прикладная

КРИПТОГРАФИЯ

Использование
и синтез
криптографических
интерфейсов

А. Щербаков, А. Домашев

РУССКАЯ РЕДАКЦИЯ

прикладная

КРИПТОГРАФИЯ

Использование
и синтез
криптографических
интерфейсов

А. Щербаков, А. Домашев

Москва 2003

РУССКАЯ РЕДАКЦИЯ

УДК 004.45
ББК 32.973.26-018.2
Щ61

Щербаков Л. Ю., Домашев А. В.
Щ61 Прикладная криптография. Использование и синтез криптографических интерфейсов. — М.: Издательско-торговый дом «Русская Редакция», 2003. — 416 с.: ил.

ISBN 5-7502-0215-1

Основное внимание в книге авторы уделяют прикладным вопросам — использованию уже существующих в операционных системах Microsoft криптографических модулей (криптопровайдеров) и созданию собственных криптопровайдеров при помощи удобного инструментария — CSPDK.

В книге очень подробно рассмотрен Microsoft Cryptographic. Application Programming Interface (CryptoAPI), впервые описаны принципы реализации интерфейса вызовов CryptoAPI, приведены подробные примеры получения информации о криптопровайдерах, использования CryptoAPI для обмена защищенными сообщениями и для реализации схемы цифровой подписи. Кроме того, подробно рассматриваются разработка криптопровайдеров и необходимый для этого инструментарий. Читатель познакомится с вопросами системной реализации криптографических средств, деталями реализации криптографических алгоритмов в рамках собственного криптопровайдера и перспективами развития криптоинтерфейсов (функции CryptoAPI 2.0).

Книга содержит эксклюзивные материалы, не публиковавшиеся ранее ни в одном из отечественных изданий, о применении штатных криптопровайдеров, разработке собственных криптопровайдеров при помощи CSPDK, а также об особенностях прикладной криптографии.

Книга состоит из 6 глав, 3 приложений и предметного указателя, снабжена библиографическим списком и содержит полный справочник функций CryptoAPI.

Это издание предназначено криптографам-разработчикам, использующим CryptoAPI, кроме того, ее можно рекомендовать в качестве учебного пособия по прикладной криптографии для профильных ВУЗов.

УДК 004.45
ББК 32.973.26-018.2

ISBN 5-7502-0215-1

Оглавление

Предмет прикладной криптографии

Современная компьютерная безопасность решает чрезвычайно широкий спектр задач — от защиты специальных государственных сетей до обеспечения закрытой электронной почты на домашнем компьютере.

Применение методов криптографической защиты характерно для решения подавляющего большинства проблем безопасности. Аутентификация, шифрование данных, контроль целостности, электронная цифровая подпись — понятия, хорошо знакомые сегодня достаточно широкому кругу разработчиков и пользователей.

В последние несколько лет появились десятки статей, монографий и переводных изданий, касающихся различных аспектов криптографии. В основном это работы теоретического плана, посвященные описанию, синтезу и анализу криптографических алгоритмов и протоколов. Понимание этих работ и использование их результатов в практическом плане достаточно затруднено.

Исключений из этого правила довольно немного. Так, на наш взгляд, одной из лучших работ является монография А.Алферова, А.Зубова, С.Кузьмина и А.Черемушкина «Основы криптографии: Учебное пособие» (М.: Гелиос АРВ, 2001).

Вместе с тем необходимо отметить, что использование криптографических алгоритмов в практической деятельности, в частности их внедрение (встраивание) в компьютерные системы, представляет собой отдельное научное направление.

В общепризнанной международной практике весь комплекс вопросов использования криптоалгоритмов описывается дисциплиной «прикладная криптография».

Наиболее известная работа по прикладной криптографии - монография Bruce Schneier «Applied Cryptography Second Edition:protocols, algorithms, and source code in C» (John Wiley & Sons, Inc. New York, 1997).

В предисловии автор так характеризует содержание своей книги*:

«Глава 1 представляет собой введение в криптографию, описывает множество терминов, к ней кратко рассматривается докомпьютерная криптография.

Главы со 2 по 6 (Часть I) описывают криптографические протоколы (что люди могут сделать с помощью криптографии) от простых (передача шифрованных сообщений от одного человека другому) до сложных (щелканье монетой по телефону) и тайных (скрытое и анонимное обращение электронных денег). Некоторые из этих протоколов очевидны, другие удивляют возможностями. Множество людей и не представляет всех проблем, которые может решать криптография.

В главах с 7 по 10 (Часть II) обсуждаются методы криптографии и наиболее распространенные способы применения криптографии. В главах 7 и 8 рассказывается о ключах: какова должна быть длина безопасного ключа, как генерировать, хранить и распределять ключи и т.д. Управление ключами представляет собой труднейшую часть криптографии и часто является ахиллесовой пятой систем, защищенных во всем остальном.

В главе 9 рассматриваются различные способы использования криптографических алгоритмов, а глава 10 описывает особенности и цели применения утих алгоритмов: как их выбирать, реализовывать и применять.

Главы с 11 но 23 (Часть 111) описывают эти алгоритмы.

В главе 11 изложена математическая база, она необходима, только если вы интересуетесь алгоритмами с открытыми ключами. Если вы собираетесь использовать DES (или что-то похожее), ее можно пропустить. В главе 12 обсуждается алгоритм DES. его история, безопасность и разновидности. В главах 13, 14 и 15 рассказывается о других блочных алгоритмах. Если нам нужно нечто более надежное, чем DES, сразу переходите к разделам о IDEA и тройном DES. Если вы желаете узнать о группе алгоритмов, некоторые из которых могут быть безопаснее DES, прочитайте всю главу целиком.

В главах 16 и 17 обсуждаются потоковые алгоритмы. В главе 18 подробно рассматриваются однонаправленные хэш-функции,

* Перевод наш. — *Прим. ант.*

среди которых самыми являются MDS и SHA, хотя я останавливаюсь и на многих других. В главе 19 рассматриваются алгоритмы шифрования с открытым ключом, а в главе 20 алгоритмы цифровой подписи с открытым ключом. В главе 21 обсуждаются алгоритмы идентификации с открытым ключом, а в главе 22 алгоритмы обмена с открытым ключом. Самыми важными являются алгоритмы RSA, DSA, ФиатШамира (FiatShamir) и ДиффиХелмана (DiffieHellman).

Глава 23 содержит ряд эзотерических алгоритмов и протоколов с открытым ключом, математика в этой главе достаточно сложна.

Главы 24 и 25 (Часть IV) переносят вас в реальный мир криптографии. В главе 24 обсуждаются некоторые современные применения алгоритмов и протоколов, в то время как глава 25 касается некоторых политических аспектов криптографии. Несомненно, эти главы не являются всеохватывающими.

В книгу также включены исходные коды 10 алгоритмов, рассмотренных в Части III».

Однако содержание прикладной криптографии по Шнаеру с современных научно-методических позиций представляется избыточным. Из предисловия к книге ясно, что значительная часть монографии посвящена классической криптографии — это описание и анализ криптографических алгоритмов и протоколов. О их практической реализации, насущно необходимой специалистам-прикладникам, рассказано только в третьей части монографии.

Исходя из практических потребностей разработчиков и пользователей защищенных компьютерных систем, мы определяем прикладную криптографию как комплексную дисциплину научно-практического характера, описывающую предметную область, связанную с методами реализации различных криптографических алгоритмов и протоколов.

Криптографические алгоритмы — обобщенное понятие, включающее алгоритмы шифрования, алгоритмы электронной цифровой подписи (ЭЦП), алгоритмы получения вспомогательной информации для алгоритмов шифрования и ЭЦП (случайных чисел, ключей, простых чисел, подстановок и т.д.)

Криптографические протоколы — формальное описание технологии использования криптоалгоритмов для решения прикладных задач.

Таким образом, *объект исследования* прикладной криптографии – реализация некоторых априорно заданных процедур (алгоритмов, протоколов), вид и свойства которых уже изучены теоретической криптографией.

Применение криптоалгоритмов и протоколов позволяет решать задачи оптимизации (преимущественно оптимизации скорости) и проблемы, связанные с обеспечением устойчивости и надежности (синтеза криптоалгоритмов и протоколов, устойчивых к внешним воздействиям платформы, на которой они реализованы, например к аппаратным или программным ошибкам и сбоям).

Области прикладной криптографии перечислены далее:

* реализация алгоритмов шифрования;
* реализация алгоритмов выработки и хранения ключей;
* методы реализации криптоалгоритмов, устойчивых к программным и аппаратным ошибкам и сбоям;
* методы проверки адекватности реализации криптоалгоритмов (в частности, методы проверки соответствия формального описания алгоритма его программному или аппаратномуисполнению);
* методы реализации и контроля качества датчиков случайных чисел;
* методы реализаций вспомогательных элементов шифров с заданными свойствами;
* методы реализации алгоритмов хеширования и контроля целостности;
* методы реализации алгоритмов и протоколов электронной цифровой подписи;
* методы реализации (создания, вычисления) вспомогательных элементов ЭЦП (генерация простых чисел, эллиптических кривых и т.д.);
* методы реализации протоколов;
* методы использования (встраивания) криптоалгоритмов в прикладных системах.

Этой книгой мы начинаем цикл работ по прикладной криптографии, предназначенных разработчикам программных средств, в значительной степени использующих криптографические механизмы. Основное внимание в данной работе уделено криптографическим интерфейсам как наиболее удобному механизму встраивания криптоалгоритмов в прикладные системы.

Глава 1

Основные понятия и алгоритмы криптографии

Кратко рассмотрим основные понятия классической криптографии, которые мы будем неоднократно использовать к этой книге.

Шифр — совокупность алгоритмов или отображений открытой (общедоступной) информации, представленной в формализованном виде, в недоступный для восприятия шифрованный текст (также представленный в формализованном виде), который зависит от внешнего параметра (ключа). Не зная последнего, практически невозможно по шифрованной информации определить открытую информацию, а по шифрованной и открытой информации — ключ.

Шифратор - аппарат или программа, реализующая шифр. В современной литературе вводится понятие *средства криптографической защиты информации* (СКЗИ), которое подразумевает и шифратор, но в целом является более широким.

Ключ — некоторый неизвестный параметр шифра, позволяющий выбрать для шифрования и расшифрования конкретное преобразование из всего множества преобразований, составляющих шифр. Простая ассоциация — ключ от замка: имеется множество одинаковых но форме ключей, но лишь некоторые (или один) откроют конкретный замок.

Шифрование — процесс создания шифрованного текста при наличии ключа.

Дешифрование — восстановление открытого текста или ключа из шифрованного текста.

Противник — субъект (или физическое лицо), который не знает и не должен знать ключа или открытого текста, но стремящийся получить его.

Как видно, понятие «противник» в криптографии тесно связано с таким понятием, как «твердое незнание ключа». Какие же из компонентов СКЗИ следует считать известными противнику? На этот вопрос отвечает *правило Кирхгофа*: противнику известно все (все параметры шифра), кроме ключа, использованного для шифрования данного текста.

Б современной криптографии считается также, что противник имеет возможность произвольным образом изменять сообщения. Поэтому необходимо удостоверить подлинность сообщения.

Подлинность — принадлежность сообщения конкретному автору и неизменность содержания сообщения.

В следующем разделе мы попытаемся дать более строгое и конструктивное определение компонентов шифра.

Шифр и требования к нему

Понятие «шифр» крайне сложно определить строго, почти любое определение будет достаточно размытым либо слишком общим. Надо заметить, что споры о точном определении шифра продолжаются по сей день. Это связано с необходимостью лицензирования деятельности в области криптографии, которое требует точного юридического определения ее границ. Однако далее нам придется оперировать компонентами шифра. Поэтому сейчас мы дадим его математическое определение, перечислим требования к шифру, вытекающие из здравого смысла, и приведем примеры шифрующих преобразований.

Пусть имеется некоторое множество X; назовем его исходным множеством или множеством открытых текстов, например тексты на некотором языке, двоичные векторы или целые числа. Чаще всего X полагается множеством векторов длины n, каждая координата которого может принадлежать множеству M, именуемому алфавитом.

Зададим также Y — множество зашифрованных текстов. Его также удобно представлять множеством векторов длины m, причем каждая координата вектора принадлежит множеству S (последнее может совпадать с M или быть отличным от него, когда для представления шифртекста используются символы другого алфавита, например «пляшущие человечки»).

Таким образом,

$X = \{(x_1, x_2, ..., x_n)\}$

$Y = \{(y_1, y_2, ..., y_m)\}$

Зададим также K — множество параметров преобразования или множество ключей. Данный параметр задает, какое именно преобразование (описанное ниже) будет использовано.

Далее введем некоторое отображение E множества X на множество Y, зависящее от параметра k из множества K, такое, что:

E(k, x) = y, где

k принадлежит K;

x принадлежит X;

y принадлежит Y.

Причем для любого x из X существует отображение D, также зависящее от параметра k такое, что:

D(E (k, x)) = x и x определяется однозначно.

Данное условие говорит лишь о том, что после расшифрования D должен получиться тот же текст, а не какой-либо другой.

Отображение E назовем шифрующим отображением, а D — расшифровывающим отображением.

Посмотрим, какие выводы можно сделать из этого определения.

1. Мощность множества X всегда не больше мощности множества Y, поскольку в противном случае не удастся добиться однозначного отображения Y в X и, значит, получить однозначного расшифрования.
2. При любом фиксированном k отображение E биективно (взаимно однозначно).

Однако в сформулированном определении мы ничем не выделили шифры среди прочих преобразований информации, например помехоустойчивого или сжимающего кодирования.

Поэтому сформулируем свойства, следующие из постулатов простой логики,

1. Множество K должно иметь настолько большую мощность, чтобы исключить возможность перебора всех различных преобразований E.
2. По зависимости y=E(k.x) очень трудно определить как x, так и k.

Дополнив основное определение данными свойствами, можно ли утверждать, что шифр будет «хорошим»? По-видимому, мет: данные требования являются *необходимыми, но недостаточными.* Мы не можем быть абсолютно уверенными в том, что по y определить k или x действительно «очень трудно». Даже

после определения дополнительных постулатов определение шифра все еще остается очень «философским», охватывающим слишком широкий класс преобразований, которые при более детальном рассмотрении не являются шифрами. Подробнее о проблеме синтеза надежных шифров мы расскажем далее. А пока приведем несколько примеров,

Пример 1

Пусть $X = \{(x_1, ..., x_n)\}$,

$Y = \{(y_1, ..., y_n)\}$,

$K = \{(k_1, ..., k_n)\}$,

x_i, k_i, y_i принадлежат M, отображение E задано операцией #:

$y_i = k_i \# x_i$ (M,#) — группа (см. Приложение 3).

Отображение D задается операцией # как:

$x_i = k_i^{-1} \# y_i$, где $k^{-1} \# k = 0$ — нейтральный элемент.

Пример 2

Пусть x_i, y_i и k_i принадлежат {0,1}.

Операция # является операцией «сумма по модулю *1*» («исключающее или»).

$y = x + k \pmod 2$ — шифрование;

$x = y + k \pmod 2$ — расшифрование.

Пример 3

Пусть имеется множество подстановок мощности r:

$P = \{p_1, ..., p_r\}$,

каждая подстановка имеет степень n.

