

Существует несколько источников доверия.

- **Этика.** Влияние этики на наше общество огромно. Хотя очень немногие (если такие люди вообще есть) ведут себя этично абсолютно во всех ситуациях, поведение большинства из нас в основном подчиняется законам морали. Согласитесь, что злоумышленников не так уж много. Все-таки большинство людей оплачивают свои покупки, даже если их можно легко украсть.
- **Репутация.** В нашем обществе очень важно иметь "доброе имя". Поэтому все его представители — от отдельно взятого человека до крупной компании — ревностно защищают свою репутацию. Зачастую угроза широкой огласки каких-то неблаговидных поступков стимулирует их вести себя должным образом.
- **Закон.** Цивилизованные страны обладают хорошо развитой правовой инфраструктурой, которая поддерживает возбуждение судебных исков и наказание людей, ведущих себя неправильно. Это вынуждает нас вести себя в соответствии с законом.
- **Угроза физической расправы.** Еще одним стимулом правильного поведения является страх перед физической расправой, которой может

[1] В протоколах с тремя или более участниками один и тот же человек может исполнять несколько ролей одновременно.

подвергнуться обманщик, пойманный своими партнерами. Это один из основных источников доверия для наркоторговцев и других людей, занимающихся незаконной торговлей. Подобная угроза может выражаться в физическом насилии или других действиях.

- **Взаимно-гарантированное уничтожение (mutually assured destruction — MAD).** Это термин времен "холодной войны". В более мягких формах он означает угрозу нанести вред и себе, и второй стороне. Если вы обманете своего лучшего друга, он может разорвать с вами отношения, что причинит боль вам обоим. Довольно часто в ситуации взаимно-гарантированного уничтожения оказываются две компании, которые возбуждают встречные иски о нарушении патентов.

Все перечисленные источники являются механизмами, стимулирующими участников общения не обманывать друг друга. Каждая сторона знает о существовании этих стимулов и потому может доверять противоположной стороне до определенной степени. Вот почему все эти стимулы не срабатывают, когда мы имеем дело с иррациональными людьми, чье поведение полностью нелогично: они далеко не всегда действуют в собственных интересах, что нарушает работу всех механизмов доверия.

Довольно трудно вызвать доверие у противоположной стороны, общаясь по Internet. Предположим, пользователь A живет за пределами США и подключается к Web-узлу ACME, у которого практически нет оснований доверять пользователю A; из всех механизмов доверия остается только этика. Предпринять какие-либо законные меры в отношении частных лиц, живущих за рубежом, практически невозможно и к тому же непозволительно дорого. Мы не можем навредить их репутации, угрожать им или даже поставить в ситуацию взаимно-гарантированного уничтожения.

Несмотря на это, у пользователя A и компании ACME все еще имеется некая основа для установления доверительных отношений. Она состоит в следующем: ACME обладает определенной репутацией, которую необходимо защищать. Это очень важно помнить, разрабатывая протоколы для электронной коммерции. В случае какого-либо сбоя или ошибки (а таковых не избежать) последние должны быть в пользу ACME, потому что у нее есть стимул корректно разрешить проблему путем ручного вмешательства[2]. Если же ошибка окажется в пользу покупателя, ситуация вряд ли разрешится должным образом. Более того, компания окажется уязвимой для злоумышленников, которые попытаются инициировать сбой системы и выиграть от этого.

[2]Практически все компании, занимающиеся распространением товаров по телефону, почте или Internet, следуют этому правилу, требуя от покупателя, чтобы товар был оплачен прежде, чем будет доставлен к месту назначения.

Понятие доверия нельзя рассматривать в черно-белых тонах: либо доверять, либо не доверять. Мы доверяем разным людям в той или иной степени. Мы можем доверить хорошему знакомому на хранение 100 долларов, но никак не лотерейный билет, который только что сорвал джек-пот в пять миллионов. Мы доверяем банку свои деньги, но сохраняем квитанции и копии аннулированных чеков, так как не полностью доверяем администрации этого банка. Вопрос: "Вы доверяете ему?" не закончен. Он должен звучать так: "Вы доверяете ему в вопросе X?".

14.2.1 Риск

Доверие является фундаментальной основой любого бизнеса, однако обычно оно выражается в форме риска. Риск можно рассматривать как противоположность доверию. Всевозможные виды рисков подвергают оцениванию, сравнению и страхованию.

Работая с криптографическими протоколами, гораздо легче думать в терминах доверия, а не рисков. Тем не менее нехватка доверия — это не что иное, как риск, а последний иногда можно обрабатывать с помощью стандартных методов управления рисками, таких, как страхование. Разрабатывая протоколы, мы говорим о доверии. Не забывайте, однако, что бизнесмены думают и говорят в терминах рисков. Чтобы общаться с этими людьми, вам придется перестраиваться с одной точки зрения на другую.

14.3 Стимул

Еще одним фундаментальным компонентом анализа криптографических протоколов является система стимулов. Каковы цели различных участников общения? Чего они хотят достигнуть? Даже в реальной жизни анализ системы стимулов позволяет сделать глубокомысленные заключения.

Каждую неделю в прессе то и дело появляются громкие заявления наподобие: "Последние исследования показали, что...". Читая подобные статьи, мы сразу же задаемся вопросом: а кто оплатил эти исследования? Исследования, результаты которых свидетельствуют в пользу лица, оплатившего их, всегда подозрительны. Здесь действует несколько факторов. Во-первых, исследователи знают, что хочет услышать их заказчик, и понимают, что могут получить повторный контракт, если предоставят "хорошие" результаты, а потому автоматически теряют объективность. Во-вторых, заказчик исследований не собирается публиковать отрицательные отчеты, а публикация только положительных результатов приводит к еще большей потере объективности. Табачные компании не раз публиковали "научные" статьи о том, что никотин

не вызывает привыкания. Компания Microsoft оплачивает исследования, которые "доказывают", что открытое программное обеспечение плохое. Никогда не доверяйте исследованиям, которые свидетельствуют в пользу компании, оплатившей их.

Авторы этой книги хорошо знакомы с подобным давлением. За долгое время работы в качестве консультантов мы многократно выполняли оценку безопасности систем для коммерческих заказчиков. Мы были безжалостны — большинство предоставленных нам продуктов оказывались довольно плохи — и редко давали положительные отзывы. Это отнюдь не добавляло нам популярности среди заказчиков. Один из них даже позвонил Брюсу и сообщил: "Остановите работу и вышлите мне счет. Я нашел человека, который пишет хорошие отчеты за меньшую плату". Догадываетесь, что в данном случае скрывалось за словом "хорошие"? Единственная причина, по которой мы могли позволить себе оставаться объективными, была в том, что у нас всегда хватало работы. Если заказов мало, а исследователю нужно кормить семью, трудно устоять перед искушением наперекор собственным амбициям говорить то, что хотят от него услышать.

Аналогичная проблема наблюдается и в других сферах деловой жизни. В наши дни газеты переполнены рассказами о банковском и бухгалтерском деле. Аналитики и аудиторы вместо отчетов, содержащих объективные оценки, слагают хвалебные оды своим клиентам. Мы обвиняем систему стимулов, которая побудила этих людей писать необъективные отчеты. Анализировать стимулы весьма поучительно. Мы занимались подобным анализом на протяжении многих лет. Имея в активе хоть немного практики, провести такой анализ совсем несложно, а его результаты поистине впечатляют. К сожалению, он заставляет подходить к оценке человеческих мотивов более цинично.

Заплатив руководящему составу компании в фондовых опционах, вы предоставите им следующую систему стимулов: увеличьте цены на акции на следующие три года, и вы сорвете большой куш; уменьшите цены, и вам покажут пальцем на дверь с утешительным призом в качестве солидного выходного пособия. Этот стимул напоминает уговор: "Если выпадет орел, я выиграю много, а если решка, я выиграю немного, но все равно выиграю"; поэтому догадайтесь, как поступает большинство менеджеров? Они выбирают краткосрочную стратегию с высокой степенью риска. Если у менеджеров появится возможность удвоить поставленную на кон сумму, они обязательно воспользуются этой возможностью, поскольку всегда получают выигрыш и никогда не платят в случае проигрыша. Если они смогут на несколько лет взвинтить цены на акции с помощью бухгалтерских штучек, то обязательно это сделают, потому что всегда смогут собрать сливки и испариться, прежде чем их изобличат. Некоторым игрокам не везет, но платят по счетам все равно другие.

Аналогичная ситуация наблюдалась в 80-х годах прошлого века на рынке ссуд и сбережений США. Федеральное правительство либерализовало правила, предоставив сберегательным банкам больше свободы в инвестировании своих средств. В это же время правительство гарантировало вкладчикам возврат всех денег по депозитам. Теперь давайте взглянем на систему стимулов. Если инвестиции окажутся удачными, банк получит прибыль, а его руководство — солидную премию и прибавку к зарплате. Если же банк потеряет деньги, федеральное правительство вернет вкладчикам их сбережения. Не удивительно, что многие сберегательные банки потеряли огромную массу средств на чрезвычайно рискованных вложениях — все равно по счетам заплатило федеральное правительство.

Подкорректировать систему стимулов не так уж сложно. Например, оплату аудита бухгалтерских книг можно поручить не самой компании, а фондовой бирже. Назначьте аудиторам солидную премию за каждую найденную ошибку, и вы получите гораздо более точный отчет.

Примеры нежелательных стимулов окружают нас повсюду. Адвокатам, занимающимся разводом, выгодно как можно более обострить течение процесса, так как им платят за каждый час, потраченный на отвоевывание имущества для своего клиента. Готовы поспорить, что они посоветуют вам заключить мировую, как только расходы на их содержание превысят стоимость самого имущества.

В Америке судебные иски по любому поводу давно стали привычной частью жизни. Каждый участник какого-нибудь неприятного происшествия имеет огромный стимул скрыться, отрицать свою причастность или каким-либо другим образом снять с себя вину. Строгие законы об ответственности и огромные компенсации за причиненный ущерб, может быть, и хороши для общества, но существенно препятствуют определению того, почему произошел этот случай и как избежать его в будущем. Законы об ответственности, которые будто бы предназначены для защиты потребителей, никогда не позволят компаниям наподобие Firestone признаться, что в их продуктах обнаружены изъяны, чтобы все узнали, как делать более качественные шины.

Криптографические протоколы взаимодействуют с системами стимулов двумя способами. Во-первых, их работа основана на системах стимулов. Некоторые протоколы электронных платежей не мешают продавцам обманывать покупателей, однако предоставляют покупателям доказательства того, что их обманывают. Этот подход приносит плоды, поскольку у продавца есть стимул не обманывать: он не хочет, чтобы у покупателя появилось доказательство его вины. Последнее может быть использовано в суде или же просто для того, чтобы навредить репутации продавца.

Во-вторых, криптографические протоколы изменяют систему стимулов. Благодаря протоколам некоторые вещи становятся невозможными, в резуль-

тате чего у злоумышленников исчезают соответствующие стимулы. С другой стороны, криптографические протоколы могут открыть новые возможности и привести к появлению новых стимулов. Реализуя систему электронных платежей через Internet, вы создаете стимул для вора взломать ваш компьютер и украсть деньги.

На первый взгляд кажется, что большинство стимулов носят материальный характер, однако это лишь часть правды. Многие поступают тем или иным образом отнюдь не из материальных побуждений. Большинство компьютерных хакеров взламывают системы не с целью получить прибыль, а исключительно ради забавы, повышения своего авторитета или из чистого бахвальства. В личных отношениях большинство стимулов не имеют практически никакого отношения к деньгам. Включите воображение и попытайтесь понять, что движет людьми. Затем используйте полученные выводы для создания криптографических протоколов.

14.4 Доверие в криптографических протоколах

Назначение криптографических протоколов состоит в минимизации объема доверия, необходимого для взаимодействия участников. Повторим еще раз: *назначение криптографических протоколов состоит в минимизации объема доверия, необходимого для взаимодействия участников.* Это означает минимизацию не только количества людей, которые должны доверять друг другу, но и необходимого уровня этого доверия.

Одним из самых мощных средств разработки криптографических протоколов является параноидальная модель. Когда пользователь A принимает участие в протоколе, он предполагает, что все другие участники общения объединились, чтобы его обмануть. Это самая что ни на есть настоящая теория тайных заговоров. Разумеется, каждый из остальных участников протокола делает те же предположения. Параноидальная модель является стандартной моделью поведения, на основе которой разрабатываются все криптографические протоколы.

Любые отклонения от упомянутой стандартной модели должны быть задокументированы в явном виде. К сожалению, этот шаг часто игнорируют. Время от времени нам попадаются протоколы, используемые в ситуациях, когда необходимый уровень доверия отсутствует. Например, большинство защищенных Web-узлов используют протокол SSL, который требует наличия у Web-узла доверенного сертификата. Между тем Internet-обозреватели пользователей зачастую принимают любые сертификаты, а получить таковой для злоумышленника не составляет труда. В результате пользователь устанавливает безопасное соединение с Web-узлом, но не знает, с каким именно.

В свое время этой особенностью воспользовалась многочисленная армия мошенников для осуществления махинаций против ничего не подозревающих пользователей системы электронных платежей PayPal.

Довольно часто необходимый уровень доверия не указывают в документации, потому что он "очевиден". Это может быть справедливо для разработчика протокола, но, как и всякий модуль системы, криптографический протокол должен иметь четко определенный интерфейс по легко объяснимым причинам.

В терминах бизнеса все внесенные в документацию требования к доверию эквивалентны рискам. Каждая потребность в определенном уровне доверия влечет за собой потенциальный риск, наличие которого следует учитывать.

14.5 Сообщения и действия

Типичное описание криптографического протокола содержит перечень сообщений, которыми обмениваются участники протокола, а также описание вычислений, которые должен выполнить каждый участник.

Следует отметить высокий уровень описания практически всех протоколов. Большинство деталей опускается. Это позволяет сконцентрироваться на основной функциональности протокола, но в то же время представляет большую угрозу для безопасности системы. Не имея четкого и досконального описания всех действий, которые должен предпринимать каждый участник, создать безопасную реализацию протокола крайне сложно.

Иногда создатели протокола снабжают его обширным описанием всех второстепенных деталей и проверок. Подобные спецификации зачастую настолько сложны, что никто не в состоянии разобраться в них до конца. Наличие подробной спецификации, безусловно, поможет разработчику, но ничто слишком сложное не может быть безопасным.

На помощь, как всегда, приходит модуляризация. Как и при работе с коммуникационными протоколами, функциональность криптографических протоколов может быть разбита на несколько уровней. Каждый уровень функционирует поверх предыдущего. Все уровни играют важную роль, но большинство из них одинаковы для всех протоколов. Лишь верхний уровень протокола весьма переменчив, и именно он всегда подлежит документированию.

14.5.1 Транспортный уровень

Пусть сетевые специалисты простят нас за то, что мы позаимствовали у них этот термин. Для криптографов транспортный уровень — это лежащая в основе протокола коммуникационная система, которая позволяет его участникам взаимодействовать друг с другом, в частности отправлять строки

байтов одним участником протокола другому участнику. Нам, как криптографам, важна лишь сама возможность отправки строки байтов от одного участника к другому. То, как именно это делается, нас не интересует. Мы можем использовать пакеты UDP, поток данных TCP, электронную почту или любой другой метод. Во многих случаях транспортному уровню требуется дополнительное кодирование. Например, если программа одновременно выполняет несколько протоколов, транспортный уровень должен доставить сообщение к месту выполнения нужного протокола. Для этого сообщению может потребоваться некоторое дополнительное поле назначения. При использовании протокола TCP к сообщению нужно добавлять поле длины, чтобы обеспечить реализацию служб, ориентированных на работу с сообщениями, поверх протокола TCP, используемого для поточной передачи данных.

Попытаемся конкретизировать ситуацию. Согласно нашим требованиям, транспортный уровень должен передавать произвольные строки байтов. Сообщение может включать в себя любые значения байтов. Длина строки является переменной. Полученная строка, безусловно, должна быть идентична той строке, которая была отослана. Удаление замыкающих нулевых байтов или любые другие модификации не допускаются.

Некоторые транспортные уровни включают в себя элементы наподобие "магических чисел", чтобы обеспечить раннее обнаружение ошибок или проверку синхронизации TCP-потока. Если магическое число полученного сообщения окажется неверным, оставшаяся часть сообщения должна быть отброшена.

Нельзя не отметить один очень важный частный случай. Иногда мы запускаем криптографический протокол поверх криптографически безопасного канала общения наподобие описанного в главе 8, "Безопасный канал общения". В подобных случаях транспортный уровень также обеспечивает конфиденциальность, аутентификацию и защиту от воспроизведения. Благодаря этому количество возможных типов атак, которые нужно учитывать при разработке, значительно сокращается, что облегчает сам процесс разработки.

14.5.2 Идентификация протоколов и сообщений

Следующий уровень (если идти снизу вверх) обеспечивает идентификацию протоколов и сообщений. Получая сообщение, мы хотим знать, какому протоколу оно принадлежит и что это за сообщение в рамках данного протокола.

Идентификатор протокола обычно состоит из двух частей. Первая — это информация о версии, которая предусматривает возможность будущих обновлений. Вторая часть указывает, какому конкретно криптографическому протоколу принадлежит сообщение. Например, в системе электронных пла-

тежей могут применяться отдельные протоколы для снятия денег со счета, оплаты, помещения на депозит, возврата и т.п. Идентификатор протокола помогает избежать путаницы между сообщениями, принадлежащими разным протоколам.

Идентификатор сообщения указывает на то, с каким именно сообщением протокола мы имеем дело. Например, если в рамках протокола есть четыре сообщения, мы хотим четко уяснять себе, что собой представляет каждое из них.

Зачем нам столько идентификационной информации, спросите вы? Разве злоумышленник не сможет подделать ее всю? Разумеется, сможет. Данный уровень не обеспечивает никакой защиты от активных фальсификаторов; вместо этого он позволяет обнаруживать случайные ошибки. Очень важно иметь хорошую систему обнаружения случайных ошибок. Предположим, отвечая за поддержку системы, вы неожиданно получаете огромное количество сообщений об ошибках. В подобной ситуации возможность отличить активные атаки от случайных ошибок наподобие проблем с конфигурацией и неправильных номеров версий окажет вам поистине неоценимую услугу.

Кроме того, идентификаторы протоколов и сообщений делают сообщение "самодостаточным", что значительно облегчает поддержку и отладку криптографических систем. Автомобили и самолеты разрабатываются с учетом того, чтобы их можно было легко обслуживать. Структура программного обеспечения еще более сложна. Вот почему в его разработку так важно закладывать простоту последующего обслуживания.

Пожалуй, наиболее важная причина, по которой мы включаем в сообщение идентификационные данные, касается принципа Хортона. Когда в протоколе применяется аутентификация (или схема цифровых подписей), мы обычно аутентифицируем несколько сообщений и полей данных. Включая в сообщение его идентификационные данные, мы полностью исключаем риск интерпретировать сообщение в неправильном контексте.

14.5.3 Кодирование и анализ сообщений

Следующим уровнем является кодирование. Каждый элемент данных сообщения должен быть преобразован в последовательность байтов. Это стандартная проблема программирования, а потому ограничимся лишь поверхностным обзором, не вдаваясь в детали.

Одним из наиболее важных моментов данной проблемы является *синтаксический анализ (parsing)*. Получатель должен иметь возможность проанализировать сообщение, чтобы преобразовать его из последовательности байтов обратно в набор полей данных. Такой анализ не должен зависеть от контекстной информации.

Поле фиксированной длины, одинаковое для всех версий протокола, легко поддается анализу. Мы знаем, какой в точности должна быть длина этого поля. Проблемы начинаются тогда, когда размер или значение поля зависят от некоторой контекстной информации, такой, как более ранние сообщения, отправленные в рамках соответствующего протокола. Это открывает массу возможностей для злоумышленников.

Многие сообщения в криптографических протоколах аутентифицируются с помощью цифровой подписи или каким-либо другим способом. Функция аутентификации работает со строкой байтов, поэтому аутентифицировать сообщение проще всего на транспортном уровне. Если интерпретация сообщения зависит от некоторой контекстной информации, проверка подписи или кода аутентичности сообщения может давать неоднозначные результаты. Мы уже взламывали несколько протоколов, пользуясь подобной ошибкой.

Для кодирования полей хорошо использовать кодировку TLV (Tag–Length–Value — дескриптор–длина–значение). Согласно ей каждое поле кодируется тремя элементами данных. Первый элемент (дескриптор) идентифицирует текущее поле, второй (длина) задает длину значения в закодированном виде, а третий (значение) — это и есть фактические данные, которые должны быть закодированы. Самой известной кодировкой TLV является ASN.1 [42], но она настолько сложна и плохо определена, что мы предпочитаем держаться от нее подальше. Впрочем, некоторые элементы ASN.1 могли бы оказаться весьма полезными.

Более новой альтернативой ASN.1 является XML. Забудьте всю шумиху, раздутую вокруг XML; мы собираемся использовать его лишь в качестве системы кодирования данных. Если для кодирования сообщений будет применяться фиксированный шаблон DTD (Document Template Definition — описание шаблона документа), синтаксический анализ будет контекстно-независимым и проблем с восстановлением исходного вида сообщений у нас не возникнет.

14.5.4 Состояние выполнения протокола

Во многих реализациях криптографических протоколов один и тот же компьютер может одновременно принимать участие в выполнении сразу нескольких протоколов. Чтобы следить за каждым из этих процессов, понадобится ввести состояние выполнения протокола. Последнее будет содержать всю информацию, необходимую для успешного завершения протокола.

Реализация протоколов требует применения *событийно-управляемого программирования (event-driven programming)*, поскольку продолжение работы каждого протокола зависит от получения внешних сообщений. Это можно реализовать разными способами, например выделяя один поток или процесс

под выполнение каждого протокола или же используя какую-нибудь систему управления событиями.

При наличии событийно-управляемой инфраструктуры реализовать криптографический протокол будет сравнительно несложно. Состояние протокола содержит конечный автомат, который указывает, какого типа должно быть следующее сообщение. В общем случае сообщения других типов не принимаются. Если в систему поступает сообщение ожидаемого типа, оно анализируется и обрабатывается согласно установленным правилам.

14.5.5 Ошибки

Криптографические протоколы всегда включают в себя массу проверок. К их числу относятся контроль типа протокола и типа сообщения, проверка того, принадлежит ли полученное сообщение к типу, заданному состоянием выполнения протокола, анализ сообщения и выполнение необходимых криптографических проверок. Если хотя бы одна из этих проверок окончится неудачей, в системе будет сгенерирована ошибка.

Поскольку ошибки представляют собой потенциальный путь нападения на систему, они нуждаются в самой тщательной обработке. Наиболее безопасное решение — при обнаружении ошибки не посылать никаких ответных сообщений и моментально удалить состояние протокола. Это минимизирует количество информации, которое злоумышленник может получить о протоколе. К сожалению, подобную систему, которая никак не сигнализирует об обнаружении ошибки, едва ли можно назвать дружественной.

Чтобы система действительно была удобна в использовании, нам могут потребоваться сообщения об ошибках. Если это возможно, не посылайте сообщения об ошибках другим участникам протокола. Запишите сообщение об ошибке в журнал, хранящийся в безопасном месте, чтобы системный администратор смог определить причину проблемы. Если же вы *должны* послать сообщение об ошибке, постарайтесь сделать его как можно менее информативным. Зачастую вполне достаточно простого сообщения, например: “Это ошибка”.

Отправка сообщений об ошибках особенно опасна при осуществлении тайминг-атак. Злоумышленник E может отправить пользователю A фальсифицированное сообщение и подождать ответа. Время, которое потребуется пользователю A для того, чтобы обнаружить ошибку и отослать ответ, часто позволяет определить, что именно в сообщении было неправильным и до какого момента все шло так, как нужно.

Вот хороший пример того, какую опасность могут представлять подобные взаимодействия. Много лет назад Нильс работал с коммерческой системой смарт-карт. Для активизации такой смарт-карты пользователю требовалось

ввести PIN-код. Четырехзначный PIN-код отсылался карте, а карта отвечала, активизирована она или нет. Если бы данная схема была реализована хорошо, для перебора всех возможных вариантов PIN-кода злоумышленнику понадобилось бы 10 000 попыток. Смарт-карта позволяла ввести неправильное значение PIN-кода пять раз, после чего она автоматически блокировалась, и разблокировать ее можно было только специальными средствами. Идея состояла в следующем: злоумышленник, который не знает PIN-кода, может перебрать всего лишь пять комбинаций, прежде чем карта будет заблокирована, в результате чего вероятность угадать PIN-код составляет 1 к 2000.

Сама идея была действительно неплоха. Аналогичные механизмы широко используются и сегодня. Вероятности 1 к 2000 вполне достаточно для большинства приложений. К сожалению, разработчик именно той конкретной системы смарт-карт был несколько беспечен. Чтобы проверить правильность четырехзначного PIN-кода, программа вначале проверяла первую цифру, затем вторую и т.п. Сообщение об ошибке PIN-кода появлялось сразу же, как только программа обнаруживала, что текущая цифра является неверной. Слабым местом этого алгоритма было то, что время, через которое смарт-карта выдавала сообщение о “неправильном PIN-коде”, зависело от количества правильных разрядов PIN-кода. Сообразительный злоумышленник мог измерить это время и получить массу полезной информации. В частности, он мог узнать, в каком по счету разряде была впервые обнаружена ошибка. В этом случае для подбора PIN-кода было бы достаточно всего 40 попыток. (Через 10 попыток мы узнаем, каков первый разряд, еще через 10 попыток — второй и т.п.) Теперь при наличии пяти возможных попыток вероятность угадать PIN-код возрастает до 1 к 143. Это намного лучше, чем вероятность 1 к 2000. Например, если у злоумышленника есть 20 попыток, вероятность угадать PIN-код возрастает до 60%, что несравнимо больше упомянутой вероятности в 0,2%.

Однако гораздо хуже то, что осуществить 20 или 40 попыток отнюдь не так невозможно, как кажется. Смарт-карта, которая блокируется после заданного числа неправильных PIN-кодов, сбрасывает свой счетчик, как только пользователь ввел нужный PIN-код. В этом случае у пользователя появляется еще пять попыток, в ходе которых он может ввести неправильный PIN-код. Предположим, что подобной смарт-картой обладает ваш сосед по комнате. Если вам удастся незаметно взять смарт-карту соседа, вы можете один-два раза попытаться подобрать PIN-код, после чего положить смарт-карту на место. Подождите, пока сосед использует ее как положено. Введя правильный PIN-код, он сбросит счетчик неправильных попыток, и вы сможете начать все сначала. Максимум через 40 попыток у вас в руках будет полный PIN-код карты соседа.

Обработка ошибок слишком сложна, чтобы пытаться описать ее простым набором правил. Криптографическое сообщество еще не проанализировало эту проблему в достаточной степени. Лучшее, что мы можем посоветовать на данный момент, — будьте предельно осторожны!

14.5.6 Воспроизведение и повторение

Атака воспроизведения происходит тогда, когда злоумышленник записывает сообщение и позднее отсылает его снова. Систему нужно защищать от воспроизведения сообщений. Обнаружить воспроизведенные сообщения довольно сложно, так как они весьма похожи на правильные. В конце концов, они ведь и *есть* правильные.

С атаками воспроизведения тесно связаны так называемые *повторения (retry)*. Предположим, пользователь А применяет протокол для общения с пользователем Б, но не получает от него ответа. Причин может быть много, но наиболее вероятно, что пользователь Б просто не получил последнего сообщения и до сих пор его ждет. В реальной жизни это случается сплошь и рядом, и мы решаем проблему, отправив еще одно письмо по обычной или электронной почте или же еще раз произнеся последнюю фразу. В автоматизированных системах это называется повторением. Пользователь А повторяет свое последнее сообщение для пользователя Б и снова ждет ответа.

Как видите, пользователь Б может получать как копии сообщений, воспроизведенные злоумышленником, так и копии сообщений, повторно отосланные ему пользователем А. Пользователю Б нужно как-то разобраться в этих сообщениях и гарантировать правильную обработку последних без ущерба для безопасности системы.

Реализация отправки повторных сообщений довольно проста. У каждого участника есть состояние выполнения протокола в том или ином виде. От вас требуется лишь установить счетчик времени и отослать последнее сообщение еще раз, если в течение заданного времени вы не получите ответа. Конкретная величина этого промежутка времени зависит от используемой коммуникационной инфраструктуры. Если вы работаете с протоколом UDP (протокол, который непосредственно использует пакеты IP), существует достаточно большая вероятность того, что сообщение будет утеряно, поэтому время ожидания ответа должно быть небольшим — порядка нескольких секунд. Если же для отправки сообщений используется TCP, последний автоматически повторяет отправку данных, которые не были получены должным образом, используя собственные значения времени ожидания. В этом случае вряд ли имеет смысл реализовать повторную отправку сообщений на криптографическом уровне, и многие системы, работающие на основе TCP, этого не делают. Тем не менее в дальнейшем будем предполагать, что повторная

отправка сообщений все-таки реализована, поскольку общие методы обработки полученных повторных сообщений работают даже в том случае, если повторные сообщения не отправляются.

Получив сообщение, следует решить, что с ним делать. Мы исходим из предположения, что все сообщения являются идентифицируемыми. Другими словами, мы точно знаем, каким именно сообщением протокола должно быть текущее полученное сообщение. Если мы получили то сообщение, которого ожидали, тогда все в порядке и мы продолжаем следовать правилам протокола. Теперь предположим, что мы получили сообщение "из будущего", т.е. то сообщение протокола, которое ожидали получить в более поздний момент времени. Справиться с таким сообщением легко — просто проигнорируйте его. Не изменяйте состояние выполнения протокола и не отправляйте ответ; просто отбросьте сообщение — и все. Вероятно, оно являлось частью атаки. Даже в самых странных протоколах, где получение сообщения "из будущего" может быть всего-навсего частью цепочки ошибок, инициированных утерянными сообщениями, игнорирование сообщения будет иметь точно такой же эффект, что и потеря сообщения в процессе его передачи. Поскольку каждый протокол должен уметь справляться с потерей сообщения, игнорирование сообщений в любом случае не нанесет вреда.

А как же быть со "старыми" сообщениями, которые уже были обработаны протоколом когда-то в прошлом? Подобное событие может произойти в трех случаях. В первом полученное сообщение совпадает с предыдущим, на которое вы только что ответили. Возможно, оно было повторно отправлено вашим собеседником, поэтому на него следует ответить так же, как вы ответили перед этим. Обратите внимание, что ответ должен быть в точности таким же. Не пересчитывайте ответ с помощью другого случайного значения. Не стоит также просто предполагать, что полученное сообщение идентично предыдущему, на которое вы ответили. Это нужно проверить.

Во втором случае идентификатор полученного сообщения равен идентификатору сообщения, на которое вы ответили последним, но содержимое сообщения отличается. Например, предположим, что в протоколе DH пользователь Б получает первое сообщение от пользователя А и затем получает еще одно сообщение, которое также помечено как первое сообщение протокола, но содержит другие данные. Это свидетельствует об атаке. Оно никогда бы не спровоцировало подобную ситуацию, поскольку повторное сообщение никогда не отличается от исходного. Либо второе сообщение, либо сообщение, на которое вы уже ответили, является поддельным. В целях безопасности подобную ситуацию рекомендуется рассмотреть как ошибку протокола со всеми вытекающими отсюда последствиями, которые уже упоминались. (Игнорирование полученного сообщения тоже вполне безопасно, но оно не позволит

распознать многие виды активных атак. Это окажет нежелательное влияние на компоненты обнаружения и отклика системы безопасности.)

В третьем случае мы получаем совсем старое сообщение, которое было впервые отправлено еще задолго до предыдущего. Сделать в данной ситуации можно не так уж много. Если вы все еще храните изначальное сообщение, которое было получено на той стадии протокола, то можете проверить, идентичны ли старое и новое сообщения. Если это действительно так, просто проигнорируйте сообщение. Если же сообщения неодинаковы, вы обнаружили атаку и должны обработать ее как ошибку протокола. Многие программы сохраняют не все сообщения, которые были получены во время выполнения протокола. В этом случае мы не сможем определить, совпадает ли новое сообщение с тем, что уже когда-то было получено. Такие сообщения рекомендуется просто проигнорировать — в любом случае это не нанесет вреда безопасности системы. Вы бы удивились, узнав, как часто подобное происходит с реальными протоколами. Иногда сообщения задерживаются на длительное время. Предположим, пользователь А отправил сообщение, которое было задержано. Через несколько секунд пользователь А посылает повторное сообщение, которое на сей раз достигает своей цели, после чего оба пользователя продолжают выполнение протокола. Через полминуты пользователь Б получает исходное сообщение. С нашей точки зрения, подобная ситуация выглядит так, как если бы пользователь Б получил копию — в терминах протокола — очень старого сообщения.

Ситуация становится намного сложнее, если у протокола есть более двух участников. Такие протоколы существуют, но их рассмотрение выходит за рамки данной книги. Если вам когда-нибудь придется разрабатывать протокол с несколькими участниками, обратите особое внимание на повторения и воспроизведения.

Еще одно замечание: узнать, было ли доставлено последнее сообщение протокола, невозможно. Если пользователь А отсылает последнее сообщение пользователю Б, он никогда не получит подтверждения того, что это сообщение действительно прибыло. Если же канал связи будет нарушен и пользователь Б не получит последнего сообщения, он попытается повторить предыдущее сообщение, но оно все равно не дойдет до пользователя А. Пользователь А, находящийся на “нормальном” конце протокола, не сможет определить, дошло сообщение или нет. Для решения этой проблемы в конец протокола можно добавить специальное уведомление, отправленное пользователем Б пользователю А, но тогда это уведомление станет новым последним сообщением, и проблема повторится. Криптографические протоколы следует разрабатывать таким образом, чтобы описанная ситуация не повлияла на безопасность системы.

Глава 15

Протокол согласования ключей

Наконец-то пришло время взяться за протокол согласования ключей. Назначением данного протокола является создание общего ключа для безопасного канала общения, описанного в главе 8, “Безопасный канал общения”.

В законченном виде криптографические протоколы довольно сложны, и знакомство сразу с окончательной версией протокола может оказаться весьма непростой задачей для неподготовленного читателя. Поэтому мы представим вам последовательность протоколов, к каждому из которых будет добавляться все больше и больше функций. Не забывайте, что промежуточные версии протокола не обладают полной функциональностью и будут иметь множество слабых мест.

15.1 Окружение

У протокола согласования ключей есть два участника: пользователь А и пользователь Б. Оба пользователя хотят обеспечить безопасность своего общения. Вначале они запускают протокол согласования ключей, чтобы установить секретный ключ сеанса k, а затем используют ключ k для обмена фактическими данными по безопасному каналу общения.

Для обеспечения безопасности согласования ключа пользователи А и Б должны иметь возможность идентифицировать друг друга. Вопросам базовой аутентификации посвящена часть 3, “Введение в криптографию”, а пока мы просто предположим, что пользователи А и Б в состоянии аутентифицировать сообщения, которыми обмениваются. Базовая аутентификация может осуществляться с помощью цифровых подписей RSA (если пользователи А и Б знают ключи друг друга или используют инфраструктуру открытого ключа) либо посредством общего секретного ключа и функции вычисления MAC.

Но зачем же проводить согласование ключа, если у нас уже есть общий секретный ключ? На то есть свои причины. Прежде всего согласование ключа позволяет отделить ключ сеанса от существующего (долговременного) общего ключа. Если ключ сеанса будет дискредитирован (например, из-за ошибок в реализации безопасного канала общения), общий секретный ключ все еще останется в безопасности. А если общий секретный ключ будет дискредитирован *после* выполнения протокола согласования ключей, злоумышленник, которому известен общий секретный ключ, все еще не сможет узнать ключ сеанса, согласованный в результате выполнения протокола. Таким образом, потеря ключа не приведет к раскрытию прежних данных. Эти свойства очень важны: они повышают надежность всей системы.

Существуют также ситуации, в которых общий секретный ключ довольно слаб (например, если это пароль). Как правило, пользователи, не желая запоминать 30-буквенные пароли, пытаются выбрать что-нибудь попроще. Стандартным типом атаки на пароли подобного рода является *атака с использованием словаря (dictionary attack)*, в ходе которой компьютер злоумышленника перебирает большое количество простых паролей. Хороший протокол согласования ключей может превратить слабый пароль в сильный ключ. Впрочем, в этой главе такие протоколы рассматриваться не будут.

15.2 Первая попытка

Начнем с самой простой структуры протокола (рис. 15.1). Это не более чем протокол Диффи–Хеллмана в рамках подгруппы с добавлением аутентификации. Пользователи А и Б выполняют протокол DH, используя первые два сообщения. (Для простоты мы опустили несколько необходимых проверок.) Затем пользователь А подсчитывает значение функции аутентификации для ключа сеанса k и посылает это значение пользователю Б, который сверяет его со значением функции аутентификации для своего ключа k. Точно так же пользователь Б посылает значение функции аутентификации для k пользователю А.

Пока неизвестно, в какой именно форме будет происходить аутентификация. Напомним: предполагается, что пользователи А и Б могут аутентифицировать сообщения, которыми они обмениваются. Таким образом, пользователь Б может проверить значение функции $\mathrm{AUTH}_A(k)$, а пользователь А — значение функции $\mathrm{AUTH}_B(k)$. То, как именно это делается — посредством цифровых подписей или функции вычисления MAC, нас не волнует. Данный протокол лишь обеспечивает возможность аутентификации по отношению к ключу сеанса.

Первая версия протокола имеет ряд недостатков.

Рис. 15.1. Первая попытка согласования ключа

- Работа протокола основана на предположении, что пользователям А и Б известен набор параметров (p, q, g). Выбор констант в качестве этих значений — не очень удачная идея.
- Данный протокол использует четыре сообщения, в то время как для достижения желаемой цели достаточно только трех.
- Ключ сеанса применяется в качестве входного значения функции аутентификации. Это не составит проблемы, если функция сильная. Предположим, однако, что функция аутентификации может допустить утечку информации о нескольких битах ключа сеанса. Это было бы очень плохо и, безусловно, потребовало бы повторного анализа всего протокола. "Золотое правило" гласит: секретные данные лучше использовать только для одной цели. В нашем случае k будет применяться в качестве ключа сеанса, поэтому использовать его еще и как аргумент функции аутентификации определенно не стоит.
- Сообщения об аутентификации слишком похожи друг на друга. Пусть, например, функция аутентификации — это простая функция вычисления MAC, которая использует значение секретного ключа, известного обоим пользователям. Тогда пользователь Б может просто отправить пользователю А полученное от него же значение MAC, а значит, пользователю Б не нужно знать секретный ключ, чтобы завершить выполнение протокола. Следовательно, пользователь А не может быть уверен в том, что последнее сообщение об аутентификации истинно.

- Ключ k не должен использоваться программой до тех пор, пока пользователи А и Б не обменяются сообщениями об аутентификации, и за этим необходимо внимательно следить. На первый взгляд данное требование выглядит очевидным, но вы бы не поверили, узнав, до чего додумываются некоторые программисты, пытаясь оптимизировать систему.

В следующих разделах главы рассматривается, как исправить все эти недостатки.

15.3 Пусть всегда будут протоколы!

Мы уже подчеркивали важность проектирования систем с запасом на будущее. Это еще более важно для протоколов. Если ограничить размер полей базы данных до 2000 байт, это может составить проблему для некоторых пользователей, но от ограничения легко избавиться в следующей же версии базы данных. С протоколами все не так просто. Протоколы запускаются для обмена данными между различными участниками, и каждая новая версия протокола должна обладать обратной совместимостью с предыдущей. Изменить протокол и в то же время сохранить его совместимость со старыми версиями чрезвычайно сложно. Чтобы осознать это, вам придется реализовать несколько версий протокола, а также систему выбора, какую версию следует использовать.