X и Y — множества векторов длины L, каждая координата которых принадлежит множеству M={1, 2, ..., n}.

$X = \{(x_1, ..., x_L)\}$,

$Y = \{(y_1, ..., y_L)\}$,

Задано также множество векторов длины L

$G = \{(g_1, ..., g_L)\}$, g_i принадлежит {1, 2, ..., r}

В данном случае множество K={P,G}.

Операция E: $y_i = p_{gi}(x_i)$ i=1, 2, ..., L

Операция D: $x_i = p_{gi}^{-1}(y_i)$ i=1, 2, ..., l

p^{-1} - обратный элемент группы подстановок степени n, т.е.

$p * p^{-1} = E$

Здесь * — операция умножения подстановок, Е — единичная подстановка. Следовательно, и данном случае первоначально формируется вектор g длины L, для шифрования первого символа выбирается подстановка с номером g_1 (первая координата вектора g) и т.д.

Пример 4

Пусть r= 1, т.е. множество Р состоит из одной подстановки, тогда такой шифр называется шифром простой замены в алфавите мощности п.

Ключ шифра в примере 4 — подстановка степени п [их число п! (факториал)] и эта подстановка воздействует на каждую букву сообщения (которая берется из алфавита 1, 2, ..., п). Мощность множества подстановок степени п совпадает со множеством возможных ключей. Легко видеть, что это число крайне велико, например, для алфавита в 32 символа (русский язык) оно составляет 32! (32 факториал). Это больше, чем 10^{80}. Исходя из этого, Леонард Эйлер считал, что шифр простой замены является очень стойким (кстати, так полагали и все те, кто в то время использовал такие шифры для переписки). Рассмотрим процесс шифрования.

Подстановка р:

1 2 3 4 5

4 1 2 5 3

Исходное сообщение в алфавите {1, 2, 3, 4, 5}:

X = 11532451134

Зашифрованное сообщение (в том же алфавите):

У = 44321534425

Мы видим, что наиболее часто встречается в зашифрованном тексте символ 4 — образ символа 1 после воздействия подстановки р. Если нам известны статистические закономерности исходного текста, то они сохранятся в шифртексте, но для образов соответствующих символов. Однако не стоит смеяться над гениальным математиком Эйлером — он окажется прав, если нам не удастся набрать нужной статистики на шифрованном тексте. По какой причине это возможно? Потому что символов шифртекста немного. Для того чтоб статистические оценки оказались сколько-нибудь достоверными, требуется не меньше символов, чем в алфавите. Увеличим алфавит — пусть, каждую «букву» файла представляет последовательность из 32 байт.

Тогда на реальных объемах данных почти никогда не удастся получить достоверной статистики. В этом состоит идея *блочного шифрования.*

Рассматривая примеры, обратите внимание, что в каждом случае объем необходимого ключа совпадает с объемом передаваемого сообщения. Хотелось бы изменить традицию: на основе не слишком длинного ключа (но все же такого, чтоб все его возможные значения не удавалось перебрать) вырабатывать длинные неповторяющиеся последовательности и использовать их для шифрования.

Пример 5

Рассмотрим линейную рекуррентную последовательность над полем GF(2), заданную характеристическим полиномом

$f(x) = x^4 + x^3 + 1$

Таблица 1-1. Преобразование, заданное полиномом

x0	x1	x2	x3
x1	x2	x3	x0+x2

$x_{t+1} = x_t + x_{t-3}$, t = 5, 6, ...n; «+» сумма по модулю 2;

0 1 1 1 1 — начальное состояние.

Таблица 1-2. Цикл преобразования

Шаг	x0	x1	x2	x3
1	1	1	1	0
2	1	1	0	1
3	1	0	1	0
4	0	1	0	1
5	1	0	1	1
6	0	1	1	0
7	1	1	0	0
8	1	0	0	1
9	0	0	1	0
10	0	1	0	0
11	1	0	0	1
12	0	0	1	1
13	0	1	1	1
14	0	1	1	1
15	1	1	1	1

Мы видим, что, задавшись начальным состоянием из 4 бит, можно получить последовательности с периодом 15. Существуют оценки, *согласно* которым для однородных линейных ре-

куррентных последовательностей (ЛРП) над конечными полями достижимы периоды T.

$T = P^k - 1$, где

P — число элементов поля,

k — степень характеристического полинома ЛРП.

Так, для тюля GF(2) P=2, следовательно, в примере достигнут максимально возможный период, равный 15 (k=4). Однако это выполнено не для любого характеристического полинома.

Вообще говоря, шифрующее преобразование можно представить как некоторый конечный автомат A,

A={I, S, O, X(I, S)-S, Y(S, I)-O}, где

I — множество входных слов;

O — множество выходных слов;

S — множество состояний;

X — отображение декартова произведения (I, S) во множество S — функция переходов автомата;

Y — отображение декартова произведения (I, S) во множество O — функция выходов автомата.

Оценка качества криптографических преобразований

Оценивая качества шифрующего отображения E, мы должны задаться некими конкретными условиями. Желательно добиться выполнения условия 2 определения шифра, а именно, чтобы по y — шифртексту или y и x — зашифрованному и открытому тексту найти параметр преобразования k оказалось достаточно трудно.

Можно сформулировать несколько типовых классов задам;

* нахождение ключа по шифртексту, а значит, в силу обратимости отображения E и открытого текста x по уравнению x = D(k, y);
* нахождение открытого текста по шифртексту без нахождения ключа шифрования;
* нахождение ключа k no паре x и y, связанной соотношением y = E(k, x) или нескольким таким нарам;
* нахождение некоторых x для известных совокупностей пар (x, y) таких, что:

  y = E(k, x), т.е. для зашифрованных па одном ключе.

Поясним сформулированные задачи примерами.

Пример 6

Наиболее непонятной с точки зрения здравого смысла кажется задача 3. Ну, откуда, скажите на милость, может взяться x для y = E(k, y)?

Предположим, однако, что текст x известен, Например, среди зашифрованных на одном ключе файлов имеется утилита DOS, которая есть у всех (открытый текст известен), следовательно, найдя в этом случае ключ, мы восстановим все файлы. При передаче данных по каналу связи может возникнуть ситуация, когда большая часть текста также известна — например, передается платежная ведомость вполне определенной структуры либо запрос типа «На Ваш исходящий — наш входящий...». Значит, в большинстве систем хранения и передачи информации необходимо обеспечить высокую трудоемкость решения задачи 3.

Пример 7

Предположим, требуется найти открытый текст для двух телеграмм, зашифрованных на одном ключе методом наложения ключа при помощи коммутативной групповой операции («сумма по модулю 2»). В этом и последующих примерах используется ASCII-кодировка.

Рассмотрим схему шифрования из примера примера 2.

y = k + x(mod 2)

Ключ длиной 21 байт:

K= 5 c 17 7f e 4e 37 d2 94 10 9 2c 22 57 ff c8 Ь Ъ2 70 54

Телеграмма 1.

T1=H a В а ш и с х о д я щ и й о т 1 2 0 4

T1=8d aO 82 aO e8 a8 el c5 ae a4 ef c9 a8 a9 ae e2 31 32 30 34

Телеграмма 2.

T2= В С е в е р н ы й ф и л и а л Б а н к

T2= 82 91 a5 a2 a5 eO ad eb a9 e4 a8 ab a8 aO ab 81 aO ad aa

Y1=T1 + K =

88 ac 95 df e6 еб d6 37 3a Ь4 e6 c7 8a fe 51 2a 3a 80 40 60

Y2= T2+K -

87 9d Ь2 dd ab ac 9a 39 3d f4 a1 85 8a t'7 54 49 ab 1f da f4

Рассмотрим Z - Y1 + Y2 = T1 + K + T2 + K = T1 + T2

Z = Y1 + Y2 =

f31 27 2 4d 48 4c e 7 40 47 42 0 9 5 63 91 9f9a 94 O

Предположим, что телеграмма начинается со слов:

«Н а В а ш ...»

8d aO 82 aO e8

Сложим по модулю 2 это сообщение с текстом Z:

+ f 31 27 2 4d

mod2 8d aO 82 aO e8

---

82 91 **a5** a2 a5

В С е в е

По-видимому, это слово «Северный», тогда, действуя описанным образом в T1, получим «исхо», вероятно, — «исходящий», В T2 читаем: «филиа» — «филиал».

Действуя таким образом, мы, скорее всего, прочитаем обе телеграммы.

Возникает парадоксальная ситуация, когда, не зная ключа и не стремясь его определить, мы читаем обе телеграммы.

Введем теперь интуитивно ясное понятие стойкости шифра, описывающее сложность преобразования E для задачи нахождения параметра преобразования. Итак, *стойкость шифрующего преобразования* — трудоемкость задачи отыскания параметра преобразования ключа k либо текста x в определенных условиях (например, тех, которые сформулированы выше).

Понятие *трудоемкости* связано с понятием алгоритма, т.е. полагается, что для поиска ключа k строится некоторый алгоритм и стойкость оценивается его трудоемкостью. Однако далее мы покажем, что иногда в результате действия алгоритма нахождения ключа или исходного текста появляются несколько ключей или осмысленных текстов, из которых придется выбирать нужный (а иногда это просто невозможно). Рассмотрим *идеальный сличай* — шифр является абсолютно стойким, т.е. текст x не удастся найти никогда. Клод Шеннон сформулировал условия для такого шифра. Эти условия в общем-то достаточно логичны: при перехвате некоторого текста y противник не должен получить никакой информации о переданном x.

Введем следующие обозначения:

* p(x/y) — вероятность того, что зашифрован текст x при перехвате текста y;

* p(y/x) — вероятность того, что при условии зашифрования x получен был именно y или суммарная вероятность использования всех ключей, которые переводят x в y;
* p(y) — вероятность получения криптограммы y;
* p(x) — вероятность отобрать для зашифрования текст x из всех возможных.

Чтобы противник не получил никакой информации о ключе или об открытом тексте, необходимо и достаточно следующее условие:

$p(x) = p(x/y)$

Т.е. вероятность выбора текста для шифрования из множества возможных текстов не должна меняться при получении криптограммы, соответствующей данному тексту.

По формуле Байеса:

$$p(x/y) = \frac{p(x)p(y/x)}{p(y)}$$

Из равенства $p(x) = p(x/y)$ следует, что $p(y) = p(y/x)$, т.е. суммарная вероятность всех ключей, переводящих x в y, должна быть равна вероятности *получения* криптограммы y и не должна зависеть от x.

Из этого равенства можно вывести также два важных следствия:

* число всевозможных ключей не должно быть меньше числа сообщений;
* для каждого y требуется ключ k, который переводит любой x в данный y (так называемое условие транзитивности), в противном случае, получив конкретный шифртекст y, можно исключить из рассмотрения некоторые ключи или открытые тексты.

Эти условия являются необходимыми, т.е. невыполнение хотя бы одного из них делает шифр не абсолютно стойким.

Пусть x_i, y_i, Ц принадлежат {0, 1}.

$y = x + k \pmod 2$ — шифрование;

$x = y + k \pmod 2$ — расшифрование.

Рассмотрим сообщения длины 1, все меньшие сообщения дополняются до длины 1 хаотической информацией.

Ключ K — двоичный вектор длины L, он выбирается случайно равновероятно из множества возможных векторов длины L

Такая система шифрования будет *абсолютно стойкой по Шеннону*.

Возможный оппонент возразит: а что, если я переберу все ключи длиной 1? Ведь я же получу нужное сообщение.

Да, но оно будет далеко не единственным осмысленным сообщением среди полученных перебором ключей.

Рассмотрим пример.

Пример 8

Вот ключ Центра:

5 c 17 7f e 4e **37** d2 94 10 9 2e 22 57 ff c8 b b2 70 54 1f

А вот сообщение Центра:

Ш т и р л и ц - В ы Г е р о й ! !

98 e2 a8 eO aЬ a8 e6 20 2d 20 82 eb 20 83 a5 eO ac a9 21 21

Такая криптограмма попала в руки Мюллеру;

Y=T+K(mod 2)=

9d ee bf 9f a5 e6 d1 ft b9 30 8b **c5** 2 **d4** 5a 28 a5 1b 51 75

Дешифровальщики попробовали ключ 1:

5 c, 17 7f e 4e 37 d2 94 10 9 2e 22 75 f4 83 7 bb fc 54 1f

и получили текст:

98 e2 a8 eO ab a8 e6 20 2d 20 82 eb 20 a1 ae ab a2 aO ad 21

Ш т и р л и ц - В ы б о л в а н !

Теперь дешифровальщики попробовали ключ 2:

c ce 12 31 7 d 7d d2 la 9c 2f 6b ae f8 fe c8 46 bc bd 9a 1f

и получили текст:

91 20 ad ae a2 cb ac 20 a3 ae a4 ae ac 2c a4 eO e3 a7 ec ef

С н о в ы м г о д о м, д р у з ь я

Пробуя новые ключи, они будут получать все новые и новые фразы, пословицы, стихотворные строфы, гениальные откровения — словом, всевозможные тексты заданной длины.

Анализируя условия Шеннона, становится ясно, что строгое их выполнение удастся реализовать только для шифров с объемом ключа, ранным передаваемому тексту. В предыдущем разделе *мы* упоминали о том. что данная ситуация на практике мало применима в связи с техническими трудностями. Применяемые схемы шифрования с ограниченным объемом ключа, предназначенным для выработки псевдослучайных последователь-

ностей, не будут абсолютно стойкими по Шеннону (в чем легко убедиться самостоятельно). Коль скоро мы сделали такой вывод, то чем следует руководствоваться при выборе систем с ограниченной длиной ключа? (Несколько опережая события: речь о синтезе шифрующих преобразований пойдет в следующем разделе).

1. Для шифров с наложением псевдослучайной последовательности при помощи коммутативной групповой операции *недопустимо использование одного и того же ключа два и более раз для различных текстов* (см. пример 7 этого раздела).
2. Псевдослучайная последовательность должна иметь большой период.
3. Вероятности встречаемости знаков в последовательности должны быть примерно одинаковы.
4. Псевдослучайная последовательность должна быть такой, чтобы по произвольному отрезку трудно было прогнозировать ее следующий и предыдущий фрагменты.

В общем, вывод очевиден: если противник будет с приблизительно равной вероятностью получать шифртексты из всего множества возможных, то условия Шеннона будут выполняться с высокой степенью точности.

Условия Шеннона достаточно сложно оценить количественно, с другой стороны, для дешифрования используется, как правило, конкретный алгоритм, в огромной степени зависящий от шифрующего преобразования. В связи с этим логично оценивать стойкость в операциях, необходимых для решения той или иной задачи нахождения ключа или открытого текста для конкретного алгоритма.

Для этого необходимо:

* сформулировать алгоритм (при этом надо помнить, что построение статистических гипотез связано с ошибками - пропуском ключа либо нахождением ложных ключей, следовательно, надо задать вероятность нахождения истинного ключа и отбрасывать все те алгоритмы, для которых она слишком мала);
* сформулировать понятие элементарных операций этого алгоритма (см. далее);
* определить число этих операций для нахождения ключа;
* определить необходимую память для алгоритма (алгоритмы с очень большой памятью тоже следует отбраковать).

Вернемся к примеру 7. В качестве характеристического полинома выберем примитивный полином n-ой степени над полем GF(2). Ключом по-прежнему будет заполнение узла реализации линейных рекуррентных последовательностей (n бит). Определим этот ключ по n битам открытого и соответствующего ему шифрованного текста. Трудоемкость полного перебора T_1 равна:

$T_1 = 2^n - 1$

Однако нетрудно заметить, что выходная последовательность и ключ связаны системой линейных уравнений. Трудоемкость T_2 решения системы из n уравнения от n неизвестных методом Гаусса:

$T_2 = O(n^3)$

Легко видеть, что при больших n трудоемкость T_1 значительно превышает T_2, следовательно, стойкость надо оценивать величиной T_2. Кроме того, при больших n эффективны алгоритмы Штрассена для решения систем линейных уравнений с еще более низкой трудоемкостью.

Итак, стойкостью в операциях будем называть минимальную трудоемкость по всем известным для данного шифрующего преобразования алгоритмам нахождения ключа или открытого текста.

Синтез качественных шифров и блочные шифры

Зададимся теперь вопросом: как же строить надежные, стойкие шифры? Почти все сложное вокруг нас можно разделить на более простые элементы. По-видимому, это соображение можно применить и в криптографии. Например, будем применять простые преобразования несколько раз. Но хорошо ли это? Предположим, мы два раза применили для шифрования каждой буквы текста две разные подстановки одинаковой степени (или даже много разных подстановок). При этом мы воздействовали на каждую букву всего одной подстановкой, которая является произведением всех, использованных для шифрования (см. Приложение 3). С другой стороны, подстановки тоже бывают разные, и их всевозможные произведение в композиции дают лишь некоторые из множества всех возможных. Это, как мы уже выяснили, плохо, поскольку некоторые символы алфавита не используются в конкретном шифртексте (т.е. шифр не будет транзитивным).

С качественной точки зрения метод синтеза стойких шифров описал, как уже говорилось, Клод Шеннон. Этот метод подробно рассмотрен в главе 5.

Однако блочные шифры распространяют искажения, произошедшие в шифрованном тексте при его передаче и хранении, а также отличаются *достаточно* низкой *скоростью* шифрования за счет большого числа итераций.

В связи с этим часто применяются системы с использование псевдослучайных последовательностей (так называемые системы *поточного шифрования).* В предыдущем разделе мы упоминали, что использовать псевдослучайные последовательности с линейной зависимостью в них не рекомендуется. Поэтому к линейной рекурренте нужно применять усложнение — вносить нелинейность, применяя те же итеративные преобразования над блоками последовательности или над отдельными ееэлементами (например, суммируя элементы последовательности в арифметическом сумматоре).

При этом важно:

* сохранить период усложненной последовательности не меньше, чем период исходной;
* получить равновероятную последовательность после усложнения;
* получить сильную нелинейную зависимость от многих переменных ключа и от большого числа знаков усложняемой последовательности.

Цель, которую нам хотелось бы достичь: стойкость для метода с наименьшей трудоемкостью должна быть не намного хуже, чем стойкость метода полного перебора ключей.

Системы открытого распределения ключей и открытого шифрования

До сих пор мы рассматривали системы шифрования, основанные на ключе, который знают и отправитель, и получатель сообщения.

Чтобы отправитель и получатель узнали спой ключ, его нужно им *доставить,* причем так, чтобы сохранить его в тайне от всех других. Кроме того, желательно для каждого сообщения использовать *новый* ключ. Очень привлекательно со всех точек зрения вырабатывать ключ для каждого сообщения, однако попятно, что детерминированные алгоритмы для этого не годятся:

противник может прогнозировать ключи. Если мы будем вырабатывать ключи каждый раз случайно, то как оповестить об этом своего корреспондента?

Решение проблемы — в *односторонних* (однонаправленных) функциях. Функцию y = f(x) назовем односторонней, если вычисление y по x имеет малую *трудоемкость* (см. Приложение 3), а вычисление x по y — высокую.

К односторонним функциям относятся: возведение в степень, обратная функция — логарифмирование) в конечном (чаще простом) поле либо в другой алгебраической структуре (например, в группе точек эллиптической кривой), разложение чисел на множители (прямая задача — умножение), задачи кодирования-декодирования линейных кодов (типа кодов Рида — Малера или Рида — Соломона). Надо отметить, что полагаемая из общих соображений «односторонность» функции часто впоследствии не подтверждается. Это связано либо с появлением новых математических методов, либо с появлением мощной вычислительной техники. Кроме того, необратимость функции часто сильно зависит от параметров этой функции.

Итак, попробуем использовать односторонние функции при выработке ключей. Например, имеются две функции f(a, x) и g(a, x). Отправитель A сообщения вырабатывает случайное значение x, вычисляет t = f(a, x), получатель B тоже вырабатывает случайное значение y, вычисляет r = g(a, y). Далее A получает r, а B — t.

Предположим, что f(r, x)=f(g(a, y), x) совпадает с g(t, y) = g(f(a, x), y) (иначе у корреспондентов не получится одно и то же значение ключа).

Какие свойства необходимы для функций f и g, чтобы противник не смог определить ключ

K = g(t, y) = g(f(a, x), y) = f(r ,x) = f(g(a, y), x)?

Из самых простых соображений следует, что свойство должно быть таким, чтобы по значению f(a, x) и g(a, x) без знания x и y сложно было бы вычислить K.

С другой стороны важно, чтобы сами функции f и g легко вычислялись, а равенство g(f(a, x),y) = f(g(a, y), x) выполнялось бы для любых x и y и хотя бы для некоторых a.

Именно такой метод был предложен в работе Диффи и Хеллмана [2], он получил название *открытое распределение ключей* (ОРК).

Действительно, данный метод описывает лишь протокол выработки ключа для *произвольного* алгоритма шифровании.

Понятно, что функций с указанными условиями окажется значительно меньше, чем просто необратимых. Позднее было показано, что любая необратимая функция с такими условиями сводится к возведению в степень (понимаемую в обобщенном смысле как многократное выполнение мультипликативной операции).

Нельзя ли построить алгоритм шифрования так, чтобы отправитель сообщения имел возможность лишь зашифровать его, а расшифровать его смог только получатель? Чтобы это стало возможным, необходимо следующее:

* отправитель и получатель сообщения должны пользоваться разными ключами или разными алгоритмами (иначе при применении одного ключа отправитель сможет легко прочитать сообщение);
* ключи отправителя и получателя должны быть связаны однозначным соотношением;
* необходимо, чтоб но одному из ключей было крайне сложно определить другой ключ.

Оказывается, так построить схему шифрования возможно. Для этого также потребуются однонаправленные алгоритмы. Такой способ шифрования называется *открытым шифрованием* (ОШ), поскольку в данном случае однонаправленная функция существенно участвует н самом процессе шифрования, а не только в выработке ключа.

Рассмотрим систему ОРК на основе дискретной экспоненты и систему ОШ RSA (но имени авторов — Райвеста, Шамира и Адлемана).

Система Диффи — Хеллмана на основе дискретной экспоненты.

Задано простое число P, и некоторое число a < P.

Пусть имеется два корреспондента. Каждый из них вырабатывает случайное число x и y соответственно, а затем вычисляет

$r = a^x \pmod P$ — первый корреспондент;

$t = a^y \pmod P$ — второй корреспондент.

В данном случае функция $f(a, x) = a^* \pmod P$.

Затем корреспонденты обмениваются этими числами и вырабатывают общий ключ K.

$K = (a^x \pmod P)^y \pmod P = (a^y \pmod P)^x \pmod P$

Противнику могут быть доступны только числа t и r, и для нахождения x и y ему придется выполнить операцию *логарифмирования а простом поле* (см. Приложение 3). Следовательно, трудоемкостью данной операции и определяется стойкость этого варианта ОРК.

Для рассмотрения *системы RSA* введем несколько положений.

1. a сравнимо с b по модулю n (обозначается $a = b \pmod n$) означает, что число n делит разность a - b.
2. Числа a и b имеют наибольший общий делитель (НОД) d, если d делит как a, так и b и является максимальным среди всех делителей.

   Обозначение: НОД(a, b) = d
3. Числа a и b взаимно просты, если НОД(a, b) = 1
4. Число p называется простым, если оно делится только на 1 и на само себя (p).
5. Если p — простое и b < p, то НОД(b, p) = 1
6. Малая теорема Ферма. p — простое число, НОД(a, p) = 1

   $a^{p-1} = 1 \pmod p$

Описание системы ОШ RSA

Пусть даны числа p и q — простые.

Обозначим n = pq, k = (p - 1)(q - 1).

Выработаем число e случайным образом, e < n.

Вычислим d из соотношения $ed = 1 \pmod k$

Рассмотрим теперь числа c и d. Предположим, что мы знаем одно из них и соотношение, которым они связаны. Можно с легкостью вычислить второе число как обратный элемент мультипликативной группы с операцией умножения по модулю k (он существует и однозначно определяется по определению группы). Однако мы не знаем чисел p и q (чтобы найти их, необходимо разложить число n на множители; если p и q примерно равны и числа просто перебирать, проверяя их делимость, то придется сделать операций порядка корня квадратного из n).

Вот выражение для k:

(p - 1) x (q - 1) = pq - p - q + 1 = n + 1 - p - q

Следовательно, одно из чисел мы смело можем подарить кому-нибудь вместе с числом n и попросить его посылать нам сообщения следующим образом.

1. Сообщения представить как векторы длины L:

   $X = (x_1, ..., x_L)$

   $0 < x_i < k$,

2. Каждое x_i возвести в степень e по модулю n.
3. Прислать нам $Y = (x_1^e \pmod n, x_2^e \pmod n, ..., x_L^e \pmod n)$

Выделим $t_i = y$

Возведем полученное число t в степень d

второго числа из пары:

$R = t^d \pmod n = (x^e \pmod n)^d \pmod n$

Каков будет результат?

Что значит, что ed сравнимо с единицей по модулю k?

e d - 1 делится на (p - 1) x (q - 1) т.е. $ed = 1 + a(p - 1)(q - 1)$,

где a — целое число, операция умножения выполняется в кольце целых чисел.

Рассмотрим выражение:

$U = x^{1+a(p-1)(q-1)}$

Утверждается, что $U \pmod n = x$.

Действительно, по малой теореме Ферма:

$x^{p-1} = 1 \pmod p$

$x^{q-1} = 1 \pmod q$

Возведем первое сравнение в степень q - 1

$(x^{p-1})^{q-1} = 1^{q-1} = 1 \pmod p$

Возведем второе сравнение в степень p - 1

$(x^{q-1})^{p-1} = 1^{p-1} = 1 \pmod q$

Следовательно, $x^{(p-1)(q-1)}$ сравнимо с 1 и по модулю p

и по модулю q, а значит,

$x^{(p-1)(q-1)} = 1 \pmod{pq}$

перемножим а раз и получим требуемое равенство.

Тем самым мы доказали, что при возведении полученного сообщения в степень числа d но модулю n получается исходное сообщение x. Если же отправлять сообщения необходимо владельцу секретного ключа, то он будет пользоваться для шифрования открытым ключом своего корреспондента.

Описанная система называется системой RSA или несимметричной системой шифрования *Райвеста — Шамира — Адлемана*. Часто применяется термин «система с открытым ключом». Надо признать его не совсем удачным — мы всегда исходим из предположения секретности хотя бы части ключа. Более корректно именовать такие системы системами *несимметричного шифрования*. Числа в паре (e,d) называются: e — открытой частью ключа, d — секретной частью ключа.

Внимательно рассмотрев описанную систему шифрования, можно заметить еще одно ее ценное качество — она обеспечивает *аутентичность* передаваемой информации при условии знания секретной части ключа только отправителем сообщения и шифровании сообщение на секретной части ключа. При этом мы полагаем, что для проверки аутентичности используется открытая часть ключа. Действительно, автором такого сообщения может быть только владелец секретной части ключа. Конечно, при этом не достигается конфиденциальность передачи.

Остановимся более подробно на стойкости рассмотренных систем. Она в конечном итоге определяется трудоемкостью задачи, положенной в основу системы. Следовательно, речь идет о трудоемкости задач разложения на множители и дискретного логарифмирования.

Эксперты полагают, что на сегодняшний день системы типа Диффи — Хеллмана должны работать с числами длиной не меньше 1024 бит, а система RSA — не меньше 2048 бит.

Электронная цифровая подпись и хеш-функция

Рассматривая систему RSA в предыдущем параграфе, мы отметили, что зашифровывать на открытой части ключа может любой, а расшифровать на секретной — только получатель сообщения. На практике очень часто возникает потребность в обратном: зашифровать некоторую информацию (например, свое имя) на секретном ключе так, чтобы определенные лица могли расшифровать его (используя разосланный ранее открытый ключ).

Очень пенно то, что в системе RSA не придется ничего менять — нужно просто использовать для шифрования спою секретную часть ключа, тогда в силу симметричности ключей e и d (которая в свою очередь возникает из-за коммутативности операции

модульного умножения) процесс расшифрования получателем выполняется аналогично.

Кроме того, мы получим неожиданный эффект — зашифрованную на секретном ключе информацию почти невозможно осмысленно изменить, ведь для этого нужно сначала ее найти (т.е. дешифровать систему RSA), а уж потом исправить. Таким образом, получатели сообщения могут аутентифицировать отправителя, т.е. они получают гарантию того, что операцию зашифрования выполнил отправитель, а не кто-то еще. С криптографическим аналогом обыкновенной подписи под некоторой информацией более или менее ясно, А как же быть с содержимым послания?

Ведь некоторые документы противнику очень хочется исправить, оставив под ними оригинальную подпись (например, документы на выплату денег). Поэтому вполне логично в качестве открытого текста использовать информацию, в значительной степени зависящую от содержания послания, но значительно меньшего объема. Такую информацию, как правило, называют хеш-значением сообщения, а способ (алгоритм) получения хеш-значения — хеш-функцией (от английского hash).

Таким образом, мы можем указать основные компоненты *электронной цифровой подписи* (ЭЦП): *хеш-функция* (алгоритм «сворачивания» текста в более короткий) и некоторый криптографический алгоритм (шифрующее преобразование), которое воздействует на хеш-значение.

Последовательно рассмотрим эти компоненты. Начнем с хеширования данных. Предположим, что имеется текст F — последовательность знаков некоторого алфавита и алгоритм А, преобразующий F в текст M меньшей длины (в этом случае может использоваться другой алфавит).

Этот алгоритм таков, что при случайном равновероятном выборе двух текстов из множества возможных соответствующие им тексты M с высокой вероятностью различны. Тогда проверка целостности данных строится так:

* после пересылки или храпения текста F получен некоторый текст T;
* рассматриваем текст T (априорно полагая, что текст F был изменен);
* по известному алгоритму А строим K = A(T);
* сравниваем M, заранее вычисленное как M = A(F) с K.

При совладении считаем текст неизменным. Алгоритм А называют, как правило, хеш-функцией или реже, контрольной суммой, а число М — хеш-значением.

Чрезвычайно важно в данном случае выполнять следующие условия:

* очень трудоемкой задачей становится нахождение по известному числу M = A(F) другого текста G, не равного F, такого, что M = A(G) (другими словами, задача компенсации хеш-значения, или задача построения коллизий, очень трудоемка);
» число М должно быть недоступно для изменения.