Переход к другой версии протокола — это, безусловно, прекрасный момент для осуществления атаки. Если старый протокол был менее безопасен, злоумышленник будет заинтересован в том, чтобы заставить вас применить именно этот старый протокол. Вы бы удивились, узнав, как много систем страдают от так называемой *атаки с откатом версий (version-rollback attack)*.

Разумеется, предусмотреть все будущие требования невозможно, поэтому в какой-то момент вам понадобится определить вторую версию протокола. Тем не менее цена одновременного существования нескольких версий протокола достаточно высока, особенно в контексте общей сложности.

Удачные протоколы живут практически вечно (а неудачные нас не интересуют). Полностью убрать протокол из всех систем в мире невозможно. Таким образом, протоколы еще более важно проектировать с запасом на будущее. Вот почему мы не можем указать в протоколе согласования ключей фиксированный набор параметров DH. Даже если они будут очень большими, существует опасность, что будущие успехи в области криптоанализа заставят нас изменить их.

15.4 Соглашение об аутентификации

Прежде чем продолжать рассмотрение протокола согласования ключей, мы познакомим вас с соглашением об аутентификации. Протоколы часто обладают множеством элементов данных, поэтому иногда бывает сложно определить, какие элементы данных должны подвергнуться аутентификации. Некоторые протоколы оказываются взломанными, поскольку не проводят проверку определенных полей данных. Во избежание этих проблем мы используем простое соглашение об аутентификации.

В наших протоколах каждый раз, когда участник посылает сообщение об аутентификации, он применяет функцию аутентификации ко всем данным, которыми до этого момента обменялся с другим участником, т.е. ко всем предыдущим сообщениям и всем полям данных, предшествующим значению функции аутентификации в самом сообщении об аутентификации. В протоколе, представленном на рис. 15.1, пользователь А должен был бы подсчитать аутентификатор не для k, а для X и Y. В свою очередь, аутентификатор пользователя Б охватывал бы X, Y и AUTH_A.

Это соглашение позволяет устранить несколько потенциальных путей атак, а его реализация обходится совсем недорого. Участники криптографических протоколов не обмениваются слишком большим количеством данных, а вычисления аутентификаторов практически всегда начинаются с хэширования входной строки. Функции хэширования работают настолько быстро, что дополнительные расходы в терминах производительности оказываются незначительными.

Помимо всего прочего, данное соглашение позволяет сократить запись функций. Вместо того чтобы писать нечто наподобие $\mathrm{AUTH}_A(X, Y)$, мы будем записывать просто AUTH_A. Поскольку данные, подлежащие аутентификации, задаются соглашением, больше не нужно указывать их явно. Все последующие протоколы, представленные в книге, соответствуют этому соглашению.

Еще раз напомним: функция аутентификации удостоверяет подлинность только строки байтов. Каждая строка байтов, подвергающаяся аутентификации, должна начинаться с уникального идентификатора, определяющего точное место протокола, в котором применяется соответствующий аутентификатор. Кроме того, преобразование предыдущих сообщений и полей данных в строку байтов должно выполняться таким образом, чтобы эти сообщения и поля могли быть восстановлены без какой-либо дальнейшей контекстной информации. Мы уже обсуждали это более подробно, однако данный момент, несмотря на всю его важность, весьма легко упустить.

15.5 Вторая попытка

Как же исправить недостатки предыдущего протокола? Мы не хотим использовать фиксированный набор параметров алгоритма DH. Пусть эти параметры выберет пользователь А и отошлет их пользователю Б.

Мы также сократим количество сообщений, которыми обмениваются пользователи А и Б, с четырех до двух (рис. 15.2). Пользователь А выбирает параметры алгоритма DH и случайное число x, вычисляет X и отсылает все это пользователю Б вместе со значением функции аутентификации. Пользователь Б должен проверить, что полученные им параметры алгоритма DH выбраны правильно и что значение X корректно. (Более подробно реализация этих проверок описана в главе 12, “Алгоритм Диффи–Хеллмана”.) Оставшаяся часть протокола полностью аналогична предыдущей версии. Пользователь А получает значение Y и AUTH_B, проверяет их и вычисляет значение ключа DH.

У нас больше нет фиксированных параметров алгоритма DH. Мы используем только два сообщения вместо четырех, нигде не применяем ключ сеанса, помимо его основного назначения, а наше соглашение об аутентификации гарантирует, что строки, подвергающиеся аутентификации, не будут одинаковыми.

Несмотря на все это, у нас появилось несколько новых проблем.

- Что делать, если пользователь Б захочет использовать простое число большего размера, чем было предложено пользователем А? Возможно, пользователь Б применяет более строгие политики безопасности, и число, выбранное пользователем А, кажется ему недостаточно безопасным. В этом случае пользователю Б придется прервать выполнение протоко-

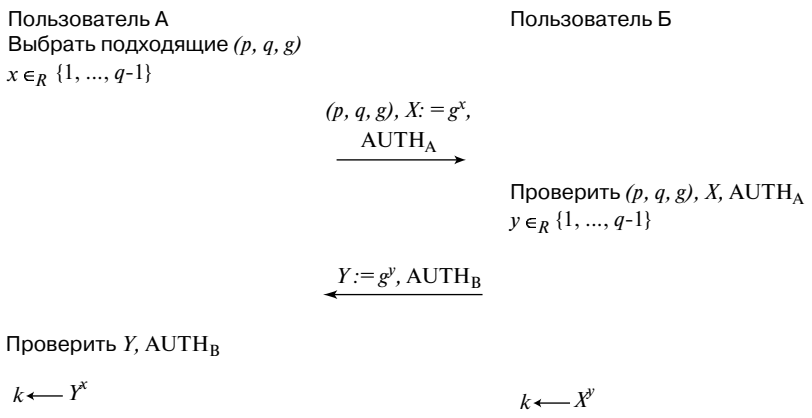

Рис. 15.2. Вторая попытка согласования ключа

ла. Он мог бы отправить сообщение об ошибке примерно следующего содержания: “Простое число DH должно быть как минимум k бит длиной”, но это внесло бы изрядную путаницу в выполнение протокола. Пользователю А остается лишь одно — перезапустить протокол с новыми параметрами.

- Другая проблема связана с аутентификацией. Пользователь Б не уверен, что он действительно общается с пользователем А. Кто-нибудь может записать первое сообщение, посланное пользователем А, и позднее переслать его пользователю Б, который решит, что сообщение пришло от пользователя А (ведь аутентификация удостоверит подлинность сообщения). Поэтому он завершит выполнение протокола, думая, что обменялся общим ключом k с пользователем А. Разумеется, злоумышленник все равно не узнает значение k, поскольку он не знает x, а без x не сможет взломать оставшуюся часть системы, которая использует k. Тем не менее журналы пользователя Б покажут успешное завершение протокола с пользователем А, что само по себе является проблемой, так как снабжает администраторов ошибочной информацией.

Проблема пользователя Б называется нехваткой “живучести”. Он не уверен, что пользователь А “жив” и что он разговаривает не с бесплотным призраком, воспроизводящим сообщения. Классический способ решения этой проблемы подразумевает включение в аутентификатор пользователя А случайного элемента, выбранного пользователем Б.

15.6 Третья попытка

Попытаемся исправить описанные выше проблемы с помощью еще нескольких изменений. Теперь параметры алгоритма DH больше не будет выбирать пользователь А. Он просто пошлет свои минимальные требования пользователю Б, а тот выберет параметры. При этом количество сообщений увеличится до трех. (Оказывается, что самые интересные криптографические протоколы требуют наличия как минимум трех сообщений. Мы не знаем, чем это вызвано, — просто так получается.) Пользователь Б посылает только одно сообщение, а именно второе. Это сообщение будет содержать его аутентификатор, поэтому вместе со своим первым сообщением пользователь А должен отослать некоторый случайный элемент. В качестве последнего мы будем применять случайную оказию.

Схема полученного протокола представлена на рис. 15.3. Вначале пользователь А выбирает s — минимально допустимый размер простого числа p. Затем он генерирует случайную 256-битовую строку в качестве оказии N_a и отсылает оба значения пользователю Б, который выбирает подходящий

Рис. 15.3. Третья попытка согласования ключа

набор параметров алгоритма DH и случайный показатель степени x. После этого он отсылает пользователю А параметры, значение g^x и свой аутентификатор. Пользователь А завершает протокол DH, отсылая пользователю Б значение Y и свой аутентификатор.

Нам осталось решить еще одну проблему. Результат k имеет переменный размер, что может быть весьма неудобно для других частей системы. Более того, k вычисляется с использованием алгебраических выражений, а наличие в криптографической системе какой-либо алгебраической структуры всегда нас пугает. Разумеется, иногда без алгебраической структуры не обойтись, но по мере возможности мы стараемся ее избегать.

Опасность использования алгебраической структуры состоит в том, что злоумышленник может извлечь из нее что-нибудь полезное. Математический аппарат — исключительно мощное средство атаки. За последние 20 лет мы видели массу новых предложений криптографических систем с открытым ключом. Практически все эти системы были успешно взломаны, и в большинстве случаев именно благодаря алгебраической структуре, которую они содержали. Всегда избавляйтесь от алгебраической структуры, если это возможно.

Очевидным решением данной проблемы является хэширование полученного ключа. Это превратит ключ в число фиксированной длины и уничтожит всю оставшуюся алгебраическую структуру.

15.7 Окончательная версия протокола

Если представить окончательную версию протокола в сокращенном виде (рис. 15.4), это сделает его читабельным и легким для понимания. Тем не менее, чтобы сделать протокол читабельным, мы опустили множество этапов проверки. Мы просто пишем "Проверить (p, q, g)", что на самом деле включает в себя сразу несколько проверок. Протокол со всеми необходимыми криптографическими проверками представлен в полном виде на рис. 15.5.

Пользователю Б нужно выбрать подходящий размер для p. Это зависит от минимального размера p, который требуется пользователю А, и минимального размера p, который требуется самому пользователю Б. Разумеется, пользователь Б должен проверить, находится ли значение s в разумных пределах. Мы не хотим, чтобы пользователь Б начал генерировать 100 000-битовые простые числа только потому, что он получил неаутентифицированное сообщение с большим значением s. Точно так же пользователь А не должен начинать проверку очень больших простых чисел только потому, что их прислал пользователь Б. Поэтому оба пользователя ограничивают максимальный размер числа p. Использование фиксированного максимума ограничивает гибкость протокола; если неожиданный прогресс в области криптоанализа заставит вас перейти к использованию чисел большего размера, наличие фиксированного максимума обернется настоящей проблемой. Использование настраиваемого

Рис. 15.4. Окончательная версия протокола в сокращенном виде

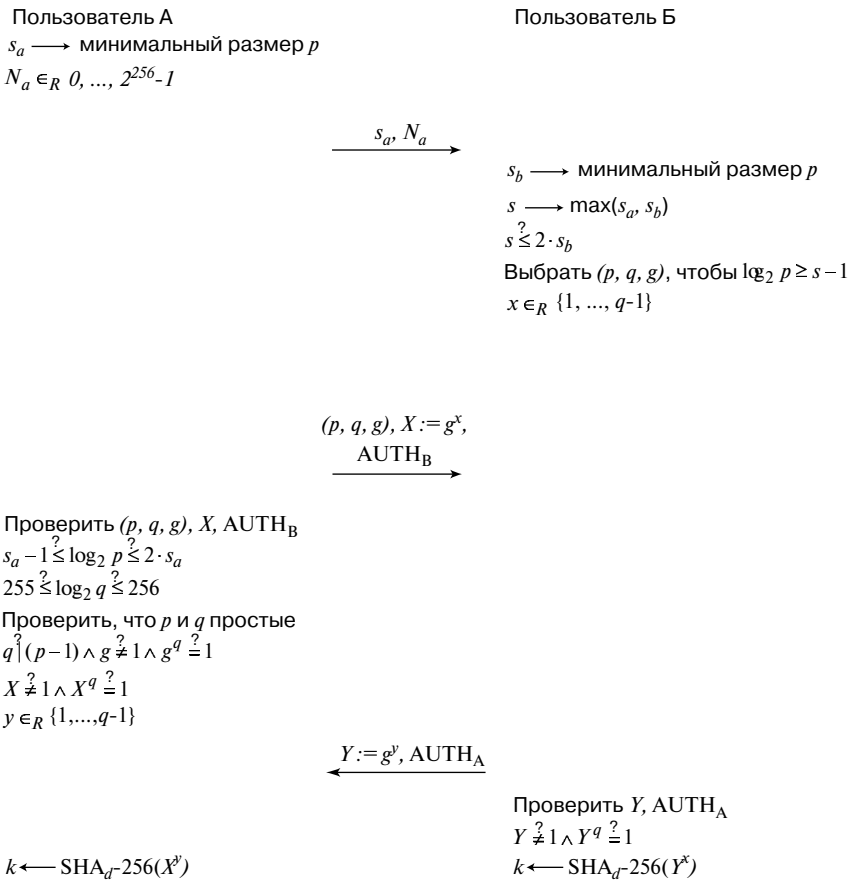

Рис. 15.5. Окончательная версия протокола в развернутом виде

максимума влечет за собой все проблемы, присущие любому настраиваемому параметру, в котором практически никто не разбирается. Поэтому мы решили использовать динамический максимум. И пользователь А и пользователь Б отказываются применять простое число, длина которого более чем в два раза превышает заданную ими длину. Наличие динамического максимума не мешает обновлению системы и в то же время позволяет избежать использования слишком больших простых чисел. Конечно, вы можете усомниться в том, действительно ли множитель 2 будет достаточно эффективным. При желании можно использовать не 2, а 3; особой роли это не играет.

Оставшаяся часть протокола (см. рис. 15.5) — это всего лишь развернутое представление этапов его сокращенной формы. Если пользователи А и Б достаточно сообразительны, они оба будут кэшировать подходящие параметры алгоритма DH. Это избавит пользователя Б от необходимости каждый раз

генерировать новые параметры алгоритма DH, а пользователя A — от необходимости каждый раз их проверять. Приложения даже могут использовать фиксированные наборы параметров алгоритма DH, в результате чего их не придется посылать явно — достаточно указать идентификатор нужного набора параметров. Все эти изменения представляют собой простые, непосредственные оптимизации протокола, поэтому мы не будем обсуждать их более подробно. Тем не менее будьте крайне осторожны. Многочисленные оптимизации могут привести к тому, что протокол будет взломан. К сожалению, мы не можем сформулировать никаких простых правил относительно того, приведет ли конкретная оптимизация к взлому протокола или нет. Проектирование протокола — это скорее искусство, нежели наука, и жестких правил, которых следует придерживаться всегда и везде, здесь не существует.

15.8 Анализ протокола с различных точек зрения

Протокол наподобие нашего весьма полезно проанализировать с различных точек зрения. Наш протокол должен обладать несколькими свойствами, и в следующих разделах речь идет о том, почему он действительно ими обладает.

15.8.1 Точка зрения пользователя A

Давайте рассмотрим протокол с точки зрения пользователя A. Он получает от пользователя Б одно сообщение. Пользователь A уверен, что это сообщение действительно пришло от пользователя Б, так как оно прошло аутентификацию, и в данные, которые подверглись аутентификации, была включена случайная оказия N_a, выбранная самим пользователем A. Такое сообщение невозможно подделать или заменить воспроизведенным старым сообщением.

Пользователь A убеждается в том, что параметры алгоритма DH были выбраны правильно, поэтому протокол DH будет обладать всеми необходимыми свойствами. Таким образом, когда пользователь A генерирует секретное значение y и посылает значение Y, он понимает, что вычислить k сможет только тот человек, который знает x, такой, что $g^x = X$. Это основное свойство протокола DH. Пользователь Б аутентифицировал X, а он может сделать это только в том случае, если выполняет все правила протокола. Следовательно, пользователь Б знает соответствующее значение x и сохраняет его в секрете. Таким образом, пользователь A может быть уверен в том, что полученный им ключ k известен только пользователю Б.

Итак, пользователь А убедился в том, что он действительно общается с пользователем Б и что полученный им ключ известен только ему и пользователю Б.

15.8.2 Точка зрения пользователя Б

Теперь давайте взглянем на протокол глазами пользователя Б. Первое сообщение, полученное им будто бы от пользователя А, практически не предоставляет ему никакой полезной информации. Оно просто утверждает, что кто-то по ту сторону протокола выбрал значение s_a и несколько случайных бит N_a.

С третьим сообщением все обстоит по-другому. Это сообщение определенно пришло от пользователя А, потому что пользователь А аутентифицировал его, а согласно нашим предположениям пользователь Б может проверить аутентификатор пользователя А. Данные, которые подверглись аутентификации, включают в себя X — случайное значение, выбранное самим пользователем Б. Это означает, что третье сообщение не могло быть воспроизведенным сообщением, посланным злоумышленником. Оно было аутентифицировано пользователем А специально для этого сеанса работы протокола. Кроме того, аутентификатор пользователя А охватывает и первое сообщение, полученное пользователем Б, поэтому теперь пользователь Б может быть уверен и в правильности первого сообщения.

Пользователь Б уверен в безопасности параметров алгоритма DH; в конце концов, он сам их выбрал. Поэтому, как и пользователь А, он знает, что вычислить ключ k сможет только тот, кто знает y, такое, что $g^y = Y$. Но пользователь А аутентифицировал высланное им значение Y, поэтому он является единственным человеком, который знает соответствующее y. Это убеждает пользователя Б в том, что вычислить ключ k кроме него сможет только пользователь А.

15.8.3 Точка зрения злоумышленника

И наконец, проанализируем протокол согласования ключей с точки зрения злоумышленника. Если мы будем просто прослушивать канал связи, то зафиксируем все сообщения, которыми обмениваются пользователи А и Б. Но ключ k вычисляется по протоколу DH, поэтому, если параметры алгоритма DH надежны, пассивная атака наподобие этой не позволит получить какой-либо информации о ключе k. Другими словами, пора переходить к активным действиям.

В качестве поучительного упражнения можно рассмотреть каждый элемент данных протокола и попытаться изменить его. В этом случае наша атака

будет моментально остановлена двумя процессами аутентификации. Последняя аутентификация, проводимая пользователем А, охватывает все данные, которыми до этого обменялись пользователи А и Б. Это означает, что мы не можем изменить какие-либо элементы данных, можем лишь попытаться воспроизвести старые сообщения из предыдущего сеанса работы протокола. Но наличие оказии и случайно выбранного X останавливает и атаку воспроизведения.

Все это отнюдь не означает, что у нас не осталось путей для нападения. Мы могли бы, например, изменить значение s_a на величину большего размера. Если это значение окажется приемлемым для пользователя Б, бóльшая часть протокола пройдет по обычному сценарию. Существует лишь три проблемы. Первая состоит в том, что увеличение s_a вообще нельзя назвать атакой, поскольку оно приводит лишь к увеличению простого числа DH, в результате чего параметры алгоритма DH становятся еще надежнее. Вторая и третья проблемы — это два процесса аутентификации, которые завершатся неудачей.

В реальной жизни существует множество протоколов с элементами данных, которые не подвергаются аутентификации. Большинство разработчиков не стали бы утруждать себя аутентификацией значения s_a, потому что его изменение не приведет к осуществлению атаки. (Пользователи А и Б независимо друг от друга проверяют, подходит ли им размер числа p.) Тем не менее, на наш взгляд, злоумышленнику никогда не следует позволять вмешиваться в общение участников протокола. Зачем предоставлять ему больше средств, чем нужно? Кроме того, мы можем легко представить себе ситуацию, в которой отказ от аутентификации s_a угрожает безопасности системы. Например, предположим, что пользователь Б предпочитает работать с одним из наборов параметров алгоритма DH, встроенных в программу, и генерирует новые параметры только тогда, когда это нужно. Если размеры простого числа, выбранные пользователями А и Б, всегда будут соответствовать параметрам, имеющимся в списке, пользователь Б никогда не сгенерирует новый набор параметров. Но это также означает, что код пользователя Б, который генерирует параметры, и код пользователя А, который проверяет эти параметры, никогда не будут использоваться. Маловероятно, что такой код будет тщательно протестирован. Ошибка в коде генерации и проверки параметров может остаться незамеченной, пока злоумышленник не увеличит s_a. Разумеется, вероятность подобного сценария крайне мала, однако существуют тысячи маловероятных сценариев, каждый из которых плохо влияет на безопасность системы. А тысячи рисков с малой степенью вероятности в совокупности могут составить риск с высокой степенью вероятности. Вот почему мы так стремимся предотвратить атаку любого типа там, где только возможно. Это дает нам ощущение истинной защищенности.

15.8.4 Взлом ключа

Теперь посмотрим, что произойдет, если одна из оставшихся частей системы будет каким-либо образом дискредитирована.

Если пользователь А потеряет свой ключ аутентификации, который не будет известен злоумышленнику, он просто не сможет запустить протокол. При этом пользователь А все еще сможет применять уже установленные ключи сеанса. Именно такого поведения обычно ожидают от протоколов. Это же справедливо и для пользователя Б, если он потеряет свой ключ аутентификации.

Если пользователь А потеряет ключ сеанса, который не будет известен злоумышленнику, ему придется еще раз запустить протокол согласования ключей с пользователем Б, чтобы установить новый ключ сеанса.

Ситуация ухудшается, если злоумышленнику удается узнать ключ. Если злоумышленник узнает ключ аутентификации пользователя А, он сможет выдавать себя за него до тех пор, пока пользователь Б не будет уведомлен о потере ключа и не перестанет принимать сообщения, аутентифицированные пользователем А. Это неизбежное следствие потери ключа. Если вы потеряете ключи от машины, всякий, кто найдет их, сможет воспользоваться вашей машиной. Главная функция ключей и состоит в том, чтобы открывать доступ к определенным функциям. К счастью, наш протокол обладает одним свойством, наличие которого всегда крайне желательно: он гарантирует, что все предыдущие сеансы общения между пользователями А и Б останутся в секрете. Даже если злоумышленник знает ключ аутентификации пользователя А, он не сможет вычислить ключ сеанса k для протокола, выполнение которого уже было завершено. Это не удастся даже в том случае, если злоумышленник записывал все предыдущие сообщения. Данное свойство протокола называется *прямой безопасностью (forward secrecy)*[1]. Сказанное справедливо и для ключа аутентификации пользователя Б.

И наконец, рассмотрим ситуацию, когда злоумышленнику удается взломать ключ сеанса. Ключ k — это хэш-код значения g^{xy}, где x и y — случайные числа. Они не предоставляют никакой информации о других ключах, например о ключах аутентификации пользователей А и Б. Значение k, применяемое в одном сеансе работы протокола, полностью независимо от значений k, применяемых в других сеансах работы этого же протокола (по крайней мере, если предположить, что пользователи А и Б применяют хороший генератор псевдослучайных чисел).

[1] Иногда это называют идеальной прямой безопасностью (perfect forward secrecy — PFS), но мы предпочитаем не использовать слов наподобие “идеальный”, поскольку идеальной безопасности не существует.

Как видите, наш протокол обеспечивает наилучшую возможную защиту от взлома ключей.

15.9 Вычислительная сложность протокола

Теперь обсудим вычислительную сложность нашего решения. Будем исходить из предположения, что процедуры выбора и проверки параметров алгоритма DH кэшированы, поэтому они не будут учитываться в объеме вычислений, необходимых для проведения одного сеанса работы протокола. Как следствие этого, у нас остаются такие вычисления:

- три операции возведения в степень в подгруппе DH для каждого из пользователей А и Б;
- одна генерация аутентификатора;
- одна проверка аутентификатора;
- различные относительно эффективные операции, такие, как генерация случайных чисел, сравнение и хэширование[2].

Если мы применяем аутентификацию с симметричным ключом, тогда время выполнения протокола будет определяться показателями степеней алгоритма DH. Давайте посмотрим, сколько вычислений для этого потребуется. Каждый из пользователей А и Б должен проделать три операции возведения в степень по модулю с 256-битовым показателем степени. Это потребует около 1150 операций умножения по модулю[3]. Чтобы получить представление о том, насколько велик этот объем вычислений, давайте сравним его с вычислительной стоимостью создания цифровой подписи RSA, если модуль RSA и простое число DH будут иметь одинаковый размер. Для s-битового модуля алгоритм создания цифровой подписи потребует $3s/2$ вычислений, если не использовать китайскую теорему об остатках (Chinese Remainder Theorem — CRT). Использование CRT-представления позволяет сократить объем вычислений в четыре раза, поэтому вычислительная стоимость цифровой подписи RSA для s-битового модуля будет эквивалентна вычислительной стоимости $3s/8$ операций умножения. Мы пришли к интересному заключению: алгоритм RSA относительно медленнее алгоритма DH при использовании модулей большого размера и относительно быстрее при использовании модулей

[2]Все сказанное в этом разделе касается объема вычислений для одного из двух участников протокола. И пользователь А и пользователь Б должны выполнить по три операции возведения в степень в подгруппе DH, по одной генерации аутентификатора, по одной проверке аутентификатора и по набору различных эффективных операций.

[3]Речь идет о простом бинарном алгоритме возведения в степень. При использовании хорошо оптимизированного алгоритма количество операций умножения можно сократить до 1000 или еще меньшего числа.

малого размера. “Переломная” точка находится примерно на уровне 3000 бит. Причина такого эффекта состоит в том, что алгоритм DH всегда использует 256-битовые показатели степеней, в то время как размеры показателей RSA возрастают вместе с размером модуля.

Мы пришли к выводу, что при используемых размерах открытых ключей объем вычислений для алгоритма DH примерно равен объему вычислений для схемы цифровых подписей RSA. Описанная выше нагрузка составляет основной объем вычислений нашего протокола, но она находится в разумных пределах.

Если для аутентификации применяются цифровые подписи RSA, вычислительная нагрузка примерно удваивается. (Проверку подписей RSA можно не учитывать, так как она выполняется очень быстро.) Тем не менее данную нагрузку все еще нельзя назвать чрезмерной. Скорость процессоров стремительно растет изо дня в день, и, как показывает практика, коммуникационные задержки и передача служебной информации отнимают гораздо больше времени, чем криптографические вычисления.

15.9.1 Методы оптимизации

Вычисления, выполняемые в рамках протокола DH, поддаются оптимизации. С помощью эвристики аддитивной цепочки (addition chain heuristics) каждую операцию возведения в степень можно осуществлять путем меньшего количества умножений. Более того, пользователь A должен вычислить значения X^q и X^y. Используя эвристику аддитивной последовательности (addition sequence heuristics), эти значения можно вычислить параллельно, сэкономив при этом около 250 операций умножения. Более подробно это рассматривается в работе Боса (Bos) [10, глава 4].

Существуют также разнообразные приемы, которые позволяют быстрее генерировать случайное число y и вычислять g^y, но они настолько усложняют систему, что мы предпочитаем обходиться без них.

15.10 Сложность протокола

Описанный протокол согласования ключей является прекрасным примером того, почему структура большинства протоколов столь сложна. Даже простой протокол наподобие нашего разрастается на целую страницу, а ведь мы еще не включили в его описание все правила генерации параметров алгоритма DH и все необходимые проверки, выполняемые в рамках схемы аутентификации (на нашем уровне абстракции они неизвестны). Несмотря на это, уследить за всем, что происходит в рамках такого протокола, уже очень трудно. Спецификации более сложных протоколов занимают еще больше места.

Одна система электронных платежей с помощью смарт-карт, над которой когда-то работал Нильс, включала в себя около десятка протоколов. Их описание представляло собой 50 страниц всевозможных символов и спецификаций протоколов, и это с использованием запатентованной, чрезвычайно компактной формы записи! Для описания критических вопросов реализации, касающихся обеспечения безопасности, потребовалось еще 50 густоисписанных страниц.

Полная документация набора криптографических протоколов может занимать сотни страниц. Протоколы усложняются так быстро, что удержать в голове все их аспекты становится крайне трудно. Это очень опасно — при наличии хотя бы малейшего недопонимания в протокол неизбежно вкрадется какая-нибудь "слабинка". Упомянутый выше проект, пожалуй, был слишком сложен для того, чтобы его в полной мере могли понять хотя бы сами разработчики.

Несколько лет спустя Нильс работал с другой системой смарт-карт, предназначенной для коммерческого распространения. Это была хорошо известная, устоявшаяся система, которая широко использовалась различными приложениями для работы со смарт-картами. Как-то раз Мариус Шилдер (Marius Schilder), коллега Нильса, задумался над одним вопросом, а точнее, над огромной "дырой" в безопасности этой системы. Оказалось, что два из ее протоколов взаимодействовали друг с другом, что оказывало негативное влияние на каждый из них. Один протокол вычислял ключ сеанса на основе долговременного ключа смарт-карты (этим он немного напоминал протокол согласования ключей, описанный в данной главе). Второй протокол вычислял значение функции аутентификации на основе все того же долговременного ключа смарт-карты. Приложив немного усилий, мы могли воспользоваться вторым протоколом, чтобы заставить смарт-карту вычислить ключ сеанса и отослать нам половину его бит. Имея на руках половину бит ключа сеанса, взломать оставшуюся часть системы не представляло никакого труда. Просто кошмар! Данная ошибка была исправлена в следующей версии системы, однако она наглядно иллюстрирует проблемы спецификаций больших протоколов.

Реальные системы всегда обладают поистине гигантскими спецификациями протоколов. Электронные коммуникации сложны сами по себе, а добавление криптографических функций и отсутствие доверия еще более усложняют ситуацию. Наш совет: будьте крайне осторожны со сложностью протоколов!

Одна из фундаментальных проблем данной области состоит в отсутствии хороших методов записи протокола, которые бы позволяли разбить его на отдельные модули. В результате компоненты протокола перемешиваются самым непостижимым образом. Мы уже наблюдали подобную ситуацию в данной главе: согласование размера параметров алгоритма DH, обмен ключами DH и аутентификация свалены в одну кучу. Это не просто комбинация несвя-

занных компонентов, а настоящая адская смесь спецификации и реализации. Все это скорее напоминает плохую, очень сложную компьютерную программу без каких-либо признаков модуляризации. Все мы знаем, к чему приводит подобная мешанина, поэтому давно разработали методы разбивки на модули, которые позволяют справиться со сложностью программы. К сожалению, нам не хватает методов модуляризации для протоколов. Если вы ищете тему для долгосрочного научно-исследовательского проекта, данный вопрос может стать хорошим предметом для изучения. С другой стороны, однажды Нильс написал исследовательскую работу именно по этой теме. Работа Нильса была принята. Он получил финансирование на четыре года исследований, но впоследствии отказался от проекта, поскольку осознал, что не имеет ни малейшего представления о том, с какой стороны можно хотя бы приблизиться к этому вопросу. Человек, которому в конце концов перепоручили проект Нильса, тоже не продвинулся в нем ни на шаг, зато провел четыре года ценных исследований совсем в другой области. Отсюда вывод: не все так просто, как кажется. Наверное, сложно получить степень доктора наук, если после многолетних исследований вы заявите комиссии: “Понятия не имею, как это делается”.

15.11 Небольшое предупреждение

Мы постарались, чтобы процесс проектирования нашего протокола выглядел как можно проще. Однако это не должно вводить вас в заблуждение. В действительности проектирование протокола — занятие весьма сложное, требующее огромного опыта. Даже опытный разработчик может легко допустить ошибку. Хотя мы и приложили максимум усилий, чтобы сделать все так, как нужно, не исключена вероятность того, что протокол согласования ключей, разработанный нами в процессе написания этой книги, окажется ненадежным.

15.12 Согласование ключей с помощью пароля

До сих пор предполагалось, что согласование ключей основывается на некоторой системе аутентификации. Зачастую, однако, у пользователей нет никаких способов аутентификации кроме пароля. В подобных случаях мы могли бы просто использовать функцию вычисления MAC, применяя пароль в качестве ключа, но это очень опасно: имея расшифровку протокола (добытую путем прослушивания канала общения), злоумышленник может легко проверить правильность любого конкретного пароля. Достаточно лишь вычислить значение MAC и посмотреть, будет ли оно правильным.

Основная проблема в этой ситуации — нежелание выбирать сложные пароли. Существуют программы, перебирающие распространенные пароли, которые с большой долей вероятности выбирают практически все пользователи. В идеале нам хотелось бы иметь такой протокол согласования ключей, который обладал бы устойчивостью к автономной атаке с использованием словаря.

Подобные протоколы существуют; пожалуй, наиболее известным из них является SRP [96]. Он обладает значительными улучшениями в плане безопасности использования паролей. К сожалению, в вопросах патентования данная сфера деятельности очень напоминает минное поле. Мы так и не смогли найти подходящий протокол согласования ключей, основанный на использовании паролей, который не был бы зажат всевозможными патентами. Университет Стэнфорда (Stanford University) предоставляет бесплатные лицензии на определенное ограниченное применение SRP, однако мы, кажется, нашли по крайней мере еще один патент, который охватывает SRP. Эта ситуация настолько сложна и запутанна, что мы предпочитаем держаться от нее подальше. А если мы считаем, что реализовать подобные протоколы не стоит, то не должны вводить в искушение и наших читателей.

Глава 16

Проблемы реализации. Часть II

Протокол согласования ключей, описанный в предыдущей главе, приводит к появлению еще нескольких проблем реализации.

16.1 Арифметика больших чисел

Все вычисления с открытым ключом подразумевают применение арифметики больших чисел. Как уже отмечалось, правильно реализовать арифметику больших чисел крайне сложно.

Реализация арифметических операций над большими числами практически всегда в той или иной мере зависит от платформы. Эффективность, которой можно добиться, учитывая специфические свойства платформы, слишком высока, чтобы ею пренебрегать. Например, большинство процессоров поддерживают инструкцию "сложение с переносом" (Add With Carry — ADC) для реализации операций сложения многократной точности. Но в C и большинстве других языков высокого уровня эта инструкция недоступна. Выполнение арифметических операций над большими числами в языке высокого уровня обычно происходит в несколько раз медленнее, чем в реализации, оптимизированной для конкретной платформы. Кроме того, вычисления с использованием больших чисел образуют "узкое место" в производительности криптосистемы с открытым ключом, поэтому применение кода, специфичного для конкретной платформы, просто необходимо для повышения эффективности.

Обсуждать вопросы реализации арифметических операций над большими числами мы не будем. Для этого существуют другие книги. В качестве хорошей отправной точки можно порекомендовать книгу Кнута [54]. Нас же, скорее, будет интересовать то, как *тестировать* арифметические операции над большими числами.

В криптографии наши цели отличаются от тех, что преследуются большинством разработчиков. Мы считаем уровень сбоя в 2^{-64} (примерно один к 18 миллионам триллионов) недопустимым, в то время как другие разработчики вполне бы довольствовались и этим. Многие программисты считают уровень сбоя в 2^{-20} (примерно один к миллиону) вполне приемлемым, а то и вовсе замечательным. Нам же, как криптографам, нужно предъявлять более строгие требования к системе, поскольку мы работаем в противоборствующем окружении.

Большинство блочных шифров и функций хэширования достаточно легко поддаются тестированию[1]. Лишь некоторые недочеты реализации приводят к появлению ошибок, которые трудно обнаружить путем тестирования. Если вы ошибетесь в реализации S-матрицы шифра AES, ошибка всплывет уже после нескольких сеансов шифрования. Простое тестирование с помощью последовательности случайных тестов исследует все пути данных в блочном шифре или функции хэширования и быстро обнаружит любые систематические ошибки. Путь кода, который выбирает программа при выполнении шифрования или хэширования, не зависит или практически не зависит от самих данных. Любой приличный набор тестов для симметричной функции шифрования исследует все возможные потоки управления, существующие в реализации.

С арифметикой больших чисел все обстоит иначе. Здесь в большинстве реализаций путь кода, который выбирает программа при выполнении операции, зависит от самих данных. Например, код, осуществляющий последний перенос при сложении, используется крайне редко. Функции деления часто содержат фрагмент кода, который применяется только один раз на каждые 2^{32} или даже 2^{64} операции деления. Ошибка в этой части кода не будет обнаружена с помощью случайного тестирования. Ситуация становится еще хуже при использовании процессоров с бóльшим количеством разрядов. Для 32-разрядного процессора мы в принципе можем выполнить2^{40} случайных теста и ожидать, что каждое 32-битовое слово встретится в каждой части пути данных, но этот способ тестирования совершенно неприемлем для 64-разрядных процессоров.

Из всего изложенного следует, что проводить тестирование арифметических операций над большими числами следует крайне осторожно. Разработчику нужно убедиться в том, что в процессе тестирования программа действительно пройдет все существующие пути кода. Для этого необходимо тщательно подбирать тестовые векторы — занятие, которое требует определенной точности и внимания. Вам понадобится не только использовать все существу-

[1] В качестве наиболее примечательных исключений можно назвать шифры IDEA и MARS, в которых используется отдельный код для тестирования частных случаев.

ющие пути кода, но и перебрать все граничные условия. Например, если у нас есть тест для $a < b$, мы должны протестировать этот же фрагмент кода для $a = b - 1$, $a = b$ и $a = b + 1$ (разумеется, если эти условия вообще могут быть достигнуты).

И без того нелегкое положение еще более усложняется с появлением оптимизации. Поскольку упомянутые выше функции создают узкое место в производительности системы, их всячески пытаются оптимизировать. Это, в свою очередь, приводит к появлению множества частных случаев, путей кода и т.п., что еще более затрудняет тестирование.

Простая арифметическая ошибка может иметь поистине катастрофические последствия для безопасности системы. Вот небольшой пример. Пусть в код, вычисляющий значение цифровой подписи RSA, вкралась небольшая ошибка, которая дает неверный результат при возведении в степень по модулю p, но верный при возведении в степень по модулю q. (Предположим также, что для ускорения этой процедуры применяются CRT-представления.) Подписывая свои сообщения, пользователь A вместо верной цифровой подписи σ отсылает подпись $\sigma + kq$, где k — некоторое число. (Чтобы результат, полученный пользователем A, был верным по модулю q, но неверным по модулю p, он должен иметь вид $\sigma + kq$.) Злоумышленник знает $\sigma^3 \bmod n$ — число, из которого пользователь A извлекает корень третьей степени и которое зависит только от сообщения. Но разность $(\sigma + kq)^3 - \sigma^3$ кратна q. Найдя наибольший общий делитель этой разности и числа n, злоумышленник узнает q и таким образом сможет разложить число n на простые множители. При одной мысли об этом нас охватывает ужас!

Вы спросите, что же в таком случае делать? Во-первых, не пытайтесь реализовать собственную библиотеку для работы с большими числами. Воспользуйтесь какой-нибудь существующей библиотекой. Если вам некуда девать свое время, лучше потратьте его на то, чтобы разобраться в существующей библиотеке и хорошенько ее протестировать. Во-вторых, разработайте действительно хорошие тесты для своей библиотеки. Убедитесь, что вы протестировали все возможные пути кода. В-третьих, вставьте дополнительные тесты в само приложение. Для этого можно воспользоваться несколькими методами.

16.1.1 Вупинг

Метод, описанный в этом разделе, имеет довольно непривычное название — *вупинг (wooping)*. Во время одного жаркого спора между Дэвидом Шомом (David Chaum) и Юрьеном Босом (Jurjen Bos) срочно понадобилось придумать имя для специальной верификационной переменной. В пылу деба-

тов кто-то предложил назвать ее словом “woop”[2]. Впоследствии это название закрепилось и за самим методом. Через некоторое время Бос описал детали вупинга в своей докторской диссертации [10, глава 6], но опустил слово “wooping”, чтобы не шокировать почтенное академическое общество.

Основная идея вупинга заключается в том, чтобы проверять правильность вычислений по модулю с помощью случайно выбранных малых простых чисел. Для большей наглядности это можно представить себе в виде криптографической проблемы. У нас есть библиотека арифметических операций над большими числами, которая пытается “обмануть” нас и предоставить неверные результаты. Наша задача состоит в том, чтобы проверить правильность этих результатов. Простая проверка результатов с помощью все той же библиотеки не принесет никакой пользы, так как ошибка может последовательно повторяться по всему коду библиотеки. Используя вупинг, мы можем проверить правильность библиотечных вычислений, если предположим, что библиотека не пытается “нарочно” навредить нам (в том смысле, что она не портит преднамеренно наши вычисления, выполняемые в рамках верификации).