Поясним смысл этих условий. Пусть злоумышленник (противник) изменил текст F. Тогда, вообще говоря, хеш-значение М для данного текста изменится. Если злоумышленнику доступно число М, то он может по известному алгоритму А вычислить новое хеш-значение для измененного им текста и заместить им исходное.

Именно с этой целью хеш-значение подвергается шифрованию, как правило, с использованием системы открытого шифрования.

С другой стороны, пусть само хеш-значение недоступно, тогда можно попытаться так построить измененный файл, чтобы его хеш-значение не изменилось [принципиально это возможно, поскольку отображение, задаваемое алгоритмом, А не биективно (неоднозначно)].

Выбор хорошего хеш-алгоритма, как и построение качественного шифра, — достаточно сложная задача.

Требования несовпадения с высокой вероятностью хеш-значений для разных файлов приходит в определенное противоречие с требованием трудоемкости компенсации их за счет подбора текста.

Поговорим теперь о хороших и плохих хеш-функциях. Представим хеш-алгоритм как конечный автомат без выхода A = {I, S, X(S, I) - S},

где I — множество входных слов, S — множество состояний,

X(S, I) => S — функция переходов автомата А — отображение декартова произведения S x 1 в S.

Процесс хеширования происходит так: текст представляется в алфавите входных символов, автомат устанавливается в некоторое детерминированное начальное состояние, текст подает-

ся на вход автомата до его исчерпания, хеш-значение — состояние автомата после подачи текста па вход.

С точки зрения несовпадения хеш-значений логично потребовать, чтобы отображение X при фиксированном s из S было бы биективным, т.е. X(s, I) представляло из себя подстановку.

Пусть при фиксированном состоянии s функция переходов X(s, 1) переводит два неодинаковых ц и i_2 в один и тот же символ t.

Это означает, что при хешировании текстов вида T, i_1 и T, i_2 хеш-значение будет одинаковым, что, вообще говоря, не очень хорошо.

С другой стороны, если найдется такой i, что $X(s_1, i) = X(s_2, i)$, где s_1 и s_2, естественно, неодинаковы, и если два разных текста T, и T_2 длины I - 1 имеют одно и то же *хеш-значение*

$h = X(s_0, T_1)=X(s_0, T_2)$, то и тексты T_1, i и T_2, i тоже будут иметь одинаковые хеш-значения. Следовательно, и отображение X(S, i) тоже должно быть подстановкой.

Наконец, должно быть достижимо покрытие всей группы подстановок. Т.е. рассматривая всевозможные произведения подстановок X(i, S) для разных i длиной p (которое назовем длиной покрытия), мы должны получить все возможные подстановки степени |S|. Причем число p должно быть не очень большим. Это будет означать, что автомат A может переходить из любого состояния в любое (в противном случае некоторые состояния окажутся принципиально недостижимыми).

Итак, мы видим, что хеш-преобразование в какой-то степени есть аналог шифрующего преобразования.

На самом деле хорошую хеш-функцию построить очень сложно, и ее надежностью в конечном итоге определяется надежность всей подписи. О российском стандарте хеш-функции мы расскажем в главе 5.

Ранее мы говорили, что после получения из текста хеш-значения его надо защитить от изменения, применяя систему шифрования (например, RSA, переставив при этом местами открытый и секретный ключи, т.е. зашифровав хеш-значение на секретном ключе и проверяя его на открытом). Теперь рассмотрим интересную систему электронной подписи, которая не может быть получена как простая модификация системы ОШ (предложена Эль Гамалем и 1985 году).

Пусть имеются простое число P и число a. Отправитель сообщения вырабатывает случайное число x, причем 1<x<P. Кроме того, для проверки своей подписи отправитель рассылает своим корреспондентам число T.

$T = a^x \pmod P$

Для сообщения M вырабатывается хеш-значение h, причем K h < P.

Кроме того, генерируется случайное число k (1 < k < P), удовлетворяющее дополнительному условию: числа P - 1 и k взаимно просты, а затем па его основе вычисляется число R.

$R = a^k \pmod P$

Вычислим $S = k^{-1} (h - xR) \pmod{P - 1}$

Для проверки корреспонденту высылается:

* содержательный текст M (сам документ);
* числа R и S.

Кроме того, проверяющий корреспондент имеет заранее:

* алгоритм хеширования (h = A(M));
* число T.

Чтобы проверить подпись, необходимо выполнить следующие операции.

1. Вычислить $Z = T^R * R^S \pmod P = (a)^{xR} (a)^{kS} \pmod P$, проводя аналогичные выкладки, что и для системы RSA, можно убедиться, что при неизменности чисел h, R, S число Z равно $a^h \pmod P$.
2. Вычислить по тексту M хеш-значение h.
3. Вычислить независимо $Y = a^h \pmod P$.
4. Проверить на равенство числа Y и Z.

В случае совпадения подлинность (принадлежность текста владельцу секретного ключа и его неизменность после того, как он поставил подпись) подтверждается.

Как определить надежность данной системы? Можно предположить, что в распоряжении противника оказались хеш-алгоритм, сам текст M, числа T, R, S (T он перехватил при рассылке).

Действия противника, вероятно, таковы:

* по T определить x, чтобы получить возможность самостоятельно изменить и заново подписать измененное сообщение — задача дискретного логарифмирования;

* исходя из S, попытаться определить x [h, R известны, а в уравнении присутствуют только операции *и + поля GF(p)]; эта задача с водится к поиску k, а оно неизвестно и присутствует только в виде R — снова дискретное логарифмирование;
* исходя из хеш-алгоритма, изменить текст так, чтобы для нового текста N, не равного M, хеш-значение осталось прежним.

Обозначим F(DLOG, X) — трудоемкость дискретного логарифмирования числа X в поле, F(=,A) — трудоемкость компенсации хеш-значения для алгоритма хеширования A.

Тогда трудоемкость подделки подписи Эль-Гамаля (стойкость)

T = min { F(DLOG,T), F(DLOG,R), F(=,A) }

Алгоритм **ЭЦП** ГОСТ Р 34.10-94

Прежде чем привести краткое описание параметров и алгоритмов ГОСТ Р 34.10-94, введем необходимые обозначения:

* $a = b \pmod{n}$ — целые числа a и b имеют одинаковые остатки при делении нацело на натуральное число n (говорят, что числа a и b сравнимы по модулю n);
* $a \pmod{n}$ — остаток от деления числа a на число n;
* $< s >_n$ — строка длиной n бит, содержащая двоичную запись числа s;
* $|A|$ — длина строки битов A;
* $A||B$ — конкатенация битовых строк A и B (строка длины $|A|+|B|$ бит, в которой левые $|A|$ символов образуют строку A, а правые $|B|$ символов образуют строку B).

Параметры ЭЦП ГОСТ Р 34.10-94

В качестве криптографической хэш-функции h регламентируется использовать хеш-функцию российского стандарта ГОСТ Р 34.11-94 [2]. Далее перечислены ее параметры:

* простое число p: $2^{509} < p < 2^{512}$ либо $2^{1020} < p < 2^{1024}$;
* простое число q: $2^{254} < q < 2^{256}$ и q является делителем числа $(p-1)$;
* целое число a: $1 < a < p-1$, при этом $a^q \pmod{p} = 1$.

ГОСТ Р 34.10-94 содержит точно описанную процедуру генерации указанных параметров,

Секретным ключом пользователя для формирования подписи выбирается число x: $0 < x < q$.

Открытый ключ пользователя для проверки подписи есть число y, которое вычисляется так:

$y = a^x \pmod p$.

Вычисление ЭЦП ГОСТ Р 34.10-94

Подпись для сообщения M вырабатывается следующим образом;

1. вычислить $h(M)$ — значение хэш-функции h от сообщения M; если $h(M) \pmod q = 0$, присвоить $h(M)$ значение $0^{255}1$ (битовая последовательность, состоящая из 255 нулей и одного единичного бита);
2. выработать целое число k: $0 < k < q$;
3. вычислить два значения:
   $r = a^k \pmod p$ и $r' = r \pmod q$;
4. если $r = 0$, перейти к этапу 2 и выработать другое значение числа k;
5. с использованием секретного ключа x пользователя, подписывающего сообщение, вычислить значение
   $s = (xr + kh(M)) \pmod q$;
6. если $s = 0$, перейти к этапу 2, в противном случае закончить работу алгоритма.

Подписью сообщения M является строка битов $< r' >_{256} SS < s >_{256}$

Проверка ЭЦП ГОСТ Р 34.10-94

Процедура проверки подписи $< r' >_{256} SS < s >_{256}$ сообщения M_1 использует открытый ключ y и состоит из следующих этапов:

1. проверить условия: $0 < 5 < q$ и $0 < r < q$; если хотя бы одно из этих условий не выполнено, то подпись считается недействительной;
2. вычислить $h(M_1)$ — значение хэш-функции А от полученного сообщения M_1;
3. если $h(M_1) \pmod q = 0$, присвоить $h(M_1)$ значение $0^{255}1$;
4. вычислить значение $v = (h(M_1))^{q-2} \pmod q$;
5. вычислить значения $z_1 = sv \pmod q$ и $z_2 = (q - r')v \pmod q$;
6. вычислить значение $u = (a^{z_1} \cdot y^{z_2} \pmod p) \pmod q$;
7. проверить условие $r = u$; если условие выполнено, подпись считается действительной, в противном случае — недействительной.

Алгоритм цифровой подписи **ГОСТ Р 34.10-2001**

Теперь перейдем к описанию параметров и алгоритмов ГОСТ Р 34.10-2001. Понятие эллиптической кривой определено в математическом приложении.

Параметры ЭЦП ГОСТ Р 34.10-2001

В качестве криптографической хэш-функции h регламентируется использование хеш-функции российского стандарта ГОСТ Р 34.11. Далее перечислены ее параметры:

* простое число p — модуль эллиптической кривой, удовлетворяющий неравенству $p > 2^{255}$;
* эллиптическая кривая E, задаваемая своими коэффициентами a, b е $GF(p)$;
* простое число q — порядок циклической подгруппы группы точек эллиптической кривой E, являющееся делителем порядка группы точек кривой E и удовлетворяющее условию $2^{254} < q < 2^{256}$;
* точка $P(x_p, y_p)$ № O эллиптической кривой E, удовлетворяющая равенству $qP = O$.

\Ia параметры схемы подписи налагаются следующие условия:

* должно быть выполнено условие $p^t \neq 1 \pmod q$ для всех целых $t = 1, 2, \ldots B$, где B удовлетворяет неравенству $B > 31$;
* количество точек кривой E не должно быть равно p;
* J-инвариант кривой E, вычисляемый по формуле

$$J(E) \equiv 1728 \frac{4a^3}{4a^3 + 27b^2} \pmod p$$

должен удовлетворять условиям $J(E) \neq 0$ и $J(E) \neq 1728$.

В качестве (секретного) ключа подписи выбирается целое число d, удовлетворяющее неравенству $0 < d < q$.

Открытым ключом проверки подписи является точка $Q(x_q, y_q)$ эллиптической кривой E, удовлетворяющая равенству $Q = dP$.

Формирование цифровой **подписи ГОСТ Р** 34.10-2001

Для получения цифровой подписи z под сообщением M с использованием ключа подписи d необходимо выполнить следующие действия:

1. вычислить хеш-код (хеш-значение) сообщениям: $\bar{h} = h(M)$;
2. вычислить целое число а. двоичным представлением которого является вектор $\bar{h}$, и определить e е а (mod q);

если $e = 0$, то определить $e = 1$;

3. сгенерировать случайное (псевдослучайное) целое число k, удовлетворяющее неравенству $0 < k < q$;
4. вычислить точку эллиптической кривой $C = kP$ и определить $r = x_c \pmod q$,

   где x_c - А" — координата точки С;

   если $r = 0$, то вернуться к этапу 3;
5. вычислить значение $s = (rd + ke) \pmod q$;

   если 5 = 0, то вернуться к шагу 3;
6. вычислить двоичные векторы $\bar{r}$ и $\bar{s}$, соответствующие r и s, и определить цифровую подпись $\zeta = (\bar{r}\ \bar{s})$ как конкатенацию двух двоичных векторов.

Проверка цифровой подписи **ГОСТ Р** 34.10-2001

Для проверки цифровой подписи z под полученным сообщением M необходимо выполнить следующие действия:

по полученной подписи z вычислить целые числа r и s;

если выполнены неравенства $0 < r < q$, $0 < 5 < q$, то перейти к следующему шагу; в противном случае подпись неверна;

вычислить хэш-код полученного сообщения M: $\bar{h} = h(M)$;

вычислить целое число а, двоичным представлением которого является вектор $\bar{h}$, и определить $e = a \pmod q$;

если $e = 0$, то определить $e = 1$;

вычислить значение v s $e^{-1} \pmod q$;

вычислить значения $z_1 = sv \pmod q$ и $z_2 = -rv \pmod q$;

вычислить точку эллиптической кривой $C = z_1P + z_2Q$ и определить $R = x_c \pmod q$;

если выполнено равенство $R = r$. то подпись принимается,

в противном случае подпись неверпа,

Глава 2

Криптографические интерфейсы

Общие положения

Подсистема безопасности *компьютерной системы* (КС) нуждается в реализации ряда общих функций, связанных с преобразованием содержания объектов КС (файлов, записей базы данных) либо с вычислением некоторых функций со специальными свойствами, которые существенно зависят от содержания объектов. К таким функциям относятся алгоритмы *контроля целостности* (КЦ) объектов КС, алгоритмы аутентификации или авторизации субъектов (или пользователей, которые управляют субъектами), алгоритмы поддержания конфиденциальности содержания объектов (функции шифрования).

Международные и национальные стандарты описывают ряд хороню изученных функций защитного характера, в частности алгоритмы хеширования MD2, MD5, SHA, ГОСТ Р 34.11-94, алгоритмы генерации и проверки ЭЦП RSA, DSS, ГОСТ Р 34.10-94 (ГОСТ Р34.11-02). Все эти алгоритмы имеют различную спецификацию вызовов (в частности, различную длину аргументов) и, естественно, логически несовместимы между собой.

При организации территориально распределенных защищенных компьютерных систсем можно использовать функционально одинаковые, но семантически разные функции логической защиты (например, вычислять функции КЦ с применением различных алгоритмов). Особенно актуальной эта проблема становится, когда дело касается стандартизированных и сертифицированных аппаратных модулей.

Задача использования внешних функций логической защиты важна не только для защитных субъектов, но и для произвольного субъекта, входящего в КС (например, применение субъекта вычисления ЭЦП для фиксации целостности и авторизации информации, передаваемой во внешнюю сеть). В связи с этим

взаимодействие с функциями логической защиты мы рассмотрим без привязки к конкретным функциям субъекта (исключения будут отмечаться особо).