Вначале мы случайным образом генерируем относительно малое простое число t длиной около 64-128 бит. Значение t не должно быть фиксированным или предсказуемым, но именно для этого и предназначен генератор псевдослучайных чисел. Полученное значение t сохраняется в секрете от остальных участников общения. Затем для каждого большого числа x, которое фигурирует в вычислениях, мы подсчитываем $\tilde{x} := (x \bmod t)$. Число $\tilde{x}$ — это и есть функция $\mathrm{WOOP}(x)$. Значения функции $\mathrm{WOOP}(x)$ имеют фиксированный размер. Обычно они намного меньше соответствующих больших чисел. Поэтому вычисление $\mathrm{WOOP}(x)$ не требует больших расходов системных ресурсов.

Итак, для каждого целого числа x мы сохраняем значение $\mathrm{WOOP}(x)$. Для каждого значения, которое подается на вход нашего алгоритма, мы сразу же вычисляем $\tilde{x}$ как $x \bmod t$. Во всех внутренних вычислениях нашей библиотеки выражения над большими числами будут дублироваться аналогичными выражениями над значениями WOOP. Следовательно, мы сможем подсчитать WOOP результата выражения, подставив в него значения WOOP входных чисел, а не применяя функцию WOOP к конечному результату выражения, как при выполнении операции взятия по модулю.

Обычная операция сложения выполняется как $c := a + b$. Мы, в свою очередь, можем вычислить $\tilde{c}$, используя равенство $\tilde{c} = \tilde{a} + \tilde{b} (\mathrm{mod} t)$. Анало-

[2] На русский язык слово “wooping” (или “whooping”) можно перевести как “вопль”, “улюлюканье” или “приступ кашля”. Действительно, такое название могло родиться только в пылу жарких споров. — *Прим. перев.*

гичным образом можно продублировать и операцию умножения. Мы можем проверять правильность $\tilde{c}$ после каждого сложения или умножения, определяя, выполняется ли соотношение $c \bmod t = \tilde{c}$. Впрочем, гораздо эффективнее проводить все эти проверки в самом конце.

Выполнять операцию сложения по модулю n лишь немного труднее, чем обычную операцию сложения. Вместо того чтобы просто записать $c = (a + b) \bmod n$, мы представляем это как $c = a+b+k \cdot n$, где число k выбрано таким образом, что результат c находится в диапазоне $0, \ldots, n-1$. Это всего лишь другой способ записи взятия числа по модулю. В данном случае k равно 0 или -1 при условии, что числа a и b лежат в диапазоне $0, \ldots, n-1$. Соответствующее выражение с использованием вупинга выглядит как $\tilde{c} = (\tilde{a}+\tilde{b}+\tilde{k}\cdot\tilde{n}) \bmod t$. Отметим, что внутренняя реализация операции сложения по модулю "знает" k. Нам необходимо лишь получить от библиотеки значение k, чтобы можно было вычислить $\tilde{k}$.

Умножение по модулю выполнить гораздо сложнее. В этом случае нужно снова записать $c = a \cdot b + k \cdot n$. Чтобы вычислить $\tilde{c} = \tilde{a} \cdot \tilde{b} + \tilde{k} \cdot \tilde{n} (\mathrm{mod} t)$, мы должны знать $\tilde{a}, \tilde{b}, \tilde{n}$ и $\tilde{k}$. Первые три числа у нас уже есть, а вот значение $\tilde{k}$ нужно каким-то образом извлечь из функции умножения по модулю. Это можно сделать при создании библиотеки "с нуля", однако модифицировать существующую библиотеку будет весьма сложно. В качестве универсального решения можно предложить следующее: вычислить $a \cdot b$, а затем поделить полученный результат на n, используя деление "в столбик". Полученное частное и будет тем самым значением k, которое необходимо для вычисления WOOP(k). Остаток от деления — это результат c. Единственным недостатком универсального метода является крайне медленная скорость его выполнения.

Применив вупинг к операции умножения по модулю, это несложно сделать и для операции возведения в степень по модулю. Большинство функций возведения в степень по модулю вычисляют результат посредством ряда обычных умножений по модулю. (Некоторые из них используют отдельную функцию возведения в квадрат по модулю, но и ее можно продублировать с помощью вупинга как обычную операцию умножения по модулю.) Достаточно лишь сохранять для каждого большого числа x значение WOOP(x) и вычислять WOOP каждого произведения как произведение значений WOOP входных чисел.

Как видите, алгоритмы, использующие вупинг, вычисляют WOOP результата выражения на основе значений WOOP входных чисел. Если значение WOOP одного или нескольких входных чисел окажется неверным, WOOP результата выражения практически наверняка также окажется неверным. Другими словами, ошибка хотя бы в одном значении WOOP распространится и на конечный результат.

По окончании вычислений мы проверяем правильность значений WOOP. Если результат операции над большими числами равен x, нам нужно проверить, что $(x \bmod t) = \tilde{x}$. Если в реализации библиотеки была допущена ошибка, значения $x \bmod t$ и $\tilde{x}$ не совпадут. Мы исходим из предположения, что ошибки библиотеки не были специально подобраны таким образом, чтобы давать нужные результаты в зависимости от того, какое значение t мы выберем. В конце концов, библиотека разрабатывалась задолго до того, как мы выбрали t, а код готовой библиотеки уже не находится под контролем злоумышленника. Легко показать, что любая ошибка библиотеки может быть обнаружена с помощью практически любого значения t. Таким образом, добавление к существующей библиотеке верификации на основе вупинга — прекрасный способ проверки правильности вычислений.

После этого остается лишь найти библиотеку арифметических операций над большими числами со встроенной системой верификации на основе вупинга. Но нам такая библиотека еще не попадалась.

Какой величины должны быть значения WOOP(x)? Это зависит от многих факторов. Для случайных ошибок вероятность того, что применение вупинга не обнаружит ошибку, составляет около $1/t$. К сожалению, в нашем мире не бывает случайностей. Пусть в коде библиотеки содержится программная ошибка, и, к нашему огорчению, злоумышленник знает о ее существовании. Он может выбирать входные данные наших вычислений и не только инициировать ошибку, но и выбирать отклонение этой ошибки от правильного значения. Вот почему значение t должно быть случайным секретным числом; не зная t, злоумышленник не сможет выбрать такую разницу, которая точно была бы пропущена нами в ходе вупинга.

Как бы мы поступили в подобной ситуации на месте злоумышленника? Мы бы, конечно же, инициировали ошибку. Кроме того, мы бы попытались свести отклонение от правильного результата к нулю для наибольшего количества модулей t. Самая простая защита от атак подобного рода — выбрать t так, чтобы оно было простым числом. Если злоумышленник хочет обмануть нас, скрыв ошибку вычислений для шестнадцати 64-битовых простых t, ему придется тщательно выбрать минимум $16 \cdot 64 = 1024$ бит входных данных. Поскольку большинство вычислений ограничивают количество входных битов, которые могут быть выбраны злоумышленником, применение простого t снижает вероятность того, что атака пройдет успешно.

Чем больше значение t, тем лучше. Существует так много простых чисел большого размера, что вероятность успеха атаки быстро снижается с увеличением размера t. Если бы мы, как обычно, стремились придерживаться 128-битового уровня безопасности, длина числа t должна была бы составлять 128 бит или около того.

Отметим, что вупинг ни в коей мере не является основным средством обеспечения безопасности системы; скорее, это вспомогательная мера. Если верификация на основе вупинга окончится неудачей, мы поймем, что в программном обеспечении содержится ошибка, которая должна быть устранена. Такая программа должна аварийно завершить свое выполнение и выдать сообщение о неисправимой ошибке (fatal error). Подобное поведение еще более усложняет проведение повторных атак на систему. На практике мы предлагаем использовать в качестве t 64-битовое случайное простое число. Это значительно сократит расходы по сравнению с использованием 128-битового простого числа и обеспечит достаточный уровень безопасности. Если вы не можете позволить себе 64-битовое t, используйте 32-битовое — это всегда лучше, чем ничего. Особенно хорошо 32-битовые значения t подходят для использования в системах с 32-разрядными процессорами, поскольку к ним могут быть применены прямые инструкции умножения и деления.

Если функции библиотеки будут включать в себя вычисления, позволяющие злоумышленнику выбирать большой объем данных, вам понадобится проверять и промежуточные значения WOOP(x). Эти проверки имеют чрезвычайно простой вид: $(x \bmod t) \stackrel{?}{=} \tilde{x}$. Проверяя промежуточные значения, которые зависят лишь от ограниченного числа битов, выбранных злоумышленником, мы значительно затрудняем атаку последнего на систему вупинга.

Мы всячески поддерживаем использование библиотеки арифметических операций над большими числами со встроенной системой верификации на основе вупинга. Этот, казалось бы, сравнительно простой метод позволяет избежать многих потенциальных проблем безопасности. Кроме того, нам кажется, что гораздо проще один раз добавить к библиотеке поддержку вупинга, нежели реализовать отдельные специфические системы верификации в каждом из приложений, которые будут использовать эту библиотеку.

16.1.2 Проверка вычислений алгоритма DH

Если у вас нет библиотеки с поддержкой вупинга, вам придется работать без нее. Протокол DH, описанный в одной из предыдущих глав, сам по себе содержит целый ряд проверок. В частности, проверку того, что результат не должен быть равен единице и что порядок результата должен быть равен q. К сожалению, эти проверки осуществляются не тем участником, который выполняет вычисления, а тем, кто получает результат этих вычислений. В общем случае мы не хотим отсылать какие-либо ошибочные результаты, так как они могут оказаться полезными для злоумышленника, но в данной ситуации особого вреда это не нанесет. Если результат ошибочен, протокол так или иначе перестанет работать, поэтому ошибку в любом случае заметят. Безопасность протокола нарушается только тогда, когда библиотека арифме-

тических операций вместо g^x возвращает x, но ошибку такого типа без труда обнаружит любое нормальное тестирование.

При необходимости можно использовать алгоритм DH с библиотекой, которая не поддерживает вупинг. Тот тип весьма редких арифметических ошибок, который нас беспокоит, вряд ли позволит злоумышленнику узнать значение x при вычислении g^x. Любая другая ошибка выглядит вполне безобидной, особенно вследствие того, что в алгоритме DH нет долгосрочных секретных данных. Тем не менее там, где это возможно, мы все-таки предпочитаем использовать библиотеку с поддержкой вупинга — просто чтобы чувствовать себя в безопасности.

16.1.3 Проверка шифрования RSA

Алгоритм шифрования RSA более уязвим для нападения злоумышленников, а потому требует дополнительных проверок. Если что-нибудь пойдет не так, злоумышленник может узнать секретные данные, которые вы пытались зашифровать, или даже ваш секретный ключ.

Если реализовать верификацию на основе вупинга невозможно, мы можем порекомендовать вам еще два метода проверки правильности шифрования RSA. Предположим, что реальный алгоритм шифрования RSA вычисляет $c = m^5 \bmod n$, где m — это сообщение, а c — шифрованный текст. Чтобы проверить правильность этой операции, мы могли бы подсчитать $c^{1/5} \bmod n$ и сравнить его с m. Недостаток этого подхода состоит в крайне медленной скорости верификации (что особенно неудобно для такого быстрого алгоритма, как этот). Кроме того, данный метод требует знания закрытого ключа, который при шифровании с помощью RSA обычно недоступен.

Пожалуй, более удачное решение — выбор случайного значения z и проверка того, что $c \cdot z^5 = (m \cdot z)^5 \bmod n$. Здесь нам придется выполнить три операции возведения в пятую степень: вычислить $c = m^5$ и z^5, а также убедиться, что $(mz)^5$ совпадает с $c \cdot z^5$. Такая верификация с высокой степенью вероятности позволит перехватить случайные арифметические ошибки. Выбирая случайное значение z, мы лишаем злоумышленника возможности инициировать ошибки с нужным отклонением. В наших системах алгоритм шифрования RSA применяется только для шифрования случайных значений, поэтому злоумышленнику вообще не удастся никоим образом повлиять на результат верификации.

16.1.4 Проверка цифровых подписей RSA

Проверять цифровые подписи RSA очень просто. Пользователю, подписавшему сообщение, достаточно лишь запустить стандартный алгоритм вери-

фикации подписи. Последний выполняется относительно быстро и с большой долей вероятности позволяет перехватить арифметические ошибки. Каждая процедура вычисления подписи RSA должна верифицировать результат, проверяя только что сгенерированную подпись. Исключений из этого правила нет.

16.1.5 Заключение

Попытаемся немного прояснить ситуацию. Проверки, о которых шла речь в предыдущих разделах, должны выполняться *в дополнение* к обычному тестированию библиотек арифметических операций над большими числами. Они ни в коей мере не заменяют обычного тестирования, которому должен подвергнуться любой элемент программного обеспечения, особенно если речь идет о системе обеспечения безопасности.

Если хотя бы одна из перечисленных выше проверок окончится неудачей, это означает, что программное обеспечение содержит ошибку. Возможных вариантов действия здесь немного. Продолжать работу небезопасно; мы ведь не знаем, что именно привело к возникновению ошибки. Пожалуй, единственное, что стоит сделать в подобной ситуации, — это записать ошибку в журнал и аварийно завершить работу программы.

16.2 Быстрое умножение

Существует множество способов ускорить умножение по модулю таким образом, чтобы системе не приходилось выполнять полное умножение и последующее деление “в столбик”. В протоколах с большим количеством умножений широко используется метод Монтгомери [67]. К сожалению, статья самого Монтгомери слишком трудна для понимания. Более понятное описание этого метода содержится в [25].

В основе метода Монтгомери лежит способ вычисления $(x \bmod n)$, где число x намного больше n. Классический метод деления “в столбик” подразумевает вычитание из числа x подходящих чисел, кратных n. Идея Монтгомери гораздо проще: многократное деление x на 2. Если число x четное, мы делим x на 2, сдвигая его двоичное представление на один бит вправо. Если же x нечетное, мы вначале прибавляем к нему n (что, разумеется, не изменяет значения x по модулю n), а затем делим полученный четный результат на 2. (Данный метод работает только для нечетных n. Напомним, что в наших системах всегда применяются нечетные n. Впрочем, данный метод легко обобщить и на четные значения n.) Если длина n равна k бит, а значение x не превышает $(n-1)^2$, мы выполним k делений на 2. Полученный резуль-

тат всегда будет находиться в диапазоне $0, \ldots, 2n-1$, что уже совсем близко к искомому значению по модулю n.

И все же что-то здесь не так! Ведь мы все время делили на 2, поэтому ответ неверен. В действительности метод Монтгомери дает нам не $(x \bmod n)$, а $x/2^k \bmod n$ для некоторого k. Данная операция выполняется намного быстрее, но "в нагрузку" мы получаем дополнительный множитель 2^{-k}. Существует несколько соображений относительно того, как убрать этот дополнительный множитель.

Одна из самых неудачных идей — просто переопределить протокол, чтобы включить в вычисления дополнительный множитель 2^{-k}. Данный прием никуда не годится, так как приводит к смешению уровней. Он изменяет спецификацию криптографического протокола, адаптируя ее к конкретному методу реализации. Возможно, в будущем вам захочется реализовать этот же протокол на другой платформе без применения метода Монтгомери. (Например, это может быть медленная платформа, имеющая сопроцессор для работы с большими числами, который выполняет умножение по модулю напрямую.) В этом случае наличие в спецификации протокола множителей 2^{-k} будет только мешать.

Стандартный прием, к которому прибегают в данной ситуации, — изменить представление чисел. Число x будет внутренне представлено как $x \cdot 2^k$. Если требуется перемножить x и y, мы применим умножение Монтгомери к соответствующим внутренним представлениям. Перемножив эти числа, мы получим $x \cdot 2^k \cdot y \cdot 2^k$, но вследствие взятия числа $x \cdot 2^k \cdot y \cdot 2^k$ по модулю n у нас появится множитель 2^{-k}. Конечный результат операции умножения будет выглядеть как $x \cdot y \cdot 2^k$, а это и есть внутреннее представление произведения xy. Таким образом, дополнительные расходы, связанные с применением метода Монтгомери, включают в себя стоимость преобразования входных данных во внутреннее представление (умножение на 2^k) и стоимость обратного преобразования выходных данных для получения фактического результата (деление на 2^k). Первое преобразование может быть выполнено путем умножения Монтгомери числа x на $(2^{2k} \bmod n)$. Второе преобразование, в свою очередь, можно выполнить, сдвинув число $x \cdot y \cdot 2^k$ еще на k бит вправо, поскольку это будет эквивалентно делению на 2^k. Конечный результат метода Монтгомери не обязательно будет меньше n, однако в большинстве случаев можно показать, что он будет меньше $2n-1$. В подобных случаях для получения корректного результата достаточно лишь небольшой проверки и (при необходимости) одного вычитания числа n.

На практике взятие числа по модулю методом Монтгомери выполняется не по битам, а по словам. Предположим, что процессор оперирует w-битовыми словами. Для заданного числа x необходимо найти такое малое z, чтобы наименее значимое слово числа $x + zn$ равнялось нулю. Можно пока-

зать, что искомое z является одним словом и может быть вычислено путем умножения наименее значимого слова числа x на слово-константу, которое зависит только от n. Если наименее значимое слово числа $x + zn$ будет равно нулю, мы можем поделить это число на 2^w, сдвигая двоичное представление сразу на слово вправо. Это намного быстрее, нежели сдвигать его по биту.

16.3 Атаки с использованием побочных каналов

Мы уже затрагивали тему тайминг-атак и других атак с использованием побочных каналов в разделе 9.5. Причина, по которой мы были столь немногословны, состоит отнюдь не в их безобидности. Напротив — наибольшую опасность для криптографических систем с открытым ключом представляют именно тайминг-атаки.

Как уже отмечалось, в криптосистемах с симметричным ключом программа для каждого вычисления использует простой путь кода. Следовательно, время, необходимое для выполнения одной операции шифрования с помощью блочного шифра, будет постоянным. Вернее, относительно постоянным — существует масса мелких операций с участием кэша, время выполнения которых варьируется самым неожиданным образом. Впрочем, пока еще никто не пытался нападать на криптосистемы с симметричным ключом, используя разницу во времени кэширования, хотя теоретически такие атаки все же возможны.

Некоторые шифры требуют программных реализаций, применяющих различные пути кода для обработки тех или иных частных случаев. В качестве примера таких шифров можно привести IDEA [60, 61] и MARS [13]. Другие шифры используют операции процессора, время выполнения которых зависит от обрабатываемых данных. У некоторых процессоров время выполнения умножения (используемого шифрами RC6 [77] и MARS) или циклического сдвига битов (используемого шифрами RC6 и RC5), величина которого зависит от данных, во многом определяется спецификой входных данных. Все это делает возможным осуществление тайминг-атак. Отметим, что функции, с которыми мы работали при написании этой книги, подобные типы операций не используют.

Криптографические системы с открытым ключом еще более уязвимы по отношению к тайминг-атакам. Путь кода, который выбирает программа для выполнения операций над открытым ключом, зачастую зависит от самих данных. Это практически всегда приводит к тому, что обработка разных данных занимает разное время. Информация о времени выполнения той или иной операции, в свою очередь, может привести к осуществлению атаки. Представьте себе защищенный Web-сервер электронной коммерции. В рам-

ках процедуры согласования по протоколу SSL сервер должен расшифровать сообщение RSA, выбранное клиентом. Злоумышленник может подключиться к серверу, попросить его расшифровать выбранное значение RSA и подождать ответа. Точное время, которое уйдет у сервера на расшифровку этого значения, может снабдить злоумышленника ценной информацией. Зачастую оказывается, что если конкретный бит ключа равен единице, то входные данные из множества A обрабатываются немного быстрее, чем входные данные из множества B, а если этот бит ключа равен нулю, то скорость обработки данных будет одинаковой. Злоумышленник может использовать это различие для нападения на систему. Он генерирует миллионы запросов с входными данными из множеств A и B и пытается найти статистическую разницу между временем отклика сервера на запросы с данными первой и второй групп. Существует множество других факторов, которые влияют на точное время отклика, однако, используя достаточное количество запросов, это влияние можно усреднить. В конце концов у злоумышленника наберется достаточно данных, на основании которых он сможет понять, отличается ли время отклика для множеств A и B. Это позволит получить один бит информации о ключе, после чего злоумышленник сможет осуществить аналогичную атаку на следующий бит ключа.

Описанная ситуация выглядит несколько неправдоподобно. Между тем атаки такого рода были успешно проведены в лабораторных условиях, а значит, с тем же успехом могут применяться и на практике [55].

16.3.1 Меры предосторожности

Существует несколько способов защитить свою систему от тайминг-атак. Наиболее очевидный из них — гарантировать, что время выполнения каждого вычисления будет фиксированным. К сожалению, это требует особого подхода к разработке всей библиотеки в целом. Еще печальнее то, что существуют источники, влияющие на длительность операций, поведение которых практически не поддается контролю. Некоторые процессоры выполняют умножение одних значений быстрее, чем других. Многие процессоры обладают сложными системами кэш-памяти, поэтому, если модель доступа к памяти зависит от закрытых данных, время обращения к кэшу, а следовательно, время выполнения операции также будет варьироваться. Избавить криптосистемы с открытым ключом от всех нюансов, связанных со временем выполнения тех или иных действий, практически невозможно. Поэтому нам нужны другие решения.

Для борьбы с тайминг-атаками в конец каждого вычисления можно было бы добавить случайную задержку. Это, однако, не устраняет разницу во времени выполнения операции, а лишь искажает ее с помощью “шума”, вы-

званного задержкой. Злоумышленник, у которого есть возможность провести большое количество измерений (например, добраться к вашей машине), может усреднить полученные результаты и определить примерную величину случайной задержки. Точное количество попыток, необходимое злоумышленнику для осуществления тайминг-атаки, зависит от порядка разницы во времени выполнения, которую он стремится обнаружить, и порядка случайной задержки, добавляемой в конец операции. В реальных системах всегда присутствует множество шумов, поэтому злоумышленник, пытающийся осуществить тайминг-атаку, все равно проводит усреднение результатов. Единственным вопросом для злоумышленника остается отношение объема основного сигнала к объему шумов.

Еще один метод борьбы с тайминг-атаками — стандартизация времени выполнения операции. В процессе разработки системы мы выбираем длительность d, которая превышает любое потенциальное время выполнения данного вычисления. Затем мы замечаем момент времени t, в который было начато вычисление, и выжидаем по окончании вычисления до момента времени $t + d$. Данная идея подразумевает несколько расточительное использование системных ресурсов, но вообще-то она неплоха. Нам нравится это решение, но оно защищает систему только от тайминг-атак в "чистом" виде. Если злоумышленник способен измерять выдаваемое компьютером радиоизлучение или, скажем, потребление памяти, он, вероятно, сможет обнаружить задержку между фактическим окончанием вычисления и моментом времени $t + d$. Это, в свою очередь, позволит злоумышленнику осуществлять тайминг-атаки наравне с атаками других типов. Тем не менее для проведения атаки, основанной на измерении радиоизлучения, злоумышленник должен находиться в непосредственной близости от атакуемой машины. Это значительно сокращает вероятность угрозы по сравнению с тайминг-атаками, которые с успехом можно осуществлять и через Internet.

Помимо всего прочего, для защиты от тайминг-атак можно применять методы, основанные на использовании слепых подписей [55]. Такие методы позволяют скрыть практически все различия во времени выполнения некоторых типов вычислений.

Идеального решения проблемы тайминг-атак не существует. Мы просто не в состоянии защитить все продаваемые компьютеры от действительно изощренных атак, подобных основанным на измерении радиоизлучения. Тем не менее, несмотря на невозможность создания идеального решения, мы можем создать *хорошее* решение. Просто будьте предельно внимательны к времени выполнения операций над открытым ключом. Существует и более удачное решение. Оно состоит в том, чтобы сделать постоянным время выполнения не только операций над открытым ключом, но и всей транзакции в целом. Другими словами, нам следует зафиксировать не только время выполнения

каждой операции с участием открытого ключа, но и время, которое проходит между принятием запроса и отправкой ответа. Если запрос поступает в момент времени t, мы должны отправить ответ в момент времени $t + C$, где C — некоторая константа. Впрочем, чтобы окончательно исключить всякую возможность утечки информации о времени выполнения вычислений, нам следовало бы гарантировать, что ответ на запрос действительно будет готов к моменту времени $t + C$. Для достижения этой цели частоту принятия входных запросов, вероятно, понадобится ограничить до некоторого фиксированного верхнего уровня.

16.4 Протоколы

В контексте реализации криптографические протоколы не очень отличаются от коммуникационных. Самый простой метод реализации протокола — отслеживание его состояния с помощью программного счетчика и поочередное выполнение всех необходимых шагов протокола. Если в подобной системе не используется многопоточность, на время ожидания ответа ее жизнедеятельность замирает. Поскольку ответ не всегда приходит вовремя (а иногда не приходит вообще), данную идею нельзя назвать удачной.

Более правильно сохранять явное состояние протокола и обновлять его каждый раз при получении нового сообщения. Реализовать данный подход несколько сложнее, но в то же время он обеспечивает гораздо больше гибкости, нежели предыдущий.

16.4.1 Выполнение протоколов поверх безопасного канала общения

Многие криптографические протоколы выполняются поверх небезопасных каналов общения. Иногда, впрочем, имеет смысл запускать протокол поверх безопасного канала общения. В качестве примера можно привести ситуацию, когда каждый пользователь взаимодействует с центром распространения ключей (ЦРК) по безопасному каналу общения. ЦРК применяет простой протокол для распространения ключей между пользователями системы, чтобы те могли общаться друг с другом. (Что-то подобное выполняет протокол Kerberos.) Если вы применяете криптографический протокол для взаимодействия с участником, с которым вы уже обменялись ключом, следует использовать полную функциональность безопасного канала общения. В частности, требуется реализовать защиту от атак воспроизведения. Это не составит труда, зато позволит предотвратить большое количество атак на криптографические протоколы.

Иногда наличие безопасного канала общения позволяет сократить функциональность протокола. Например, если безопасный канал общения обеспечивает защиту от атак воспроизведения, самому протоколу этого делать не нужно. Тем не менее старое доброе правило модуляризации гласит, что протокол должен минимизировать свою зависимость от безопасного канала общения.

В дальнейшем при обсуждении реализации нашего протокола будем предполагать, что протокол выполняется поверх небезопасного канала общения. Некоторые моменты нашего обсуждения будут не вполне применимы к выполнению протокола поверх безопасного канала общения, но предложенные решения не повредят и в этом случае.

16.4.2 Получение сообщения

Когда состояние протокола получает новое сообщение, нам нужно провести несколько проверок. Вначале следует убедиться, принадлежит ли полученное сообщение заданному протоколу. Каждое сообщение должно начинаться описанными ниже полями.

- **Идентификатор протокола.** В точности идентифицирует протокол и его версию. Особо важную роль здесь играют идентификаторы версий.
- **Идентификатор экземпляра протокола.** Идентифицирует, к какому экземпляру протокола принадлежит данное сообщение. Пользователи А и Б могут одновременно запустить два сеанса протокола согласования ключей, и мы не хотим их перепутать.
- **Идентификатор сообщения.** Идентифицирует сообщение в рамках заданного протокола. Самый легкий способ идентификации — просто пронумеровать сообщения.

В зависимости от ситуации, некоторые из перечисленных идентификаторов могут быть неявными. Например, для протоколов, которые выполняются поверх собственного соединения TCP, номер порта и связанный с ним сокет однозначно идентифицируют экземпляр протокола. Отметим также, что идентификатором протокола и информацией о его версии достаточно обменяться лишь однажды. С другой стороны, пользователи должны как минимум один раз обменяться этими сведениями, чтобы впоследствии они охватывались всеми процедурами аутентификации или проверки цифровых подписей, присутствующими в протоколе.

Проверив идентификаторы протокола и его экземпляра, мы узнаем, какому состоянию протокола следует переслать данное сообщение. Предположим, состояние протокола только что отослало сообщение номер $n - 1$ и ожидает прихода сообщения номер n.

Если полученное сообщение действительно имеет номер n, значит, все в порядке. Просто обработайте его в соответствии с правилами протокола. Но что, если номер сообщения будет другим?

Если номер полученного сообщения больше n или меньше $n-1$, это свидетельствует о том, что в системе не все в порядке. Подобное сообщение не могло быть сгенерировано системой, а значит, является подделкой. Содержимое такого сообщения следует проигнорировать.

Если полученное сообщение имеет номер $n-1$, значит, отправленное сообщение n могло быть утеряно. По крайней мере это могло случиться, если протокол выполняется поверх ненадежного транспортного уровня. Поскольку мы хотим минимизировать зависимость протокола от других частей системы, то предполагаем именно наличие ненадежного канала общения.

Вначале проверьте, совпадает ли только что полученное сообщение $n-1$ с сообщением $n-1$, полученным ранее. Если они отличаются, новое сообщение следует проигнорировать. Отправка второго ответа приведет к нарушению безопасности многих криптографических протоколов. Если же сообщения абсолютно идентичны, просто еще раз отправьте сообщение n. Разумеется, оно должно быть полностью идентичным предыдущему сообщению n, которое было отослано ранее.

Если в соответствии с одним из перечисленных выше правил сообщение было проигнорировано, необходимо принять еще одно решение. Нужно ли инициировать аварийное завершение протокола? Ответ на этот вопрос в некоторой степени зависит от используемого приложения и самой ситуации. Если протокол выполнялся поверх безопасного канала общения, наличие подобных сообщений свидетельствует о взломе безопасного канала общения либо о злонамеренных действиях второго участника. И в том и в другом случае выполнение протокола и канала общения должно быть аварийно завершено. Просто удалите состояние протокола и состояние канала общения вместе с ключом последнего.

Если протокол выполняется поверх небезопасного канала общения, тогда отправка каждого из отброшенных сообщений могла быть делом рук злоумышленника, пытающегося вмешаться в выполнение протокола. В идеале нам нужно было бы проигнорировать сообщения злоумышленника и просто завершить протокол. Это, конечно же, не всегда возможно. Например, если поддельное сообщение $n-1$ дойдет до нас первым, мы отправим ответ. Если позднее мы получим "настоящее" сообщение $n-1$, то будем вынуждены его проигнорировать. Разрешить данную проблему невозможно, поскольку мы не можем отправить второй ответ, не подвергая риску безопасность системы. Но мы не знаем, какое из двух полученных сообщений с номером $n-1$ было настоящим. Поэтому следует просто занести в журнал сведения об ошибке, вызванной получением второго сообщения с номером $n-1$, и про-

должать выполнение протокола как обычно. Если сообщение, на которое мы ответили, было создано злоумышленником, выполнение протокола рано или поздно даст сбой, поскольку криптографические протоколы специально разрабатываются таким образом, чтобы помешать злоумышленнику успешно завершить протокол с одним из его участников.

16.4.3 Время ожидания

Выполнение каждого протокола включает в себя ожидание ответа. Если вы не получаете ответ на сообщение в течение определенного периода времени, вы можете повторно отослать последнее сообщение. После нескольких безуспешных попыток отослать сообщение с этой идеей придется расстаться. Зачем продолжать выполнение протокола, если вы не можете взаимодействовать со вторым участником?

Самый простой способ реализовать время ожидания — это отсылать состоянию протокола сообщения об истекшем времени. Для контроля за временем ожидания можно применять таймеры, значения которых явно устанавливаются протоколом, или же сообщения, которые отсылаются каждые несколько секунд или около того.

Одним из распространенных типов атак является отправка на конкретный компьютер огромного количества сообщений типа “начало протокола”. Каждый раз, когда система получает подобное сообщение, она инициирует новое состояние выполнения протокола. Через несколько миллионов подобных сообщений у компьютера заканчивается память и система “зависает”. В качестве хорошего примера подобной атаки можно привести *синхронную атаку (SYN flood attack)*. Защититься от таких атак очень сложно, но они показывают, как важно удалять устаревшие состояния протокола. Если выполнение протокола замерло и не возобновляется в течение слишком долгого времени, его состояние необходимо удалить.

Величина времени ожидания постоянно является предметом ожесточенных споров. По своему опыту мы знаем, что пакет, отправленный по Internet, доходит до адресата примерно через секунду или же не доходит вообще. Если в течение пяти секунд на сообщение не было получено ответа, имеет смысл отправить его еще раз. Трех попыток должно быть вполне достаточно; если уровень потери сообщений настолько высок, что мы потеряли четыре последовательных сообщения за 15 с, данное соединение вряд ли годится для выполнения протокола. Мы предпочитаем проинформировать пользователя о проблеме уже через 20 с, а не заставлять его пребывать в напрасном ожидании ответа несколько минут или более.

Часть III

Управление ключами

Глава 17

Часы

Прежде чем переходить к подробному обсуждению управления ключами, необходимо рассмотреть еще одну примитивную функцию: часы. На первый взгляд это совершенно *не* криптографическая функция. Тем не менее, поскольку в криптографических системах часто используется текущее время, нам определенно не обойтись без надежных часов.

17.1 Зачем нужны часы

В криптографии часы имеют несколько областей применения. Функции управления ключами часто привязаны к предельным срокам. Кроме того, текущее время может пригодиться как в качестве уникального значения, так и для полного упорядочения событий. Обсудим каждую из этих областей применения подробнее.

17.1.1 Срок действия

Довольно часто требуется ограничить срок действия документа. В реальном мире ограниченный срок действия имеют многие вещи — чеки, авиабилеты с открытой датой, ваучеры, купоны и даже авторские права. Стандартный способ ограничить срок действия цифрового документа — включить время истечения срока действия в сам документ. Но, чтобы проверить, не истек ли срок действия документа, необходимо знать текущее время. Вот где на помощь приходят часы.

17.1.2 Уникальные значения

Еще одной удобной особенностью текущего времени является возможность применять его в качестве уникального значения в рамках одного ком-

пьютера. В предыдущих главах не раз упоминались оказии. Важным свойством оказии является то, что никакое ее значение не используется дважды, по крайней мере на протяжении заданного периода времени. Иногда этот период времени ограничен, например использование оказии в рамках сеанса безопасного канала общения. В этой ситуации оказию можно генерировать с помощью счетчика. В других случаях оказия должна оставаться уникальной до следующей перезагрузки компьютера. Существует два универсальных способа генерации значений оказии. Первый заключается в использовании текущего времени часов совместно с некоторым механизмом, гарантирующим, что одно и то же значение времени никогда не будет использовано дважды. Второй способ подразумевает использование генератора псевдослучайных чисел, который рассматривается в главе 10, "Генерация случайных чисел". Недостатком использования случайной оказии является то, что она должна быть очень большой. Чтобы достигнуть уровня безопасности в 128 бит, следовало бы использовать 256-битовую оказию. Не все криптографические функции поддерживают оказии таких размеров. Более того, на некоторых платформах очень сложно реализовать генератор псевдослучайных чисел. В качестве замечательного альтернативного способа генерации оказий можно предложить надежные часы.

17.1.3 Монотонность

Время, как известно, всегда идет вперед, оно никогда не останавливается и не обращает свой ход вспять. Это весьма полезное свойство применяется в некоторых криптографических протоколах. Включив текущее время в криптографический протокол, мы мешаем злоумышленнику пересылать старые сообщения, сгенерированные в рамках предыдущих сеансов работы протокола, под видом новых. Действительно, время, закодированное в тех сообщениях, не попадает во временны́е рамки текущего сеанса работы протокола.

Еще одной важной областью применения часов является аудит и ведение журналов. В любой системе управления транзакциями очень важно вести журнал всего происшедшего. Если в системе когда-нибудь возникнет спорная ситуация, журналы аудита позволят отследить точную последовательность происшедших событий. В каждую запись о событии очень важно включать точное время; не имея временно́й метки, сложно определить, какие события принадлежали к одной и той же транзакции и в каком порядке они выполнялись. Если часы различных компьютеров хорошо синхронизированы и их показания не отклоняются (или практически не отклоняются) друг от друга, наличие временны́х меток позволяет соотносить события из различных журналов, находящихся на разных машинах.

17.1.4 Выполнение транзакций в режиме реального времени

Следующий пример основан на опыте работы Нильса с системами электронных платежей. Чтобы обслуживать платежи “на лету”, банку нужна система финансовых транзакций, работающая в режиме реального времени. Чтобы обеспечить возможность аудита, подобная система должна соответствовать нескольким требованиям. Прежде всего, в системе должна существовать строгая последовательность транзакций. Для каждых двух транзакций A и B очень важно знать, какая из них была выполнена первой, поскольку результат одной из них может зависеть от того, была ли уже выполнена другая. Самый простой способ зафиксировать последовательность выполнения транзакций — это присвоить каждой из них временну́ю метку. Данный подход, однако, сработает только в том случае, если у нас будут надежные часы.

Ненадежные часы могут показать неправильное время. Если часы случайно станут показывать время в прошлом, ничего не произойдет: легко проверить, что текущее время больше временно́й метки последней выполненной транзакции. Настоящие проблемы возникнут тогда, когда часы станут показывать время в будущем. Предположим, в течение получаса выполнения транзакций часы показывали 2020 год. Мы не можем просто изменить временны́е метки этих транзакций: изменять финансовые записи вручную не допускается. Мы также не можем выполнять новые транзакции, временны́е метки которых будут находиться ранее 2020 года, так как это нарушило бы порядок выполнения транзакций, который определяется именно временно́й меткой. Данную проблему, разумеется, можно решить, однако иметь надежные часы все же предпочтительнее.

17.2 Использование микросхемы датчика времени

Большинство настольных компьютеров содержат микросхему датчика времени и маленькую батарею. Это самые настоящие маленькие цифровые часы. Именно с их помощью компьютер определяет, сколько сейчас времени, когда его включают после ночного “сна”. Так почему же просто не воспользоваться временем этих часов?

Микросхема датчика времени вполне пригодна для обычной жизни, однако в системе безопасности к часам предъявляются более строгие требования. Будучи одним из элементов системы безопасности, часы должны показывать точное время, даже если злоумышленник попытается ими манипулировать. Вторая причина связана с последствиями, к которым может привести применение испорченных часов. В обычной жизни часы, показывающие неправиль-

ное время, могут раздражать, но не более. Если же часы являются частью системы безопасности, их сбой может привести к значительным повреждениям.

Датчики времени, встроенные в бóльшую часть аппаратного обеспечения, не обладают той степенью надежности и защищенности, которая нам нужна. На протяжении последних 10 лет мы несколько раз сталкивались со сбоями микросхемы датчика времени. Что еще хуже, эти сбои происходили спонтанно, без всякого участия злоумышленников, пытающихся повредить наши часы. Большинство сбоев подобного рода довольно просты. В старом компьютере батарея может "сесть" и часы остановятся либо будут "сброшены" на 1980 год. Кроме того, в один прекрасный день вы можете включить компьютер и обнаружить, что часы сами собой перевелись на какой-то день 2028 года.

Помимо случайных ошибок датчиков времени, необходимо учитывать и возможность осуществления активных атак. Кто-нибудь может попытаться манипулировать часами в своих интересах. В зависимости от специфики компьютера, изменить показания датчика времени может быть проще или сложнее. В одних системах для изменения времени пользователю требуются привилегии администратора, в других перевести часы может каждый.

17.3 Виды угроз

Существует несколько типов атак на систему с часами.

17.3.1 Перевод часов назад

Предположим, злоумышленнику удалось перевести часы на какое-нибудь произвольное время в прошлом. Это может повлечь за собой самые разные неприятности. Компьютер ошибочно предполагает, что находится в прошлом. Возможно, когда-то злоумышленник был нанят на временную работу и имел доступ к определенным данным, но сейчас срок действия этого доступа истек. Компьютер, часы которого показывают неправильное время, может предоставить злоумышленнику доступ к важным данным. Эта проблема возникает всякий раз, когда у пользователя отбирают какие-нибудь права доступа. В зависимости от архитектуры остальных частей системы, перевод часов назад может восстановить этот доступ.

Не менее интересным объектом атак являются автоматизированные задачи. Компьютер, стоящий в отделе кадров, автоматически начисляет зарплату в конце каждого месяца, используя прямое зачисление на депозит по платежной ведомости. Выполнение автоматизированных задач наподобие этой инициирует программа, которая проверяет текущее время и просматривает список назначенных задач. Постоянный перевод часов назад может привести

к повторяющемуся выполнению одних и тех же задач. Если выполнение задачи должно начаться в полночь, злоумышленник переводит часы на 23:55 (11:55 p.m.) и ждет, пока система начнет выполнять задачу. По окончании выполнения задачи злоумышленник снова переводит часы назад. Это может повторяться до тех пор, пока запасы средств на банковском счете компании не будут исчерпаны.

Еще одна проблема, касающаяся перевода часов назад, свойственна финансовым системам. В подобных системах очень важно знать правильное время выполнения транзакции, поскольку расчеты процентов дают различные результаты в зависимости от того, когда была выполнена транзакция. Если на вашей кредитной карточке находится большая сумма, почему бы не убедить компьютер вашего банка в том, что осуществленный несколько минут назад электронный платеж на самом деле произошел на шесть месяцев раньше? Это бы избавило вас от уплаты процентов за целых шесть месяцев.

17.3.2 Остановка часов

Каждый разработчик живет с инстинктивным пониманием того, что время не стоит на месте. Это негласное предположение слишком очевидно даже для того, чтобы фиксировать его в документации. Системы, которые создают разработчики, основаны на нормальном поведении времени. Но, если часы останавливаются, время будто бы замирает. Многие процессы не будут завершены, а многие системы начнут вести себя непредсказуемо.

Наиболее простые проблемы, вызываемые остановкой часов, — это ошибки наподобие неправильного указания времени в отчетах и журналах аудита. Точное время транзакции может иметь большие финансовые последствия, и отправка официального документа с неправильной датой и временем может привести к серьезным осложнениям.

Другие проблемы могут касаться программ, отображающих изменение данных в режиме реального времени. Предположим, разработчик графического интерфейса использует простую систему для отображения текущего положения дел на дисплее биржевого брокера. Каждые 10 секунд содержимое дисплея обновляется. Но не все отчеты о выполнении финансовых транзакций поступают с одинаковой скоростью из-за различных задержек. Простое отображение полученных данных может создать несогласованное представление о сложившейся финансовой ситуации. Возможно, об одной части транзакции уже был подан отчет, а о другой еще нет. Поступление денежных средств на банковский счет может отобразиться еще до фактического перемещения акций. Бухгалтеры не любят получать отчеты, суммы в которых не сходятся.

Во избежание подобных проблем разработчик применяет хитрый прием. Каждому отчету о выполнении финансовой транзакции присваивается вре-

менна́я метка, после чего он помещается в базу данных. Чтобы создать согласованный отчет, разработчик выбирает конкретный момент времени и отображает финансовую ситуацию, соответствующую тому моменту. Например, если отчет от самой медленной системы поступает с задержкой в пять секунд, разработчик отображает положение дел, которое существовало семь секунд назад. Это немного увеличивает задержку, с которой данные поступают на дисплей брокера, однако гарантирует их согласованность. Если же часы неожиданно остановятся, на дисплее будет постоянно отображаться одна и та же ситуация: положение дел по состоянию на семь секунд назад относительно времени, которое показывают испорченные часы. Катастрофа!

17.3.3 Перевод часов вперед

Перевод часов вперед заставляет компьютер думать, что он живет в будущем. Это приводит к появлению множества простых атак типа "отказ в обслуживании". Если часы будут переведены на четыре года вперед, все транзакции с участием кредитных карточек будут неожиданно отклонены, так как карточки окажутся недействительными. Вы также не сможете заказать авиабилеты через Internet, поскольку расписания полетов на четыре года вперед еще не существует.

Самые ожесточенные баталии на аукционах eBay происходят в последние секунды торгов. Если вам удастся перевести часы eBay немного вперед, вы сможете неожиданно выбить из игры массу потенциальных покупателей и приобрести товар по более низкой цене.

Наш друг однажды столкнулся с подобной проблемой при разработке биллинговой системы. Из-за ошибки в программном обеспечении часы перескочили на 30 лет вперед. Как следствие, биллинговая система начала предъявлять своим клиентам горы неоплаченных счетов за 30 лет. В данном случае это не привело к прямым финансовым потерям, но если бы система автоматически снимала деньги с банковских счетов или кредитных карточек, результат был бы весьма плачевным. Разумеется, это никак не способствовало установлению хороших отношений с клиентами.

Существуют и прямые угрозы безопасности, связанные с переводом часов вперед. Довольно часто определенные данные должны храниться в секрете до заданного момента времени, по достижении которого их следует открыть широкой общественности. В автоматизированной системе перевод часов вперед позволяет получить доступ к таким данным. Если в секрете хранится предупреждение о том, что прибыль компании может оказаться ниже ожидаемой, кто-нибудь изрядно наживется на продаже акций, получив преждевременный доступ к этим данным.

17.4 Создание надежных часов

Простое решение проблемы создания надежных часов нам не известно. Мы можем лишь предложить несколько методов и идей. Детали этого процесса столь существенно зависят от конкретного рабочего окружения и анализа рисков, что универсальный ответ невозможен.

Большинство компьютеров имеют или могут реализовать некое подобие счетчика, который начинает функционировать при загрузке компьютера. Это может быть счетчик количества циклов процессора, счетчик обновлений или что-нибудь подобное. Такой счетчик может применяться для отслеживания времени, прошедшего с момента последней перезагрузки компьютера. Разумеется, его нельзя рассматривать как часы, так как он не предоставляет никакой информации о текущем времени. Тем не менее он может использоваться для измерения времени, прошедшего между событиями, которые случились с момента последней перезагрузки.

Основное назначение счетчика (по крайней мере применительно к проблеме создания надежных часов) состоит в проверке датчика времени на предмет обнаружения случайных ошибок. Если датчик времени работает неправильно, его показания начнут расходиться со счетчиком. Наличие данного свойства может применяться в качестве признака некоторых ошибочных режимов работы датчика времени, а обнаружить расхождение совсем несложно. Если системное время было изменено авторизованным пользователем, соответствие между временем часов и значениями счетчика придется подкорректировать.

Вторая простая проверка состоит в том, чтобы отслеживать время последнего выключения компьютера или последней записи данных на диск. Часы не должны перескакивать назад. Если компьютер неожиданно загрузится и покажет 1980 год, с датчиком времени, очевидно, что-то не так. Кроме того, мы можем помешать часам “перескакивать” на большой период времени вперед. Большинство компьютеров загружаются, как минимум, раз в неделю. Систему можно настроить таким образом, чтобы пользователь подтверждал правильность даты, если компьютер не включался больше недели[1]. Это позволило бы перехватить ошибку, связанную с переводом часов более чем на неделю вперед.

Существуют и другие методы проверки времени. Чтобы узнать точное время, можно обратиться к серверу времени в Internet или intranet. Многие системы широко используют протоколы синхронизации времени, такие, как NTP [65] или SNTP [66]. Некоторые из этих протоколов даже обеспечивают

[1] Поскольку большинство пользователей щелкают на кнопке OK, даже не удосужившись прочитать сообщение, гораздо лучше попросить пользователя самого ввести текущую дату, не показывая ему, какая дата выставлена на встроенном датчике времени.

аутентификацию временны́х данных, чтобы злоумышленник не смог обмануть компьютер. Разумеется, для выполнения подобной аутентификации системе требуется некоторая инфраструктура ключей. В качестве общего ключа, используемого совместно с сервером времени, можно было бы применять ключ симметричного шифрования и настраивать его вручную, но настройка ключей вручную — занятие не из приятных. Обмен ключами с сервером времени можно было бы проводить и с помощью инфраструктуры открытого ключа (PKI). К сожалению, как отмечается в главе 19, "PKI: красивая мечта", большинству систем PKI нужны часы, что напоминает старую добрую историю о курице и яйце. Будьте крайне осторожны, полагаясь на криптографическую защиту, которую предоставляют протоколы синхронизации времени. В этом случае от безопасности протокола может зависеть безопасность всей вашей системы, а мы еще не видели ни одного криптографического обзора подобных протоколов.

17.5 Проблема одного и того же состояния

Мы подошли к серьезной проблеме, свойственной некоторым аппаратным платформам. Речь идет о небольших встроенных компьютерах наподобие дверного замка или удаленного устройства считывания смарт-карт. Такие компьютеры, как правило, состоят из небольшого процессора, небольшого объема оперативной памяти, энергонезависимой памяти (например, флэш-памяти), в которой хранится программа, нескольких каналов связи и дополнительного оборудования, специфичного для конкретной задачи.

В подобные устройства редко включают датчики времени. Добавление датчика времени требует установки дополнительной микросхемы, кварцевого резонатора и, что самое важное, батареи. Помимо увеличения стоимости устройства добавление батареи усложняет его. Необходимо заботиться о том, чтобы заряд батареи не закончился. Батареи весьма чувствительны к колебаниям температуры, а наличие в некоторых батареях токсичных веществ может привести к проблемам с доставкой оборудования. Все это и послужило причиной того, почему большинство малых компьютеров не имеют датчика времени.

Малый компьютер всегда загружается в одном и том же состоянии. Он считывает одну и ту же программу из одной и той же энергонезависимой памяти, инициирует аппаратное обеспечение и начинает выполнение операций. Поскольку наша книга посвящена криптографии, будем исходить из предположения, что для взаимодействия малого компьютера с другими частями системы применяется некий криптографический протокол. Вот тут-то и начинаются проблемы: при отсутствии датчика времени или аппаратного гене-

ратора случайных чисел встроенная система всегда будет демонстрировать одно и то же поведение. Представьте себе злоумышленника, который ждет, пока к воротам предприятия подъедет грузовик, и компьютеру ворот понадобится их открыть. Злоумышленник перезагружает компьютер ворот непосредственно перед тем, как они должны открыться (например, на мгновение прервав подачу электропитания). После нескольких процедур инициализации центральная система по каналу связи скомандует компьютеру открыть ворота. На следующий день злоумышленник снова перезагружает компьютер ворот и посылает ему в точности те же сообщения, что и в первый раз. Поскольку компьютер ворот запускается в том же состоянии и получает те же входные данные, он поведет себя точно так же и откроет ворота. Это, конечно же, очень и очень плохо. Отметим, что использование компьютером ворот протокола синхронизации времени в данном случае не имеет никакого значения. Злоумышленник может воспроизвести вчерашние сообщения протокола, а встроенный компьютер ворот не умеет распознавать подобные вещи. Проблему одного и того же состояния не решит никакой протокол.

Для решения этой проблемы достаточно воспользоваться микросхемой датчика времени. Малый встроенный компьютер может зашифровать текущее время вместе с фиксированным секретным ключом, чтобы сгенерировать данные, обладающие высокой степенью случайности. Эти данные, в свою очередь, могут быть использованы в качестве оказии в криптографическом протоколе. Поскольку состояние датчика времени никогда не повторяется, встроенный компьютер сможет избежать постоянного пребывания в одном и том же состоянии.

Аналогичного эффекта можно добиться с помощью аппаратного генератора случайных чисел. Он позволяет встроенному компьютеру изменять свое поведение после каждой перезагрузки.

А как быть, если в системе нет датчика времени или генератора случайных чисел? К сожалению, это оборачивается большими проблемами. Разумеется, проявив смекалку, вы можете попытаться извлечь случайность из фазового сдвига синхронизирующих импульсов между локальным резонатором и сетевым источником времени, но получить достаточное количество энтропии за такое короткое время практически невозможно. Вряд ли кто-нибудь захочет, чтобы встроенный компьютер загружался 10 минут кряду!

В своей практике мы постоянно сталкиваемся с проблемой одного и того же состояния. Главный урок, который мы смогли вынести из всех подобных ситуаций, таков: чтобы действительно можно было сделать что-нибудь стоящее в плане криптографии на таких маленьких компьютерах, аппаратное обеспечение должно коренным образом измениться. Эту идею сложно внушить руководству, особенно если аппаратное обеспечение уже установлено. Руководство не желает слышать о том, что чего-то нельзя сделать. Но у нас

нет волшебной палочки, которой можно прикоснуться к существующей небезопасной системе и превратить ее в безопасную. Если система с самого начала не будет разрабатываться с оглядкой на безопасность, хороший уровень безопасности практически недостижим.

Существует еще одно возможное решение, хотя оно редко применимо на практике. Иногда в энергонезависимую память удается поместить счетчик перезагрузок. Значение этого счетчика будет увеличиваться на единицу при каждой перезагрузке компьютера. Данное решение изобилует всевозможными недостатками. Некоторые типы энергонезависимой памяти могут претерпеть лишь несколько тысяч обновлений. Постоянное обновление счетчика, хранящегося в такой памяти, приведет к быстрому износу компьютера. Еще некоторые типы энергонезависимой памяти могут быть запрограммированы только при наличии дополнительного напряжения, которое в обычных условиях зачастую недоступно. В некоторых системах мы можем лишь устанавливать определенные биты энергонезависимой памяти или очищать все ее содержимое. Последняя возможность нас не устраивает, поскольку в результате полной очистки памяти мы бы потеряли главную программу компьютера. Впрочем, даже если мы справимся со всеми перечисленными выше проблемами, нам будет крайне сложно модифицировать энергонезависимую память таким образом, чтобы значение счетчика всегда гарантированно увеличивалось, даже если в произвольные моменты времени система будет неожиданно отключаться от электропитания. На практике мы встречали лишь несколько ситуаций, в которых данное решение оказывалось действительно жизнеспособным.

17.6 Время

Обсуждая часы, нельзя не поговорить и о том, какое время следует использовать. Наш совет: держитесь подальше от местного времени. Местное время — это время, которое выставлено на наших наручных, комнатных и других часах. Проблема местного времени состоит в его изменении при переходе на летнее время. Подобные изменения приводят к неприятностям: некоторые значения времени повторяются дважды в году, когда осенью часы переводят на час назад. В результате подобных манипуляций время теряет свою уникальность и монотонность. Аналогичным образом некоторые значения времени пропускаются, когда весной часы переводят на час вперед. Более того, точные даты перехода на летнее и зимнее время различаются от страны к стране. В некоторых странах правило перехода на летнее время изменяется каждые несколько лет, что привело бы к необходимости постоянно обновлять программное обеспечение. Ну и наконец те, кто путешествует со

своими ноутбуками, могут изменять установленное на них время в соответствии с местным временем, что еще более усугубит проблему.

Очевидное решение — использовать время UTC. Это международный стандарт времени, основанный на использовании атомных часов, который широко применяется во всем мире. Любой компьютер может отслеживать смещение местного времени относительно времени UTC и использовать эти знания для взаимодействия с пользователем.

Время UTC имеет один недостаток — *дополнительные секунды (leap seconds)*. Чтобы сохранить синхронизацию времени UTC с периодом вращения Земли, каждые несколько лет вводится дополнительная секунда. До сих пор все дополнительные секунды были положительными: в последний день конкретного месяца выбирается минута, которая будет длиться 61 с. Теоретически дополнительная секунда может быть и отрицательной. Все зависит от вращения Земли. Проблема дополнительных секунд состоит в том, что они непредсказуемы. Игнорирование дополнительных секунд приводит к неточности измерения времени, если соответствующий временной интервал включает в себя дополнительную секунду. Вообще-то данную проблему нельзя назвать криптографической, но, если вы хотите создать хорошие часы, ее нельзя не учитывать. В любом программном обеспечении всегда предполагается, что в каждой минуте содержится ровно 60 с. Если вы синхронизируете свою систему непосредственно со шкалой времени UTC, неожиданная вставка дополнительной секунды может вызвать проблемы. В большинстве случаев это приведет к тому, что значения внутренних часов на протяжении одной секунды будут повторяться дважды. Это второстепенная проблема, но она нарушает уникальность и монотонность значений времени.

Зачастую точная синхронизация часов менее важна, чем монотонность и уникальность временны́х меток. Вы можете выбрать любой способ решения данной проблемы — важно лишь гарантировать, что в дополнительную секунду часы никогда не будут перескакивать назад.

Более подробно о различных шкалах времени, таких, как UTC, GMT, TAI или UT1, можно прочитать в Internet.

17.7 Заключение

К сожалению, мы так и не смогли предложить идеальное решение. Создание надежных часов — нелегкое занятие, особенно если речь идет о криптографическом окружении, битком набитом злоумышленниками. Выбор того или иного решения полностью зависит от конкретной ситуации. Удачи!

Глава 18

Серверы ключей

Пришло время вплотную заняться управлением ключами. Это, без сомнения, самый трудный момент всех криптографических систем. Именно поэтому мы решили рассмотреть управление ключами в конце книги. Мы уже объясняли, как шифровать данные, проводить их аутентификацию и согласовывать секретный ключ между двумя участниками общения. Теперь нужно найти способ, с помощью которого пользователи А и Б могли бы идентифицировать друг друга по Internet. Вы вскоре убедитесь, что сложность данной проблемы разрастается с быстротой снежного кома. Управление ключами является таким трудным еще и потому, что вместо математических формул в нем задействованы люди, а понять и предсказать человеческую логику намного сложнее.

Прежде чем начинать, давайте проясним еще один момент. Мы будем говорить исключительно о криптографических аспектах управления ключами. Такие организационные аспекты, как политики, описывающие, кому следует выдавать ключи, какие ключи будут предоставлять доступ к каким ресурсам, как проверить личность человека, получившего ключ, политики безопасности хранения ключей, механизмы верификации, проверяющие, действительно ли в системе придерживаются этих политик, и прочее, затронуты не будут. Каждая организация реализует эти аспекты по-своему, в зависимости от своих требований и существующей организационной инфраструктуры. Поэтому нас интересуют только те элементы управления ключами, которые непосредственно влияют на криптографическую систему.

Один из способов управления ключами — создание доверенной сущности, которая будет раздавать пользователям все необходимые ключи. Назовем такую сущность *сервером ключей (key server)*.

18.1 Основная идея

Идея, лежащая в основе применения сервера ключей, очень проста. Мы предполагаем, что каждый пользователь обменивается общим секретным ключом с сервером ключей. Например, пользователь А выбирает ключ K_A, который будет известен только ему и серверу ключей. Пользователь Б выбирает ключ K_B, который будет известен только ему и серверу ключей. Аналогичным образом свои секретные ключи выбирают и остальные участники общения.

Теперь предположим, что пользователь А хочет начать общаться с пользователем Б. У него нет ключа, посредством которого он мог бы общаться с пользователем Б, зато он может безопасно общаться с сервером ключей. Сервер ключей, в свою очередь, может безопасно общаться с пользователем Б. В подобной ситуации мы могли бы просто направлять весь трафик на сервер ключей, который выполнял бы роль гигантского почтового отделения. К сожалению, обработка такого объема трафика оказалась бы слишком сложным заданием для одного сервера. Гораздо лучше, если сервер ключей сгенерирует ключ K_{AB}, который будет совместно применяться пользователями А и Б.

18.2 Kerberos

Описанная идея легла в основу Kerberos — системы управления ключами, которая широко используется во всем мире [57]. Родоначальником Kerberos стал протокол Нидхэма–Шредера (Needham–Schroeder) [75].

Базовый принцип работы Kerberos можно объяснить следующим образом. Когда пользователь А хочет отправить сообщение пользователю Б, он вначале связывается с сервером ключей. Сервер ключей посылает пользователю А новый секретный ключ K_{AB}, а также ключ K_{AB}, зашифрованный с помощью ключа K_B пользователя Б. Оба сообщения зашифрованы с помощью ключа K_A, чтобы их мог прочитать только пользователь А. Он отсылает ключ K_{AB}, зашифрованный с помощью ключа K_B, пользователю Б, который расшифровывает это сообщение и получает ключ K_{AB}. После этого ключ K_{AB} начинает выступать в роли ключа сеанса, известного только пользователям А и Б (ну и, разумеется, серверу ключей).

Одно из свойств Kerberos состоит в том, что серверу ключей, который в терминологии Kerberos называется *центром распространения ключей* или *ЦРК (Key Distribution Center — KDC)*, не нужно обновлять свое состояние слишком часто. Разумеется, сервер ключей должен помнить все общие ключи, которыми он обменялся с каждым участником. Тем не менее, когда пользователь А просит ЦРК создать ключ для общения между ним и пользова-

телем Б, ЦРК выполняет свою функцию и тут же об этом забывает. Он не следит за тем, какие ключи были сгенерированы им для общения пользователей друг с другом. Это очень удобное свойство, так как оно позволяет без особых трудностей распределить сверх меры загруженный сервер ключей между несколькими компьютерами. Поскольку серверу ключей не нужно обновлять свое состояние, пользователь А может в один момент общаться с одной копией сервера, а в следующий момент — уже с другой его копией.

Оказывается, что криптографические протоколы, которые требуются для работы систем на основе Kerberos, очень сложны. Вначале проектирование подобных протоколов выглядит довольно простым, но даже разработки опытных криптографов в конце концов оказывались взломанными. Изъяны, которым удается проникнуть в эти протоколы, крайне тонки и незаметны. Мы не будем рассматривать протоколы такого типа, поскольку считаем их слишком опасными. Даже мы избегаем разрабатывать их "с нуля". Если вам действительно нужен подобный протокол, воспользуйтесь последней версией Kerberos. Он уже довольно давно применяется на практике и тщательно изучен многими компетентными критиками.

18.3 Решения попроще

Иногда использовать Kerberos невозможно. Структуру этого протокола никак не назовешь простой. Кроме того, она накладывает на систему некоторые ограничения. Серверы должны запоминать все принятые ими билеты, а каждому участнику общения нужны надежные часы[1]. В некоторых ситуациях удовлетворить подобные требования невозможно.

Между тем мы можем создать более простое и надежное решение, если не будем слишком заострять внимание на эффективности. Оказывается, что сохранение сервером ключей своего состояния может весьма пригодиться для распространения ключей. Современные компьютеры намного мощнее, чем вычислительные машины тех лет, когда создавался Kerberos. Поэтому они не должны испытывать затруднений, сохраняя состояние сервера для десятков тысяч участников. Даже наличие большой системы со 100 000 участников не представляет особых проблем: если на каждого участника потребуется по 1 Кбайт для сохранения состояния на сервере ключей, сохранение всех состояний потребует лишь 100 Мбайт памяти. Разумеется, сервер ключей все же должен быть достаточно быстрым, чтобы генерировать все запрашиваемые ключи, но это не составит проблем для современных быстрых компьютеров.

Мы ограничимся тем, что рассмотрим использование только одного сервера ключей. Существуют методы, с помощью которых состояние сервера

[1] Билет (ticket) — это сообщение, которое пользователь А отсылает пользователю Б. Это стандартная терминология Kerberos.

ключей может быть распределено между несколькими компьютерами. Вдаваться в подробности этих методов мы не будем, так как не собираемся создавать сервер ключей для десятков тысяч участников: это слишком рискованно. Опасность больших серверов ключей состоит в том, что *все* ключи находятся в одном и том же месте. Это делает сервер ключей весьма привлекательной мишенью для нападения. Кроме того, сервер ключей должен все время находиться в рабочем состоянии, а значит, злоумышленник может связаться с ним в любой удобный момент. Текущие средства безопасности не очень хорошо защищают компьютеры от сетевых атак, а размещение всех ключей в одном и том же месте равносильно подготовке к убийству. В системах меньшего размера общая "сумма" ключей, которые хранятся на сервере, относительно невелика, поэтому вероятность угрозы снижается[2]. В следующих нескольких главах мы рассмотрим инфраструктуру управления ключами, которая хорошо подходит для крупных систем. В свою очередь, для описания сервера ключей ограничимся сравнительно небольшими системами — до нескольких тысяч участников.

18.3.1 Безопасное соединение

Попробуем кратко описать наше простое решение. Прежде всего предположим, что пользователь A и сервер ключей совместно применяют общий ключ K_A. Вместо того чтобы использовать этот ключ непосредственно для обмена данными, они запускают с его помощью протокол согласования ключей наподобие описанного в главе 15, "Протокол согласования ключей". (Если K_A — это пароль, вам следовало бы использовать один из протоколов, предназначенных для обработки паролей с низкой степенью энтропии (см. раздел 15.12). К сожалению, это влечет за собой все проблемы, связанные с их лицензированием.) Протокол согласования ключей генерирует для сервера ключей и пользователя A новый ключ K'_A. Все другие участники общения запускают этот же протокол между собой и сервером ключей, в результате чего каждый из них получает по новому ключу для обмена данными с сервером ключей.

Пользователь A и сервер ключей применяют ключ K'_A для создания безопасного канала общения (см. главу 8, "Безопасный канал общения"). Используя последний, пользователь A и сервер ключей могут безопасно взаимодействовать друг с другом. Безопасный канал общения обеспечивает и конфиденциальность, и аутентификацию, и даже защиту от атак воспроизведения. Весь последующий обмен данными между пользователем A и сервером ключей происходит по безопасному каналу общения. Аналогичные каналы для взаимодействия с сервером создают и другие участники.

[2] Вообще-то нам не нравится оставлять в системе хоть какую-нибудь вероятность угрозы, но в управлении ключами всегда приходится выбирать компромиссное решение.

18.3.2 Создание ключа

Теперь нам намного легче разработать протокол, который будет генерировать ключ для взаимодействия между пользователями А и Б. Требуется рассмотреть только те случаи, когда сообщения будут утеряны или удалены злоумышленником; от остальных манипуляций последнего нас защитит безопасный канал общения. Таким образом, наш протокол будет иметь довольно простую структуру. Пользователь А обращается к серверу ключей с просьбой сгенерировать ключ для обмена данными между ним и пользователем Б. Сервер ключей отвечает на запрос, посылая новый ключ K_{AB} обоим пользователям. Более того, сервер ключей может послать сообщение пользователю Б через пользователя А, поэтому ему нет необходимости связываться с пользователем Б напрямую.

Этот протокол накладывает на систему одно ограничение: пользователь Б должен выполнить протокол согласования ключей с сервером ключей прежде, чем пользователь А обратится к последнему с просьбой сгенерировать общий ключ для него и пользователя Б. Степень серьезности данной проблемы, а также ее возможные решения зависят от конкретных обстоятельств.

18.3.3 Обновление ключа

Как и все ключи, ключ K'_A должен иметь ограниченное время жизни. Реализовать данное свойство несложно, потому что пользователь А может в любой момент перезапустить протокол согласования ключей (используя для аутентификации исходный ключ K_A), чтобы сгенерировать новый ключ K'_A. Как правило, время жизни ключа рекомендуется ограничивать несколькими часами.

Поскольку обновление ключа может быть выполнено в любой момент, серверу ключей не обязательно сохранять состояние безопасного канала общения очень надежным способом. Предположим, что в результате сбоя сервер ключей потерял все сведения о состоянии. Если сервер ключей сохранил ключ K_A (и соответствующие ключи для взаимодействия с другими участниками), восстановить состояние безопасного канала общения будет несложно. Все, что для этого требуется, — еще раз запустить протокол согласования ключей между сервером и каждым участником общения. Таким образом, хотя в нашем случае сервер ключей и не лишен состояния, ему не обязательно изменять свое долговременное состояние (ту часть данных, которая хранится на энергонезависимых носителях) при выполнении криптографических протоколов.

18.3.4 Другие свойства

Возможно, реализация нашего решения ничуть не проще, чем Kerberos, однако его концепция более проста. Наличие безопасного канала общения намного облегчает наблюдение за возможными путями атак на протокол. Использование уже созданных нами протокола согласования ключей и безопасного канала общения является хорошим примером того, как модуляризация может пригодиться при разработке криптографических протоколов.

Использование протокола согласования ключей для установки безопасного канала общения имеет еще одно преимущество — обеспечивает прямую безопасность. Даже если ключ K_A пользователя A будет взломан, это не повлияет на предыдущие ключи безопасного канала общения K'_A, а значит, все прежние сообщения, которыми пользователь A обменивался с другими участниками, останутся в безопасности.

В предыдущих частях книги приведены подробные примеры реализации рассматриваемых криптографических функций. Больше мы этого делать не будем, поскольку криптографическая часть системы управления ключами довольно проста. Разумеется, мы могли привести пример реализации сервера ключей, однако он бы вряд ли кому-нибудь пригодился. Разработка систем управления ключами в большей степени касается сбора требований к конкретному приложению и доведения до ума пользовательского интерфейса, чем криптографии как таковой. Чтобы объяснить методы реализации системы управления ключами на конкретном примере, нам потребовалось бы создать и задокументировать полную социальную и организационную структуру окружения, модель угроз, а также приложение, которому требуется система управления ключами. Это подошло бы, скорее, для научной фантастики, нежели для книги о практической криптографии.

18.4 Что выбрать

Если вы хотите реализовать централизованный сервер ключей, старайтесь по возможности применять Kerberos. Эта система имеет широкое распространение и уже долгие годы успешно применяется на практике.

Там, где использовать Kerberos невозможно, придется разработать и реализовать нечто наподобие решения, описанного в данной главе. Обратите, однако, внимание на то, что выполнение данной операции отнюдь не так тривиально, как кажется. В наиболее распространенных типах криптографических приложений разработка сервера ключей занимает практически столько же времени, сколько разработка всей оставшейся части приложения.

Глава 19

PKI: красивая мечта

В этой главе рассматривается стандартная концепция инфраструктуры открытого ключа (PKI) и то, как она позволяет решить проблему управления ключами. Очень важно, чтобы вы хорошо поняли этот материал, прежде чем продолжать чтение книги. В следующей главе мы объясним, почему на практике инфраструктура открытого ключа работает совсем не так хорошо, как ожидалось. Но это все будет потом, а пока совершим экскурсию в идеальный мир, где инфраструктура открытого ключа одним махом решает все наши проблемы.

19.1 Краткий обзор инфраструктуры открытого ключа

Инфраструктура открытого ключа (Public-Key Infrastructure — PKI) представляет собой систему, с помощью которой можно определять, кому принадлежит тот или иной открытый ключ. Рассмотрим ее классическое описание.

Существует некий центральный орган, который называется *центром сертификации* или *ЦС (Certificate Authority — CA)*. Центр сертификации обладает парой "открытый ключ-закрытый ключ" (например, парой ключей RSA) и публикует свой открытый ключ. Будем исходить из предположения, что открытый ключ центра сертификации известен всем. Поскольку данный ключ остается неизменным на протяжении длительных периодов времени, достичь этого будет несложно.

Чтобы присоединиться к инфраструктуре открытого ключа, пользователь A генерирует собственную пару "открытый ключ-закрытый ключ". Закрытый ключ он сохраняет в секрете, а открытый ключ PK_A передает центру сертификации со следующим пояснением: "Привет, я пользователь A, и у ме-

ня есть открытый ключ РК_A". Центр сертификации проверяет, действительно ли пользователь А является тем, за кого себя выдает, а затем подписывает цифровое утверждение примерно следующего содержания: "Ключ РК_A принадлежит пользователю А". Это подписанное утверждение называется *сертификатом (certificate)*. Оно удостоверяет, что данный ключ принадлежит пользователю А.

Теперь, если пользователь А захочет пообщаться с пользователем Б, он может отослать ему свой открытый ключ и сертификат. У пользователя Б есть открытый ключ центра сертификации, с помощью которого он может проверить цифровую подпись последнего. Если пользователь Б доверяет центру сертификации, он может довериться и тому, что ключ РК_A действительно принадлежит пользователю А.

Выполнив описанные действия, пользователь Б может заверить в центре сертификации свой открытый ключ, после чего отослать открытый ключ и сертификат пользователю А. Теперь пользователи А и Б знают открытые ключи друг друга. Последние, в свою очередь, могут быть использованы для запуска протокола согласования ключей, который сгенерирует ключ сеанса для последующего безопасного общения.

Для реализации описанной схемы нужен главный центр сертификации, которому все будут доверять. Каждый участник общения должен заверить свой открытый ключ в центре сертификации, и каждый участник должен знать открытый ключ самого центра сертификации. После этого все участники смогут безопасно общаться друг с другом.

На первый взгляд звучит довольно просто.

19.2 Примеры инфраструктуры открытого ключа

Чтобы облегчить понимание материала оставшейся части главы, рассмотрим несколько примеров реализации и использования инфраструктуры открытого ключа.

19.2.1 Всеобщая инфраструктура открытого ключа

Самой заветной мечтой разработчиков является всеобщая инфраструктура открытого ключа, т.е. большая организация (наподобие почты), которая сертифицирует открытый ключ каждого человека. Прелесть данной идеи состоит в том, что каждому человеку понадобится сертифицировать лишь один ключ, который он мог бы использовать где угодно. Поскольку все жители Земли доверяют почте (или любой другой организации, которая будет выполнять роль всеобщей инфрастуктуры открытого ключа), они смогут безопасно общаться друг с другом и жить в мире и согласии до конца дней своих.

Неудивительно, если все это напоминает вам сказку, — так оно и есть. Всеобщей инфраструктуры открытого ключа нет и быть не может.

19.2.2 Доступ к виртуальным частным сетям

В качестве более реального примера можно привести компанию, у которой есть виртуальная частная сеть (virtual private network — VPN). С ее помощью сотрудники компании могут осуществлять доступ к корпоративной сети прямо из дома или, скажем, из гостиничного номера во время путешествия. Точки доступа VPN должны уметь распознавать людей, которым разрешено получать доступ к сети, и предоставлять им соответствующий уровень доступа. В этом случае IT-департамент компании выступает в качестве центра сертификации. Он предоставляет каждому сотруднику сертификат, благодаря которому точка доступа VPN сможет распознать этого сотрудника.

19.2.3 Электронные платежи

Пусть некоторый банк хочет предоставить своим клиентам возможность осуществлять финансовые транзакции на Web-узле банка. Данному приложению критически важна правильная идентификация клиентов, а также возможность генерировать доказательства, которые будут приниматься в суде. Такой банк может сам выполнять роль центра сертификации и заверять открытые ключи своих клиентов.

19.2.4 Нефтеперегонный завод

Структура нефтеперегонного комплекса очень сложна. По километрам труб и подъездных дорог разбросаны сотни датчиков, которые измеряют такие показатели, как температура, дебит и давление. Умному злоумышленнику не составит большого труда подделать показания датчиков, отправляемые в диспетчерскую, чтобы действия, ошибочно предпринятые операторами, привели к колоссальному взрыву. Таким образом, диспетчеры должны быть уверены в том, что они действительно получают корректные показания датчиков. Мы можем использовать стандартные механизмы аутентификации, чтобы предотвратить подделку данных датчиков, но для обеспечения идентификации этих датчиков потребуется некоторая инфраструктура ключей. В этом случае компания может выступить в качестве центра сертификации и развернуть инфраструктуру открытого ключа для всех датчиков, чтобы в диспетчерской могли распознать каждый датчик, установленный на предприятии.

19.2.5 Ассоциация кредитных карт

Эта организация объединяет на кооперативных началах несколько тысяч банков по всему миру. Все эти банки могут обмениваться платежами. Например, пользователь, открывший кредитную карту в банке А, может заплатить продавцу, которого обслуживает банк Б. При этом банк А должен некоторым образом рассчитаться с банком Б, для чего им потребуется безопасное взаимодействие. При наличии инфраструктуры открытого ключа все банки, входящие в ассоциацию, могут идентифицировать друг друга и выполнять безопасные транзакции. В этом случае ассоциация кредитных карт может выступить в качестве центра сертификации, который заверяет ключи каждого банка.

19.3 Дополнительные детали

В реальной жизни все гораздо сложнее, поэтому простая схема PKI часто обрастает разнообразными расширениями.

19.3.1 Многоуровневые сертификаты

Довольно часто центр сертификации разбивается на несколько частей. Например, ассоциации кредитных карт было бы крайне утомительно самой сертифицировать каждый банк. Сертификацией отдельных банков занимаются ее региональные отделения. Это приводит к появлению двухуровневой структуры сертификатов. Главный ЦС выдает сертификат на открытый ключ регионального отделения. Этот сертификат утверждает нечто наподобие "Ключ РК_X принадлежит региональному отделению X и может применяться для сертификации других ключей". После этого региональное отделение может сертифицировать ключи отдельных банков. Сертификат открытого ключа банка состоит из двух подписанных сообщений: сообщения главного ЦС, которое авторизует ключ регионального отделения, и сертификата, выданного региональным ЦС ключу банка. Такая структура называется *цепочкой сертификатов (certificate chain)* и может быть расширена на любое количество уровней.

Подобные многоуровневые структуры сертификации могут оказаться крайне полезными. С их помощью функции главного ЦС могут быть распределены между уровнями иерархической структуры ЦС, что вполне подходит для многих организаций. Практически все системы PKI имеют многоуровневую структуру. Один из недостатков такой структуры состоит в том, что размер сертификатов становится больше, а для их верификации требуется большее количество вычислений, но, как правило, стоимость последних от-

носительно невелика. Следует также отметить, что каждый из центров сертификации, добавляемых в систему, представляет собой еще один потенциальный объект атаки, а следовательно, снижает безопасность системы в целом.

Один из способов устранить недостатки больших многоуровневых сертификатов, который, к сожалению, еще не встречался нам на практике, — это свернуть иерархию сертификатов. Проиллюстрируем его на примере, рассмотренном выше. Получив двухуровневый сертификат, банк может отослать его главному ЦС. Последний проверяет двухуровневый сертификат и высылает банку одноуровневый сертификат, заверенный ключом главного ЦС. Если структура сертификатов будет сворачиваться подобным образом, стоимость добавления в иерархию дополнительных уровней станет очень небольшой. Однако добавление уровней иерархии — не всегда удачное решение. Многоуровневые иерархические структуры редко оказываются эффективными.

Следует быть крайне осторожным, выстраивая цепочки сертификатов наподобие описанной выше. Они усложняют систему, а сложность — это всегда плохо. Вот лишь один пример из недалекого прошлого. Защищенные Web-узлы в Internet используют систему PKI, чтобы обозреватели пользователей могли идентифицировать эти узлы. На практике эта система оказалась не очень защищенной. Если бы все дело было только в пользователях, которые не утруждают себя проверкой имени посещаемого Web-узла! К сожалению, около года назад критическая ошибка была обнаружена и в библиотеке, которая удостоверяет правильность сертификатов всех операционных систем Microsoft. Каждый элемент цепочки сертификатов содержит флажок, который показывает, является ли сертифицируемый ключ ключом центра сертификации. Ключам центров сертификации разрешено сертифицировать другие ключи. Ключи остальных субъектов этого делать не могут. Это очень важное различие. К сожалению, упомянутая библиотека не проверяла данный флажок. Как следствие, злоумышленник мог приобрести сертификат, скажем, для домена `nastyattacker.com` и воспользоваться им, чтобы самому заверить подделанный ключ узла `amazon.com`. Обозреватель Microsoft Internet Explorer использовал ту самую библиотеку, в которой была обнаружена ошибка. Он мог принять сертификат фальшивого ключа Amazon, выданный узлом `nastyattacker.com`, и отобразить поддельный Web-узел под видом настоящего узла Amazon. Подумать только: глобальную систему безопасности, на разработку которой было затрачено столько средств и усилий, удалось обмануть с помощью малюсенькой ошибки в одной-единственной библиотеке! Как только о наличии ошибки стало известно широкой общественности, Microsoft выпустила исправление библиотеки (специалистам компании понадобилось несколько попыток, чтобы устранить эту проблему). Тем не менее это хороший пример того, как одна маленькая ошибка может разрушить без-

опасность всей системы. Вне всяких сомнений, еще есть люди, которые до сих пор пользуются неисправленной версией библиотеки.

19.3.2 Срок действия

Ни один криптографический ключ не должен жить вечно; он всегда рискует оказаться взломанным. Регулярное изменение ключа позволяет пусть медленно, но все же восстановить систему после взлома. Аналогичным свойством должны обладать и сертификаты, так как и ключ самого центра сертификации, и открытый ключ, заверенный сертификатом, имеют ограниченный срок действия. Вдобавок ко всему ограничение срока действия помогает поддерживать информацию в актуальном состоянии. По окончании срока действия сертификата ЦС должен выдать новый сертификат, а заодно и обновить содержащиеся в нем данные. Обычно срок действия сертификата составляет от нескольких месяцев до нескольких лет.