Особенности внешнего разделяемого сервиса безопасности

Субъекты (программы, процессы) компьютерной системы, связанные с выполнением защитных функций, как говорилось ранее, могут использовать некоторое общее подмножество функций логического преобразования объектов (в частности, алгоритмы шифрования и контроля целостности объектов). При проектировании и реализации субъектов КС, как правило, реализуется исторически сложившийся подход относительно использования общего ресурса — применение разделяемых субъектов, выполняющих функции, общие для некоторого подмножества внешних по отношению к данному субъектов функций (динамически загружаемых библиотек для Microsoft Windows). Логично распространить данный подход на функции реализации защиты. Среди множества функций, относящихся к защитной компоненте КС, можно выделить три класса: ***функции, связанные с работой средств идентификации и авторизации пользователей, субъектов и объектов; функции, связанные с контролем неизменности объектов; функции, связанные с логическим преобразованием объектов КС и поддержанием функций конфиденциальности (криптографические функции).*** Иногда отдельно выделяют функции *генерации случайных последовательностей*, необходимых, в частности, для формирования индивидуальных аутентификационных признаков или ключей пользователей.

Далее мы будем говорить о защитных функциях КС. объединяя в данном термине все три класса функций и уточняя по мере необходимости те особенности при их реализации, которые следуют из их свойств,

Проблему проектирования субъектов, реализующих функции логической защиты, можно рассматривать в нескольких аспектах:

* ***оптимальную реализацию субъектов*** в рамках некоторого субъекта КС. к которому обращаются остальные субъекты за выполнением соответствующих функций (в данном случае речь идет о задаче оптимизации параметров «быстродействие — память» при реализации защитных функций);

» ***мобильность*** *субъектов,* использующих защитные функции при изменении внутреннего наполнения, которое реализует защитные функции субъекта (имеется в виду задача максимальной переносимости субъектов, использующих защитные функции на иные алгоритмы, например на иной алгоритм шифрования);

* ***корректное использование субъекта,*** реализующего защитные функции со стороны вызывающих его модулей.

Теперь определим, что такое разделяемая технология применения функций логической защиты. Это порядок использования средств логической защиты информации в КС, при котором:

* не требуется изменений в программном обеспечении (в содержании и составе субъектов КС) при изменении алгоритмов защиты;

* в КС однозначно выделяется *модуль реализации защитных функций* (МРЗФ).

Открытым интерфейсом (ОИ) МРЗФ назовем детальную спецификацию функций, реализованных в МРЗФ, позволяющую организовать исполнение этих функций из внешних субъектов.

«Открытость» интерфейса понимается как его полное описание для использования реализуемых в МРЗФ функций внешними субъектами. Для проектирования схем информационных потоков КС, в том числе имеющих отношение к обеспечению безопасности, такое описание играет ключевую роль.

В настоящее время существует довольно много криптографических интерфейсов. Но мы рассмотрим только базовые криптографические преобразования. Нас будут интересовать криптографические интерфейсы, в явном виде реализующие в себе основные криптографические операции: симметричное шифрование, хеширование, цифровую подпись и несимметричный обмен ключами. Один из них, получивший самое широкое распространение, — Microsoft CryptoAPI 2.0. Распространение CryptoAPI связано не только с его удобством, документированностью и другими объективными факторами. Важнейшим фактором является, то, что Microsoft самым активным образом интегрировала его в свои операционные системы и прикладные программы. Современные операционные системы Microsoft (Windows 2000, Windows XP, Windows ME) содержат множество криптографических подсистем различного назначения как прикладного уровня, так и уровня ядра, и, как уже отмечалось,

ключевую роль в реализации этих подсистем играет интерфейс CryptoAPI. Разумеется, в подсистемах уровня ядра базовые криптографические преобразования происходят непосредственно в драйверах, реализующих эти подсистемы. Функции CryptoAPI в таких случаях используются для вспомогательных операций на прикладном уровне.

Далее мы подробно рассмотрим работу именно интерфейса CryptoAPI, точнее, набора *базовых криптографических функций* (base cryptography functions), который называют также интерфейсом CryptoAPI 1.0.

Microsoft Cryptographic Application Programming Interface (CryptoAPI)

Интерфейс Microsoft CryptoAPI 2.0 содержит как функции, осуществляющие базовые криптографические преобразования, так и функции, реализующие преобразования более высокого уровня — работу с сертификатами X.509, работу с криптографическими сообщениями PKCS#7 и другие функции, поддерживающие так называемую *инфраструктуру открытых ключей* (Public Key Infrastructure). Как уже отмечалось, набор функций из CryptoAPI 2.0, реализующих базовые криптографические преобразования, называют также CryptoAPI 1.0. Именно этот набор функций и станет предметом нашего рассказа.

Дело в том, что функции высокого уровня, предназначенные для реализации криптографических преобразований, вызывают именно функции CryptoAPI 1.0. Таким образом, именно интерфейс CryptoAPI 1.0 является криптографическим ядром прикладного уровня современных операционных систем Microsoft. Если вы научитесь пользоваться всеми возможностями базовых криптографических функций, то не только более глубоко будете понимать работу всего интерфейса CryptoAPI 2.0, но и сами сможете создавать криптографические подсистемы любого уровня.

Начнем с обзора функций криптографического интерфейса CryptoAPI 1.0.

Обзор функции CryptoAPI 1.0

Прежде чем перейти к рассказу о криптоинтерфейсе, мы познакомим вас с его функциональными возможностями. Это позволит вам составить общее представление о построении CryptoAPI 1.0 и поможет при изучении материалов следующих

параграфов. В тех местах книги, где это необходимо, будут даны ссылки на Приложение 2 — полный справочник функций криптографического интерфейса.

В этом разделе мы приводим краткое описание 31 функции, составляющих интерфейс Microsoft CryptoAPI 1.0. Для удобства функции размещены в разных таблицах, в зависимости от назначения. В таблице 2-1 перечислены функции управления криптопровайдерами и контекстами криптопровайдеров (см. раздел «Принципы реализации интерфейса вызовов CryptoAPI 1.0»), в таблице 2-2 — функции, которые применяются приложениями для создания, конфигурирования и уничтожения криптографических ключей, а также для передачи ключей другим приложениям, в таблице 2-3 — функции, реализующие операции зашифрования и расшифрования с использованием симметричных ключей, в таблице 2-4 — функции, которые используются приложениями для вычисления значений хеш-функций, а также создания и проверки цифровой подписи сообщений.

Таблица 2-1. Функции управления криптопровайдерами и контекстами криптопровайдеров

Функция	Краткое описание
CryptAcquireContext	**Используется** для создания дескриптора определенного ключевого контейнера в рамках определенного криптопровайдера.
CryptContextAddRef	Увеличивает па единицу счетчик ссылок надескриптор **криптопровайдера.**
CryptEnumProviders	**Используется** для получения первого и следующего доступного криптопровайдера.
CryptEnumProviderTypes	**Используется** для получения первого и следующего типа доступных криптопровайдеров.
CryptGetDefaultProvider	Находит криптопровайдер, используемый по умолчанию, для указанного типа криптопровайдера.
CryptGetProvParam	Возвращает параметры криптопровайдера.
CryptReleaseContext	Используется для освобождения дескриптора криптопровайдера, созданного *CryptA cquire Context.*
CryptSetProvider и *CryptSetProviderEx*	Используется для задания имени и типа криптопровайдера, используемого по умолчанию.
CryptSetProvParam	Устанавливает параметры криптопровайдера.

Таблица 2-2. Функции создания, конфигурирования, уничтожения криптографических ключей, а также обмена ключами с другими приложениями

Функция	Краткое описание
CryptDeriveKey	Создает сессионные криптографические ключи из ключевого материала.
CryptDestroyKey	Освобождает дескриптор ключа.
CryptDuplicateKey	Делает точную копию ключа, его параметров и внутреннего состояния.
CryptExportKey	Используется для экспорта криптографических ключей и ключевых пар из ключевого контейнера криптопровайдера
CryptGenKey	Генерирует случайные сессионные ключи и ключевые пары.
CryptGenRandom	Вырабатывает случайную последовательность и сохраняет ее в буфер.
CryptGetKeyParam	Возвращает параметры ключа.
CryptGetUserKey	Возвращает дескриптор одной из постоянных ключевых пар.
CryptImportKey	Используется для импорта криптографического ключа из ключевого блоба в контейнер криптопровайдера.
CryptSetKeyParam	Устанавливает параметры ключа.

Таблица 2-3. Функции, реализующие операции зашифрования и расшифрования с использованием симметричных ключей

Функция	Краткое описание
CryptDecrypt	Используется для расшифрования данных.
CryptEncrypt	Используется для зашифрования данных.

Таблица 2-4. Функции, используемые для вычисления значений хеш-функций, а также создания и проверки цифровой подписи сообщений

Функция	Краткое описание
Crypt Create Hash	Используется для инициализации хеширования потока данных.
CryptDestroyHash	Уничтожает объект хеш-функции.
CryptDuplicateHash	Делает точную копию объекта хеш-функции.
Cryp tGetHashParam	Возвращает параметры объекта хеш-функции.
CryptHashData	Используется для добавления данных к объекту хеш-функции.
CryptHashSessionKey	Используется для добавления к объекту хеш-функции значения сессионного ключа,
CryptSetHashParam	Устанавливает параметры объекта хеш-функции.
CryptSignHash	Вычисляет значение ЭЦП от значения хеша, определенного дескриптором объекта хеширования.
Crypt Verify Signature	Осуществляет проверку подписи, соответствующей объекту хеширования.

Принципы реализации интерфейса вызовов CryptoAPI 1.0

Принципы реализации интерфейса вызовов функций продемонстрируем с помощью восстановленного кода функции ***CryptAcquireContextA*** (ANSI-вариант функции ***CrypiAcquireContext***).

Общая архитектура CryptoAPI 1.0 показана на рис. 2-1. Все функции интерфейса, описанные в разделе «Обзор функции CryptoAPI 1.0», содержатся в библиотеке advapi32.dll. За исключением нескольких сервисных функций (например, ***CryptSetProvider***), эти процедуры выполняют ряд вспомогательных операций и вызывают библиотеку, в которой непосредственно реализованы соответствующие криптографические преобразования. Такие библиотеки называются *криптопровайдерами* (Cryptographic Service Provider, CSP). Криптопровайдеры имеют стандартный набор функций, который состоит из 23 обязательных и 2 необязательных процедур (см. листнг 2-1).

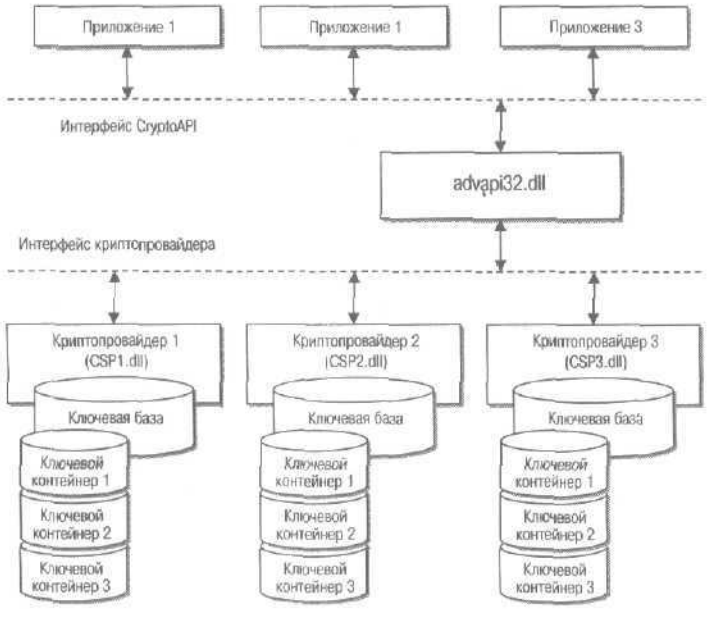

Рис. 2-1. Архитектура CryptoAPI 1.0

Программист, работающий с интерфейсом CryptoAPI 1.0, может получить всю необходимую информацию об определенном криптопровайдере, средствами функции ***CryptGetProvParam***.

Самое главное, что необходимо при этом знать, — наборы криптографических стандартов (шифрования, хеширования, цифровой подписи и несимметричного обмена ключами), которые реализуют криптопровайдеры, установленные в системе (см. раздел «Получение информации о криптопровайдерах, установленных в системе»).

Кроме различия в реализуемых стандартах, криптопровайдеры отличаются способом физической организации ключевой базы. С точки зрения программирования, способ физической организации ключевой базы значения не имеет. Однако он весьма важен с точки зрения эксплуатации и безопасности системы. Существующие криптопровайдеры Microsoft хранят свою ключевую базу па жестком диске (в Реестре или в файлах), а криптопровайдеры фирм Gemplus, Schlumberger и Infenion, которые включены в поставку Windows 2000, Windows XP и Windows ME, — на смарт-картах.

В то время как способы физической организации ключевой базы разных криптопровайдеров отличаются, логическая структура, которая определяется самим интерфейсом, для всех одинакова. Ключевая база представляется набором ключевых контейнеров. Каждый ключевой контейнер имеет имя, которое присваивается ему при создании, а затем используется для работы с ним. В ключевом контейнере сохраняется долговременная ключевая информация. В криптопровайдерах Microsoft долговременными являются ключевые пары цифровой подписи и несимметричного обмена ключами, которые также генерируются функциями CryptoAPI.

Теперь рассмотрим подробно, каким образом функций CryptoAPI из библиотеки advapi32.dll вызывают запрошенный пользователем криптопровайдер и какие операции происходят, прежде чем вызывается конкретный криптопровайдер.

Каждый криптопровайдер характеризуется собственным именем и типом. Его имя — просто строка, по которой система распознает криптопровайдер. Так, базовый криптопровайдер Microsoft назван «Microsoft Base Cryptographic Provider v1.0». Тип криптопровайдера — целое число (в нотации языка C - DWORD), значение которого идентифицирует набор поддерживаемых алгоритмов цифровой подписи и несимметричного обмена ключей. Уже упомянутый криптопровайдер «Microsoft Base Cryptographic Provider v1.0» имеет тип 1 (в файле WinCrypt.h определена константа PROV_RSA_FULL), этот тип криптопровайдеров реализует в качестве алгоритмов цифровой подписи и

несимметричного обмена ключей стандарт RSA. Другой базовый криптопровайдер Microsoft — «Microsoft Base DSS and Diffie-Hellman Cryptographic Provider» имеет тип 13 (в файле WinCrypt.h определена константа PROV_DSS_DH), этот тип криптопровайдеров реализует в качестве алгоритма цифровой подписи стандарт DSS, а для реализации ключевого обмена использует алгоритм Диффи — Хеллмана.

Разумеется, в системе могут существовать несколько различных криптопровайдеров одного и того же типа. Криптопровайдеры фирм Gemplus, Schlumberger и Infenion имеют тот же тип, что и криптопровайдер «Microsoft Base Cryptographic Provider v1.0» и реализуют цифровую подпись и ключевой обмен по алгоритму RSA.

Таким образом, для работы с набором криптопровайдеров в системном реестре содержится список имен всех криптопровайдеров. С каждым именем связан тип криптопровайдера и имя библиотеки, реализующей криптопровайдер.

Кроме того, в системе содержится информация о том, какой криптопровайдер применять, если пользователь при вызове не определил конкретное имя, а задал только тип необходимого ему криптопровайдера. Такие криптопровайдеры называются *криптопровайдерами, используемыми по умолчанию* (default cryptographic service provider) для данного типа. Для типа 1 криптопровайдером но умолчанию является «Microsoft Base Cryptographic Provider vl.O», а для типа 13 — «Microsoft Base DSS and Diffie-Hellman Cryptographic Provider». Для определения имени криптопровайдера по умолчанию можно воспользоваться функцией ***CryptGetDefaultProvider*** (см. Приложение 2), а для изменения этого параметра — функциями ***CryptSetProvider*** или ***CryptSetProviderEx*** (см. Приложение 2). Из описания этих функций следует, что криптопровайдер по умолчанию задается как для *текущего пользователя* (current user), так и для *системы в целом* (local computer). Вторые задаются для всех типов установленных криптопровайдеров при установке операционной системы и сохраняются в ключе реестра HKEY_LOCAL_MACHINE. В дальнейшем можно изменить эти параметры или назначить криптопровайдер по умолчанию только для текущего пользователя. Параметры для текущего пользователя имеют приоритет перед общесистемными и сохранятся в ключе реестра HKEY_CURRENT_USER. Разумеется, если параметры для текущего пользователя в явном виде отсутствуют, то применяются общесистемные.

Теперь разберем, каким образом пользователь начинает работу с криптопровайдером, который ему необходим, и каким образом система вызывает конкретную библиотеку, соответствующую выбранному криптопровайдеру. Для этого рассмотрим действие функции ***CryptAcquireContext*** (см. Приложение 2), С се вызова начинается любая программа, работающая с CryptoAPI. Именно в этой функции пользователь задает имя используемого криптопровайдера, его тип и имя рабочего ключевого контейнера. Функция ***CryptAcquireContext*** возвращает пользователю *дескриптор криптопровайдера* (handle of a cryptographic service provider), который в дальнейшем пользователь передает в процедуры для проведения всех необходимых операций с криптопровайдером.

Для того чтобы вы получили полное представление о структуре контекста криптопровайдера, последовательности его формирования и дальнейшей работе с ним, рассмотрим восстановленный код функции ***CryptAcquireContextA***. Полностью исходные тексты процедуры и пример ее использования находятся в файле Chapter2/CpContext/CpContext.c на Web-сайте издательства «Русская Редакция».

Листинг 2-1. Текст процедуры **OwnCryptAcquireContextA**

```
// См. комментарий 1
// Константы, определяющие версию используемой OG
// От этого зависит структура контекста криптопровайдера
// и его инициализация
// Определяется для Windows NT 4.0
#define _WIN32_WINNT  0x0400
// Определяется для Windows 2000 и старше
// (Windows 98 и старше)
//#define _WIN32_WINNT 0x0500
// Определение максимального значения типа криптопровайдера
ffdefine  MAX_PROV_TYPE      999
// Определение идентификатора контекста криптопровайдера
tfdefine PROVIDER_CONTEXT    0x11111111

. . .
// См. комментарий 2
// Определение прототипа функции CPAcquireContext
typedef BOOL (WINAPI *PFN_CP_ACQUIRE_CONTEXT)(
            HCRYPTPROV          *phProv,
            CHAR                *pszContainer,
            DWORD               dwFlags,
            PVTableProvStruc pVTable
            ):
```

(см. след. стр.)

```
// См. комментарий 3
// Определение структуры контекста криптопровайдера
// (структура зависит от опереционной системы)
typedef struct _PROV_CONTEXT_ {
   // Массив указателей на функции криптопровайдера
   FARPROC           CSPFuncPtr[24];
#if (_WIN32_WINNT >= 0x0500)
   // Массив указателей на необязательные функции
   // криптопровайдера(используется, начиная с Windows 2000
   // и Windows 98)
   FARPROC           CSPOptionalFuncPtr[3];
ffendif
   // Дескриптор библиотеки криптопровайдера
   HMODULE           hCSPDll;
   // Дескриптор ключевого контекста, который вернула
   // функция криптопровайдера CPAcquireContext
   HCRYPTPROV        hCSP;
   // Идентификатор контекста
   DWORD             ContextId;
   // Счетчик использования ключевого контекста
   DWORD             dwContextUseCount;
#if (_WIN32_WINNT >= 0x0500)
   .// Дополнительный счетчик использования ключевого
   // контекста, инкрементируется функцией CryptContextAddRef
   // (используется, начиная с Windows 2000 и Windows 98)
   DWORD             dwContextRefCount;
#else
   // Критическая секция ключевого контекста
   CRITICAL_SECTION CSPLock;
ttendif
} PROV_CONTEXT, *PPROV_CONTEXT;

// См. комментарий 4
// Определение массива имен функций криптопровайдера
LPTSTR FunctionNames[]={
         "CPAcquireContext",    "CPReleaseContext",
         "CPGenKey",            "CPDeriveKey",
         "CPDestroyKey",        "CPSetKeyParam",
         "CPGetKeyParam",       "CPExportKey",
         "CPImportKey",         "CPEncrypt",
         "CPDecrypt",           "CPCreateHash",
         "CPHashData",          "CPHashSessionKey",
         "CPDestroyHash",       "CPSignHash",
         "CPVerifySignature",   "CPGenRandom",
         "CPGetUserKey",        "CPSetProvParam",
         "CPGetProvParam",      "CPSetHashParam",
```

(см. след. стр.)

```
          "CPGetHashParam",      NULL};
// Определение массива имен необязательных функций
// криптопровайдера
LPTSTR OptionalFunctionNames[]={
          "CPDuplicateKey", "CPDuplicateHash", NULL};
// См. комментарий 5
// Имя ключа реестра (базовый ключ HKEY_CURRENT_USER),
// в котором хранятся имена криптопровайдеров, используемых
// по умолчанию, для текущего пользователя
LPTSTR szUserType=
          "SOFTWARE\\Microsoft\\Cryptography
          \\Providers\\Type";
// Имя ключа реестра (базовый ключ HKEY_LOCAL_MACHINE),
// в котором хранятся имена криптопровайдеров, используемых
// по умолчанию, для системы
LPTSTR szMachineType=
          "SOFTWARE\\Microsoft\\Cryptography\\Defaults
             \\Provider Types\\Type";
// Имя ключа реестра (базовый ключ HKEY_LOCAL_MACHINE),
// в котором хранятся имена всех криптопровайдеров,
// установленных на компьютере
LPTSTR szProvider=
          "SOFTWARE\\Microsoft\\Cryptography\\Defaults
          \\Provider\\";
PBYTE pbContextInfo=NULL;
DWORD cbContextInfo=0;
// См. комментарий 6
// Определение реконструированной функции CryptAcquireContextA
BOOL
WINAPI
OwnCryptAcquireContextA(
      HCRYPTPROV    *phProv,
      LPCSTR        pszContainer,
      LPCSTR        pszProvider,
      DWORD         dwProvType,
      DWORD         dwFlags
       )
{
   BOOL             Result=TRUE;
   LPTSTR           pszRegStr=NULL,  pszDllName=NULL,
                    pszProvName=NULL;
   LPWSTR           pwszDllName=NULL;
   DWORD            dwRegError=ERROR_SUCCESS;
   DWORD            dwCryptError=ERROR_SUCCESS;
   HKEY             hkResult=0;
   DWORD            dwRegType,  dwGueryProvType;
```

(см. след. стр.)

```
    DWORD              cbData=0;
    HMODULE              hProvDll=0;
    PPROV_CONTEXT   pProvContext=NULL;
    DWORD            dwFuncCount=0;
    VTableProvStruc VTable;
    // Если значение типа криптопровайдера не попадает
    // EI заданные границы, то возвращаем ошибку
    if (dwProvType==0 || dwProvType > MAX_PROV_TYPE) {
       Result=FALSE;
       dwCryptError=NTE_BAD_PROV_TYPE;
       goto ReleaseResource;
    }
    if (pszProvider == NULL) {
// См. комментарий 7
       // Если имя криптопровайдера не определено, то
       // пытаемся определить криптопровайдер, используемый
       ,// по умолчанию для указанного типа
       pszRegStr=LocalAlloc(LMEM_ZEROINIT,
          lstrlen(szUserType)+4);
       if (!pszRegStr) {
          Result=FALSE;
          dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
          goto  ReleaseResource;
       }
       // Формируем строку реестра для типа криптопровайдера
       // текущего пользователя
       lstrcpy(pszRegStr,szUserType);
       sprintf(pszRegStr+lstrlen(szUserType),
          "%03d",dwProvType);
       // Пытаемся открыть ключ реестра для типа
       // криптопровадера текущего пользователя
       dwRegError=RegOpenKeyEx(
                HKEY_CURRENT_USER,
                pszRegStr,
                0,
                KEY_READ,
                &hkResult);
       if (dwRegError != ERROR_SUCCESS) {
II См. комментарий 8
          // Если ключ отсутствует то обращаемся
          // к системному ключу реестра
          LocalFree(pszRegStr);
          pszRegStr=LocalAlloc(LMEM_ZEROINIT,
             lstrlen(szMachineType)+4);
          if (!pszRegStr) {
             Result=FALSE;
```

(см. след. стр.)

```
            dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
            goto ReleaseResource;
        }
        // Формируем строку реестра для типа
        // криптопровайдера, определенного в системе
        lstrcpy(pszRegStr,szMachineType);
        sprintf(pszRegStr+lstrlen(szMachineType),
            "%03d",dwProvType);
        // Пытаемся открыть ключ реестра для типа
        // криптопровадера, определенного в системе
        dwRegError=RegOpenKeyEx(
              HKEY_LOCAL_MACHINE,
              pszRegStr,
              0,
              KEY_READ,
              &hkResult);
        if (dwRegError != ERROR_SUCCESS) {
            Result=FALSE;
            dwCryptError=NTE_PROV_TYPE_NOT_DEF;
            goto ReleaseResource;
        }
    }
// См. комментарий 9
    // По открытому ключу определяем имя
    // криптопровайдера, используемого по умолчанию
    dwRegError=RegQueryValueEx(
              hkResult,
              "Name",
              NULL,
              &dwRegType,
              NULL,
              &cbData);
    if (dwRegError != ERROR_SUCCESS) {
        Result=FALSE;
        dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
        goto ReleaseResource;
    }
    // Выделяем память под строку с именем
    // криптопровайдера
    pszProvName=LocalAlloc(LMEM_ZEROINIT,cbData);
    if (!pszProvName) {
        Result=FALSE;
        dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
        goto ReleaseResource;
    }
    // Считываем из реестра строку с именем
```

(см. след. стр.)

```
        // криптопровайдера
        dwRegError=RegQueryValueEx(
                  hkResult,
                  "Name",
                  NULL,
                  &dwRegType,
                  pszProvName,
                  &cbData);
        if (dwRegError != ERROR_SUCCESS) {
           Result=FALSE;
           dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
           goto ReleaseResource;
        }
        // Освобождаем выделенные ресурсы
        LocalFree(pszRegStr); pszRegStr=NULL;
        RegCloseKey(hkResult); hkResult=0;
     }
     else {
        // Если имя криптопровайдера определено явно,
        // то копируем его в рабочий буфер
        pszProvName=LocalAlloc(LMEM_ZEROINIT,
           lstrlen(pszProvider)+1);
        if (!pszProvName) {
           Result=FALSE;
           dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
           goto ReleaseResource;
        }
        lstrcpy(pszProvName,pszProvider);
     }
// См. комментарий 10
     // Формируем строку для доступа к записи о выбранном
     // криптопровайдере
     pszRegStr=LocalAlloc(
                    LMEM_ZEROINIT,

lstrlen(szProvider)+lstrlen(pszProvName)+1);
     if (!pszRegStr) {
        Result=FALSE;
        dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
        goto ReleaseResource;
     }
     lstrcpy(pszRegStr,szProvider);
     lstrcpy(pszRegStr+lstrlen(szProvider),pszProvName);
     // Открываем ключ реестра с информацией о
     // выбранном криптопровайдере
     dwRegError=RegOpenKeyEx(
```

(см. след. стр.)

```
            HKEY_LOCAL_MACHINE,
            pszRegStr,
            0,
            KEY_READ,
            &hkResult);
    if (dwRegError != ERROR_SUCCESS) {
        Result=FALSE;
        dwCryptError=NTE_KEYSET_NOT_DEF;
        goto ReleaseResource;
    }
    LocalFree(pszRegStr); pszRegStr=NULL;
    // По открытому ключу считываем тип криптопровайдера
    cbData=sizeof(DWORD);
    dwRegError=RegQueryValueEx(
            hkResult,
            "Type",
            NULL,
            &dwRegType,
            (PBYTE)&dwQueryProvType,
            &cbData);
    if (dwRegError != ERROR_SUCCESS) {
        Result=FALSE;
        dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
        goto ReleaseResource;
    }
    // Если считанный тип не совпадает с заданным при
    // вызове функции, то возвращаем ошибку
    if (dwQueryProvType != dwProvType) {
        Result=FALSE;
        dwCryptError=NTE_PROV_TYPE_NO_MATCH;
        goto ReleaseResource;
    }
// См. комментарий 11
    // По открытому ключу считываем имя библиотеки
    // криптопровайдера
    dwRegError=RegQueryValueEx(
            hkResult,
            "Image Path",
            NULL,
            &dwRegType,
            NULL,
            &cbData);
    if (dwRegError != ERROR_SUCCESS) <
        Result=FALSE;
        dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
        goto ReleaseResource;
```

(см. след. стр.)

```
    }
    pszRegStr=LocalAlloc(LMEM_ZEROINIT,cbData);
    if (!pszRegStr) {
       Result=FALSE;
       dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
       goto ReleaseResource;
    }
    dwRegError=RegQueryValueEx(
              hkResult,
              "Image Path",
              NULL,
              &dwRegType,
              pszRegStr,
              &cbData);
    if (dwRegError != ERROR_SUCCESS) {
       Result=FALSE;
       dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
       goto ReleaseResource;
    }
    // По считанному имени определяем полное имя библиотеки
    cbData=ExpandEnvironmentStrings(pszRegStr,NULL,0);
    pszDllName=LocalAlloc(LMEM_ZEROINIT,cbData);
    if (!pszDllName) {
       Result=FALSE;
       dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
       goto ReleaseResource;
    }
    cbData=ExpandEnvironmentStrings(pszRegStr,
       pszDllName,cbData);
    // Проверяем наличие данной библиотеки в кэше
    if (!CSPInCacheCheck(pszDllName,&hProvDll)) {
       // Если библиотека отсутствует, то загружаем ее...
       hProvDll=LoadLibrary(pszDllName);
       if (!hProvDll) {
          Result=FALSE;
          dwCryptError=NTE_PROVIDER_DLL_FAIL;
          goto ReleaseResource;
       }
       // Проверяем цифровую подпись библиотеки
       // криптопровайдера
       . . .
       // ... и помещаем дескриптор в кэш
       AddHandleToCSPCache(hProvDll);
    }
    // Освобождаем выделенные ресурсы
LocalFree(pszRegStr); pszRegStr=NULL;
```