Практически все сертификаты содержат дату и время окончания срока действия. По истечении этого времени сертификат приниматься не должен. Вот для чего участникам инфраструктуры открытого ключа нужны часы.

Довольно часто в сертификат включаются и другие данные. Многие сертификаты содержат не только дату окончания срока действия, но и дату его начала. В число других дополнительных данных могут входить классы сертификатов, серийные номера сертификатов, время и дата выдачи и т.п. Некоторые из этих данных действительно полезны, а некоторые — нет.

Самым распространенным форматом сертификатов является X.509 v3, который содержит просто немыслимое количество всякого мусора. Если вы не боитесь сойти с ума, почитайте справочник Питера Гутманна (Peter Guttmann) [39]. Если ваша система не должна обладать возможностью взаимодействия с другими системами, забудьте об X.509 v3 раз и навсегда. Если же вам все-таки придется использовать X.509 v3 — примите наши искренние соболезнования.

19.3.3 Отдельный центр регистрации

Некоторые системы вдобавок к центру сертификации содержат еще и отдельный центр регистрации. Данная проблема относится к разряду политических. Как известно, решение о принятии на работу того или иного сотрудника осуществляет отдел кадров. Но работой центра сертификации руководит IT-департамент; он ни за что не разрешит заниматься этим отделу кадров.

Существует два неплохих способа решения данной проблемы. Первый — использовать многоуровневую структуру сертификации и разрешить отделу кадров самому стать подчиненным ЦС. Это автоматически обеспечит гиб-

кость, необходимую для поддержки нескольких регионов. Второе решение во многом напоминает первое, за исключением того, что пользователь обращается к главному ЦС и обменивает полученный им двухуровневый сертификат на одноуровневый. Это избавит нас от необходимости проверять двухуровневый сертификат каждый раз, когда он используется, а дополнительные расходы составят лишь стоимость добавления к системе простого протокола, состоящего из двух сообщений.

Одно из наиболее неудачных решений — добавить к криптографическому протоколу третьего участника. В этом случае в спецификациях проекта будет фигурировать центр сертификации и еще один участник, который может называться, скажем, *центром регистрации* или *ЦР (Registration Authority — RA)*. ЦС и ЦР будут рассматриваться как две совершенно отдельные сущности, в результате чего к системе добавится еще по крайней мере 100 страниц документации. Это плохо уже само по себе. Кроме того, нам понадобится описать взаимодействие между ЦР и ЦС. Мы даже встречали протоколы с тремя участниками, в которых ЦР авторизует ЦС, чтобы последний мог выдать сертификат. Это не просто безумие, но и хороший пример того, как требования пользователей могут навязать неудобоваримое техническое решение. Требования пользователей задают только внешнее поведение системы. Компании нужно обладать отдельными наборами функций для отдела кадров и IT-департамента, но это не значит, что программное обеспечение должно содержать для них отдельный код. Во многих ситуациях, в том числе и в данной, оба отдела могут с успехом использовать бóльшую часть одной и той же функциональности, а следовательно, бóльшую часть одного и того же кода. Использование одного и того же набора функций сертификации упрощает структуру системы, делает ее менее дорогостоящей, а также более мощной и гибкой, чем структура, основанная непосредственно на требованиях пользователей, которые включают в себя и ЦС и ЦР. Двухуровневая схема сертификации позволяет отделу кадров и IT-департаменту совместно использовать бóльшую часть кода и протоколов. Разница будет касаться в основном пользовательского интерфейса, что совсем несложно реализовать. Применение подобной схемы, может, и приведет к появлению нескольких сотен дополнительных строк кода, но никак не нескольких сотен дополнительных страниц спецификации, для описания которых программе потребуются десятки тысяч строк кода.

19.4 Заключение

То, что описано в этой главе, — всего лишь мечта, но мечта очень важная. Инфраструктура открытого ключа — это первое и последнее слово в управ-

лении ключами для большей части нашей компьютерной индустрии. Люди впитали эту мечту с молоком матери и относятся к ней как к чему-то такому очевидному, что не нуждается в подтверждении. Чтобы понять их, вы должны понять мечту об инфраструктуре открытого ключа, так как практически все, что они говорят, следует воспринимать в контексте этой мечты. Кроме того, очень приятно думать, что мы нашли решение всех проблем управления ключами...

Глава 20

PKI: жестокая реальность

Пришло время разрушить мечту. Основная концепция инфраструктуры открытого ключа страдает от ряда фундаментальных проблем. Разумеется, в теории все было в порядке, однако теория на то и теория, чтобы разительно отличаться от практики. Инфраструктуры открытого ключа не функционируют в реальном мире так, как можно было ожидать. Вот почему массированная реклама инфраструктуры открытого ключа, проводившаяся несколько лет назад, так никогда и не приспособилась к законам суровой действительности.

20.1 Имена

Начнем с относительно простой проблемы — понятия имени. Инфраструктура открытого ключа привязывает открытый ключ пользователя к его имени. А что же такое имя?

Для начала рассмотрим простое окружение. В маленькой деревне все жители знают друг друга в лицо. У каждого есть свое имя, и это имя либо уникально, либо его делают уникальным. Если в деревне живут два Джона, их очень быстро станут называть, скажем, Маленьким Джоном и Большим Джоном. Каждому такому имени соответствует только один житель деревни, но у одного жителя может быть сразу несколько имен. Например, Большого Джона могут еще называть Шерифом.

В данной главе речь идет не об официальных именах, которые указаны в документах, а о тех, которые употребляются в обиходе. В действительности имя — это данные любого типа, которые применяются для ссылки на некоего человека или, в более общем случае, на некую сущность. “Официальное” имя человека — это всего лишь одно из многих имен (и далеко не самое употребительное), которыми его называют в повседневной жизни.

Когда деревня разрастается до размеров небольшого городка, число людей увеличивается настолько, что запомнить их всех становится невозможно. Как следствие этого, имя начинает терять свою непосредственную ассоциацию с конкретным человеком. В городе может быть только один человек по имени Дж. Смит, но вы можете быть с ним незнакомы. Теперь имена начинают жить своей собственной жизнью, в отрыве от самого человека. Вы можете говорить о людях, которых вообще никогда не встречали. Возможно, вы сидите в пивной и обсуждаете некоего богатенького Дж. Смита, который только что переехал в ваш город и собирается в следующем учебном году спонсировать школьную команду по футболу. Через две недели оказывается, что Дж. Смит — это тот самый парень, который пару месяцев назад начал играть вместе с вами в бейсбол и которого вы знаете просто как Джона. Как видите, у людей все еще может быть несколько имен, просто теперь мы не всегда знаем, какие из этих имен принадлежат одному и тому же человеку и на какого конкретно человека они указывают.

Когда небольшой городок превращается в настоящий мегаполис, ситуация изменяется еще сильнее. Оказывается, что на самом деле вы знакомы лишь с крохотным подмножеством жителей своего города. Что еще хуже, имена окончательно перестают быть уникальными. Вам будет очень сложно найти конкретного человека, зная о нем только то, что его зовут Джон Смит — ведь людей с таким именем в городе могут быть сотни! Значение имени начинает зависеть от контекста. Девушка по имени Алиса может быть знакома с тремя Джонами, но когда на работе она говорит о “Джоне”, всем понятно, что речь идет о том самом Джоне из отдела продаж. Позднее, говоря о “Джоне” уже в домашней обстановке, она может иметь в виду сына соседей. Связь между именем и конкретным человеком становится еще более туманной.

Теперь возьмем Internet. Вскоре число пользователей всемирной сети достигнет миллиарда. Что значит имя “Джон Смит” в Internet? Практически ничего: людей с такими именем и фамилией слишком много. Поэтому для идентификации людей вместо традиционных имен используются адреса электронной почты. Теперь вы общаетесь не с Джоном Смитом, а с `jsmith533@yahoo.com`. Это имя, конечно же, уникально, но для вас оно не привязано к конкретной личности, т.е. к человеку, которого вы наглядно знаете. Даже если вы узнаете его адрес и номер телефона, он все равно останется для вас неким абстрактным человеком, живущим где-то на другом конце мира. Вы никогда не встретите его лично, если только не сделаете этого специально. Неудивительно, что многие пользователи Internet зачастую пытаются представить себя совсем другими людьми с другими именами, возрастом, увлечением и привычками. И естественно, у каждого человека может быть масса имен. Многие пользователи в конце концов обрастают горами адресов электронной почты. (У нас их, например, никак не меньше десятка.) Крайне

сложно определить, ссылаются ли два адреса электронной почты на одного и того же человека. Ситуацию окончательно усложняет тот факт, что некоторые люди совместно используют один и тот же адрес электронной почты, поэтому "имя" ссылается на них обоих.

Существуют большие организации, которые пытаются присвоить каждому человеку единственное собственное имя. Это, как правило, государственные учреждения. Руководящие органы большинства стран требуют, чтобы у каждого гражданина было одно официальное имя, которое фигурирует в паспорте и других документах, удостоверяющих личность. Само по себе имя не уникально — существует множество людей с одинаковыми именем и фамилией; поэтому на практике имя часто расширяют такими данными, как адрес, номер водительских прав и дата рождения. Это, однако, все еще не гарантирует уникальность идентификатора конкретного человека[1]. Кроме того, в течение жизни человека некоторые из этих идентификаторов могут измениться. Люди меняют адреса, номера водительских прав, имена и даже пол. Пожалуй, единственное, что не изменяется, — это дата рождения, но не следует забывать, что многие намеренно ее искажают.

Кстати, то, что у каждого человека есть единственное, утвержденное правительством официальное имя, тоже не совсем верно. Кто-то живет на нелегальном положении и вообще не имеет документов. Кто-то может иметь двойное гражданство, причем в каждой стране имя человека могут написать посвоему. Если в языках двух стран применяются различные алфавиты, имя человека не будет одинаковым по определению. В некоторых странах требуют, чтобы имя писалось в соответствии с правилами государственного языка, и "подгоняют" иностранные имена под похожие, "правильные" имена на своем языке.

Во избежание путаницы во многих странах жителям присваивают уникальные номера наподобие номеров социального страхования (Social Security number — SSN) в США или номеров SoFi в Нидерландах. Главное назначение этого номера состоит в присвоении каждому человеку уникального и неизменного имени, чтобы государственные органы могли отслеживать его действия и связывать их друг с другом. В большинстве случаев подобные схемы именования оказываются успешными, однако и они имеют свои недостатки. Связь между конкретным человеком и присвоенным ему номером не слишком крепка, и многие предприятия с успехом оперируют фальшивыми номерами. Кроме того, поскольку эти схемы именования работают только в рамках конкретной страны, они не обеспечивают глобального охвата людей или глобальной уникальности их номеров.

[1] Номера водительских прав действительно уникальны, но права есть не у всех.

Нельзя не отметить еще один аспект имен. В европейских странах существуют законы об охране частной жизни. Они ограничивают объем информации о человеке, которую разрешено иметь той или иной организации. Например, супермаркет, выдающий вам дисконтную карту, не имеет права спрашивать вас, хранить или каким-либо другим способом обрабатывать ваш SSN или SoFi. Это ограничивает область возможного применения государственных схем именования.

Так какое же имя следует использовать для идентификации пользователя в инфраструктуре открытого ключа? Поскольку многие люди имеют множество разных имен, это становится настоящей проблемой. Возможно, девушке по имени Алиса хочется иметь два разных ключа — один для личной, а второй для деловой переписки. В этом случае для получения одного ключа она может использовать свою девичью фамилию, а для получения второго — фамилию по мужу. При попытке разработать всеобщую инфраструктуру открытого ключа подобные вещи быстро приводят к серьезным проблемам. Это одна из причин того, почему множество маленьких инфраструктур открытого ключа, созданных специально для конкретных приложений, функционируют гораздо лучше, чем одна большая.

20.2 Полномочный орган

Кто выступает в роли центра сертификации, уполномоченного присваивать ключи именам? Почему он уполномочен выдавать ключи данному кругу имен? Кто решает, является ли человек с данным именем сотрудником компании, которому следует предоставить доступ к виртуальной частной сети, или же, скажем, клиентом банка с ограниченным доступом?

В большинстве рассмотренных нами примеров ответить на последний вопрос было несложно. Работодатель знает, кто является, а кто не является его сотрудником. Банк знает своих клиентов. Это дает нам первое указание на то, какая организация должна выступать в качестве центра сертификации. К сожалению, в нашем мире нет авторитетного источника, который мог бы выступать в качестве ЦС всеобщей инфраструктуры открытого ключа. Вот почему создать такую инфраструктуру невозможно.

Планируя инфраструктуру открытого ключа, следует подумать о том, кто будет уполномочен выдавать сертификаты. Например, компания может прекрасно справляться с ролью полномочного органа по отношению к служащим. Компания не выбирает официальное имя для своего сотрудника, но она знает, под каким именем этот человек известен *внутри компании*. Никого не волнует, если сотрудника по имени Фред Смит на самом деле зовут Альфре-

дом. Имя Фред Смит вполне подходит для идентификации этого человека среди служащих компании.

20.3 Доверие

Управление ключами — самая трудная проблема криптографии, а инфраструктура открытого ключа — одно из лучших имеющихся у нас средств для ее решения. К сожалению, успешность последнего полностью зависит от безопасности инфраструктуры открытого ключа, а значит, и от степени доверия к центру сертификации. Подумайте, какой ущерб может быть нанесен компании, если ЦС начнет фабриковать поддельные сертификаты. Центр сертификации сможет выдавать себя за любого пользователя системы, а значит, о безопасности можно забыть.

Всеобщая инфраструктура открытого ключа — объект мечтаний любого разработчика, однако в плане доверия она не выдерживает никакой критики. Как по-вашему, стал бы банк, которому нужно взаимодействовать со своими клиентами, доверять кому бы то ни было на другом конце мира? Или бюрократическому аппарату своей собственной страны? Сколько денег вы можете потерять, если центр сертификации начнет вести себя неподобающим образом? Какую степень ответственности согласен взять на себя центр сертификации? Позволят ли вам местные законы о банковском деле взаимодействовать с иностранным центром сертификации? Все эти проблемы очень серьезны. Только подумайте, какой ущерб может быть нанесен компаниям по всему миру, если закрытый ключ центра сертификации будет опубликован на каком-нибудь Web-узле...

Для большей наглядности попробуем представить себе эту проблему в более традиционных терминах. Центр сертификации — это организация, которая раздает ключи к офисам. Большинство крупных офисов имеют охрану, нанятую извне в какой-нибудь охранной фирме. Охрана следит за тем, чтобы никто не нарушал правила входа в офис; это довольно простая и вполне понятная задача. Между тем решение о том, кому следует выдавать ключи для входа в офис, — задача, которую редко перепоручают кому-нибудь со стороны, поскольку это фундаментальная часть политики безопасности. По этой же причине обязанности ЦС не следует возлагать на внешний источник.

Ни одной организации в мире не доверяют абсолютно все. Не существует даже такой организации, которой бы доверяло подавляющее *большинство* населения. Как видите, создать всеобщую инфраструктуру открытого ключа невозможно. Следовательно, придется использовать множество небольших инфраструктур. Именно такое решение предлагается во всех наших примерах. Банк может выполнять роль собственного ЦС; в конце концов, банк дове-

ряет сам себе, а его клиенты уже высказали доверие банку, вложив в него свои деньги. Компания может выполнять роль собственного ЦС для виртуальной частной сети. Собственный ЦС может быть и у ассоциации кредитных карт.

Интересно отметить, что ЦС использует существующие отношения доверия, основанные на договорных отношениях. Это всегда учитывается при разработке криптографических систем: все базовые отношения доверия, от которых мы отталкиваемся при разработке системы, основаны на договорных отношениях.

20.4 Непрямая авторизация

Мы подошли к большой проблеме классической мечты о PKI. Инфраструктура открытого ключа привязывает ключи к именам людей, но большинство систем не интересует имя конкретного человека. Банковская система должна знать, какие транзакции следует авторизовать; виртуальная частная сеть — к каким каталогам следует предоставить доступ. Ни одну из этих систем не волнует, *кому* принадлежит ключ. Их интересует лишь то, *что* позволено делать владельцу этого ключа.

Для разрешения данной проблемы большинством систем используются так называемые *списки контроля доступа (access control list — ACL)*. Список контроля доступа — это всего лишь база данных, в которой указано, кому из пользователей системы какие действия разрешено выполнять. Иногда содержимое списка контроля доступа сортируется по имени пользователя (например, пользователю по имени Боб разрешено выполнять следующее: открывать файлы в каталоге `/usr/bob`, использовать офисный принтер, осуществлять доступ к файловому серверу), но в большинстве случаев список контроля доступа индексируется по действию (например, вносить деньги на данный счет может только Боб либо Бетти). Зачастую для упрощения списков контроля доступа пользователей объединяют в группы, однако базовая функциональность при этом не изменяется.

Итак, теперь у нас есть три объекта: ключ, имя и разрешение на выполнение какого-нибудь действия. Система при этом должна знать, какой ключ какое действие авторизует, другими словами, есть ли у заданного ключа разрешение на выполнение определенного действия. Классическая инфраструктура открытого ключа решает эту проблему, привязывая ключи к именам и используя список контроля доступа, чтобы привязать имена к разрешениям. Это непрямой метод авторизации, который к тому же создает больше потенциальных объектов для нападения [27].

Первый объект нападения — это сертификат типа “имя-ключ”, выданный инфраструктурой открытого ключа. Второй объект нападения — база

данных со списком контроля доступа, который привязывает имена к разрешениям. И наконец, третий объект нападения — путаница в именах. Однако понятие имени, как уже отмечалось, весьма расплывчатое. Как определить, является ли имя, указанное в списке контроля доступа, тем самым именем, что фигурирует в сертификате? И как по ошибке не присвоить двум разным людям одно и то же имя?

Если вы проанализируете эту ситуацию, то поймете, что в данном случае техническое проектирование следует наивной формулировке требований пользователей. Эта проблема рассматривается в терминах идентификации владельца ключа и предоставления доступа тому или иному человеку — именно так поступают охранники в офисе. Автоматизированные системы могут использовать гораздо более прямой подход. Дверному замку все равно, кто именно держит в руках ключ. Он пропустит в помещение любого, у кого этот ключ есть.

20.5 Прямая авторизация

Намного эффективнее привязать разрешения непосредственно к ключу, используя для этого инфраструктуру открытого ключа. Сертификат больше не будет привязывать ключ к имени; вместо этого он связывает ключ с набором разрешений [27].

Все системы, использующие сертификаты инфраструктуры открытого ключа, теперь могут непосредственно решать, предоставлять доступ или нет. Для этого им достаточно взглянуть на сертификат и выяснить, обладает ли соответствующий ключ необходимыми разрешениями. Это прямой и простой метод.

Прямая авторизация избавляет от необходимости поддерживать списки контроля доступа и работать с именами, тем самым устраняя соответствующие объекты нападения. Правда, некоторые из этих проблем вновь всплывут на стадии выдачи сертификатов. Кто-то должен определить, какому пользователю будет разрешено осуществлять какие действия, и гарантировать, что это соответствие будет правильно закодировано в сертификате. База данных всех этих соответствий станет эквивалентом списка контроля доступа, но атаковать ее будет намного сложнее. Во-первых, принятие подобных решений легко распределить между несколькими людьми, что позволяет избавиться от централизованной базы данных и всех уязвимых мест, связанных с ее наличием. Лица, принимающие решения, могут просто выдать пользователю соответствующий сертификат, не прибегая к развертыванию какой-либо дополнительной инфраструктуры и обеспечению ее безопасности. Во-вторых, это позволяет снизить зависимость процесса выдачи сертификатов

от имен, поскольку лица, принимающие решения, занимают более низкую ступень в руководстве компании и работают с гораздо меньшими группами людей. Как правило, они лично знакомы с пользователями или хотя бы знают их в лицо, что во многом позволяет избежать путаницы с именами.

Так, может быть, стоит совсем избавиться от имен в сертификатах?

Вообще-то нет. Хотя имена и не будут фигурировать в обычных операциях, сведения о пользователях могут понадобиться для проведения аудита и т.п. Предположим, банк только что обработал выдачу заработной платы, авторизованную одним из четырех ключей, имеющих полномочия переводить средства компании на данный счет. Через три дня главный финансовый администратор компании звонит в банк и спрашивает, на каком основании был выполнен этот платеж. Банк знает, что выдача заработной платы была авторизована некоторым ключом, но финансовому администратору нужно нечто большее, чем несколько тысяч случайных бит открытого ключа. Вот почему в сертификат все-таки нужно включать имя пользователя. Тогда банк сможет сообщить администратору, что ключ, посредством которого была авторизована выдача заработной платы, принадлежит некоему Дж. Смиту. Этого будет вполне достаточно, чтобы администратор понял, что произошло. Отметим, однако, что в данном случае имена, указанные в сертификатах, должны иметь значение только для людей. Компьютеру никогда не придется выяснять, относятся ли два разных имени к одному и тому же человеку или на кого конкретно ссылается заданное имя. Люди гораздо лучше управляются с такими расплывчатыми понятиями, как имена, в то время как компьютерам нравятся более простые и строго определенные вещи наподобие наборов разрешений.

20.6 Системы мандатов

Продолжая развивать описанный принцип доступа на основе разрешений, мы получим полноценную систему мандатов. Это уже не просто инфраструктура, а суперинфраструктура открытого ключа. В подобной системе для выполнения каждого действия пользователю нужно иметь *мандат (credential)* в виде подписанного сертификата. Если у пользователя А есть мандат, который разрешает ему открывать конкретный файл для чтения и записи, этот пользователь может передать полностью или частично свои полномочия пользователю Б. Например, пользователь А может подписать сертификат открытого ключа пользователя Б, в котором будет утверждаться нечто наподобие: “Ключу РК_B разрешено открывать файл X для чтения, так как ему были переданы полномочия ключа РК_A”. Если пользователь Б хочет считать файл X, он должен предъявить этот сертификат, а также сертификат, дока-

зывающий, что пользователю А действительно разрешено открывать файл X для чтения.

Система мандатов может обладать и дополнительными функциями. Например, пользователь А может ограничить срок действия передачи своих полномочий, включив его в сертификат. Кроме того, пользователь А может ограничить возможность пользователя Б передавать полномочия на чтение файла X другим пользователям[2].

В теории система мандатов является исключительно мощной и гибкой. На практике же по ряду причин подобные системы применяются крайне редко.

Во-первых, системы мандатов довольно сложны, а их применение влечет за собой массу дополнительных расходов. Права пользователя на доступ к ресурсу могут зависеть от цепочки из пяти-шести сертификатов, каждый из которых должен быть передан и тщательно проверен.

Во-вторых, применение мандатов требует своеобразного микроуправления доступом. Полномочия так легко разбивать на части все меньшего и меньшего размера, что пользователи начинают тратить слишком много времени на обдумывание того, сколько именно полномочий следует передать своему коллеге. Это время, как правило, уходит впустую. Еще большей проблемой, однако, является потеря времени того самого коллеги, когда оказывается, что у него не хватает прав доступа для выполнения работы. Возможно, проблема микроуправления может быть решена путем обучения пользователей и улучшения пользовательских интерфейсов, но вопрос все еще остается открытым. Некоторые пользователи успешно избегают проблемы микроуправления, передавая все (или практически все) свои полномочия любому, кто нуждается в каком-либо уровне доступа, что подрывает безопасность всей системы.

В-третьих, чтобы реализовать систему мандатов, необходимо разработать специальный язык передачи полномочий. Сообщения о передаче полномочий должны быть написаны на логическом языке, который будет понятен компьютерам. Этот язык должен быть достаточно мощным для того, чтобы на нем можно было выразить всю требующуюся функциональность, и вместе с тем достаточно простым, чтобы обеспечить быстрое сцепление заключений. Он также должен быть разработан с запасом на будущее. После того как система мандатов будет развернута, в каждую программу понадобит-

[2]Многие клиенты требуют, чтобы система обладала подобным свойством, однако нам это кажется в корне неверным. Ограничение возможности пользователя Б передавать свои полномочия другим лишь вынуждает его запустить какую-нибудь службу прокси, благодаря которой другие пользователи смогут применять мандат пользователя Б для доступа к нужным ресурсам. Использование подобных программ подрывает инфраструктуру безопасности и должно быть запрещено, но это имеет смысл только тогда, когда у людей нет практических причин передавать свои полномочия и, следовательно, запускать прокси. А такие причины всегда найдутся.

ся включить код для интерпретации языка передачи полномочий. Переход к новой версии языка может оказаться слишком сложным, особенно из-за того, что функции обеспечения безопасности проникают в каждый уголок системы. Между тем мы просто не в состоянии разработать настолько общий язык, чтобы он был способен удовлетворить все потенциальные будущие требования, так как не знаем, что может случиться в будущем. Данный вопрос остается открытым для дальнейших исследований.

И наконец, четвертая проблема систем мандатов, скорее всего, никогда не будет преодолена. Все дело в том, что концепция избранной передачи полномочий слишком сложна для среднестатистического пользователя. Мы не знаем, как ознакомить пользователей с понятием правил доступа, чтобы они это поняли. Попросив пользователей подумать о том, какие полномочия следует передавать другому человеку, вы автоматически обрекаете себя на провал. Подобные ситуации можно повсеместно наблюдать в реальной жизни. Среди студентов весьма распространено поведение, когда одного человека просят пойти к банкомату и получить деньги сразу за всех. Другие студенты отдают ему свои банковские карточки и сообщают PIN-коды. Это в высшей степени глупый поступок, вместе с тем подобное поведение свойственно и более опытным людям. Работая консультантами, мы посещали множество компаний. Иногда в ходе работы у нас возникала необходимость получения доступа к локальной сети. Мы были просто потрясены тем, какие возможности доступа открывались перед нами, в общем-то совершенно посторонними людьми. Системные администраторы предоставляли нам неограниченный доступ к данным, полученным в ходе исследований, в то время как нам было нужно лишь взглянуть на пару файлов. Если с подобным заданием не справляются и системные администраторы, что уж говорить о рядовых пользователях!

Как криптографы, мы бы одобрили идею системы мандатов, если бы только пользователи смогли справиться с ее сложностью. Несомненно, вопрос взаимодействия человека с системами безопасности должен подвергнуться более тщательному изучению.

Существует, однако, одна область, в которой применение мандатов приносит огромную пользу и должно быть обязательным. В иерархической структуре центров сертификации главный ЦС подписывает сертификаты ключей подчиненных ЦС. Если эти сертификаты не включают в себя какие-либо ограничения полномочий, каждый подчиненный ЦС будет наделен неограниченной властью. Это очень плохо влияет на безопасность системы, поскольку лишь увеличивает количество мест, в которых хранятся ее критические ключи.

В иерархии центров сертификации возможности подчиненного ЦС должны быть ограничены путем включения соответствующих ограничений в сертификат его ключа. Для выполнения операций над сертификатами ЦС раз-

работчику понадобится язык передачи полномочий наподобие того, который упоминался несколько выше. Выбор конкретного типа ограничений зависит от приложения. Просто подумайте, какой тип подчиненных ЦС нужен вашему приложению и как следует ограничить их возможности.

20.7 Измененная мечта

Давайте подытожим все критические замечания, высказанные в адрес инфраструктуры открытого ключа, и немного подкорректируем нашу мечту. Это будет гораздо более реалистическое представление о том, как должна выглядеть PKI.

Прежде всего каждое приложение должно иметь свою инфраструктуру открытого ключа с собственным центром сертификации. Мир состоит из огромного количества небольших инфраструктур открытого ключа. Каждый пользователь одновременно является членом множества разнообразных инфраструктур.

В каждой инфраструктуре открытого ключа пользователь должен применять разные ключи. Использование одного и того же ключа в разных системах было бы возможно только при тщательном координировании процессов их разработки. По этой причине хранилище ключей каждого пользователя будет содержать десятки ключей и занимать десятки килобайт дискового пространства.

Основное назначение инфраструктуры открытого ключа состоит в том, чтобы привязать ключ к мандату. Например, инфраструктура открытого ключа банка привязывает ключ пользователя A к мандату, который разрешает осуществлять доступ к счету этого пользователя. Инфраструктура открытого ключа компании, в свою очередь, привязывает ключ пользователя A к мандату, который предоставляет доступ к виртуальной частной сети. Значительные изменения полномочий пользователя требуют выдачи нового сертификата. Сертификаты все еще включают в себя имя пользователя, однако последнее применяется лишь в нуждах управления и аудита.

Как видите, измененная мечта стала более реалистичной. Она также обладает бóльшими возможностями, большей гибкостью и безопасностью, чем первоначальная. Было бы так приятно считать, что измененная мечта решит все наши проблемы управления ключами! К сожалению, существует проблема (она описана в следующем разделе), которая никогда не будет решена до конца и всегда будет требовать компромиссов.

20.8 Отзыв

Самая сложная проблема инфраструктуры открытого ключа — это *отзыв (revocation)* сертификата. Иногда сертификат, выданный пользователю, следует аннулировать. Например, злоумышленник мог взломать компьютер пользователя Б и узнать его закрытый ключ или, скажем, пользователь А мог перейти в другой департамент либо вообще уволиться из компании. Мы можем придумать огромное количество ситуаций, в которых действующий сертификат пользователя должен быть немедленно отозван.

Проблема отзыва состоит в том, что сертификат — это всего лишь последовательность битов. Эти биты используются и хранятся в огромном количестве мест. Сколько бы мы ни старались, мы не сможем заставить весь мир забыть об этом сертификате. Около 10 лет назад Брюс потерял свой ключ PGP. Между тем сообщения, зашифрованные с помощью соответствующего сертификата, приходят к нему до сих пор[3]. Сама попытка заставить мир забыть о существовании сертификата выглядит абсолютно нереальной. Если злоумышленник взломает компьютер пользователя Б и украдет его закрытый ключ, можете быть уверены, что он не забудет сделать копию сертификата соответствующего открытого ключа.

Каждая система выдвигает собственные требования к отзыву сертификатов, однако в общем случае они характеризуются тремя параметрами.

- **Скорость отзыва.** Какое максимально допустимое количество времени может пройти между поступлением команды на отзыв и последним использованием сертификата?
- **Надежность отзывов.** Приемлема ли для системы ситуация, в которой при некоторых обстоятельствах отзыв сертификата окажется не вполне эффективным? Какой остаточный риск допускает система?
- **Количество отзывов.** Сколько отзывов могут обрабатываться одновременно?

Существует два практических решения проблемы отзывов: списки отзыва и быстрое устаревание.

20.8.1 Список отзыва

Список отзыва сертификатов (certificate revocation list — CRL) представляет собой базу данных, в которой перечислены отозванные сертификаты.

[3]Программа PGP использует собственную довольно странную инфраструктуру, которая по своему назначению напоминает PKI и называется *сетью доверия (web of trust)*. На практике данная концепция слишком сложна для понимания пользователей, а потому применяется только в ограниченном масштабе.

Каждый, кто хочет проверить подлинность сертификата, должен обратиться к списку отзыва и посмотреть, не был ли этот сертификат отозван.

Наличие централизованной базы данных отзыва сертификатов имеет целый ряд преимуществ. Прежде всего следует отметить, что отзыв сертификатов выполняется практически мгновенно. Как только сертификат был добавлен в список отзыва, ни одна из последующих транзакций с его участием не будет авторизована. Кроме того, система отзыва очень надежна, а прямого ограничения на количество отзываемых сертификатов не существует.

Несмотря на это, у централизованной базы данных отзыва есть и существенные недостатки. Каждый член PKI должен быть постоянно подключен к центру сертификации, чтобы он мог в любой момент обратиться к списку отзыва сертификатов. Кроме того, при использовании централизованной базы данных у системы появляется так называемая *одна точка сбоя (single point of failure)*: если база данных окажется недоступной, выполнение каких-либо действий станет невозможным. Некоторые решают эту проблему, позволяя членам PKI продолжать свою работу, даже если база данных отзыва сертификатов окажется недоступной. В этом случае, однако, злоумышленники будут использовать атаки типа "отказ в обслуживании", чтобы деактивизировать базу данных и вывести из строя систему отзыва.

В качестве альтернативы можно предложить использование распределенной базы данных. Мы могли бы создать избыточные зеркала базы данных, используя для этого десяток серверов по всему миру, и надеяться, что такая система отзыва окажется достаточно надежной. К сожалению, построение и поддержка избыточных баз данных требуют такого количества средств, что обычно подобный вариант даже не рассматривается. Не забывайте: в нашем мире никто не хочет тратить деньги на безопасность.

Некоторые системы просто рассылают полные копии базы данных отзыва сертификатов всем своим устройствам. Подобный принцип применялся военными силами США при шифровании телефонных разговоров с помощью оборудования STU-III (Secure Telephone Unit, Third Generation — безопасный телефонный аппарат третьего поколения). Точно так же поступают и банки, когда рассылают по магазинам маленькие книжечки с номерами украденных кредитных карт. Реализовать такую схему отзыва сравнительно несложно. Каждое устройство системы может периодически (скажем, раз в полчаса) загружать с Web-сервера обновленную версию списка отзыва сертификатов. Разумеется, это несколько снизит скорость отзыва. К сожалению, данное решение накладывает ограничения на размер базы данных отзыва сертификатов. Мало кто может позволить себе, чтобы на каждое устройство системы постоянно копировался список, состоящий из сотен тысяч сертификатов. Нам попадались системы, в которых размер списка отзыва был ограничен 50 элементами. Думаем, способ нападения на такую систему понятен и школьнику.

По своему опыту мы знаем, что реализация и поддержка списков отзыва сертификатов обходится в немалую сумму. Системам отзыва требуется собственная инфраструктура, управление, пути коммуникации и т.п. Значительный объем дополнительной функциональности нужен только для того, чтобы обрабатывать сравнительно редко используемые механизмы отзыва.

20.8.2 Быстрое устаревание

Вместо списков отзыва можно использовать принцип быстрого устаревания. Данный принцип подразумевает использование существующего механизма устаревания сертификатов. Согласно ему центр сертификации должен выдавать сертификаты с очень малым сроком действия (от 10 мин до 24 ч). Каждый раз, когда пользователь хочет применить свой сертификат, он обращается к ЦС за новым сертификатом. Затем полученный сертификат может использоваться на протяжении всего срока действия. Точное значение скорости устаревания может быть выбрано в соответствии с требованиями приложения, но на практике выдавать сертификаты со сроком действия меньше 10 мин нет смысла.

Главное преимущество этой схемы состоит в использовании уже существующего механизма выдачи сертификатов. Системе больше не требуется отдельный список отзыва сертификатов, что значительно снижает ее сложность. Все, что нужно для отзыва разрешений, — это проинформировать ЦС о новых правилах доступа. Разумеется, для обновления сертификатов каждый член PKI должен быть постоянно подключен к ЦС.

Одним из главных критериев проектирования для нас является простота, поэтому базе данных отзыва сертификатов мы предпочитаем быстрое устаревание. Возможность реализации последнего зависит главным образом от того, требуется ли приложению мгновенный отзыв, или же оно допускает некоторую задержку.

20.8.3 Отзыв обязателен

Поскольку реализовать отзыв сертификатов довольно сложно, его всячески пытаются избежать. В некоторых предложениях по реализации инфраструктуры открытого ключа нет ни слова об отзыве сертификатов. В других список отзыва сертификатов упоминается лишь как возможность последующего расширения. В действительности инфраструктура открытого ключа без какой-либо схемы отзыва оказывается бесполезной. Обстоятельства реальной жизни таковы, что ключи *действительно* подвергаются взлому, а следовательно, доступ к системе с помощью таких ключей должен быть аннулирован. Применять инфраструктуру открытого ключа без работающей системы отзыва — это все равно, что выходить в открытое море на корабле без трюмных

насосов. Теоретически корабль должен быть водонепроницаемым, а значит, ему не нужны трюмные насосы. На практике же в трюме корабля всегда собирается вода, и если от нее не избавиться, корабль в конце концов утонет.

20.9 Где может пригодиться инфраструктура открытого ключа

В самом начале обсуждения инфраструктуры открытого ключа отмечалось, что ее назначение состоит в том, чтобы позволить участникам общения сгенерировать общий секретный ключ. Последний может применяться для создания безопасного канала общения, который, в свою очередь, позволяет двум участникам безопасно общаться друг с другом. Пользователь А хочет аутентифицировать сообщения пользователя Б (и наоборот) без привлечения третьего участника. Все это предполагалось сделать возможным с помощью инфраструктуры открытого ключа.

Ничего не вышло.

Не существует системы отзыва, которая бы работала в автономном режиме. Это очень легко объяснить. Если ни пользователь А, ни пользователь Б не будут обращаться к стороннему участнику, они никогда не узнают, что один из их ключей был отозван. Поэтому им нужно иметь постоянное подключение к центру сертификации. Оба решения, предложенные нами в качестве схемы отзыва, требуют наличия постоянных подключений.

Но, если мы все время подключены к ЦС, нам вовсе не нужна большая и сложная инфраструктура открытого ключа. Желательной безопасности можно достичь гораздо проще, с помощью централизованного сервера ключей наподобие описанного в главе 18, “Серверы ключей”.

Рассмотрим преимущества инфраструктуры открытого ключа по сравнению с использованием сервера ключей.

- Сервер ключей требует, чтобы каждый участник общения был все время к нему подключен. Если вы не можете связаться с сервером ключей, вы вообще ничего не сможете сделать. Пользователи А и Б никак не смогут распознать друг друга. В этом плане инфраструктура открытого ключа обладает некоторыми преимуществами. Если в качестве схемы отзыва применяется быстрое устаревание, участнику общения достаточно лишь время от времени связываться с главным сервером. Для приложений, работающих с сертификатами, срок действия которых исчисляется несколькими часами, это означает значительное послабление требований к наличию постоянного подключения. Даже при использовании базы данных отзыва сертификатов можно установить, как следует поступать в том случае, если база данных отзыва недоступна. По-

добные правила применяются в системах кредитных карт. Если вы не можете получить автоматическую авторизацию, система разрешает выполнение некоторого объема транзакций. Правила такого рода должны разрабатываться на основе анализа рисков, включая риск атаки типа "отказ в обслуживании" на список отзыва сертификатов. Тем не менее наличие подобных правил дает возможность хоть как-то продолжить работу. Сервер ключей подобной возможностью не обладает.

- Сервер ключей представляет собой одну точку сбоя. Распределить сервер ключей сложно, потому что он содержит все ключи системы. Думаем, никому не захочется разбрасывать свои ключи по всему миру. В отличие от этого, база данных отзыва сертификатов менее критична для безопасности системы, а распределить ее гораздо проще. В схеме быстрого устаревания точкой сбоя становится ЦС. Тем не менее в больших системах практически всегда применяется иерархия центров сертификации. Это означает, что ЦС уже распределен и его сбой повлияет только на небольшую часть системы.
- Теоретически инфраструктура открытого ключа должна гарантировать соблюдение пользователями своих обязательств. Оно состоит в следующем: если пользователь A подписал сообщение своим ключом, он не может впоследствии отрицать факт подписывания этого сообщения. Сервер ключей подобным свойством не обладает. Главный сервер имеет доступ к ключу, который применяется пользователем A, а потому может легко отправить поддельное сообщение, подписанное якобы пользователем A. На практике, однако, принцип соблюдения обязательств не срабатывает, потому что люди не в состоянии обеспечить достаточную надежность хранения своих секретных ключей. Если пользователь A захочет отрицать факт подписывания сообщения, он может просто заявить, что его компьютер подвергся нападению вируса, который и украл закрытый ключ.
- Самым важным ключом PKI является ключ корневого центра сертификации. Его не обязательно хранить на компьютере, подключенном к серверу, а можно поместить в безопасное место и загружать только в автономный компьютер, да и то по мере необходимости. Ключ корневого ЦС применяется только для подписывания сертификатов подчиненных ЦС, что происходит крайне редко. В противоположность этому, содержимое сервера ключей должно располагаться на компьютере, подключенном к сети. Атаковать автономные компьютеры гораздо сложнее, чем компьютеры, подключенные к сети, поэтому наличие данного свойства делает инфраструктуру открытого ключа потенциально более безопасной, чем сервер ключей.