(см. след. стр.)

```
LocalFree(pszDllName); pszDllName=NULL;
RegCloseKey(hkResult); hkResult=0;
// См. комментарий 12
   // Выделяем память под структуру контекста
   // криптопровайдера
   (PBYTE)pProvContext=LocalAlloc(LMEM_ZEROINIT,
      sizeof(PROV_CONTEXT));
   if (!pProvContext) {
      Result=FALSE;
      dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
      goto ReleaseResource;
   }
   // Сохраняем в контексте адреса обязательных
   // функций криптопровадера
   for (dwFuncCount=0; ;dwFuncCount++) {
      pProvContext->CSPFuncPtr[dwFuncCount]=

GetProcAddress(hProvDll,FunctionNames[dwFuncCount]);
      if ( !FunctionNames[dwFuncCount]) break;
      if (!pProvContext->CSPFuncPtr[dwFuncCount]) {
         Result=FALSE;
         dwCryptError=NTE_PROVIDER_DLL_FAIL;
         goto ReleaseResource;
      }
   }
// См. комментарий 13
#if (_WIN32_WINNT >= 0x0500)
   // Сохраняем в контексте адреса необязательных
   // функций криптопровайдера
   // (начиная с Windows 2000 и Windows 98)
   for (dwFuncCount=0; ;dwFuncCount++) {
      (FARPROC)pProvContext->CSPOptionalFuncPtr[dwFuncCount]=
            GetProcAddress(hProvDll,
               OptionalFunctionNames[dwFuncCount]);
      if (!OptionalFunctionNames[dwFuncCount]) break;
   >
#endif
   // Сохраняем в контексте дескриптор библиотеки
   // криптопровайдера
   pProvContext->hCSPDll=hProvDll;
// См. комментарий 14
   // Заполняем структуру VTableProvStruc версии 3
   VTable.Version=3;
   VTable.FuncVerifyImage=FuncVerifyImage;
   VTable.FuncReturnhWnd=FuncReturnhWnd;
   VTable.dwProvType=dwProvType;
```

(см. след. стр.)

```
   VTable.pbContextInfo=NULL;
   VTable.cbContextInfo=0;
   VTable.pszProvName=pszProvName;
// См. комментарий 15
   // Вызываем функцию CPAcquireContext из
   // выбранного криптопровайдера
   if  (!(PFN_CP_ACQUIRE_CONTEXT)(pProvContext->CSPFuncPtr[0])(
            &pProvContext->hCSP,
            (CHAR*)pszContainer,
            dwFlags,
            &VTable)) {
      Result=FALSE;
      dwCryptError=GetLastError();
      goto ReleaseResource;
   }
   // Если установлен флаг CRYPT_DELETEKEYSET,
   // то освобождаем ресурсы и возвращаемся из функции
   if (dwFlags == CRYPT_DELETEKEYSET) {
      FreeLibrary(hProvDll);
      LocalFree(pProvContext);
      goto ReleaseResource;
   }
   // Сохраняем в контексте идентификатор
   pProvContext->ContextId=PROVIDER_CONTEXT;
// См. комментарий 16
ttif (_WIN32_WINNT >= 0x0500)
   // Инициализируем в контексте значение счетчика
   // и дополнительного счетчика ссылок
   // (начиная с Windows 2000 и Windows 98)
   pProvContext->dwContextUseCount=1;
   pProvContext->dwContextRefCount=1;
«else
   // Инициализируем в контексте счетчик ссылок
   pProvContext->dwContextUseCount=0;
   // Инициализируем в контексте критическую секцию
   InitializeCriticalSection(&pProvContext->CSPLock);
#endif
   // Возвращаем указатель на контекст
   *phProv=(HCRYPTPROV)pProvContext;
ReleaseResource:
   // Освобождаем выделенные ресурсы
   . . .
   return  Result;
}
```

Теперь *прокомментируем* показанный фрагмент.

1. Значение константы _WIN32_WINNT определяет версию операционной системы, для которой компилируется программа. От значения константы зависит структура контекста криптопровайдера и некоторые шаги по его формированию.
2. Определяем тип для функции ***CPAcquireContext***. Функция ***CPAcquireContext*** — одна из 23 обязательных функций, которые должен экспортировать криптопровайдер.
3. Определяем структуру контекста криптопровайдера. Указатель на эту структуру возвращается функцией ***CryptAcquireContext*** через параметр *phProv* и используется приложением в качестве дескриптора криптопровайдера. Инициализация и использование переменных структуры будут рассмотрены в соответствующих местах программы. Как уже отмечалось, состав структуры контекста зависит от версии используемой операционной системы,
4. Определяем дна массива имен функций. Массив *FunctionNames* содержит имена 23 обязательных функций, которые должен экспортировать криптопровайдер. Массив *OptionalFunctionNames* содержит имена 2 необязательных функций. Значения терминов «обязательный» и «необязательный» будут объяснены далее.
5. Определяем базовые строки для рабочих ключей реестра.

   Строка *szUserType* содержит строку, на базе которой в программе формируется запрос к ключу реестра HKEY_CURRENT_USER\SOFTWARE\Microsoft\Cryptography\Providers\Type NNN. Ключ содержит информацию о криптопровайдере, используемом по умолчанию текущим пользователем. NNN — три десятичные цифры, определяющие тип криптопровайдера. Например, информация о криптопровайдере, используемом текущим пользователем, для типа PROV_RSA_FULL (имеет значение 1, см. файл WinCrypt.h), содержится в ключе HKEY_CURRENT_USER\SOFTWARE\Microsoft\Cryptography\Providers\Type 001.

   Строка *szMachineType* содержит строку, на базе которой в программе формируется запрос к ключу реестра HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider Types\ Type NNN. Ключ содержит информацию о криптопровайдере, который по умолчанию используется системой. NNN — три десятичные цифры, определя-

ющие тип криптопровайдера. Информация о криптопровайдере, используемым системой для типа PROV_DSS_DH (имеет значение 13, см. файл WinCrypt.h), содержится в ключе HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider Types\ Type 013.

Строка *szProvider* содержит строку, на базе которой в программе формируется запрос к ключу реестра HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider\ Provider Name. Ключ содержит информацию о криптопровайдере с именем «Provider Name». Например, информацию о криптопровайдере «Microsoft Base Cryptographic Provider v 1.0» содержит ключ реестра HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider\Microsoft Base Cryptographic Provider v1.0.

6. Определяем функцию ***OwnCryptAcquireContextA*** Она реконструирована из функции ***CryptAcquireContextA*** Windows 2000. Разумеется, реконструкция не является абсолютно точной. Имена функций и глобальных переменных соответствуют отладочной информации для данной версии операционной системы. В функцию также включены особенности, характерные для Windows NT 4.0. Как уже отмечалось, при компиляции версия операционной системы определятся константой _WIN32_WINNT. Поскольку наша задача — показать структуру контекста криптопровайдера и последовательность его формирования, а не полностью восстанавливать функцию, то процедуры ***CSPInCacheCheck***, ***AddHandleToCSPCache***, ***CheckSignatureInFile***, ***FuncVerifyImage*** определены пустыми. Описание функций ***CheckSignatureInFile***, ***FuncVerifyImage*** смотрите в главе 3.
7. Если параметр *pszProvider* нулевой, то приложение запрашивает криптопровайдер, используемый по умолчанию, для типа, определенного параметром *dwProvType*. Поэтому на базе строки *szVserType* в переменной *pszRegStr* формируется имя ключа реестра (см. комментарий 5). Через вызов функции ***RegOpenKeyEx*** с параметрами HKEY_CURRENT_USER и *pszRegStr* делается попытка открыть ключ реестра с информацией о криптопровайдере, который назначен по умолчанию, для текущего пользователя.
8. Запрашиваемый ключ реестра не найден, следовательно, явные параметры, заданные для текущего пользователя, отсутствуют, и вызов будет обработан с использованием

общесистемных параметров. На базе строки *szMachineType*, в переменной *pszRegStr* формируется имя ключа реестра (см. комментарий 5). Через вызов функции ***RegOpenKeyExc*** параметрами HKEY_LOCAL_MACHINE и *pszRegStr* открываем ключ реестра с информацией о криптопровайдере по умолчанию для системы в целом.

9. Независимо от того, какой ключ реестра открыт, он содержит два параметра. Первый параметр (имя «Name», тип REG_SZ) — строка с именем криптопровайдера. Второй параметр (имя «TypeName», тип REG_SZ) — строка с названием типа криптопровайдера. Нас интересует только первый параметр, он и является искомым именем криптопровайдера по умолчанию. Сохраняем имя по адресу *pszProvName*.

10. На базе строки *szProvider* в переменной *pszRegStr* формируется имя ключа реестра (см. комментарий 5). Через вызол функции *RegOpenKeyEx* с параметрами HKEY_LOCAL_MACHINE и *pszRegStr* открываем ключ реестра с информацией о выбранном криптопровайдере.

11. Набор параметров, которые содержит открытый ключ, зависит от используемой операционной системы. Но два наиболее важных параметра присутствуют всегда. Первый параметр (имя «Image Path», тип REG_SZ) — строка с именем библиотеки, в которой реализован данный криптопровайдер. Второй параметр (имя «Type», тип REG_DWORD) — значение типа криптопровайдера. Последнее сохраняем в переменной *dwQueryProvType* этот параметр необходим для сравнения с запрошенным типом криптопровайдера *dwProvType*. Имя библиотеки криптопровайдера сохраняем по адресу *pszRegStr*. На основании этого параметра производим вызов функции ***LoadLibrary*** и загружаем библиотеку криптопровайдера в память процесса. В файле CpContext.c содержится также проверка подписи библиотеки, этот вопрос будет рассмотрен в следующей главе, посвященной разработке криптопровайдеров.

12. Получив необходимую информацию о криптопровайдерс, начинаем формировать структуру контекста криптопровайдера. На основании массива имен обязательных функций криптопровайдера *FunctionNames* формируем массив указателей на обязательные функции криптопровайдера *pProvContext->CSPFuncPtr*. Обратите внимание, что если какая

либо из функций отсутствует в криптопровайдере, то инициализация прерывается и возвращается ошибка. Именно поэтому данный набор функций является обязательным.

13. На основании массива имен необязательных функций криптопровайдера *OptionalFunctionNames* формируем массив указателей на необязательные функции криптопровайдера *pProvContext->CSPOptionalFuncPtr*. Этот член структуры и его инициализация присущи только *операционным системам* (ОС), начиная с Windows 2000. В отличие от обработки обязательных функций, в отсутствии необязательной инициализация не прерывается.

14. Инициализируем структуру *VTable*. Инициализация производится в соответствии с версией 3 структуры *VtableProvStruc*. Необходимо отметить, что версия 3 используется, только начиная с Windows 2000. Более подробно параметр рассмотрен *VTable* в главе 3.

15. Вызываем функцию ***CPAcquireContext*** из запрошенного криптопровайдера. В параметре *pProvContext->hCSP* сохраняем дескриптор, который возвращается криптопровайдером. Сущность этого дескриптора определяется конкретным криптопровайдером.

    Если приложение установило флаг CRYPT_DELETEKEYSET, то работа функции на этом заканчивается. В противном случае инициализируем параметр *pProvContext->ContextId* значением 0x11111111. Оно показывает, что этот контекст является контекстом криптопровайдера. При передаче указателя на контекст криптопровайдера в какую либо функцию CryptoAPI проверяется значение данного поля, и если оно не равно 0x11111111, то возвращается ошибка. Аналогичные поля имеются также в контексте объекта ключа, и объекта хеш-функции, отличаются только идентификаторы. Для объекта ключа он равен 0x22222222, а для объекта хеш-функции 0x33333333.

16. Инициализируются параметры контекста криптопровайдера, отвечающие за счетчика обращений. В Windows NT 4.0 и Windows 2000 реализованные механизмы отличаются, соответственно отличаются и параметры.

    В Windows NT 4.0 при изменении счетчика обращений *pProvContext->dwContextUseCount* вызывается функция ***EnterCriticalSection*** с параметром *pProvContext->CSPLock*. После изменения критическая секция освобождается,

В Windows 2000 для изменения счетчика обращений используются функции ***InterlockedIncrement*** и ***InterlockedDecrement***, и критической секции не требуется. Также в Windows 2000 появилась возможность увеличивать счетчик обращений без вызова функции ***CryptAcquireContext.*** Для этого используется функция ***CryptContextAddRef*** (см Приложение 2). Однако ***CryptContextAddRef*** увеличивает не основной счетчик *pProvContext->dwContextUseCount* а дополнительный *pProvContext->dwContextRefCount*.

В функции ***CryptReleaseContext*** **(см.** Приложение 2) счетчики обращений проверяются, и если они находятся не в исходном состоянии, возвращается ошибка ERROR_BUSY (контекст занят). Кроме того, в Windows 2000 функция ***CryptReleaseContext*** декрементирует счетчик *pProvContext->dwContextRefCount* Поэтому, чтобы освободить контекст, необходимо вызвать функцию ***CryptRelease Context*** столько же раз, сколько вызывается функция ***CryptContextAddRef***.

Итак, мы рассмотрели структуру и последовательность формирования контекста крипто провайдера. Как видно из программы, дескриптор криптопровайдера, который возвращает функция ***CryptAcquireContext*** является указателем па контекст криптопровайдера. Когда мы говорим «дескриптор криптопровайдера» или «контекст криптопровайдера», это не совсем точно. С помощью функции ***CryptAcquireContext*** создастся контекст не всего криптопровайдера, а определенного ключевого **контейнера криптопровайдера.** Конечно, используя полученный дескриптор, мы можем получать параметры криптопровайдера и устанавливать параметры, которые будут распространяться на все ключевые контексты. Но, по сути, функция ***CryptAcquireContext*** записывает ключевой контейнер в память и возвращает приложению дескриптор, позволяющий работать с этим образом ключевого контейнера,

Когда вы вызываете функцию ***CryptReleaseContext*** то разрушается только образ ключевого контейнера, сам ключевой контейнер остается в физическом хранилище **криптопровайдера (см.** рис. 2-1). Для удаления ключевого контейнера из физического хранил ища **необходимо использовать** функцию ***CryptAcquireContext*** *с* флагом CRYPT_DELETEKEYSET. Исключение составляет ключевой контекст, созданный с флагом CRYPT_VERIFYCONTEXT. Он не является отражением ключевого контейнера. В таком контексте нельзя создавать долго-

временные ключевые пары подписи и обмена. Он предназначен для проверки подписи сообщений, поскольку в этом случае нет необходимости доступа к секретному ключу пары. Кроме того, если вы работаете только с симметричными ключами, то также нет необходимости создавать ключевые контейнеры (см. раздел «Использование CryptoAPI 1.0 для реализации схемы симметричного шифрования»).

При создании объектов ключей (функция ***CryptGenKey***) и хеш-функций (функция ***CryptCreateHash***) создается контекст с аналогичной структурой. Однако аналогия не полная, есть несколько отличий. В контексте ключа или хеш-функции сохраняются указатели только на те функции, в работе которых принимают участие дескрипторы этих объектов. Отсутствует дополнительный счетчик обращений. Кроме того, как уже отмечалось, отличаются идентификаторы объектов (см. комментарий 15).

Теперь, мы надеемся, вы получили представление о принципах вызовов функций CryptoAPI, а также о том, что такое ключевой контейнер, контекст криптопровайдера (или ключевой контекст) и дескриптор криптопровайдера (или дескриптор ключевого контекста).

Получение информации о криптопровайдерах, установленных в системе

Теперь займемся непосредственно программированием с использованием интерфейса CryptoAPI 1.0.

Прежде всего необходима программа, которая получает полную информацию обо всех криптопровайдерах, установленных в системе. Для этого воспользуемся функциями ***CryptEnumProviders*** и ***CryptGetProvParam*** (см. Приложение 2). Функция ***CryptEnumProviders*** позволяет последовательно перечислить все криптопровайдеры, установленные на рабочем месте, а функция ***CryptGetProvParam*** — получить необходимую информацию о криптопровайдере. Исходный текст программы находится в файле Chapter2/EnumCSP/EnumCSP.c.