Как видите, инфраструктура открытого ключа имеет ряд преимуществ, но ни одно из них не является действительно критичным. За каждое из них приходится платить довольно дорого. Инфраструктура открытого ключа намного сложнее, чем сервер ключей, а операции над открытым ключом требуют от системы гораздо большей вычислительной мощности.

20.10 Что выбрать

Так как же настроить систему управления ключами? Что выбрать — сервер ключей или инфраструктуру открытого ключа? Как всегда, это зависит от требований конкретного приложения.

В небольших системах сложность инфраструктуры открытого ключа себя не оправдывает. Мы рекомендуем в подобных случаях применять сервер ключей. Это объясняется главным образом тем, что преимущества инфраструктуры открытого ключа перед сервером ключей гораздо лучше проявляются в крупных системах.

Для больших систем дополнительная гибкость инфраструктуры открытого ключа все еще остается крайне привлекательной. Инфраструктура открытого ключа лучше поддается распределению, нежели сервер ключей. Расширения наподобие системы мандатов позволяют ограничивать полномочия подчиненных центров сертификации. Это, в свою очередь, облегчает настройку небольших подчиненных ЦС, каждый из которых охватывает определенный диапазон операций. Поскольку возможности подчиненного ЦС по выдаче сертификатов ограничены сертификатом его собственного ключа, он вряд ли может представлять угрозу безопасности системы в целом. В больших системах наличие подобной гибкости и ограничения риска действительно играет важную роль.

Тем, кто разрабатывает большую систему, мы бы посоветовали внимательно приглядеться к инфраструктуре открытого ключа, но все же сравнить ее с решением на основе сервера ключей. Подумайте, стоят ли преимущества инфраструктуры открытого ключа ее дороговизны и сложности. Особенно трудно придется тем, кому действительно необходимо ограничить полномочия подчиненных центров сертификации с помощью системы мандатов. Для описания ограничений вам понадобится некоторая логическая структура (logical framework). Универсальной логической структуры для выполнения задач такого рода не существует, а следовательно, проектирование данной части системы будет полностью определяться требованиями ваших заказчиков. Это, вероятно, также означает, что вы не сможете воспользоваться готовыми инфраструктурами открытого ключа от других производителей, поскольку подобные продукты вряд ли будут содержать язык описания ограничений.

Глава 21

Практические аспекты PKI

Если вам нужна инфраструктура открытого ключа, ее можно купить или разработать самому. В первом случае вам, вероятно, не понадобилась бы наша помощь. *Практическая криптография* — это книга по инженерии, а не справочник покупателя. Поэтому мы полагаем, что вы решили развернуть собственную инфраструктуру. В этой главе приводится несколько практических соображений, которые могут пригодиться при разработке инфраструктуры открытого ключа.

21.1 Формат сертификата

Что бы вы ни делали, держитесь подальше от сертификатов X.509. Почему? Ответ на этот вопрос вы найдете в [39]. Определить формат сертификата самому и то легче. В действительности сертификат — это всего лишь тип данных с обязательными и необязательными полями. Поскольку эта проблема из области программной инженерии, вдаваться в ее детали мы не будем. Однако обратите внимание: кодирование конкретной структуры данных должно быть уникальным, поскольку в криптографии структура данных часто подвергается хэшированию, чтобы подписать ее или сравнить с другой структурой данных. Работая с форматом наподобие XML, допускающим несколько представлений одной и той же структуры данных, необходимо тщательно следить за тем, чтобы цифровые подписи и хэш-коды всегда функционировали так, как положено.

21.1.1 Язык разрешений

В любой инфраструктуре открытого ключа (кроме, пожалуй, самых простых) вам придется ограничить полномочия сертификатов, которые может

выдавать подчиненный центр сертификации. Для этого необходимое ограничение нужно закодировать в сертификате подчиненного ЦС, что, в свою очередь, требует наличия языка, с помощью которого можно было бы описывать разрешения, предоставленные ключу. Это, вероятно, наиболее сложный момент разработки инфраструктуры открытого ключа, и здесь мы ничем не можем вам помочь. Ограничения, которые следует наложить на подчиненный ЦС, зависят от специфики конкретного приложения. Если вы не можете придумать подходящих ограничений, пересмотрите свое решение насчет использования PKI. При отсутствии ограничений в сертификате каждый подчиненный ЦС, по сути, получает в свое распоряжение главный ключ с неограниченными полномочиями, а это очень плохо для безопасности системы. Разумеется, вы можете ограничиться использованием единственного ЦС, но это лишит вас многих преимуществ инфраструктуры открытого ключа перед сервером ключей.

21.1.2 Ключ корневого ЦС

Чтобы центр сертификации мог выполнять какие-либо действия, ему нужна пара "открытый ключ-закрытый ключ". Генерация пары ключей — задание довольно простое. Открытый ключ ЦС вместе с некоторыми дополнительными данными (например, сроком действия) должен быть распространен между всеми участниками общения. Для упрощения системы это обычно выполняется с помощью сертификата, подписанного самим ЦС. Вообще-то это довольно странная конструкция: ЦС подписывает сертификат своего же открытого ключа. Хотя данный процесс иногда называют *самосертификацией (self-certification)*, на деле он не имеет с ней ничего общего. Это название — всего лишь историческая ошибка, которой мы почему-то придерживаемся до сих пор. Данный сертификат вообще не сертифицирует ключ и никоим образом не доказывает его безопасность. Каждый может создать открытый ключ и лично сертифицировать его. Все, что делает подобный сертификат, — это привязывает к ключу дополнительные данные. Сюда относятся список разрешений, срок действия, контактные данные соответствующего сотрудника и т.п. Сертификат, подписанный самим ЦС, использует тот же формат данных, что и другие сертификаты системы. Все остальные участники могут воспользоваться существующим кодом, чтобы проверить эти дополнительные данные. Сертификат, подписанный корневым ЦС, называется *корневым сертификатом (root certificate)* инфраструктуры открытого ключа.

Следующий шаг — безопасным образом распространить корневой сертификат между всеми участниками системы. Каждый должен знать ключ корневого ЦС, и каждый должен иметь правильный корневой сертификат.

Когда компьютер впервые присоединяется к инфраструктуре открытого ключа, ему следует безопасным образом передать корневой сертификат. Иногда для этого компьютер просто перенаправляют на локальный файл или файл, размещенный на доверенном Web-сервере, и сообщают ему, что это и есть корневой сертификат необходимой инфраструктуры открытого ключа. Начальное распределение корневого сертификата невозможно защитить криптографическими методами, так как у нас еще нет ключей, которые могли бы использоваться для аутентификации. Аналогичная ситуация возникает тогда, когда злоумышленнику удается взломать закрытый ключ центра сертификации. Поскольку ключ корневого ЦС больше не является безопасным, последнему необходимо инициировать совершенно новую инфраструктуру открытого ключа, а это подразумевает безопасное распространение корневого сертификата между всеми участниками системы. Неплохой мотив для сохранения ключа корневого ЦС в безопасности!

Через некоторое время срок действия ключа корневого ЦС будет окончен, и корневому ЦС понадобится издать новый ключ. Распространить новый корневой сертификат гораздо проще. Участники системы могут совершенно спокойно загрузить новый корневой сертификат из небезопасного источника. Поскольку этот сертификат подписан с помощью старого ключа корневого ЦС, он не может быть изменен злоумышленником. Единственная возможная проблема возникает в том случае, если участник не получит новый корневой сертификат. В большинстве систем, однако, сроки действия старого и нового ключей корневого ЦС перекрываются друг с другом на несколько месяцев. Этого времени вполне достаточно, чтобы перейти на использование нового ключа корневого ЦС.

В связи с изложенным следует отметить небольшой аспект реализации. Возможно, новый корневой сертификат должен иметь две цифровых подписи — сгенерированную с помощью старого ключа (чтобы пользователи смогли удостовериться в подлинности нового корневого сертификата) и сгенерированную с помощью нового ключа (который будет применяться новыми устройствами, добавленными в систему уже после окончания срока действия старого ключа). Этого можно добиться, включив в формат сертификата поддержку нескольких подписей или же просто выдав два отдельных сертификата для одного и того же нового ключа корневого ЦС.

21.2 Жизненный цикл ключа

Рассмотрим жизненный цикл одного ключа. Это может быть ключ корневого ЦС или любой другой открытый ключ. В процессе своего существования ключ проходит несколько этапов. В зависимости от специфики приложения,

некоторые ключи могут проходить не все этапы полного жизненного цикла. В качестве примера будем применять открытый ключ пользователя A.

- **Создание.** Первым этапом жизненного цикла каждого ключа является его создание. Пользователь A создает пару "открытый ключ-закрытый ключ" и сохраняет закрытый ключ в безопасном месте.
- **Сертификация.** Следующий этап — это сертификация. Пользователь A передает свой открытый ключ корневому или подчиненному ЦС, который сертифицирует данный ключ. Именно на этом этапе центр сертификации решает, какие полномочия следует предоставить открытому ключу пользователя A.
- **Распространение.** В зависимости от приложения, пользователю A может понадобиться распространить свой сертифицированный открытый ключ, прежде чем он сможет его применять. Если, например, пользователь A применяет свой ключ для подписывания сообщений, каждый из потенциальных собеседников пользователя A должен предварительно получить его открытый ключ. Для достижения наилучшего эффекта открытый ключ рекомендуется распространить за некоторое время до момента его первого использования. Это особенно важно в отношении нового корневого сертификата. Когда центр сертификации переходит на использование нового ключа, каждому участнику общения следует предоставить возможность познакомиться с новым корневым сертификатом, прежде чем он получит сертификат, подписанный посредством нового ключа.
- Необходимость реализации отдельного этапа распространения диктуется спецификой приложения, однако лучше, если без него можно обойтись. Назначение отдельного этапа распространения придется разъяснять пользователям. Кроме того, его наличие отражается и на пользовательском интерфейсе. Это, в свою очередь, создает массу дополнительных сложностей, поскольку многие пользователи не поймут, что означает этап распространения, и будут применять систему неправильно.
- **Активное использование.** Следующий этап жизненного цикла — активное использование ключа в транзакциях. Это и есть нормальная жизнедеятельность ключа.
- **Пассивное использование.** По окончании активного использования в жизненном цикле ключа должен наступить период, на протяжении которого ключ больше не применяется в новых транзакциях, но все еще принимается остальными участниками. Транзакции не всегда выполняются мгновенно; иногда их откладывают. Доставка подписанного письма по электронной почте вполне может затянуться на день-другой. Пользователю A следует прекратить активное применение ключа

и оставить достаточный запас времени, чтобы все неоконченные транзакции могли завершиться прежде, чем срок действия ключа будет исчерпан.

- **Окончание срока действия.** В конце своего жизненного цикла ключ устаревает и больше не считается действительным.

Как определяют этапы жизненного цикла ключа? Самое распространенное решение — явно задать в сертификате время перехода к каждому следующему этапу. Такой сертификат будет содержать дату начала этапа распространения, дату начала этапа активного использования, дату начала этапа пассивного использования и дату окончания срока действия. К сожалению, со всеми этими датами необходимо ознакомить и пользователя, так как они влияют на способ работы сертификата. Между тем для среднестатистических пользователей все это может оказаться слишком сложно.

Более гибкая схема подразумевает наличие централизованной базы данных, которая будет содержать список этапов жизненного цикла для каждого ключа. Этот подход, однако, влечет за собой массу совершенно новых проблем безопасности, которыми мы предпочли бы не заниматься. Кроме того, если у вас есть список отзыва сертификатов, он может аннулировать содержимое базы данных и привести к моментальному устареванию ключа.

Ситуация еще более усложняется, если пользователь A хочет применять один и тот же ключ в нескольких различных инфраструктурах открытого ключа. В общем случае этот подход кажется нам неудачным, но иногда без него просто не обойтись. Предположим, пользователь A носит с собой небольшой модуль, защищенный от несанкционированного вмешательства. Этот модуль содержит закрытые ключи пользователя A и выполняет вычисления, необходимые для создания цифровой подписи. Подобные модули имеют ограниченный объем памяти для хранения данных. Сертификаты открытых ключей пользователя A могут храниться в корпоративной сети без каких-либо ограничений на их размер, но маленький модуль не сможет вместить в себя неограниченное количество закрытых ключей. В подобных ситуациях пользователю A приходится применять один и тот же ключ в нескольких различных инфраструктурах. Из этого также следует, что жизненный цикл ключа должен быть одинаков для всех инфраструктур открытого ключа, в которых состоит пользователь A. Скоординировать подобные параметры довольно сложно.

Если вам когда-нибудь придется работать с системами такого рода, убедитесь, что цифровая подпись, используемая в одной инфраструктуре открытого ключа, не будет фигурировать в других инфраструктурах. Всегда придерживайтесь одной и той же схемы цифровой подписи, например описанной в разделе 13.7. Подписываемая строка байтов никогда не должна сов-

падать для двух разных приложений или инфраструктур открытого ключа. Этого легко добиться, включив в подписываемую строку данные, которые уникальным образом идентифицируют приложение и инфраструктуру открытого ключа.

21.3 Почему ключи изнашиваются

Уже не раз отмечалось, что ключи следует периодически менять, но почему?

В идеальном мире один и тот же ключ мог бы использоваться на протяжении очень долгого времени. При отсутствии слабых мест в защите системы злоумышленнику не остается ничего другого, как проводить поиск путем полного перебора вариантов. Теоретически это сводит нашу проблему к задаче выбора ключей достаточно большого размера.

К сожалению, наш мир не идеален. Секретность ключа всегда находится под угрозой. Ключ нужно где-то хранить, и злоумышленник может туда добраться. Ключ также может находиться в использовании, а каждое применение ключа несет в себе еще одну потенциальную угрозу его безопасности. Ключ должен быть передан из места своего хранения туда, где будут проводиться вычисления. Зачастую процесс передачи ключа ограничивается рамками одного компьютера, но и это открывает для злоумышленника очередную возможность нападения. Если злоумышленник может прослушивать канал общения, который применяется для передачи ключа, он непременно получит копию этого ключа. И наконец, над ключом выполняются разнообразные криптографические операции. Мы не знаем сколько-нибудь полезных криптографических функций, которые бы имели доказательства своей безопасности. В их основе обычно лежат сомнительные аргументы наподобие: “Пока что никому из нас не удалось обнаружить способ нападения на эту функцию, поэтому она выглядит вполне безопасной”[1].

Чем дольше мы храним ключ и чем больше его используем, тем выше вероятность того, что злоумышленнику удастся его взломать. Если мы хотим ограничить вероятность того, что злоумышленник узнает ключ, следует ограничить время жизни этого ключа. По сути, ключ как будто изнашивается.

Существует еще одна причина, требующая ограничить время жизни ключа. Предположим, в системе произойдет что-нибудь неприятное и злоумыш-

[1] То, что часто называют “доказательством безопасности” криптографических функций, на деле таковым не является. Подобные доказательства представляют собой не более чем простое логическое приведение: если мы можем взломать функцию А, то можем взломать и функцию Б. Это позволяет сократить количество примитивных операций, выполнение которых предполагается безопасным, однако это ни в коей мере не является реальным доказательством безопасности.

ленник все-таки узнает ключ. Это нарушит безопасность системы и приведет к нанесению некоторого ущерба. (Отзыв сертификата эффективен только в том случае, если вы узнаете, что злоумышленник взломал ключ. Умный злоумышленник никогда не даст себя обнаружить.) Нанесение ущерба будет продолжаться до тех пор, пока вы не смените ключ. Ограничивая время жизни одного ключа, мы ограничиваем время воздействия злоумышленника на систему в случае его успешных действий.

Как видите, короткое время жизни ключа имеет два преимущества. Оно снижает вероятность того, что злоумышленник узнает ключ, и ограничивает объем ущерба, который будет нанесен системе, если злоумышленник все же добьется успеха.

Каким же должно быть разумное время жизни ключа? Все зависит от ситуации. Изменение ключей влечет за собой дополнительные расходы, поэтому менять их слишком часто тоже не нужно. С другой стороны, если вы меняете ключи только раз в 10 лет, можете не сомневаться: когда через 10 лет вы запустите функцию обновления ключей, она не сработает. Как показывает практика, функция или процедура, которая редко используется или тестируется, на деле оказывается неисправной[2]. Пожалуй, наибольшая опасность применения "долгоживущих" ключей состоит в том, что функция обновления ключей никогда не используется, а следовательно, в случае необходимости вряд ли сработает так, как нужно.

Процедуры изменения ключей, требующие участия пользователя, относительно дороги, а потому должны выполняться нечасто. Рекомендуемый срок жизни таких ключей составляет от одного месяца и выше. Управление ключами с меньшим временем жизни должно осуществляться автоматически.

21.4 Так что же нам делать?

К сожалению, мы не можем ответить на этот вопрос, ничего не зная о приложении, с которым вы работаете, и об окружении, в котором оно будет функционировать. Мы можем дать массу подробных советов относительно функций шифрования и даже протоколов согласования ключей, но управление ключами — проблема не столько криптографическая, сколько касающаяся взаимодействия с реальным миром. В этой и предыдущих главах вы познакомились с общими принципами, которых, конечно же, недостаточно для разработки системы управления ключами. Тем не менее, на наш взгляд, невозможно дать хороший совет, ничего не зная о конкретной ситуации. Впрочем, давать личные советы в книге вряд ли уместно.

[2] Данный стереотип применим ко всем сферам нашей жизни и является главной причиной того, почему мы должны всегда проверять процедуры экстренного реагирования (например, проводить пожарные учения).

Глава 22

Хранение секретов

В разделе 9.3 обсуждается хранение временных секретов, таких, как ключи сеанса. А где же хранить долгосрочные секреты — скажем, пароли или закрытые ключи? В контексте безопасности хранение секретов должно соответствовать двум прямо противоположным требованиям. Во-первых, секреты должны сохраняться в секрете. Во-вторых, риск полной потери секретов (т.е. невозможность найти собственноручно спрятанный секрет) должен быть минимальным.

22.1 Диск

Пожалуй, наиболее очевидное решение — хранить секреты на жестком диске компьютера или другого постоянного хранилища данных. Это, однако, приносит пользу только в том случае, если компьютер находится в безопасности. Если пользователь А хранит на компьютере свои ключи (в незашифрованном виде), доступ к этим ключам смогут получить все, кто работает за этим компьютером. Большинство компьютеров хотя бы от случая к случаю используются другими людьми. Возможно, пользователь А не против, чтобы за его компьютером поработал кто-нибудь другой, но, конечно же, он не собирается предоставлять этому человеку доступ к своему банковскому счету! Еще одна проблема заключается в том, что пользователь А может работать с несколькими компьютерами. Если его ключи хранятся на домашнем компьютере, он не сможет использовать их на работе или во время путешествия. Да и вообще, где лучше хранить ключи — на домашнем настольном компьютере или в ноутбуке? Совсем не к чему хранить копии ключей в нескольких местах; это еще больше ослабляет систему.

Гораздо лучше, если ключи пользователя А будут храниться в его карманном компьютере (КПК), который гораздо реже одалживают другим людям

и, кроме того, всегда носят с собой. (В качестве альтернативы КПК можно было бы применять мобильный телефон или наручные часы, но использование их для хранения секретов потребовало бы обновления соответствующей инфраструктуры, что выходит за рамки наших возможностей.)

Вам, вероятно, кажется, что шифрование секретов улучшило бы их безопасность. Разумеется, это так, но шифрование с помощью чего? Для шифрования секретов нужен главный ключ, который, в свою очередь, тоже должен где-то храниться. Хранение его вместе с зашифрованными секретами не принесло бы никакой пользы. Тем не менее шифрование секретных данных — *действительно* хороший метод сокращения их количества и размера, а потому оно широко используется в сочетании с другими методами. Например, закрытый ключ RSA может иметь несколько тысяч бит в длину. Применив к нему симметричное шифрование, мы можем значительно сократить объем пространства, необходимый для хранения этого ключа.

22.2 Человеческая память

Следующая идея — хранить ключ в памяти самого пользователя А. Мы попросим его запомнить пароль, а затем зашифруем с помощью этого пароля все остальные ключи. Зашифрованные ключи могут быть помещены в любое удобное место, например на жесткий диск. Кроме того, ключи можно разместить на Web-сервере, чтобы пользователь А загружал их на любой компьютер, с которым работает в данный момент.

Человечество славится своими никудышными способностями к запоминанию паролей. Если мы выберем очень простые пароли, то не получим никакой безопасности. Простых паролей слишком мало, чтобы они обеспечивали необходимую секретность: злоумышленник может просто их перебрать. Использование в качестве пароля, скажем, девичьей фамилии своей матери не обеспечит должной защиты; ее фамилия, скорее всего, известна многим. Впрочем, даже если это и не так, злоумышленнику будет достаточно перебрать несколько тысяч фамилий, чтобы отыскать нужную.

Хороший пароль должен быть непредсказуем. Другими словами, он должен содержать большое количество энтропии. Обычные слова не обладают достаточной энтропией. Английский язык насчитывает около полумиллиона слов (и это включая все длинные и непонятные слова, внесенные в полный словарь языка), поэтому применение в качестве пароля одного слова из словаря обеспечит не более 19 бит энтропии. Оценки количества энтропии на один символ текста, написанного на английском языке, несколько отличаются, но в среднем составляют примерно 1,5-2 бит на один символ.

Как вы помните, в наших системах для достижения 128-битового уровня безопасности использовались 256-битовые ключи. В большинстве случаев использование 256-битовых ключей лишь немного повышает общие расходы. Тем не менее в данной ситуации пользователь должен запомнить пароль (или ключ), а дополнительная стоимость ключей большого размера весьма высока. Использовать пароли с энтропией 256 бит слишком обременительно, поэтому ограничим количество энтропии, содержащейся в паролях, до 128 бит[1].

Исходя из оптимистической оценки в 2 бита энтропии на символ, для получения 128 бит энтропии нужны пароли в 64 символа длиной. Это неприемлемо. Пользователи просто откажутся запоминать пароли такого размера.

Что, если мы пойдем на компромисс и согласимся с энтропией 64 бит? Это уже минимально допустимый уровень безопасности. Если на каждый символ пароля приходится по 2 бита энтропии, нужно, чтобы пароль содержал по крайней мере 32 символа. Даже подобное послабление окажется непосильной задачей для среднестатистического пользователя. Не забывайте — на практике длина большинства паролей составляет лишь 6-8 символов.

Для решения этой проблемы пароли можно было бы назначать автоматически. Впрочем, вы когда-нибудь пытались работать с системой, которая радостно сообщает, что ваш пароль выглядит как “719327582742994690518 6”? Или, скажем, “aoekjk3ncmakwe”? Обычный человек просто не в состоянии запомнить подобные вещи, поэтому данный ход не пройдет. (На практике пользователь, как правило, записывает свой пароль на бумажке, но об этом речь идет в следующем разделе.)

Гораздо лучше использовать в качестве пароля какую-нибудь *идентификационную фразу (passphrase)*. Идентификационная фраза — это практически то же самое, что и пароль. В действительности эти понятия настолько схожи, что мы предпочитаем не проводить между ними различий. По сути, различие лишь одно: идентификационная фраза намного длиннее однословного пароля.

В качестве пароля пользователь А мог бы применять какую-нибудь фразу наподобие: “Розовые занавески порхают над океанами”. Это совершенно бессмысленная фраза, но запомнить ее нетрудно. Кроме того, данная фраза состоит из 38 символов, поэтому содержит примерно 57-76 бит энтропии. Если же пользователь А расширит эту фразу примерно до следующей: “Занавески в розовый горошек порхают над морями любви”, он получит 52 символа, которые могут применяться в качестве вполне неплохого ключа с энтропией 78-104 бит. Набрать на клавиатуре такую фразу можно всего за несколько секунд, что, разумеется, гораздо быстрее, нежели набирать строку случай-

[1] Для математиков: это пароли, выбранные с помощью вероятностного распределения, обладающего энтропией в 128 бит.

ных чисел. Мы исходим из факта, что запомнить идентификационную фразу намного проще, чем случайные числа. Большинство мнемонических приемов основаны на идее преобразования случайных данных в нечто похожее на наши идентификационные фразы.

Некоторые пользователи не слишком любят стучать пальцами по клавиатуре, поэтому выбирают свои пароли по-другому. Возьмем, например, поистине замечательное слово "Длспус,иносивсссцмбпсн?". Оно выглядит полнейшей абракадаброй, если не знать, что это первые буквы слов некоторого вполне осмысленного предложения. В данном случае мы воспользовались цитатой из шекспировского "Гамлета": "Достойно ль смиряться под ударами судьбы, иль надо оказать сопротивленье и в смертной схватке с целым морем бед покончить с ними?"[2] Разумеется, на практике использовать цитаты из классической литературы не следует. Литературные тексты слишком доступны злоумышленнику, а сколько подходящих фраз может содержаться в книгах, стоящих на полке у пользователя А? Поэтому пользователю А следует придумать что-нибудь оригинальное, чего никто не мог бы разгадать.

По сравнению с полной идентификационной фразой для составления такого пароля требуется предложение, состоящее из большего количества слов. Тем не менее подобный прием позволяет обеспечить необходимый уровень безопасности путем ввода меньшего количества символов, так как первые буквы слов обладают большей случайностью, чем последовательные буквы предложения. К сожалению, мы не знаем ни одной оценки количества энтропии, которая приходится на каждый символ подобного слова. Возможно, впоследствии кому-нибудь удастся провести исследования и написать статью о различных способах выбора идентификационных фраз.

Идентификационные фразы — это, безусловно, самый лучший способ сохранить секрет в человеческой памяти. К сожалению, многие пользователи все равно испытывают серьезные затруднения с их применением. Кроме того, даже используя идентификационные фразы, весьма сложно сохранить в человеческом мозгу сразу 128 бит энтропии.

22.2.1 Солим и растягиваем

Чтобы выжать как можно больше безопасности из пароля или идентификационной фразы с ограниченным объемом энтропии, можно воспользоваться двумя любопытными приемами. Их названия наводят на мысль о средневековой камере пыток. Данные приемы настолько просты и очевидны, что должны применяться в каждой системе, работающей с паролями. Исключений из этого правила не существует.

[2] Перевод Б. Пастернака. — *Прим. перев.*

Первый прием состоит в том, чтобы добавить *соли (salt)*. "Соль" — это случайное число, которое хранится вместе с данными, зашифрованными с помощью пароля. Если это возможно, используйте 256-битовую соль.

Следующий шаг — это *растягивание (stretching)* пароля. По своей сути растягивание представляет собой очень длинный процесс вычисления. Пусть p — это пароль, а s — соль. Используя любую криптографически сильную функцию хэширования h, можно вычислить следующее:

$$\begin{aligned} x_0 &:= 0, \\ x_i &:= h(x_{i-1} \parallel p \parallel s) \text{для } i = 1, \ldots, r, \\ K &:= x_r. \end{aligned}$$

Полученное значение K применяется в качестве ключа для фактического шифрования данных. Параметр r — это количество итераций данного алгоритма, которое должно быть настолько большим, насколько это практически возможно. (Думаем, нам не нужно лишний раз подчеркивать, что значения x_i и K должны быть 256 бит длиной.)

Рассмотрим описанный процесс с точки зрения злоумышленника. Зная соль s и некоторые данные, зашифрованные с помощью ключа K, мы пытаемся найти K, перебирая различные пароли. Для этого мы выберем некоторый пароль p, вычислим соответствующее значение K, расшифруем данные и посмотрим, имеет ли смысл полученный открытый текст. Если расшифрованный текст окажется бессмысленным, выбранный пароль p должен быть неправильным. Чтобы проверить одно значение p, необходимо выполнить r различных операций хэширования. Чем больше значение r, тем больше работы придется проделать злоумышленнику.

В нормальных ситуациях растягивание пароля должно выполняться при каждом использовании последнего. Напомним, что это происходит в тот момент, когда пользователь только что ввел пароль. Обычно ввод пароля занимает несколько секунд, поэтому вполне допустимо потратить на его обработку, скажем, еще 200 мс. Вот наше правило выбора r: значение r должно быть таким, чтобы вычисление $K(s, p)$ занимало от 200 до 1000 мс на оборудовании пользователя. С течением времени компьютеры работают все быстрее и быстрее, поэтому r должно постепенно увеличиваться. В идеале приемлемое значение r следует определять экспериментальным путем, когда пользователь впервые выбирает пароль, и хранить r вместе с s. (Обязательно убедитесь, что значение r не слишком мало и не слишком велико.)

Какой выигрыш мы получили? Если $r = 2^{20}$ (это чуть больше миллиона), тогда злоумышленнику для проверки каждого варианта пароля придется проделать по 2^{20} операции хэширования. Перебор 2^{60} паролей требует выполнения 2^{80} операций хэширования, поэтому использование $r = 2^{20}$ экви-

валентно увеличению размера пароля на 20 бит. Чем больше значение r, тем больший выигрыш в размере пароля мы получаем.

Теперь посмотрим на эту ситуацию с другой стороны. Параметр r не дает злоумышленнику извлечь пользу из появления высокоскоростных компьютеров, поскольку с увеличением мощности компьютеров возрастает и значение r. Это нечто наподобие компенсатора закона Мура (Moore's law), но только в долгосрочной перспективе. Через 10 лет злоумышленник может воспользоваться технологией следующего десятилетия, чтобы взломать пароли, с которыми мы работаем сегодня. Поэтому нам все еще нужно обеспечивать приличный уровень безопасности и использовать пароли, которые содержат большое количество энтропии.

Это, между прочим, еще один аргумент в пользу применения протокола согласования ключей с прямой безопасностью. Каким бы ни было наше приложение, вполне вероятно, что закрытые ключи пользователя A будут защищены паролем. Через 10 лет у злоумышленника появится возможность определить пароль пользователя A. Но если ключ, зашифрованный с помощью пароля, применялся только для запуска протокола согласования ключей с прямой безопасностью, тогда, даже взломав ключ, злоумышленник не узнает ничего ценного. К тому времени ключ пользователя A больше не будет действительным (он давным-давно устареет), а знание его старого закрытого ключа никак не поможет раскрыть ключи сеанса, которые применялись 10 лет назад.

Наличие соли мешает злоумышленнику сэкономить свои усилия, когда он пытается одновременно взломать большое количество паролей. Предположим, что у системы есть миллион пользователей. Каждый из них хранит на диске файл со своими ключами, зашифрованный с помощью растянутого пароля соответствующего пользователя. Если бы мы не использовали соль, злоумышленник мог бы атаковать систему следующим образом: взять пароль p, вычислить растянутый ключ K и попытаться расшифровать каждый из этих файлов с помощью K. Для каждого пароля значение функции растягивания нужно подсчитать только один раз, а полученный ключ может быть использован злоумышленником, чтобы попытаться расшифровать сразу миллион файлов.

Добавляя к функции растягивания соль, мы делаем описанную ситуацию невозможной. Все значения соли — случайные числа, поэтому каждый пользователь будет применять свою соль. Теперь злоумышленнику придется подсчитывать значение функции растягивания не для каждого пароля, а для каждой комбинации "пароль-файл". Это потребует от него намного бóльше усилий, а для пользователей системы расходы окажутся незначительными.

Какого размера должно быть значение соли? Мы не будем отвлекаться на подробный анализ этого вопроса. Возможно, вам хватит и 128-битового зна-

чения (если система не допускает проведения атак, в основе которых лежит парадокс задачи о днях рождения), но зачем экономить на таких мелочах? В наше время биты очень дешевы, и использование 256-битового значения соли удовлетворит любые потребности.

Кстати, будьте очень внимательны, используя соль и растягивание. Однажды нам попалась система, в которой были замечательно реализованы оба этих приема. Затем, однако, какому-то программисту пришло в голову улучшить пользовательский интерфейс, предоставляя более быстрый ответ относительно того, правилен ли введенный пользователем пароль. Для этого программист сохранял контрольную сумму каждого пароля, что свело на нет всю процедуру применения соли и растягивания. Если время отклика системы оказывается недопустимо медленным, вы можете немного уменьшить значение r, но убедитесь, что злоумышленник сможет проверить правильность пароля, только выполнив по меньшей мере r операций хэширования.

22.3 Портативное хранилище

Очередная идея — хранить ключи где-нибудь за пределами компьютера. Самая простая форма внешнего хранилища — это клочок бумаги, на котором записаны пароли. Многие заводят хранилища подобного рода даже для абсолютно не криптографических систем наподобие Web-узлов. Очень часто нам (в том числе и авторам этой книги) приходится помнить по меньшей мере полдюжины паролей, а это не очень-то легко, особенно если речь идет о системах, в которых пароль используется крайне редко. Чтобы запомнить пароли, пользователи просто их записывают. Ограничение этого решения состоит в том, что пароль все еще должен обрабатываться глазами, мозгом и пальцами пользователей при каждом его применении. Чтобы раздражение пользователя и совершаемые им при вводе ошибки не превышали разумных пределов, данный прием может применяться только к паролям и идентификационным фразам с относительно низкой степенью энтропии.

Как разработчику, вам не придется предпринимать каких-либо действий для реализации данного метода хранения. Пользователи всегда будут записывать пароли на клочках бумаги, какие бы правила вы ни установили и как бы ни настроили систему паролей.

Более совершенная форма хранилища представляет собой некоторую разновидность переносимой памяти. Это может быть карта флэш-памяти, дискета, пластиковая карта с магнитной полосой или любой другой вид цифрового хранилища. Системы цифрового хранения всегда обладают достаточной емкостью для хранения, как минимум, 256-битового секретного ключа, поэтому можно забыть об использовании паролей с низкой степенью энтропии. По су-

ти, переносимая память сама превращается в ключ. Тот, у кого есть ключ, автоматически получает доступ к системе, поэтому переносимую память нужно хранить в безопасном месте.

22.4 Идентификатор безопасности

Более удачное (и более дорогостоящее) решение — это использование устройства, названного *идентификатором безопасности (secure token)*. Это маленький компьютер, который пользователь может носить с собой. Внешний вид идентификатора безопасности может варьироваться в широких пределах — от смарт-карты (которая выглядит в точности как кредитная карта) до приборов iButton, USB-устройств или карт формата PCMCIA. Главными особенностями такого устройства являются наличие энергонезависимой памяти (т.е. памяти, которая сохраняет свои данные после выключения компьютера) и центрального процессора.

Идентификатор безопасности функционирует аналогично портативному хранилищу данных, однако имеет ряд улучшений в плане безопасности. В частности, доступ к ключам, хранящимся в таком устройстве, может быть ограничен паролем или каким-либо другим способом. Прежде чем идентификатор безопасности позволит вам применить ключ, вы должны предоставить ему правильный пароль. При этом идентификатор безопасности сам может защитить себя от злоумышленников, пытающихся подобрать пароль, блокируя доступ после трех или пяти неудачных попыток ввода. Разумеется, некоторые пользователи то и дело ошибаются при вводе пароля. Для восстановления доступа к системе таких горе-пользователей применяются специальные, гораздо более длинные ключи или идентификационные фразы с более высокой степенью энтропии.

Как видите, подобные системы обеспечивают многоуровневую защиту данных. Прежде всего пользователь защищает идентификатор физически; например, он может хранить его в кошельке или на брелке. Чтобы добраться к данным, злоумышленник должен украсть сам идентификатор или по крайней мере получить к нему какой-нибудь доступ. Затем ему придется либо физически взломать идентификатор и извлечь из него данные, либо узнать пароль, чтобы разблокировать идентификатор. Для усложнения физической атаки подобные устройства часто делают устойчивыми к несанкционированному вмешательству[3].

[3]Такие идентификаторы устойчивы к несанкционированному вмешательству, но отнюдь не защищены от него полностью. Физический взлом устройств возможен всегда; вопрос лишь в том, во сколько он обойдется.

На сегодняшний момент идентификаторы безопасности являются одним из лучших и самых практичных методов хранения секретных ключей. Они относительно недороги и имеют довольно небольшой размер, что позволяет носить их с собой.

Одна из проблем практического применения идентификаторов безопасности — халатное поведение пользователей. Они то и дело оставляют свой идентификатор безопасности подключенным к компьютеру, когда уходят на обед или деловую встречу. Поскольку пользователям ужасно не хочется повторно вводить пароль, их система будет настроена таким образом, чтобы послушно предоставлять доступ к данным на протяжении многих часов с момента последнего ввода пароля. В подобной ситуации злоумышленнику достаточно подойти к компьютеру и воспользоваться секретными ключами, хранящимися в идентификаторе.

Вы можете попробовать решить эту проблему путем обучения. Среди любимых средств воздействия на сознание пользователей можно назвать видеопрезентации "Корпоративная безопасность в офисе", отвратительные и совсем не смешные плакаты "Возьмите свои смарт-карты с собой на обед", а также лекции на тему "Если я еще когда-нибудь увижу смарт-карту вставленной в компьютер, хозяин которого ушел покурить, он прослушает эту лекцию еще 50 раз". Впрочем, существуют и другие пути решения этой проблемы. Сделайте так, чтобы идентификатор безопасности был ключом не только к секретным данным, но и к дверям офиса. Теперь пользователь будет вынужден брать идентификатор с собой, чтобы попасть обратно в офис. Настройте автомат, продающий кофе, таким образом, чтобы он выдавал стаканчик с кофе только при наличии идентификатора безопасности. Это не позволит пользователям оставлять идентификаторы подключенными к компьютеру на время своего отсутствия. Иногда безопасность системы полностью обеспечивается подобными примитивными мерами, но они оказываются гораздо эффективнее многочисленных плакатов, правил и призывов.

22.5 Безопасный пользовательский интерфейс

Несмотря на бесспорные достоинства, идентификатор безопасности имеет один существенный недостаток. Пароль, который используется для доступа к содержимому идентификатора, должен вводиться в компьютер или другое аналогичное устройство. Это не составит проблемы, если мы полностью доверяем своему компьютеру, однако, как известно, компьютеры не настолько хорошо защищены, чтобы можно было на них надеяться. Действительно, смысл хранения секретных ключей за пределами компьютера состоит именно в том, что мы не совсем ему доверяем. Можно достичь более высокого уровня

безопасности, если сам идентификатор будет оснащен встроенным безопасным пользовательским интерфейсом. Теперь пароль (или, скорее всего, PIN-код) можно вводить прямо в идентификатор безопасности, не полагаясь на внешнее устройство.

Наличие у идентификатора безопасности собственной клавиатуры защитит PIN-код от взлома. Разумеется, после введения PIN-кода компьютер все равно узнает ключ и может делать с этим ключом все, что ему вздумается. Таким образом, мы все еще ограничены безопасностью компьютера в целом.

Чтобы предотвратить подобные проблемы, необходимо внедрить криптографические процессы, выполняемые при участии секретного ключа, в сам идентификатор безопасности. Это требует применения в идентификаторе специализированного кода, благодаря чему тот быстро превращается в полноценный компьютер. На сей раз, однако, это будет доверенный компьютер, который пользователь может носить с собой. Доверенный компьютер может реализовать компоненты каждого приложения, критические для безопасности, на самом идентификаторе безопасности. В подобной ситуации для идентификатора безопасности важно наличие дисплея, чтобы пользователь видел, какие действия он авторизует, вводя свой PIN-код. В типичной ситуации пользователь управляет приложением с помощью мыши и клавиатуры основного компьютера. Когда, например, пользователь должен авторизовать банковский платеж, компьютер отправляет соответствующие данные идентификатору безопасности. Последний отображает на своем дисплее сумму платежа и другие параметры транзакции, а пользователь авторизует платеж, введя свой PIN-код. Затем идентификатор подписывает параметры транзакции, а основной компьютер завершает ее выполнение.

На практике идентификаторы с безопасным пользовательским интерфейсом слишком дороги для большинства приложений. Пожалуй, самой близкой альтернативой идентификатору безопасности является КПК (например, Palm). Тем не менее пользователи загружают на свои КПК многочисленные программы, а разработчики КПК не позиционировали их изначально как безопасные устройства, поэтому, на наш взгляд, КПК ненамного безопаснее обычных компьютеров.

Описанная ситуация может служить хорошим примером конфликта между функциональностью и безопасностью. Для пользователей важно иметь возможность загружать программы и запускать их, когда они пожелают. Кроме того, им хотелось бы доверять своему компьютеру ценную информацию. Учитывая текущее положение дел в обеспечении безопасности операционных систем, мы просто не в состоянии совместить в одном и том же устройстве гибкость, которую предоставляет возможность загрузки программ, и уровень безопасности, который требуется системе. А если придется делать выбор между безопасностью и возможностью загрузки программы,

которая отображает на экране танцующих свинок, пользователи, вне всяких сомнений, выберут последнее.

22.6 Биометрика

Те, кто интересуется последними веяниями моды, могут добавить к своему арсеналу средств безопасности еще и биометрику. В идентификатор безопасности можно встроить что-нибудь наподобие сканера отпечатков пальцев или сетчатки глаза. В настоящее время биометрические устройства еще не получили широкого распространения. Сканеры отпечатков пальцев стоят не так уж и дорого, но безопасность, которую они обеспечивают, крайне сомнительна. В 2002 году криптограф Цутому Мацумото (Tsutomu Matsumoto) со своими тремя учениками показал, как обмануть все имеющиеся в продаже сканеры отпечатков пальцев, используя только подручные материалы [63]. Оказалось, что с созданием искусственного пальца по свежему отпечатку (как те, которые мы оставляем на гладкой поверхности) справится любой сообразительный старшеклассник.

В действительности нас шокировало не то, что устройства считывания отпечатков пальцев можно обмануть, а то, *как* это просто. Что еще хуже, производители биометрических устройств постоянно рассказывали о том, какую высокую безопасность обеспечивает биометрическая идентификация. Они никогда не говорили, как легко подделать отпечатки пальцев. И вдруг на сцене появляется математик (даже не эксперт по биометрике!), который одним взмахом руки развенчивает все устоявшиеся мифы. У нас есть два предположения: либо производители биометрических устройств знали об этом и бесстыдно обманывали нас, либо они не знали об этом, а значит, совершенно некомпетентны в своей области. Нам нужно действительно хорошо подумать о том, насколько безопасны остальные биометрические системы.

Тем не менее, даже несмотря на то что сканеры отпечатков пальцев легко обмануть, они все еще могут сослужить хорошую службу. Предположим, у нас есть идентификатор безопасности с небольшим дисплеем, небольшой клавиатурой и сканером отпечатков пальцев. Чтобы добраться к ключу, необходимо получить физический доступ к идентификатору безопасности, узнать PIN-код и подделать отпечаток пальца. Все это требует от злоумышленника намного бóльших усилий, чем любое из предыдущих решений. Это, пожалуй, самая лучшая из практических схем хранения ключей, которую можно реализовать на данный момент. С другой стороны, подобный идентификатор безопасности будет стоить недешево, поэтому использовать его смогут далеко не все.

Сканеры отпечатков пальцев могут применяться для обеспечения безопасности данных и на более примитивном уровне. Прикосновение пальца к сканеру занимает всего лишь доли секунды, поэтому вполне допустимо, чтобы пользователи проделывали эту процедуру относительно часто. Таким образом, применение сканера отпечатков пальцев может повысить уверенность в том, что действия компьютера были авторизованы именно тем человеком, которому это поручено. Теперь сотруднику предприятия будет сложнее разрешить коллеге пользоваться своим паролем. Вместо того чтобы защищать систему от хитроумных злоумышленников, сканер отпечатков пальцев будет предотвращать случайные нарушения правил безопасности. Это может принести гораздо бóльшую пользу безопасности системы, нежели попытка использовать сканер отпечатков пальцев в качестве высоконадежного средства защиты.

Отметим также, что работа Мацумото может иметь серьезные последствия для использования отпечатков пальцев в качестве криминальных улик. Оказывается, что сделать искусственный палец по имеющемуся отпечатку не так уж сложно. Так почему бы не смазать искусственный палец чем-нибудь жирным и не наставить чужих отпечатков в нужных местах? До сих пор подобные трюки можно было увидеть только в шпионских фильмах, однако, как вы убедились, реализовать подобную идею несложно и на практике. Выходит, что наличие отпечатков пальцев на орудии убийства значит отнюдь не так много, как мы привыкли думать. Пройдет не менее 10 лет, прежде чем судебная система свыкнется с этой мыслью, а пока что создание искусственных пальцев остается безотказным средством оклеветать невинного человека.

22.7 Однократная регистрация

Как уже не раз отмечалось, среднестатистическому пользователю приходится запоминать слишком много паролей. В связи с этим особую привлекательность приобретает идея *однократной регистрации (single sign-on — SSO)*. Она состоит в том, чтобы предоставить пользователю один главный пароль, который, в свою очередь, будет использоваться для расшифровки всех остальных паролей, применяемых пользователем в различных приложениях.

Для реализации этой схемы все приложения должны уметь взаимодействовать с системой однократной регистрации. Каждый раз, когда приложению нужен пароль, оно должно запрашивать его не у пользователя, а у системы однократной регистрации. Однако на практике это не срабатывает. Данный процесс еще не стандартизирован, а пока этого не случится, он не будет происходить автоматически. Только подумайте, сколько различных

приложений придется подвергнуть изменениям, чтобы они автоматически получали свои пароли у системы однократной регистрации.

Более простая идея состоит в том, чтобы создать небольшую программу, которая будет хранить пароли в текстовом файле. Пользователь А вводит свой главный пароль, а затем использует привычные операции копирования и вставки, чтобы скопировать пароль из системы однократной регистрации в нужное приложение. Брюс однажды представил широкой общественности свою программу под названием Password Safe, которая занималась именно этим. К сожалению, данная программа не более, чем цифровая версия клочка бумаги, на котором пользователь А записывает свои пароли. Разумеется, она очень полезна и определенно лучше клочка бумаги, если мы всегда используем один и тот же компьютер. Тем не менее она никак не является тем самым глобальным решением, которым должна была бы стать идея однократной регистрации.

22.8 Риск утраты

А что, если идентификатор безопасности сломается? Или пользователь забудет свой клочок бумаги с паролями в кармане брюк и затем бросит их в стиральную машину? Утрата секретных ключей — это всегда плохо. Ее цена может варьироваться от необходимости перерегистрации в каждом приложении для получения нового ключа до безвозвратной потери доступа к важным данным. Если вы зашифруете с помощью секретного ключа свою докторскую диссертацию, над которой усердно работали на протяжении пяти лет, а затем потеряете этот ключ, у вас больше не будет докторской диссертации. Все, что у вас останется, — это файл с грудой случайных битов. Какой кошмар!

Довольно сложно создать систему хранения ключей, которая была бы проста в использовании и вместе с тем обладала бы высокой надежностью. Поэтому данные функции необходимо разнести. Сохраните две копии ключа — одну, которая будет проста в использовании, и другую, обладающую высокой надежностью. Если первая система когда-нибудь потеряет ключ, вы сможете восстановить его с помощью второй системы, которая вовсе не обязательно должна быть сложной. Как насчет того, чтобы написать ключ на листке бумаги и положить его в банковский сейф?

Несомненно, разрабатывать надежную систему хранения следует очень тщательно. Благодаря своим особенностям она быстро станет местом хранения всех секретных ключей, а следовательно, превратится в крайне привлекательный объект для нападения. Необходимо провести анализ рисков, чтобы определить, какой вариант хранения вам больше подходит: несколько небольших надежных систем или же одна большая.

22.9 Совместное владение секретом

Некоторые ключи должны содержаться в условиях повышенной безопасности, например закрытый ключ корневого центра сертификации. Как уже отмечалось, сохранение ключа в безопасности может оказаться весьма сложной задачей. Еще труднее одновременно обеспечить его безопасность и надежность.

Существует одно криптографическое решение, которое может применяться для хранения секретных ключей. Оно называется *совместным владением секретом (secret sharing)*, что не совсем корректно, поскольку означает, что ваш секрет известен еще нескольким людям. Это не так. Идея состоит в том, чтобы разбить секрет на несколько различных частей. Это можно сделать таким образом, чтобы для восстановления секрета требовалось собрать вместе, скажем, три из пяти частей. Затем каждая часть секрета передается одному из руководителей компании. Вся хитрость такого приема заключается в том, что любые вместе взятые два человека не знают о секретном ключе абсолютно ничего.

С академической точки зрения системы совместного владения секретом выглядят крайне привлекательно. Каждую часть секрета можно хранить с помощью описанных ранее методов. Правило "k из n" сочетает в себе высокую безопасность (для восстановления ключа нужны, как минимум, k людей) с высокой надежностью ($n-k$ частей секрета могут быть утеряны без каких-либо негативных последствий). Существуют и более изощренные схемы совместного владения секретом, которые подчиняются более сложным правилам доступа, например: "пользователь А и пользователь Б или пользователь А, и пользователь В, и пользователь Г".

В реальной жизни схемы совместного владения секретом применяются крайне редко, потому что они слишком сложны. Их трудно реализовать, но, что гораздо важнее, ими сложно руководить. В большинстве компаний нет группы высокопоставленных сотрудников, которые бы не доверяли друг другу. Попытайтесь сказать членам правления, что каждому из них будет выдан идентификатор безопасности с частью секретного ключа и что в случае необходимости они должны немедленно явиться в офис, даже если им позвонят в три часа утра в воскресенье. Не забудьте подчеркнуть, что они должны не только перестать доверять друг другу, но и сохранять собственные части ключа в секрете даже от остальных членов правления. Им также придется спускаться в безопасную комнату управления ключами, чтобы получить новую часть ключа, каждый раз, когда кто-нибудь уйдет из правления или появится в нем. На практике это означает, что использование членов правления обречено на провал.

Директор компании также не слишком подходит для хранения части ключа, поскольку, как правило, постоянно находится в отъезде. Когда вы это, наконец, осознаете, круг подходящих лиц сузится до двух-трех старших IT-менеджеров. Они могли бы использовать схему совместного владения секретом, но высокая стоимость и сложность этого решения делает его малопривлекательным.

Почему бы не воспользоваться чем-то попроще, например сейфом? Использование сейфов или банковских хранилищ имеет ряд преимуществ. Все знают, как они функционируют, поэтому вам не придется слишком долго обучать своих сотрудников. Кроме того, сейфы и банковские хранилища уже были тщательно протестированы, в то время как процесс воссоздания секрета крайне сложно тестировать из-за огромного количества взаимодействий с пользователями. Думаем, никому не хочется, чтобы какая-нибудь незначительная ошибка в процессе воссоздания секрета привела к потере ключа корневого центра сертификации.

Мы не будем рассматривать функционирование схем совместного владения секретом в деталях, поскольку, во-первых, для этого нужна довольно серьезная математическая основа, а во-вторых, схемы совместного владения секретом применяются крайне редко.

22.10 Уничтожение секретов

Любой, даже самый важный долгосрочный секрет в конце концов приходится уничтожать. Как только секрет перестает быть нужным, мы должны очистить место его хранения, чтобы избежать последующего взлома информации. Проблемы очистки памяти уже обсуждались в разделе 9.3. Удалить же долгосрочные секреты из постоянного хранилища еще сложнее.

Схемы хранения долгосрочных секретов, рассмотренные в этой главе, используют целый ряд технологий хранения данных: жесткие диски, бумагу, дискеты, пластиковые карты с магнитной полосой, память EPROM, EEPROM, флэш-память или память RAM, работающую от батарейки. Ни одна из этих технологий не обладает задокументированной функциональностью удаления данных, которая бы гарантировала, что их восстановление невозможно.

22.10.1 Бумага

Уничтожение пароля, записанного на клочке бумаги, обычно подразумевает уничтожение самого клочка бумаги. Один из возможных методов уничтожения — сжечь бумажку с паролем, после чего растереть оставшийся пепел в мелкий порошок или перемешать его с небольшим количеством воды.

Вместо этого листок с паролем можно пропустить через машинку для уничтожения бумаг, хотя большинство подобных устройств измельчают бумагу на куски довольно большого размера, что позволяет сравнительно легко восстановить содержавшийся на ней текст.

22.10.2 Магнитное хранилище

Магнитные носители с большим трудом поддаются очистке. Между тем посвященная этому вопросу литература удивительно малочисленна. Лучшая работа, которая нам известна, — это статья Питера Гутманна [38], хотя ее технические аспекты, вероятно, уже устарели.

Магнитные носители хранят свои данные в маленьких магнитных доменах. Направление намагниченности домена указывает на то, какие данные в нем закодированы. Когда данные подвергаются перезаписи, направление намагниченности домена изменяется в соответствии с новыми данными. Существует, однако, несколько механизмов, которые препятствуют полной потере старых данных. Головка чтения/записи, которая пытается перезаписать данные, никогда не выравнивается в точности по старым данным, поэтому некоторые части старых данных остаются нетронутыми. Таким образом, перезапись носителя не приводит к полному уничтожению старых данных. Для большей наглядности это можно сравнить с перекрашиванием стены. Если мы покроем ее одним слоем краски, через него кое-где будет просвечивать старая краска. Вдобавок ко всему магнитные домены способны “убегать” от головки чтения/записи к краю дорожки или, наоборот, вглубь магнитного вещества, где могут задержаться на длительное время. В большинстве случаев перезаписанные данные нельзя восстановить с помощью обычной головки чтения/записи, но злоумышленник может украсть диск и воспользоваться специальным оборудованием, которое позволит ему частично или даже полностью восстановить старую информацию.

На практике лучшим способом уничтожения секретной информации, пожалуй, является многократное перезаписывание последней с помощью случайных данных. При этом следует учесть ряд моментов.

- В каждой процедуре перезаписи необходимо применять новые случайные данные. Некоторые исследователи разработали конкретные наборы данных, которые, как предполагается, позволяют лучше уничтожить старые данные, однако выбор того или иного набора зависит от специфики самого носителя. Применение случайных данных для достижения того же эффекта может потребовать большего количества операций перезаписи, однако оно срабатывает во всех ситуациях и потому безопаснее.

- Убедитесь, что вы перезаписали именно ту область носителя, в которой хранился секрет. Если вы просто измените файл, поместив в него новые данные, файловая система может сохранить его в другом месте носителя, вследствие чего исходные данные останутся нетронутыми.
- Убедитесь, что в ходе каждой операции перезаписи новые данные действительно записываются на диск, а не в один из его кэшей. Особую опасность в этом отношении представляют жесткие диски, имеющие собственные кэши записи, так как они могут кэшировать новые данные, чтобы объединить многочисленные операции перезаписи в одну.
- Очистке должны подвергнуться и те области носителя, которые находятся непосредственно перед секретными данными, а также непосредственно после них. Поскольку скорость вращения диска никогда не бывает абсолютно постоянной, место расположения новых данных не будет в точности совпадать с местом расположения старых.

Нам не известны надежные оценки того, сколько операций перезаписи требуется для полного уничтожения старых данных, однако мы не видим причин ограничиваться небольшим количеством подобных операций. Все, что от нас требуется, — это уничтожить один-единственный ключ. (Если у вас есть большое количество секретных данных, храните их зашифрованными с помощью некоторого ключа и уничтожайте только этот ключ.) На наш взгляд, вполне разумно выполнить 50 или 100 операций перезаписи с использованием случайных данных.

Теоретически магнитную ленту или диск можно очистить с помощью размагничивающего устройства. К сожалению, современные магнитные носители с высокой плотностью записи не поддаются размагничиванию в той степени, которая обеспечивает гарантированное уничтожение данных. Впрочем, на практике у пользователей нет доступа к размагничивающим устройствам, поэтому такого вопроса не возникает.

Даже при многократном перезаписывании данных следует ожидать, что высококвалифицированный и хорошо оснащенный злоумышленник сможет восстановить секретную информацию, которая когда-то хранилась на магнитном носителе. Чтобы полностью уничтожить данные, вам, вероятно, придется уничтожить и сам носитель. Если магнитный слой носителя заключен в пластиковый корпус (как в дискете или кассете), вы можете измельчить и затем сжечь магнитный носитель. Если же речь идет о жестком диске, попробуйте воспользоваться шлифовальным станком, чтобы удалить с пластин магнитный слой, или же с помощью паяльника переплавить пластины на жидкий металл. На практике вы вряд ли убедите пользователей прибегнуть к таким радикальным мерам, поэтому лучшим практическим решением было и остается многократное перезаписывание.

22.10.3 Полупроводниковые записывающие устройства

Аналогичные проблемы возникают и при очистке энергонезависимой памяти, такой, как EPROM, EEPROM и флэш-память. Перезаписывание старых данных не удаляет все их следы. Определенную роль в этом играют и механизмы удерживания данных (см. раздел 9.3.4). Единственным практическим решением данной проблемы является многократное перезаписывание секретной информации с помощью случайных данных, но оно ни в коей мере не обеспечивает идеальной защиты. Когда полупроводниковое записывающее устройство становится ненужным, оно должно быть уничтожено.

Часть IV

Разное

Глава 23

Стандарты

В данной книге мы всячески старались избегать упоминания стандартов. На это есть серьезные причины. Стандарты безопасности редко срабатывают на практике. К сожалению, если вы занимаетесь инженерией криптографических систем, вам так или иначе придется иметь дело со стандартами, поэтому небольшое знакомство с ними отнюдь не помешает.

23.1 Процесс стандартизации

Для тех, кто еще не принимал участия в процессе стандартизации, опишем, как создаются стандарты. Все начинается с какой-нибудь организации, занимающейся стандартизацией, например Проблемной группы проектирования Internet (Internet Engineering Task Force — IETF), Института инженеров по электротехнике и электронике (Institute of Electrical and Electronics Engineers — IEEE), Международной организации по стандартизации (International Organization for Standardization — ISO) или Европейского комитета по стандартизации (European Committee for Standardization — CEN). Осознав необходимость принятия нового или улучшенного стандарта, данная организация назначает комитет. Этот комитет может называться по-разному: рабочая группа, проблемная группа или как-нибудь иначе. Иногда для принятия стандартов создаются иерархические структуры комитетов, но основная идея остается той же. Членство в комитете, как правило, основано на добровольных началах. Люди обращаются с просьбой присоединиться к комитету, и принимают практически каждого. Иногда, чтобы попасть в комитет, желающему приходится проходить несколько обязательных процедур, но какого-либо видимого критерия отбора членов комитета не существует. Размер комитета может доходить до нескольких сотен человек, однако большие комитеты разбиваются на несколько подкомитетов меньшего размера (они называются

проблемными группами, исследовательскими группами и т.п.). Как правило, бóльшая часть работы выполняется группой из нескольких десятков человек.

Раз в несколько месяцев комитет по стандартизации устраивает собрание. Все члены комитета приезжают в назначенный город и на протяжении нескольких дней заседают в гостинице. В промежутках между собраниями члены комитета выполняют свою работу, создают предложения и презентации и т.п. На собраниях комитет решает, в каком направлении двигаться дальше. Обычно среди членов комитета выбирают редактора, которому поручается собрать все предложения и организовать их в единый документ. Создание стандарта — весьма медленный процесс, который зачастую растягивается на долгие годы.

Так кто же входит в состав комитета? Вообще говоря, быть членом комитета по стандартизации — дело весьма дорогостоящее. Помимо неизбежной траты времени, члену комитета приходится оплачивать поездки и проживание в гостиницах. Поэтому каждый член комитета выдвигается туда своей компанией. У компаний есть несколько мотивов быть представленными в комитете. Иногда компания хочет продавать продукт, который должен обладать совместимостью с продуктами других компаний. Для этого требуются стандарты, а лучший способ следить за процессом стандартизации — это самому принимать в нем участие. Кроме того, компании хотят контролировать действия своих конкурентов. Если позволить конкуренту самому написать стандарт, он обязательно поставит вас в невыгодное положение, например приспособит стандарт к собственным требованиям или включит в него собственноручно запатентованные технологии. Иногда компаниям, напротив, невыгодна разработка стандарта. Представители подобных компаний появляются на собраниях комитета только затем, чтобы как можно более затянуть процесс стандартизации и дать время захватить рынок своему собственному решению. В реальной жизни все эти и еще некоторые мотивы сочетаются в разнообразных пропорциях, образуя весьма сложное политическое окружение.

Неудивительно, что работа множества комитетов по стандартизации заканчивается провалом. Подобные комитеты либо не создают никакого стандарта, либо создают что-нибудь из рук вон плохое, либо заходят в тупик и настолько подпадают под влияние индустрии, что стандартизируют первую попавшуюся систему, которой удалось завоевать рынок. Впрочем, работа некоторых комитетов все же завершается успехом и через несколько лет у нас появляется документ с новым стандартом.

После принятия стандарта начинается его реализация разработчиками. Это приводит к появлению массы несовместимых систем, а значит, к необходимости запуска вторичного процесса стандартизации, в ходе которого проводится тестирование на совместимость. Затем разные разработчики начина-

ют адаптировать свои реализации, чтобы они могли взаимодействовать друг с другом.

Описанный процесс обычно изобилует проблемами. Фракции комитета практически не уделяют внимания разработке хорошего технического стандарта. Самое главное для комитета — достичь согласия. Окончательный вариант стандарта принимается тогда, когда каждый член комитета *не*доволен результатом. Чтобы удовлетворить интересы различных фракций, стандарты наделяются массой возможностей, расширенной функциональностью, бесполезными вариантами и т.п. А поскольку каждая фракция имеет собственные идеи, собственное мнение и собственный взгляд на проблему, наилучший компромисс зачастую оказывается противоречивым. Многие стандарты содержат внутренние нестыковки или даже противоречат сами себе.

Ситуацию еще более усложняет то, что компании начинают реализацию своих продуктов прямо в процессе принятия стандарта, основываясь на его черновиках. Как следствие этого, в стандарт очень трудно внести изменения, поскольку кто-то уже реализовал свой проект и не желает начинать все сначала. Разумеется, различные компании реализуют проекты разными способами, поэтому на собрании комитета они будут бороться за то, чтобы максимально подогнать стандарт под свою реализацию. Иногда единственным возможным решением является выбор варианта, который не реализовала ни одна компания, — просто для того, чтобы они оказались одинаково не удовлетворены результатом. Технические же качества стандарта никого не волнуют.

23.1.1 Стандарт

Один из результатов описанного процесса состоит в том, что стандарты невероятно сложно читать. Документ, описывающий стандарт, разрабатывается договорным путем, поэтому ни у кого нет стимула сделать стандарт понятным, четким, точным или читабельным. В действительности, чем ниже читабельность документа, тем легче с ним работать. Такой документ поймут лишь немногие члены комитета, а потому они смогут спокойно проработать его, не отвлекаясь на препирательства с остальными членами. Вникать в сотни страниц плохо написанной документации — занятие не из приятных, поэтому большинство членов комитета не станут читать весь текст черновика и ограничатся просмотром лишь тех небольших фрагментов стандарта, которые их интересуют.

23.1.2 Функциональность

Как уже отмечалось, в процессе принятия стандарта всегда требуется проводить тестирование продуктов разных поставщиков на совместимость.

Разумеется, каждая компания реализует один и тот же стандарт по-своему. Зачастую реализация немного отклоняется от того, что определено в стандарте, а поскольку компания уже продвигает свои продукты на рынке, вносить изменения, как правило, слишком поздно. Мы не раз наблюдали, как продукты двух различных компаний настолько отклоняются от стандарта, что становятся несовместимыми, в результате чего разработчики вынуждены подгонять один продукт под другой.

Довольно часто стандарты включают в себя огромное количество возможностей. В реализациях, однако, применяется лишь ограниченный набор этих возможностей, конечно же, с некоторыми ограничениями и расширениями, так как сам по себе стандарт описывает нечто нежизнеспособное. И разумеется, отличие реализации от стандарта нигде не документируется.

В большинстве случаев продуктам различных поставщиков удается кое-как взаимодействовать друг с другом, но только в рамках базовой функциональности. Точка доступа беспроводной сети позволит клиенту подключиться к последней, однако управлять этим клиентом в окружении с оборудованием другого поставщика она, скорее всего, не сможет. Простые HTML-страницы будут корректно отображаться во всех обозревателях, однако отображение страниц, содержащих более сложные элементы, в разных обозревателях даст различные результаты. Мы все настолько привыкли к подобным вещам, что практически не обращаем на них внимания.

В действительности же такое положение дел можно назвать более чем печальным. Наша индустрия, и мы вместе с ней, не может создать стандарт, который был бы по крайней мере корректным или читабельным, не говоря уже об обеспечении совместимости различных продуктов сверх базовой функциональности.

23.1.3 Безопасность

Таким образом, обычный процесс разработки стандартов никак не подходит для обеспечения безопасности. Здесь мы имеем дело с активным злоумышленником, который проберется в самые удаленные уголки стандарта. Кроме того, общая безопасность системы определяется ее слабым звеном, а значит, любая отдельно взятая ошибка может оказаться роковой.

Мы уже неоднократно подчеркивали, насколько важной является простота. Стандарты же обладают чем угодно, только не простотой. Процесс разработки стандарта договорным путем исключает простоту и неизбежно приводит к появлению чего-то настолько сложного, что будут не в состоянии понять сами члены комитета. Уже одного этого вполне достаточно, чтобы стандарт нельзя было считать безопасным.

Когда мы пытались донести эту проблему до тех, кто занимается стандартизацией, то часто получали в ответ высказывания примерно такого рода: “Этим технарям всегда подавай идеальный стандарт...”, “Политические реалии таковы, что мы вынуждены идти на компромисс...”, “Именно так и работает система...”, “Да посмотрите же, чего мы достигли...”, “Все и так хорошо...” и т.п. Простите за резкость, но в сфере безопасности это не пройдет. Даже того, что после принятия стандарта необходимо проводить тестирование на совместимость, вполне достаточно, чтобы четко уяснить: стандарты, разработанные комитетом, не подходят для обеспечения безопасности. Если функциональная часть стандарта (т.е. самая легкая его часть) недостаточно хороша, чтобы гарантировать совместимость систем различных разработчиков без предварительного тестирования, тогда часть стандарта, отвечающая за безопасность, скорее всего, тоже не сможет обеспечить безопасность систем без тестирования. В свою очередь, мы знаем, что протестировать систему на безопасность невозможно. Вероятно, мы могли бы создать реализацию, использующую подмножество функциональности стандарта, которое также будет безопасным, но для стандарта безопасности этого явно недостаточно. Стандарт безопасности провозглашает, что, придерживаясь его, мы непременно достигнем определенного уровня безопасности. Нет, безопасность слишком сложна, чтобы отдавать ее в руки комитета.

Мы не знаем ни одного стандарта, разработанного комитетом, который бы стоил внимания. Поэтому, когда кто-нибудь предлагает использовать криптографический стандарт, мы относимся к этому крайне неприязненно, а порой и враждебно.

Существует лишь несколько действительно стоящих криптографических стандартов, ни один из которых не был разработан комитетом. Иногда стандарт разрабатывается небольшой группой единомышленников, которые вместе создают один последовательный стандарт. А иногда проект принимают в качестве стандарта без каких-либо политических компромиссов. Такие стандарты могут оказаться довольно неплохими. Рассмотрим два наиболее важных из них.

23.2 SSL

Это протокол безопасности, используемый Web-обозревателями для безопасного подключения к Web-серверам. Первой версией данного протокола, вошедшей в широкое применение, была SSL 2, которая содержала несколько недочетов в системе безопасности. Улучшенная версия SSL получила название SSL 3 [36]. Она была разработана группой из трех человек без организа-

ции каких-либо комитетов. Протокол SSL 3 получил широкое распространение и в целом был признан хорошим.

Небольшое предупреждение: SSL — хороший протокол, но это еще не означает, что любая система, использующая его, будет безопасной. Для аутентификации Web-сервера SSL использует функциональность инфраструктуры открытого ключа (PKI), а клиент PKI, встроенный в большинство обозревателей, обладает такой толерантностью к чужим сертификатам, что общий уровень безопасности оказывается довольно низким. У одного из наших обозревателей имеется 108 различных корневых сертификатов от 35 центров сертификации. Таким образом, даже не начав рассмотрение активных атак, мы уже сталкиваемся с фактом существования 35 различных организаций, разбросанных по всему миру, которым мы должны доверять всю нашу Web-информацию.

Протокол SSL никогда не был стандартизирован по-настоящему. Он был просто разработан компанией Netscape и стал стандартом де-факто. Стандартизацией и дальнейшей разработкой протокола SSL (под именем TLS) занимается IETF. Мы не занимались подробным изучением TLS. Пока что отличия TLS от SSL выглядят довольно незначительными, поэтому у нас нет оснований думать, что TLS в чем-то уступает SSL 3. Тем не менее, учитывая последние "достижения" IETF в области разработки протоколов безопасности, таких, как IPSec [32], мы опасаемся, что эффект комитета опять проявит себя и испортит хороший стандарт.

23.3 AES: стандартизация на конкурсной основе

Для нас AES является ярким примером того, как следует стандартизировать системы безопасности. Стандарт AES не был разработан на собраниях комитета — его отобрали на конкурсной основе. Схема проведения подобного конкурса довольно проста. Вначале вы задаете, каким требованиям должна соответствовать система и какова ее цель. Разработка спецификаций системы может выполняться сравнительно небольшим коллективом людей с использованием многих внешних источников.

Следующий шаг — объявить конкурс. Вы обращаетесь к экспертам с просьбой разработать законченное решение, которое будет соответствовать заданным требованиям. Как только эксперты представят свои предложения, вам останется лишь выбрать одно из них. Это прямое соревнование, в котором проекты-конкурсанты оцениваются по ряду критериев. Если главным критерием отбора выступает безопасность, участники будут крайне заинтересованы в том, чтобы найти как можно больше слабых мест в безопасности систем своих конкурентов. В случае удачи это обеспечит неоценимую обрат-

ную связь. В других ситуациях для проведения анализа безопасности вам может понадобиться помощь внешних экспертов.

Если вам повезет, в ходе конкурса вы сможете отобрать одно предложение, которое и будет принято в качестве стандарта либо без изменений, либо с небольшими изменениями. Не следует объединять в общий стандарт черты различных предложений; подобные действия приведут лишь к появлению еще одного стандарта, разработанного комитетом. Если ни одно из предложений не удовлетворит заданным требованиям и вам покажется, что можно создать что-нибудь получше, подумайте о проведении нового конкурса.

Именно так Национальный институт стандартов и технологий США (NIST) проводил конкурс на лучший шифр при выборе стандарта шифрования AES, и данная схема сработала просто замечательно. К участию в конкурсе были допущены 15 предложений, из которых в ходе первого раунда отобрали пять финалистов. Второй раунд оценки проектов-финалистов привел к выявлению победителя. Мы были приятно удивлены тем, что каждый из пяти финалистов был вполне достоин называться новым стандартом; безусловно, этот стандарт был бы намного лучше, чем любой из стандартов, разработанных комитетом.

Принцип стандартизации на конкурсной основе не сработает, если у вас нет достаточного числа экспертов, способных создать, как минимум, несколько конкурентоспособных проектов. Впрочем, на наш взгляд, если у вас недостаточно экспертов, чтобы разработать несколько предложений, вам вообще не стоит заниматься стандартизацией систем безопасности. Чтобы достичь простоты и согласованности, которые являются критическими для безопасности системы в целом, система безопасности должна разрабатываться небольшой группой экспертов. Затем вам понадобится еще несколько экспертов, которые будут анализировать систему и пытаться напасть на нее, выискивая слабые места в ее безопасности. Чтобы иметь хоть какую-то надежду на получение хорошего результата (какую бы схему стандартизации вы не использовали), вам понадобится достаточное количество экспертов для формирования, как минимум, трех групп по разработке предложений. Если у вас есть такое количество экспертов, воспользуйтесь схемой стандартизации на конкурсной основе, поскольку сегодня это единственная схема, которая подтвердила свою способность создавать хорошие стандарты безопасности.

Глава 24

Патенты

Патенты имеют намного большее влияние на криптографию, чем хотелось бы. Мы понимаем потребность бизнеса в существовании патентов; более того, целый ряд патентов уже подписаны нашими именами. (В большинстве случаев эти патенты являются результатом консультативной работы, а потому принадлежат компаниям, которые нас наняли.) К сожалению, наличие патентов часто означает, что мы не можем использовать необходимые математические инструменты, и вынуждает идти на компромисс, который нам не по душе. Что еще более важно, мы не верим, что патенты приносят пользу сообществу безопасности. Существует несколько аспектов текущей системы патентования, которые делают ее нежизнеспособной на практике.

24.1 Прототип

Предполагается, что мы не можем получить патент на то, что уже всем известно. Изобретение, которое публично известно еще до того, как его попытаются запатентовать, называется *прототипом (prior art)*. Между тем все, кто работает с текущей системой патентования, знают, что она не мешает заявлять свои права на давно существующие вещи. В 2001 году некий житель Австралии запатентовал колесо, а точнее, "кругообразное устройство для облегчения транспортировки" [51]. Нечего и объяснять, что у данного устройства был прототип. Это была попытка продемонстрировать, насколько несовершенна система патентования. Большинство патентов не содержат столь вопиющего плагиата, но мы все чаще и чаще сталкиваемся с патентами, распространяющимися на уже известные изобретения. Однажды нам даже попалась технология, запатентованная неким человеком через шесть месяцев после посещения конференции, на которой эта же технология была представлена другим человеком.

Причина незаконного присвоения чужих изобретений состоит в том, что патентное бюро просто не может проверить, есть ли у патентуемой технологии прототип. В окружающем мире слишком много прототипов; мы просто не в состоянии обнаружить их все. Проблема состоит в том, что после выдачи патента его истинность *не подвергается сомнению*. Другими словами, если на протяжении многих лет вы использовали в своих продуктах некоторый шифр и кто-то вдруг получает патент на подобное изобретение, вам придется взять на себя тяжелое бремя доказательства того, что ваше изобретение является прототипом. Владелец патента может спокойно устраниться от этого и переложить всю работу на вас. Это, конечно, обойдется вам в немалую сумму. Оспаривание патента требует привлечения юристов, занимающихся патентными делами, поэтому такие патенты могут служить замечательным средством законного вымогательства: купите дешевую лицензию или же потратьте в 10 раз больше на своих адвокатов.

24.2 Расширения

Система патентования имеет еще одну странную особенность. Мы не совсем понимаем, как она работает, но это одна из уловок юристов, занимающихся патентными делами. В чем же состоит эта особенность? Для каждой запатентованной технологии один или два патента всегда оставляют "открытыми"; другими словами, процесс патентования технологии искусственно продлевается. Когда вы регистрируете патент, вы обязаны зарегистрировать и так называемый *документ о раскрытии (disclosure document)*, который описывает соответствующую технологию. Зарегистрированный документ о раскрытии не может впоследствии подвергаться изменению, а вот пункты самой формулы изобретения могут. А задержать процесс патентования с помощью процедурной волокиты не составляет никакого труда. Итак, вы затягиваете регистрацию нескольких патентов, являющихся расширениями основного патента.

Теперь предположим, что ваш конкурент начинает использовать технологию, которая значительно отличается от запатентованной вами, однако все же имеет с ней кое-что общее. Тогда вы пытаетесь переписать пункты "открытых" патентов таким образом, чтобы они начали распространяться и на технологию конкурента. Утверждение пунктов патента проводится в ходе переговоров между вами и патентным бюро, а поскольку в патентном бюро не знают, что именно использует ваш конкурент, вероятность утверждения новых пунктов патента весьма высока. Затем вы прекращаете затягивать процесс патентования и получаете патент, пункты которого непосредственно распространяются на технологию конкурента. Осталось только подать на

конкурента в суд за нарушение патента. На наш взгляд, все это серьезно отдает мошенничеством; между тем подобная практика абсолютно законна и применяется на самой широкой основе.

24.3 Расплывчатость описаний

Патент должен содержать четкое и ясное описание технологии, чтобы ее мог реализовать не только сам изобретатель, но и все, кому это потребуется. Это одна из составляющих сделки по выдаче патента. Взамен за разглашение технологии владельцу патента предоставляется 20-летняя монополия на право использования этой технологии. На практике же патенты сознательно формулируют так, чтобы их содержимое было расплывчатым, неопределенным и трудным для чтения. Очень часто владелец патента *не* описывает лучший способ использования его технологий даже несмотря на то, что это обязательное требование к процессу патентования.

Предположим, вы работаете над системой и думаете, что определенный патент может представлять для нее потенциальную угрозу. Ну что ж, вы можете достать патент, прочитать его и понять, распространяется ли его действие на вашу разработку, правильно? Нет, неправильно! Понять, распространяется ли действие патента на конкретную систему, чрезвычайно сложно. В конце концов это решает судья, у которого нет технического образования. С таким же успехом принимать решение о нарушении патента можно было бы, просто подбросив монетку.

24.4 Чтение патентов

Наш совет: никогда не читайте патенты! Возможно, вы думаете, что чтение патента позволит понять, на что он распространяется. К сожалению, все не так просто. Если вы нарушите патент, не зная об этом, вам, скорее всего, придется выплатить компенсацию его владельцу. Но если владелец патента докажет, что нарушение было сознательным (потому что вы читали текст патента), размер компенсации может возрасти втрое. Поэтому если вы читаете патент, то автоматически повышаете свою ответственность за нарушение патента в три раза.

А теперь самый больной удар: даже если вы прочитаете патент и, как эксперт в своей области, решите, что ваша работа не подпадает под его действие, судья все равно может решить, что патент был нарушен сознательно. Видите ли, эксперт не имеет полномочий решать, на что распространяется патент. Это может сделать только юрист, знающий патентное право. Поэтому, чтобы избежать опасности в виде выплаты тройной компенсации, вам придется

нанять юриста, который определит, будет ли иметь место нарушение патента. Существуют миллионы патентов, и вряд ли вы можете позволить себе заплатить юристу, чтобы он прочитал каждый из них.

Таким образом, самое безопасное решение — вообще не читать патент. По крайней мере в случае неприятности вы сможете с чистой совестью заявить, что нарушили его несознательно.

24.5 Лицензирование

Если у пользователя А есть патент, а пользователь Б хочет применять запатентованную технологию, то пользователь А может продать ему лицензию. Это весьма распространенное поведение. Многие большие компании активно практикуют взаимное лицензирование. Обычно они просто предоставляют друг другу лицензии на свои патенты без взимания какой-либо денежной оплаты. Компаниям меньшего размера приходится покупать лицензии. Во многих отраслях программной инженерии это не составляет проблемы. Если вам нужен патент, вы можете получить лицензию на вполне приемлемых условиях.

В криптографии, однако, все обстоит по-другому. В 80-х годах прошлого века обстановка в криптографическом сообществе была испорчена появлением нескольких базовых, начальных патентов на RSA и DH — единственные хорошие криптосистемы с открытым ключом, которые существовали на то время. В течение многих лет получить лицензию на эти фундаментальные патенты было невероятно сложно и дорого. Мы знаем массу трагических историй о компаниях, которые пытались приобрести лицензии и просто получили отказ.

Владельцы упомянутых патентов решили использовать их, чтобы контролировать развитие технологий путем ограничения лицензирования. Все это быстро сделало соответствующие технологии недоступными оставшейся части рынка. Теоретически патенты были предназначены именно для этого, однако общий эффект может быть крайне неудачным, если необходимая технология безопасности не может быть использована из-за наличия патента, а его владелец отказывается продавать лицензию.

Недавно мы испытывали трудности с использованием новых режимов работы блочных шифров XCBC, IACBC и OCB. По меньшей мере два из трех изобретателей этих режимов использовали в своей работе результаты, полученные ранее их конкурентами. Все трое подали заявки на патентование, но из-за задержек в процессе выдачи патентов на момент написания этой книги ни один патент так и не был выдан. Так как патенты еще не выданы, их точные формулировки неизвестны, поэтому мы не знаем, на что конкретно

будет распространяться тот или иной патент. А поскольку каждый претендент еще может изменить пункты своего патента с учетом наработок других претендентов (пункты патентов еще не были зафиксированы), вполне вероятно, что человеку, использующему одно из этих изобретений, придется получать лицензии на все три патента. Возможно, для использования XCBC будет достаточно и одной лицензии, однако для этого вы должны полностью поверить в честность системы патентования и безоговорочно положиться на нее, но делать это мы настоятельно не рекомендуем.

Таким образом, если вы хотите использовать, скажем, режим OCB, вам могут понадобиться три патентные лицензии на патенты, которые еще не выданы. Это означает необходимость составления трех контрактов с участием трех адвокатов (разумеется, при условии, что разработчики захотят продавать лицензии на свои патенты). Не удивляйтесь, если согласование этих вопросов затянется на полгода или даже больше. Мы сталкивались с ситуациями, когда после изнурительных переговоров владелец патента в последний момент вдруг решал не лицензировать свой патент. Для компании, разрабатывающей продукт, это равноценно катастрофе. Процесс получения лицензии затягивает разработку продукта, влечет за собой огромные авансовые расходы (и хорошо, если только на услуги адвокатов) и ставит существование всей линии продуктов в зависимость от непредсказуемых действий владельца патента и его адвокатов. Кроме того, чем дольше вы ведете переговоры, тем менее выгодным становится ваше положение в этих переговорах, поскольку вы уже вложили слишком много денег в процесс разработки продукта и должны вывести его на рынок чем скорее, тем лучше.

Казалось бы логичным, чтобы все три изобретателя собрались вместе и объединили свои патенты. Это бы позволило всем желающим получать одну лицензию на все патенты сразу. Попытки подобного объединения действительно имели место, но, согласно последним сведениям, изобретатели никак не могут договориться.

Именно проблемы лицензирования, касающиеся режимов XCBC, IACBC и OCB, стимулировали разработку режима CCM (см. раздел 8.5). Режим CCM работает в два раза медленнее, однако его разработчики решили обойтись без патентования. Это преимущество намного перевешивает все дополнительные вычисления, которые включает в себя данный режим.

Даже если патент уже выдан и его формулировка является довольно четкой, стоимость получения патентной лицензии может быть очень большой. Многие владельцы патентов невероятно усложняют получение лицензий. Стандартные условия лицензирования зачастую нереальны или же слишком обременительны. Довольно часто владельцы патентов просто не понимают, что у запатентованной технологии есть альтернативы и что процесс получения лицензии должен быть простым и относительно недорогим.

24.6 Защищающие патенты

Многие компании подают заявки на получение патентов не столько для ограничения использования соответствующих технологий, сколько для собственной защиты. Предположим, у вас есть целая груда патентов. Если ваш конкурент подает в суд за нарушение одного из его патентов, вы всегда сможете найти у себя патент, который либо был нарушен этим конкурентом, либо нарушение которого можно ему приписать. Это напоминает уже знакомую ситуацию взаимно-гарантированного уничтожения. Если вы подадите в суд на меня, я подам в суд на вас, и мы оба потеряем огромное количество денег (на судебных издержках и утраченных возможностях). Время от времени нам попадаются компании, занимающиеся этой игрой в кошки-мышки. Но как только более трезвые головы поймут, что соглашение о взаимном самоубийстве не принесет большого дохода, подобные игры прекращаются.

24.7 Как исправить систему патентования

Поведение текущей системы патентования полностью выходит за рамки приличий. В лучшем случае патенты — это неизбежное зло, в худшем — узаконенная форма мошенничества и шантажа. Предполагалось, что патенты будут помогать изобретателям, вознаграждая их за результаты проделанной работы. На наш взгляд, стоимость текущей системы патентования для компьютерной индустрии намного перевешивает преимущества патентов. Патенты остаются действительными на протяжении 20 лет, что для IT-мира равносильно вечности. Одно лишь получение патента затягивается на годы, что эквивалентно нескольким жизненным циклам программного или аппаратного обеспечения. Время в IT-индустрии течет настолько быстро, что система патентования за ним попросту не поспевает.

Существует, разумеется, группа людей, которые постоянно выигрывают от текущего положения дел в системе патентования, — это юристы. Думаем, не стоит лишний раз объяснять, представители какой профессии утверждают, что система патентования вполне работоспособна или даже хороша.

Мы часто слышим, что патенты защищают маленького безобидного разработчика от больших грозных компаний. Какая нелепость! Вероятно, отдельные примеры подобных ситуаций все же существуют, однако в большинстве случаев все происходит наоборот. Разработчик-одиночка или небольшая компания создают действительно новый продукт. Они вкладывают свое время и деньги в разработку этого продукта и выставляют его на рынке. Но как только такие разработчики начинают угрожать рынку крупной компании, их тут же забрасывают исками о нарушении патентов. Небольшие компании не могут позволить себе оспаривать патент судебным путем, что делает их

прекрасным объектом шантажа. Мы лично знаем, как минимум, одну небольшую компанию с инновационным продуктом, которая очутилась в подобной ситуации. В конце концов им пришлось продать свой продукт крупной фирме за весьма скромную цену. Текущая система патентования наделяет огромной властью большие компании, которые имеют штатных адвокатов, занимающихся патентованием, и достаточное количество денег, чтобы инициировать серьезную угрозу в виде патентной тяжбы.

Мы думаем, что IT-индустрии было бы гораздо лучше без патентов, чем с ними. Некоторые из наших друзей придерживаются иного мнения, и данный вопрос, разумеется, открыт для обсуждения. Но текущая система патентования попросту не работает так, как положено.

Для начала мы бы хотели ускорить процесс патентования. Мы полагаем, что выдавать патент следует в течение трех месяцев с момента подачи заявки, а срок его действия необходимо ограничить до трех-пяти лет. Кроме того, нужно разрешить всем и каждому подвергать сомнению истинность патентов на основе существующих прототипов даже после того, как патент был выдан. И наконец, следует создать систему, временной масштаб функционирования которой будет исчисляться месяцами, а не годами. Компьютерная индустрия никак не выиграет от правила, позволяющего осудить разработчика, нарушившего патент пять лет назад. Нам нужно, чтобы ответ был дан в рамках временного масштаба компьютерной индустрии, который составляет не более нескольких месяцев.

Разумеется, все это — наше личное мнение. Система патентования не будет исправлена просто потому, что это не принесет никакой политической выгоды. Итак, мы вынуждены жить с существующей системой, а значит, придется патентовать все изобретения только для того, чтобы защитить себя от своих конкурентов. А разработчики, занимающиеся криптографическими системами, будут по-прежнему тратить время и усилия на то, чтобы старательно избегать запатентованных областей.

24.8 Отречение

Мы не юристы. Мы даже не похожи на юристов. Не принимайте наши советы в правовых вопросах. Другими словами, вы можете полагаться только на самого себя, если не наймете адвоката, занимающегося патентными делами за несколько сотен долларов в час. Добро пожаловать в увлекательный мир патентов!

Глава 25

Привлечение экспертов

Криптографии присуще одно странное свойство: каждый думает, что знает о ней достаточно для того, чтобы разрабатывать собственные системы. Мы никогда не попросим второкурсника физического факультета спроектировать атомную электростанцию. Мы бы никогда не легли под нож медсестры-стажерки, объявившей, что она придумала революционный метод операций на сердце. Тем не менее, прочитав одну-две книги по криптографии, многие почему-то начинают думать, что могут разработать собственную криптографическую систему. Что еще хуже, иногда им удается убедить руководство компании, предпринимателей, готовых идти на риск, и даже некоторых клиентов в том, что их система будет представлять собой самое удачное воплощение человеческой мысли со времен изобретения бутерброда.

Первая книга Брюса, *Applied Cryptography* [86, 87], является как самой популярной, так и самой печально известной среди криптографов. Она снискала популярность за привлечение к криптографии внимания десятков тысяч людей. Печальную же известность ей принесли системы, которые эти люди пытались разработать и реализовать, прочитав ее.

В качестве одного из недавних примеров можно привести стандарт беспроводных сетей 802.11. Первоначальный стандарт включал в себя безопасный канал общения, который называется *WEP* (*wired equivalent privacy — эквивалент конфиденциальности проводных сетей)* и применяется для шифрования и аутентификации беспроводного обмена данными. Этот стандарт был разработан комитетом, в составе которого не было криптографов. Результат оказался ужасным. Решение использовать алгоритм шифрования RC4, возможно, было и не самым удачным, но не таким уж роковым. Тем не менее RC4 — это поточный шифр, для работы которого требуется уникальная оказия. Алгоритм WEP не выделял достаточного количества битов для оказии, в результате чего одно и то же значение оказии приходилось использовать несколько раз. Это, в свою очередь, привело к тому, что многие пакеты были

зашифрованы с помощью одного и того же ключевого потока. Безопасность поточного шифра RC4 была нарушена, и сообразительный злоумышленник мог без труда взломать шифр. Еще один, более тонкий момент состоял в следующем: система не проводила хэширования комбинации секретного ключа и оказии прежде, чем использовать ее в качестве ключа RC4, что в конце концов привело к осуществлению атак с восстановлением ключа [35]. Для аутентификации применялась контрольная сумма CRC (cyclic redundancy check — контроль с помощью циклического избыточного кода), но, поскольку при ее подсчете используются линейные вычисления, подделать пакет и контрольную сумму (с помощью аппарата линейной алгебры) не составляло труда, а обнаружить такую подделку было невозможно. Все пользователи сети применяли один и тот же общий ключ, что значительно усложняло его обновление. Сетевой пароль непосредственно применялся для шифрования всех сообщений без использования какого-либо протокола согласования ключей. И наконец, по умолчанию шифрование было отключено. Это означало, что в большинстве реализаций никто не утруждался его включить. Алгоритм WEP не просто был взломан; он был взломан окончательно и бесповоротно.

Разработать замену WEP было нелегко, поскольку ее требовалось адаптировать к существующему аппаратному обеспечению. Но выбора не было: безопасность исходного стандарта оказалась отвратительной. Замена получила название WPA (wireless protected access — защищенный беспроводной доступ) и на момент выхода этой книги уже должна быть доступной.

История WEP — не исключение. Она привлекла к себе более пристальное внимание прессы, чем остальные плохие криптографические алгоритмы, только из-за популярности стандарта 802.11, однако подобные ситуации можно повсеместно наблюдать и в других системах. Как сказал однажды коллега Брюса: “В мире полно плохих систем безопасности, разработанных людьми, которые прочитали *Applied Cryptography*”.

Появление книги *Практическая криптография*, скорее всего, приведет к аналогичным результатам.

Все это делает нашу книгу крайне опасной. Всегда найдутся люди, которые, прочитав ее, попытаются разработать криптографический алгоритм или протокол. Готовый результат может выглядеть довольно неплохим и даже работоспособным, однако о безопасности этой системы лучше не говорить. Возможно, полученное решение окажется правильным на 70%. Возможно, разработчикам повезет, и оно окажется правильным даже на 90%. Но криптография не терпит понятия “почти правильный”. Система безопасности надежна настолько, насколько надежно ее самое слабое звено. Чтобы система была безопасной, правильным должно быть *все*. А этому нельзя научиться, просто читая книги.

Так для чего же мы написали эту книгу, если она приводит к появлению плохих систем? Мы написали ее потому, что люди, которые хотят научиться разрабатывать криптографические системы, должны где-то этому учиться, а других подходящих книг мы не знаем. Рассматривайте данную книгу как введение в разработку криптографических систем, хотя она и не является руководством или учебником. Мы также написали ее для других специалистов, участвующих в разработке системы. Каждая часть системы безопасности играет критически важную роль, и все, кто работает над проектом, должны обладать базовым пониманием проблем безопасности и методов ее обеспечения. К числу этих людей относятся программисты, тестировщики, составители документации, руководство и даже персонал из отдела продаж. Каждый сотрудник должен иметь достаточное понимание проблем безопасности, чтобы правильно выполнять свою работу. Мы надеемся, что данная книга снабдит вас адекватной информацией о практической стороне криптографии.

Если вы не против получить хоть какой-нибудь дельный совет, вот он: по мере возможности всегда привлекайте к сотрудничеству экспертов в области криптографии. Если в проекте каким-либо образом задействован криптографический аппарат, вам не обойтись без опытного разработчика криптографических систем. Подключите его к работе в самом начале проекта. Чем раньше вы проконсультируетесь с экспертом в области криптографии, тем легче и дешевле обойдется разработка проекта. Довольно часто нас звали взглянуть на проект только затем, чтобы заткнуть “дыры” в частях системы, которые уже давно были спроектированы или реализованы. Конечный результат всегда обходился слишком дорого — либо в терминах затраченных усилий, срока сдачи проекта и денежных средств, либо в терминах безопасности пользователя конечного продукта.

Создать правильную криптографическую систему невероятно сложно. Даже системы, разработанные экспертами, время от времени дают сбой. Неважно, насколько вы умны или какой опыт работы имеете в других областях. Проектирование и реализация криптографических систем требуют наличия специальных знаний и опыта, а единственный способ набраться опыта — разрабатывать системы снова и снова. Однако этот процесс никогда не обходится без ошибок. Так для чего же нужен эксперт, если он тоже делает ошибки? По той же причине, по которой мы отдаем свою жизнь в руки квалифицированного хирурга. Нельзя сказать, что хирурги вообще не допускают ошибок; просто они их делают намного реже, да и сами ошибки гораздо менее серьезны. Кроме того, хирурги придерживаются консервативных правил, поэтому небольшие ошибки не приводят к катастрофическим последствиям; хирурги знают достаточно для того, чтобы ошибка оказалась безобидной.

Реализация криптографических систем — это практически такой же специализированный вид деятельности, как и их проектирование. Найти проектировщиков криптографических систем не так уж сложно. Найти программистов, занимающихся реализацией криптографических систем, намного труднее, отчасти потому, что их требуется гораздо больше. Один проектировщик может создать работу для 10–20 программистов. Большинство людей не рассматривают реализацию криптографических систем как специализированный вид деятельности. Один и тот же программист может перейти от написания баз данных к написанию графического интерфейса, а затем заняться реализацией криптографических систем. Разумеется, написание баз данных и графического интерфейса — тоже специализированные виды деятельности, однако опытный программист, немного подучившись, может вполне успешно заниматься и тем и другим. Это не касается реализации криптографических систем, где *все* должно быть сделано правильно.

Лучший из известных нам способов реализации криптографических систем — нанять компетентных программистов и натренировать их для выполнения данной задачи. Одной из составляющих тренинга может стать и наша книга, однако в большинстве своем написание криптографических систем требует опыта и правильного отношения к проблеме. Кроме того, на овладение написанием криптографических систем, как и любым другим специальным навыком, уходят годы. Учитывая, сколько времени требуется для приобретения данного опыта, вы должны уметь удержать подготовленных вами специалистов. Это еще одна проблема, решение которой мы с радостью перекладываем на других членов нашего общества.

Вероятно, даже более важную роль, чем эта или любая другая книга, играет культура ведения проекта. Фраза “безопасность в первую очередь” не должна оставаться просто девизом; она должна быть вплетена в саму материю проекта и его разработчиков. Каждый участник проекта должен постоянно жить с безопасностью, дышать безопасностью, говорить и думать о безопасности. Достичь этого невероятно трудно, но все-таки возможно. Подобная команда сотрудников была у компании DigiCash в 1990-х годах. Аналогичная культура всеобъемлющей безопасности характерна для авиационной промышленности. Разумеется, культуры безопасности нельзя достичь в короткий срок, однако к этому нужно стремиться. Наша книга — это всего лишь начальное пособие по наиболее важным проблемам безопасности, предназначенное для технических специалистов команды разработчиков.

В книге *Secrets and Lies* [88] Брюс написал следующее: “Безопасность — это процесс, а не продукт”. Помимо культуры безопасности, нам нужен еще и процесс безопасности. Авиационная промышленность обладает всесторонним, строго отлаженным процессом контроля безопасности. Большинство производителей программных продуктов не имеют даже устоявшегося про-

цесса выпуска программного обеспечения, не говоря уже о процессе выпуска высококачественного программного обеспечения и тем более систем безопасности. Написание хороших систем безопасности, вероятно, выходит за рамки текущих качеств нашей компьютерной индустрии. Это, однако, не означает, что мы должны сдаться. Поскольку информационные технологии играют все более и более критическую роль в нашей инфраструктуре, нашей свободе и нашей безопасности, мы *должны* улучшать безопасность своих систем. Мы просто обязаны делать все, что только в наших силах.

Надеемся, что эта книга внесет свою лепту в улучшение систем безопасности, обучая их разработчиков основам практической криптографии.

Благодарности

Данная книга основана на совместном опыте, приобретенном нами за долгие годы работы в области криптографии. Мы глубоко признательны всем, кто сотрудничал с нами. Именно они сделали нашу работу интересной и помогли нам достичь того понимания, без которого было бы невозможно написать эту книгу. Мы также хотим поблагодарить своих клиентов, которые предоставили нам не только средства, позволившие продолжить наши криптографические исследования, но и возможность приобрести практический опыт, необходимый для написания книги.

Особого упоминания заслуживают Бет Фридман (Beth Friedman), которая оказалась неоценимым выпускающим редактором, и Дениз Дик (Denise Dick), значительно улучшившая наше творение, тщательно вычитав его. Следует отметить Джона Келси, снабдившего нас ценными советами по поводу криптографического содержимого. А состоялось наше сотрудничество благодаря Internet. Особую признательность мы хотим выразить Кэрол Лонг (Carol Long) и другим сотрудникам издательства Wiley за претворение наших идей в реальность.

И наконец, мы благодарим всех программистов мира, которые самоотверженно продолжают писать криптографический код и предоставлять его широкой общественности на бесплатных началах.

Список основных источников информации

1. Anderson R.J. *Security Engineering: A Guide to Building Dependable Distributed Systems.* ? John Wiley & Sons, Inc., 2001 (ISBN 0-471-38922-6).
2. Anderson R., Biham E. and Knudsen L. *Serpent: A proposal for the Advanced Encryption Standard.* ? In: National Institute of Standards and Technology, August 1998 (см. `http://www.cl.cam.ac.uk/~rja14/serpent.html` или `http://www.nist.gov/aes`).
3. Bellare M., Canetti R. and Krawczyk H. *Keying Hash Functions for Message Authentication.* ? In: Koblitz N. (ed.). *Advances in Cryptology — CRYPTO '96. Lecture Notes in Computer Science.* Springer-Verlag, 1996, vol. 1109, p. 1–15.
4. Bellare M., Killian J. and Rogaway P. *The Security of Cipher Block Chaining.* ? In: Desmedt Y.G. (ed). *Advances in Cryptology — CRYPTO '94. Lecture Notes in Computer Science.* Springer-Verlag, 1994, vol. 839, p. 341–358.
5. Bennett C.H. and Brassard G. *An update on quantum cryptography.* ? In: Blackley G.R. and Chaum D. (ed). *Advances in Cryptology, Proceedings of CRYPTO 84. Lecture Notes in Computer Science.* Springer-Verlag, 1984, vol. 196, p. 475–480.
6. Biham E., Dunkelman O. and Keller N. *The Rectangle Attack — Rectangling the Serpent.* ? In: Pfitzmann B. (ed). *Advances in Cryptology — EUROCRYPT 2001. Lecture Notes in Computer Science.* Springer-Verlag, 2001, vol. 2045, p. 340–357.
7. Biham E. *New Types of Cryptoanalytic Attacks Using Related Keys.* ? In: Helleseth T. (ed). *Advances in Cryptology — EUROCRYPT '93. Lecture Notes in Computer Science.* Springer-Verlag, 1993, vol. 765, p. 398–409.
8. Black J., Halevi S., Krawczyk H., Krovetz T. and Rogaway P. *UMAC: Fast and Secure Message Authentication.* ? In: Michael Wiener (ed). *Advances in Cryptology — CRYPTO '99. Lecture Notes in Computer Science.* Springer-Verlag, 1999, vol. 1666, p. 216–233.

9. Bos J. *Booting problems with the JEC computer.* ? Personal Communications, 1983.
10. Bos J. *Practical Privacy.* ? PhD thesis : Eindhoven University of Technology, 1992 (см. `http://www.macfergus.com/niels/lib/bosphd.html`).
11. Brassard G. and Crepeau C. *Quantum Bit Commitment and Coin-Tossing Protocols.* ? In: Menezes A.J. and Vanstone S.A. (ed). *Advances in Cryptology — CRYPTO '90. Lecture Notes in Computer Science.* Springer-Verlag, 1990, vol. 537 p. 49–61.
12. Brincat K. and Mitchell C.J. *New CBC-MAC forgery attacks.* ? In: Varadharajan V. and Mu Y. (ed). *Information Security and Privacy, ACISP 2001. Lecture Notes in Computer Science.* Springer-Verlag, 2001, vol. 2119, p. 3–14.
13. Burwick C., Coppersmith D., D'Avignon E., Gennaro R., Halevi S., Jutla C., Matyas S.M. Jr., O'Connor L., Peyravian M., Safford D., Zunic N. *MARS — a candidate cipher for AES.* ? In: National Institute of Standards and Technology, August 1998 (см. `http://www.research.ibm.com/security/mars.html` или `http://www.nist.gov/aes`).
14. Cachin C. *Entropy Measures and Unconditional Security in Cryptography.* ? PhD thesis, ETH : Swiss Federal Institute of Technology (Zurich, 1997) (см. `ftp://ftp.inf.ethz.ch/pub/publications/dissertations/th12187.ps.gz`).
15. Carroll L. *The Hunting of the Snark: an Agony, in Eight Fits.* ? London : Macmillan and Co., 1876.
16. Chabaud F. and Joux A. *Differential Collisions in SHA-0.* ? In: Krawczyk H. (ed). *Advances in Cryptology — CRYPTO '98. Lecture Notes in Computer Science.* Springer-Verlag, 1998, vol. 1462, p. 56–71.
17. Courtois N. and Pieprzyk J. *Cryptanalysis of Block Ciphers with Overdefined Systems of Equations.* ? Cryptology ePrint Archive, Report 2002/044, 2002 (см. `http://eprint.iacr.org/`).
18. Daemen J. and Rijmen V. *AES Proposal: Rijndael.* ? In: National Institute of Standards and Technology, August 1998 (см. `http://www.esat.kuleuven.ac.be/~rijmen/rijndael` или `http://www.nist.gov/aes`).
19. Davis D., Ihaka R. and Fenstermacher P. *Cryptographic Randomness from Air Turbulence in Disk Drives.* ? In: Desmedt Y.G. (ed). *Advances in Cryptology — CRYPTO '94. Lecture Notes in Computer Science.* Springer-Verlag, 1994, vol. 839, p. 114–120.
20. Den Boer B. and Bosselaers A. *Collisions for the compression function of MD5.* ? In: Helleseth T. (ed). *Advances in Cryptology — EUROCRYPT '93.*

Lecture Notes in Computer Science. Springer-Verlag, 1993, vol. 765, p. 293–304.

21. Diffie W. and Hellman M.E. *New Directions in Cryptography* // IEEE Transactions on Information Theory. — 1976. — IT-22(6). — P. 644–654.
22. Dijkstra E.W. *The Humble Programmer* // Comm. of the ACM. — 1972. — 15(10). — P. 859–866. (Также издана как EWD340, `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF`.)
23. Di Crescenzo G., Ferguson N., Impagliazzo R. and Jakobsson M. *How To Forget a Secret. ?* In: Meinel C. and Tison S. (ed). *STACS 99. Lecture Notes in Computer Science*. Springer-Verlag, 1999, vol. 1563, p. 500–509.
24. Dobbertin H. *Cryptanalysis of MD4* // J. Cryptology. — 1998. — 11(4). — P. 253–271.
25. Dusse S.R. and Kaliski B.S. Jr. *A Cryptographic Library for the Motorola DSP56000.* ? In: Damgård I.B. (ed). *Advances in Cryptology — EUROCRYPT '90. Lecture Notes in Computer Science*. Springer-Verlag, 1990, vol. 473, p. 230–244.
26. Dworkin M. *Recommendation for Block Cipher Modes of Operation — Methods and Techniques.* ? National Institute of Standards and Technology, December 2001 (см. `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`).
27. Ellison C. *Improvements on Conventional PKI Wisdom.* ? In: Smith S. (ed). *1st Annual PKI Research Workshop — Proceedings*. 2002, p. 165–175 (см. `http://www.cs.dartmouth.edu/~pki02/Ellison/`).
28. Evertse J.-H. and Van Heyst E. *Which New RSA-Signatures Can Be Computed from Certain Given RSA-Signatures?* // J. Cryptology. — 1992. — 5(1). — P. 41–52.
29. Feistel H., Notz W.A. and Smith J.L. *Some Cryptographic Techniques for machine-to-Machine Data Communications* // Proc. of the IEEE. —1975. — 63(11). — P. 1545–1554.
30. Ferguson N., Kelsey J., Lucks S., Schneier B., Stay M., Wagner D., Whiting D. *Improved Cryptanalysis of Rijndael.* ? In: Schneier B. (ed). *Fast Software Encryption, 7th International Workshop, FSE 2000. Lecture Notes in Computer Science*. Springer-Verlag, 2000, vol. 1978, p. 213–230.
31. Ferguson N., Kelsey J., Schneier B. and Whiting D. *A Twofish Retreat: Related-Key Attacks Against Reduced-Round Twofish.* ? Twofish Technical Report 6, Counterpane Systems, February 2000 (см. `http://www.counterpane.com/twofish.html`).

32. Ferguson N. and Schneier B. *A Cryptographic Evaluation of IPsec*, 1999 (см. `http://www.counterpane.com/ipsec.html`).
33. Ferguson N., Schroeppel R. and Whiting D. *A simple algebraic representation of Rijndael.* ? In: Vaudenay S. And Youssef A.M. (ed). *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001. Lecture Notes in Computer Science.* Springer-Verlag, 2001, vol. 2259.
34. Ferguson N. *Collision attacks on OCB.* ? Comments to NIST, February 11, 2002 (см. `http://csrc.nist.gov/CryptoToolkit/modes/` и `http://csrc.nist.gov/CryptoToolkit/modes/comments/Ferguson.pdf`).
35. Fluhrer S., Mantin I. and Shamir A. *Weaknesses in the Key Schedule Algorithm of RC4.* ? In: Vaudenay S. and Youssef A.M. (ed). *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001. Lecture Notes in Computer Science.* Springer-Verlag, 2001, vol. 2259.
36. Freier A.O., Karlton P. and Kocher P.C. *The SSL Protocol, Version 3.0.* ? Internet draft, Transport Layer Security Working Group, November 18, 1996 (см. `http://home.netscape.com/eng/ssl3/`).
37. Goldberg I. and Wagner D. *Randomness and the Netscape Browser* // Dr. Dobb's Journal. — January 1996. — P. 66–70 (см. `www.cs.berkeley.edu/~daw/papers/ddj-netscape.html`).
38. Gutmann P. *Secure Deletion of Data from Magnetic and Solid-State Memory.* ? In: *USENIX Security Symposium Proceedings*, 1996 (см. `http://www.auckland.ac.nz/~pgut001`).
39. Gutmann P. *X.509 Style Guide*, October 2000 (см. `http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt`).
40. Harkins D. and Carrel D. *The Internet Key Exchange (IKE).* ? RFC 2409, November 1998.
41. Intel. *Intel 82802 Firmware Hub: Random Number Generator, Programmers Reference Manual*, December 1999 (см. на Web-узле Intel `http://www.intel.com`).
42. International Telecommunication Union. *X.680-X.683: Abstract Syntax Notation One (ASN.1).* ? X.690-X.693: ASN.1 encoding rules, 2002 (см. `www.itu.int/ITU-T/studygroups/com17/languages/x680-x693_0702.pdf`).
43. Jonsson J. *On the Security of CTR+CBC-MAC.* ? In: *Selected Areas in Cryptography, 9th Annual International Workshop, SAC 2002*, 2002 (см. `http://csrc.nist.gov/encryption/modes/proposedmodes/ccm/ccm-ad.pdf`).
44. Jueneman R.R. *Analysis of Certain Aspects of Output Feedback Mode.* ? In: Chaum D., Rivest R.L., and Sherman A.T. (ed). *Advances in Cryptology, Proceedings of Crypto 82.* Plenum Press, 1982, p. 99-128.

45. Kahn D. *The Codebreakers, The Story of Secret Writing.* ? New York : Macmillan Publishing Co., 1967.

46. Kelsey J., Schneier B. and Ferguson N. *Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator.* ? In: Heys H. and Adams C. (ed). *Selected Areas in Cryptography, 6th Annual International Workshop, SAC '99. Lecture Notes in Computer Science.* Springer-Verlag, 1999, vol. 1758.

47. Kelsey J., Schneier B., Wagner D. and Hall C. *Cryptanalytic Attacks on Pseudorandom Number Generators.* ? In: Vaudenay S. (ed). *Fast Software Encryption, 5th International Workshop, FSE '98. Lecture Notes in Computer Science.* Springer-Verlag, 1998, vol. 1372, p. 168–188.

48. Kelsey J., Schneier B., Wagner D. and Hall C. *Side Channel Cryptanalysis of Product Ciphers //* Journal of Computer Security. — 2000. — 8(2-3). — P. 141–158 (см. также `http://www.counterpane.com/side_channel.html`).

49. Kelsey J., Schneier B. and Wagner D. *Key-Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES.* ? In: Koblitz N. (ed). *Advances in Cryptology — CRYPTO '96. Lecture Notes in Computer Science.* Springer-Verlag, 1996, vol. 1109, p. 237–251.

50. Kent S. and Atkinson R. *Security Architecture for the Internet Protocol.* ? RFC 2401, November 1998.

51. Granted Innovation Patent No. AU2001100012 A4. *Circular transportation facilitation device.* Keogh J. Australian Patent Office, August 2001 (см. `http://www.ipmenu.com/archive/AUI_2001100012.pdf`).

52. Killian J. and Rogaway P. *How to Protect DES Against Exhaustive Key Search.* ? In: Koblitz N. (ed). *Advances in Cryptology — CRYPTO '96. Lecture Notes in Computer Science.* Springer-Verlag, 1996, vol. 1109, p. 252–267.

53. Knudsen L.R. and Rijmen V. *Two Rights Sometimes Make a Wrong.* ? In: *Workshop on Selected Areas in Cryptography (SAC '97).* 1997, p. 213–223 (см. `http://adonis.ee.queensu.ca:8000/sac/sac97/papers.html`).

54. Knuth D.E. *Seminumerical Algorithms. Vol. 2 of The Art of Computer Programming.* ? Addison-Wesley, 1981. (Кнут Д. *Искусство программирования. Т. 2. Получисленные алгоритмы, 3-е изд.*: Пер. с англ. ? М. : Издат. дом "Вильямс", 2000.)

55. Kocher P.C. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems.* ? In: Koblitz N. (ed). *Advances in Cryptology — CRYPTO '96. Lecture Notes in Computer Science.* Springer-Verlag, 1996, vol. 1109, p. 104–113.

56. Kocher P., Jaffe J. and Jun B. *Differential Power Analysis.* ? In: Wiener M. (ed). *Advances in Cryptology — CRYPTO '99. Lecture Notes in Computer Science.* Springer-Verlag, 1999, vol. 1666, p. 388–397.

57. Kohl J. and Neuman C. *The Kerberos Network Authentication Service (V5).* ? RFC 1510, September 1993.

58. Krawczyk H., Bellare M. and Canetti R. *HMAC: Keyed-Hashing for Message Authentication.* ? RFC 2104, February 1997.

59. Krovetz T., Black J., Halevi S., Hevia A., Krawczyk H. and Rogaway P. *UMAC: Message Authentication Code using Universal Hashing.* ? RFC draft draft-krovetz-umac-01.txt, 2000 (см. `http://www.cs.ucdavis.edu/~rogaway/umac/`).

60. Lai X. and Massey J.L. *A Proposal for a New Block Encryption Standard.* ? In: Damgård I.B. (ed). *Advances in Cryptology — EUROCRYPT '90. Lecture Notes in Computer Science.* Springer-Verlag, 1990, vol. 473, p. 389–404.

61. Lai X., Massey J.L. and Murphy S. *Markov Ciphers and Differential Cryptanalysis.* ? In: Davies D.W. (ed). *Advances in Cryptology — EUROCRYPT '91. Lecture Notes in Computer Science.* Springer-Verlag, 1991, vol. 547, p. 17–38.

62. Lenstra A.K. and Verheul E.R. *Selecting Cryptographic Key Sizes* // J. Cryptology. — 2001. — 14(4). — P. 255–293.

63. Matsumoto T., Matsumoto H., Yamada K. and Hoshino S. *Impact of Artificial "Gummy" Fingers on Fingerprint Systems.* — In: *Proc. of SPIE, Vol #4677, Optical Security and Counterfeit Deterrence Techniques IV* (см. также `www.itu.int/itudoc/itu-t/workshop/security/present/s5p4.pdf`).

64. Menezes A.J., Van Oorschot P.C. and Vanstone S.A. *Handbook of Applied Cryptography.* ? CRC Press, 1996 (ISBN 0-8493-8523-7).

65. Mills D.L. *Network Time Protocol (Version 3).* ? RFC 1305, March 1992.

66. Mills D. *Simple Network Time Protocol (SNTP) Version 4.* ? RFC 2030, October 1996.

67. Montgomery P. *Modular Multiplication without Trial Division* // Mathematics of Computation. — 1985. — 44(170). — P. 519–521.

68. National Institute of Standards and Technology. *DES Modes of Operation.* — FIPS PUB 81, December 1980 (см. `http://www.itl.nist.gov/fipspubs/`).

69. National Institute of Standards and Technology. *Data Encryption Standard (DES).* — FIPS PUB 46-2, December 1993 (см. `http://www.itl.nist.gov/fipspubs/`).

70. National Institute of Standards and Technology. *Secure Hash Standard.* — FIPS PUB 180-1, April 1995 (см. `http://www.itl.nist.gov/fipspubs/`).

71. National Institute of Standards and Technology. *AES Round 1 Technical Evaluation, CD-1: Documentation*, August 1998 (см. `http://www.itl.nist.gov/aes`).

72. National Institute of Standards and Technology. *Data Encryption Standard (DES).* — DRAFT FIPS PUB 46-3, 1999 (см. `http://csrc.ncsl.nist.gov/fips/`).

73. National Institute of Standards and Technology. *Proc. 3rd AES candidate conference*, April 2000.

74. National Institute of Standards and Technology. *Secure Hash Standard (draft).* — DRAFT FIPS PUB 180-2, 2001 (см. `http://csrc.nist.gov/encryption/shs/dfips-180-2.pdf`).

75. Needham R.M. and Schroeder M.D. *Using Encryption for Authentication in Large Networks of Computers* // Comm. of the ACM. — 1978. — 21(12). — P. 993–999.

76. Preneel B. and Van Oorschot P.C. *On the Security of Two MAC Algorithms.* ? In: Maurer U. (ed). *Advances in Cryptology — EUROCRYPT '96. Lecture Notes in Computer Science.* Springer-Verlag, 1996, vol. 1070, p. 19–32.

77. Rivest R.L., Robshaw M.J.B., Sidney R. And Yin Y.L. *The RC6 Block Cipher. ?* In: National Institute of Standards and Technology, August 1998 (см. `http://www.rsasecurity.com/rsalabs/rc6/` или `http://nist.gov/aes`).

78. Rivest R.L. *The MD4 Message Digest Algorithm.* ? In: Menezes A.J. and Vanstone S.A. (ed). *Advances in Cryptology — CRYPTO '90. Lecture Notes in Computer Science.* Springer-Verlag, 1991, vol. 547, p. 17–38.

79. Rivest R.L. *The RC5 Encryption Algorithm.* ? In: Preneel B. (ed). *Fast Software Encryption, Second International Workshop, FSE '94. Lecture Notes in Computer Science.* Springer-Verlag, 1995, vol. 1008, p. 86–96.

80. Rivest R., Shamir A. and Adleman L. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems* // Comm. of the ACM. — 1978. — 21. — P. 120–126.

81. Rivest R. *The MD5 Message-Digest Algorithm.* ? RFC 1321, April 1992.

82. Rogaway P., Bellare M., Black J. and Krovetz T. *OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption.* ? In: *Eighth ACM Conference on Computer and Communications Security* (CCS-8). ACM Press, 2001, p. 196–205.

83. Rogaway P., Bellare M., Black J. and Krovetz T. *OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption*, September 2001 (см. `http://www.cs.ucdavis.edu/~rogaway`).

84. RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard*, January 2001 (см. `http://www.rsasecurity.com/rsalabs/pkcs`).

85. Schneier B., Kelsey J., Whiting D., Wagner D., Hall C., Ferguson N. *The Twofish Encryption Algorithm, A 128-bit Block Cipher.* ? Wiley, 1999.

86. Schneier B. *Applied Cryptography, Protocols, Algorithms, and Source Code in C.* ? John Wiley & Sons, Inc., 1994 (ISBN 0-471-59756-2).

87. Schneier B. *Applied Cryptography, Second Edition, Protocols, Algorithms, and Source Code in C.* ? John Wiley & Sons, Inc., 1996; ISBN 0-471-12845-7. (Шнайер Б. *Прикладная криптография, 2-е издание: протоколы, алгоритмы, исходные тексты на языке Си.* ? М. : Триумф, 2002.)

88. Schneier B. *Secrets and Lies, Digital Security in a Networked World.* ? John Wiley & Sons, Inc., 2000. ISBN 0-471-25311-1. (Шнайер Б. *Секреты и ложь. Безопасность данных в цифровом мире.* ? СПб. : Питер, 2003.)

89. Dr. Seuss. *Horton Hears a Who!* ? Random House, 1954.

90. Shannon C.E. *A Mathematical Theory of Communication* // The Bell Systems Technical Journal. — 1948. — 27. — P. 370–423; 623–656 (см. `http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html`).

91. Wagner D., Ferguson N. and Schneier B. *Cryptanalysis of FROG.* ? In: *Proc. 2nd AES candidate conference*. National Institute of Standards and Technology, March 1999, p. 175–181.

92. Wagner D. and Schneier B. *Analysis of the SSL 3.0 protocol.* ? In: *Proc. of the Second USENIX Workshop on Electronic Commerce*. USENIX Press, November 1996, p. 29–40 (исправленный вариант см. на `http://www.counterpane.com`).

93. Whiting D., Housley R. and Ferguson N. *Counter with CBC-MAC (CCM)*, June 2002 (см. `http://csrc.nist.gov/encryption/modes/proposedmodes/ccm/ccm.pdf`).

94. Wiener M.J. *Cryptanalysis of short RSA secret exponents* // IEEE Transactions on Information Theory. — 1990. — 36(3). — P. 553–558.

95. Winternitz R.S. *Producing a One-way Hash Function from DES.* ? In: Chaum D. (ed). *Advances in Cryptology, Proceedings of Crypto 83*. Plenum Press, 1983, p. 203–207.

96. Wu T. *The Secure Remote Password Protocol.* ? In: *Proc. of the 1998 Network and Distributed System Security (NDSS '98) Symposium*, March 1998.

Предметный указатель

Ш

Э

Научно-популярное издание

Нильс Фергюсон, Брюс Шнайер

Практическая криптография

Литературный редактор	*Т.П. Кайгородова*
Верстка	*А.Н. Полинчик*
Художественный редактор	*Е.П. Дынник*
Корректоры	*Л.А. Гордиенко*, *О.В. Мишутина*, *Л.В. Пустовойтова*

Издательский дом "Вильямс"
101509, г. Москва, ул. Лесная, д. 43, стр. 1
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати

Подписано в печать 19.11.2004. Формат 70x100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 21,9. Уч.-изд. л. 24,8
Тираж 3000 экз. Заказ №

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций
197110, С.-Петербург, Чкаловский пр., 15