Листинг 2-2. Текст программы EnumCSP

```
// См. комментарий 1
// Определение прототипа функции CryptEnumProvidersU
// из модуля Crypt32.dll
typedef BOOL (WINAPI *PFN_CRYPT_ENUM_PROVIDERS_U)(
```

(см. след. стр.)

```
    DWORD     dwIndex,
    DWORD     *pdwReserved,
    DWORD     dwFlags,
    DWORD     *pdwProvType,
    LPWSTR pswzProvName,
    DWORD     *pcbProvName
    );
PFN_CRYPT_ENUM_PROVIDERS_U  pfnCryptEnumProvidersU=NULL;
// См. комментарий 2
// Определение массива имен типов криптопровайдеров
LPTSTR szProvType[]={
          NULL,
          "RSA Full (Signature and Key Exchange)",
          "RSA Signature",
          "DSS Signature",
          "PROV_FORTEZZA",
          "PROV_MS_EXCHANGE",
          "PROV_SSL",
          NULL, NULL, NULL, NULL, NULL,
          "RSA SChannel",
          "DSS Signature with Diffie-Hellman Key Exchange",
          "PROV_EC_ECDSA_SIG",
          "PROV_EC_ECNRA_SIG",
          "PROV_EC_ECDSA_FULL",
          "PROV_EC_ECNRA_FULL",
          "Diffie-Hellman  SChannel",
          NULL,
          "PROV_SPYRUS_LYNKS",
          "PROV_RNG",
          "PROV_INTEL_SEC"};
// Определение массива имен типов исполнения
//  криптопровайдеров
LPTSTR szImpType[]={
          NULL,
          "Hardware",
          "Software",
          "Mixed",
          "Unknown",
          "Not define", "Not define", "Not define",
          "Removable",
          "Type 9", "Type 10", "Type 11"};
// Определение массива имен классов алгоритмов
LPTSTR szAlgClass[] = {
          NULL,
          "Signature",
          "Encrypt  ",
```

(см. след. стр.)

```
            "Encrypt    ",
            "Hash       ",
            "Exchange   ",
            "All        "};
// Определение и инициализация переменных
HMODULE                 hDll=0;
DWORD                   dwCount, cbName;
DWORD                   dwDataLen, dwIndex=0;
WCHAR                   swzProvName[100];
CHAR                    szName[100];
PROV_ENUMALGS           AlgInfo;
PROV_ENUMALGS_EX        AlgInfoEx;
CHAR                    *pszAlgType = NULL;
LPWSTR                  pswzProvName=NULL;
LPWSTR                  pszProvName=NULL;
DWORD                   dwProvType, dwCryptError,
                        dwVersion, dwImpType;
DWORD                   dwFlags=0;
HCRYPTPROV              hProv = 0;
DWORD                   dwSigKeySizeInc, dwXchKeySizeInc;
void main(void) {
// См. комментарий 3
   // Загружаем библиотеку Crypt32.dll
   hDll=LoadLibrary("Crypt32.dll");
   if (!hDll) {
      printf("Error:LoadLibrary=0x%X.\n",GetLastError());
      goto  ReleaseResource;
   }
   // Получаем адрес функции CryptEnumProvidersU
   pfnCryptEnumProvidersU=
         (PFN_CRYPT_ENUM_PROVIDERS_U)GetProcAddress(hDll,
      "CryptEnumProvidersU");
   if (!pfnCryptEnumProvidersU) {
      printf("Error:GetProcAddress=0x%X.\n",
         GetLastError());
      goto  ReleaseResource;
   }
// См. комментарий 4
   // Цикл перечисления криптопровайдеров
   while  (pfnCryptEnumProvidersU(
         dwIndex,
         NULL,
         0,
         &dwProvType,
         NULL,
         &cbName)) {
```

(см. след. стр.)

```
      if (pfnCryptEnumProvidersU(
            dwIndex++,
            NULL,
            0,
            &dwProvType,
            swzProvName,
            &cbName)) {
         ZeroMemory(szName,100);
         // Конвертируем Unicode-строку
         if (WideCharToMultiByte(
               CP_ACP,
               WC_COMPOSITECHECK,
               swzProvName,
               -1,
               szName,
               wcslen(swzProvName),
               NULL,
               NULL
               )) {
            printf("Error:WideCharToMultiByte=0x%X.\n",
               GetLastError());
            break;
         }
// См, комментарий 5
      // Открываем контекст криптопровайдера
      if (!CryptAcquireContext(
            &hProv,
            NULL,
            szName,
            dwProvType,
            CRYPT_VERIFYCONTEXT)) {
         printf("Error:CryptAcquireContext=0x%X.\n",
            GetLastErrorO);
         break;
      }
// См. комментарий 6
         // Получаем имя криптопровайдера
         dwDataLen=100;
         if (!CryptGetProvParam(
               hProv,
               PP_NAME,
               szName,
               &dwDataLen,
               0)) {
            printf("Error:CryptGetProvParam=0x%X.\n",
               GetLastErrorO);
```

(см. след. стр.)

```
            goto ReleaseResource;
         }
         // Получаем версию криптопровайдера
         dwDataLen=sizeof(DWORD);
         if (!CryptGetProvParam(
               hProv,
               PP_VERSION,
               (PBYTE)&dwVersion,
               &dwDataLen,
               0)) {
            printf("Error:CryptGetProvParam=0x%X.\n",
               GetLastError());
            goto ReleaseResource;
         }
         // Получаем тип исполнения криптопровайдера
         dwDataLen=sizeof(DWORD);
         if (!CryptGetProvParam(
               hProv,
               PP_IMPTYPE,
               (PBYTE)&dwImpType,
               &dwDataLen,
               0)) {
            printf("Error:CryptGetProvParam=0x%X.\n",
               GetLastError0);
            goto ReleaseResource;
         }
         // Распечатываем полученную информацию
         printf( "Provider name; %s\n", szName);
         printf("Provider version: %d.%d\n",
                        HIBYTE( LOWORD(dwVersion)),
                        LOBYTE(LOWORD(dwVersion) ) );
         printf("Provider type: %s\n",
            szProvType[dwProvType]);
         printf("Provider implementation type:
            %s\n",szImpType[dwImpType]);
         // Распечатываем заголовок таблицы алгоритмов
         printf("                |Length info (bits)|\n");
         printf("AlgId|Type      |Def   |Max   |Min  |Name  \n");
// См. комментарий 7
         // Перечисляем алгоритмы криптопровайдера
         for(dwCount=0 ; ; dwCount++) {
            // Устанавливаем флаг CRYPT_FIRST
            // при первом обращении в цикле.
            if(dwCount == 0) dwFlags = CRYPT_FIRST;
            else dwFlags = 0;
            // Получаем информацию об алгоритмах
```

(см. след. стр.)

```
            dwDataLen = sizeof(PROV_ENUMALGS_EX);
            dwCryptError=ERROR_SUCCESS;
            // Запрашиваем структуру PROV_ENUMALGS_EX
            if (!CryptGetProvParam(
                  hProv,
                  PP_ENUMALGS_EX,
                  (PBYTE)&AlgInfoEx,
                  &dwDataLen,
                  dwFlags)) {
               dwCryptError=GetLastError();
               if (dwCryptError == ERROR_NO_MORE_ITEMS)
                  break;
               if (dwCryptError == NTE_BAD_FLAGS
                                    || dwCryptError==
                                   NTE_BAD_TYPE) {
II См. комментарий 8
                  dwDataLen = sizeof(PROV_ENUMALGS);
                  // Если криптопровайдер не поддерживает
                  // запрос, то запрашиваем структуру
                  // PROV_ENUMALGS
                  if (!CryptGetProvParam(
                        hProv,
                        PP_ENUMALGS,
                        (PBYTE)&AlgInfo,
                        &dwDataLen,
                        dwFlags)) {
                     dwCryptError=GetLastError();
                     if (dwCryptError ==
                        ERROR_NO_MORE_ITEMS)  break;
                     printf("Error %x reading
                        algorithm!\n",GetLastError());
                     goto ReleaseResource;
                  }
                  AlgInfoEx.aiAlgid=AlgInfo.aiAlgid;
                  AlgInfoEx.dwDefaultLen=AlgInfo.dwBitLen;
                  AlgInfoEx.dwMaxLen=0;
                  AlgInfoEx.dwMinLen=0;
                  lstrcpy(AlgInfoEx.szLongName,
                     AlgInfo.szName);
               }
               else {
                  printf("Error %x reading
                     algorithm!\n",GetLastError());
                  goto ReleaseResource;
               }
            }
```

(см. след. стр.)

```
            // Распечатываем полученную информацию
            printf(
                "%4.4xh|%s|%-4d |X-5d  |%-4d |%s \n",
                AlgInfoEx.aiAlgid,
                szAlgClass[GET_ALG_CLASS(AlgInfoEx.aiAlgid)
                    >> 13],
                AlgInfoEx.dwDefaultLen,
                AlgInfoEx.dwMaxLen,
                AlgInfoEx.dwMinLen,
                AlgInfoEx.szLongName);
        }
// См. комментарий 9
        // Запрашиваем информацию о значении шага
        // инкремента ключа подписи
        dwDataLen=sizeof(DWORD);
        if (CryptGetProvParam(
              hProv,
              PP_SIG_KEYSIZE_INC,
              (PBYTE)&dwSigKeySizeInc,
              &dwDataLen,
              dwFlags)) {
           if (dwSigKeySizeInc)
              printf("Number of bits for the increment
                  length of
                                signature key - %-
                                4d\n",dwSigKeySizeInc);
        }
        // Запрашиваем информацию о значении шага
        // инкремента ключа обмена
        if (CryptGetProvParam(
              hProv,
              PP_SIG_KEYSIZE_INC,
              (PBYTE)&dwXchKeySizeInc,
              &dwDataLen,
              dwFlags)) {
           if (dwXchKeySizeInc)
              printf("Number of bits for the increment
              length of
                                exchange key - %-
                                4d\n",dwXchKeySizeInc);
        }
        printf("\n");
        // Освобождаем дескриптор криптопровайдера
        CryptReleaseContext(hProv, 0); hProv=0;
    }
}
```

(см. след. стр.)

```
ReleaseResource:
    // Освобождаем выделенные ресурсы
    . . .
    return;
}
```

Теперь прокомментируем показанный фрагмент.

1. Определяем прототип функции ***CryptEnumProvidersU***Как уже отмечалось, для последовательного перечисления криптопровайдеров, установленных в системе, используется функция ***CryptEnumProviders***из библиотеки advapi32.dll. Однако, эта функция доступна только для ОС, начиная с Windows 2000 и Windows 98. Вообще говоря, материал, изложенный в предыдущем параграфе, позволяет реализовать подобную функцию самостоятельно, но в этом нет необходимости. В Windows NT 4.0, как и других операционных системах, в библиотеке Crypt32.dll имеется функция перечисления криптопровайдеров ***CryptEnumProvidersU.*** Прототип этой функции полностью совпадает с прототипом Unicode-версии функции ***CryptEnumProviders***(см. прототип функции ***CryptEnumProvidersW***в файле WinCrypt.h). Поэтому для универсальности в программе используется функция ***CryptEnumProvidersU m*** библиотеки Crypt32.dll.
2. Для удобства вывода информации на экран определяем строковые массивы для названий типов криптопровайдеров. названий типов исполнения криптопровайдеров и имен классов. Индекс в этих массивах соответствует числовому значению типа, типа исполнения или класса.
3. Для получения адреса процедуры ***CryptEnumProvidersU*** загружаем библиотеку Crypt32.dll и вызываем функцию ***GetProcAddress***.
4. Реализуем цикл перечисления криптопровайдеров. Б качестве индекса используется переменная *dwIndex*, которая инициализируется нулем и инкрементируется в каждом цикле.
5. На основании полученного имени и типа криптопровайдера вызываем функцию ***CryptAcquireContext***Поскольку мы использовали Unicode-функцию ***CryptEnumProvidersU***то предварительно Unicode-строку *swzProvName* с именем криптопровайдера конвертируем в ANSI-строку *szName*. Поскольку мы не намерены работать с ключевыми парами,

то применим флаг CRYPT_VERIFYCONTEXT (см. приложение 2).

6. Используя полученный дескриптор криптопровайдера, вызываем функцию ***CryptGetProvParam*** для получения необходимых параметров. Запрашиваем базовые параметры криптопровайдера: имя, версию и тип исполнения. Разумеется, имя криптопровайдера у нас уже имеется, но мы все же запрашиваем и этот параметр для демонстрации работы с функцией ***CryptGetProvParam.***

7. Реализуем цикл перечисления алгоритмов, которые поддерживает криптопровайдер. Наиболее полную информацию можно получить через запрос параметра PP_ENUMALGS_EX, который возвращает структуру PROV_ENUMALGS_EX. Однако старые версии криптопровайдеров не поддерживают этот параметр, в них возможен запрос параметра PP_ENUMALGS, который возвращает структуру PROV_ENUMALGS.

8. Если запрос параметра PP_ENUMALGS_EX не удается, то запрашиваем параметр PP_ENUMALGS. Задаем для возможных полей структуры *AlgInfoEx* значения, полученные в этом запросе. Остальные поля инициализируются нулем. В поле полного имени алгоритма копируем короткое имя.

9. Запрашиваем значения шага инкремента для ключей подписи и обмена. Данную возможность поддерживается криптопровайдерами (и то не всеми) для ОС, начиная с Windows 2000. потому если запрос не обрабатывается, то информация просто не выводится на печать,

Приведем информацию, которая получена с помощью этой программы на Windows 2000 с установленным пакетом «Microsoft Windows 2000 High Encryption Pack». Этот пакет необходим для установки в двух дополнительных криптопровайдеров «*Microsoft* Enhanced Cryptographic *Provider v1.0*» (библиотека rsaenh.dll) и «Microsoft Enhanced DSS and Diffie-Hellman Cryptographic Provider» (библиотека dssenh.dll). Основное их отличие заключается *в* поддержке алгоритмов симметричного шифрования с длиной ключа свыше 56 бит. В Windows XP и Windows ME эти криптопровайдеры включены в состав операционной системы.

Листинг 2-3. Пример работы программы EnumCSP

```
Provider name: Gemplus GemSAFE Card CSP v1.0
Provider version: 0.0
```

(см. след. стр.)

```
Provider type: RSA Full (Signature and Key Exchange)
Provider implementation type: Type 11
                 ILength info (bits)|
AlgId | Type     |Def  I Max    |Min   |Name
6602h|Encrypt   |40    |128     |40    |RSA Data Security's RC2
6801h|Encrypt   |40    |128     j 40   |RSA Data Security's RC4
6601h|Encrypt   |56    | 56     |56    |Data Encryption Standard
                                        (DES)
6609h|Encrypt   |112   |112     |112   ||Two Key Triple DES
6603h|Encrypt   |168   |168     |168   ||Three Key Triple DES
8004h|Hash      |160   ||160     |160   |Secure Hash Algorithm
                                        (SHA-1)
8001h|Hash      |128   |128     ||128  ||Message Digest 2 (MD2)
8002h|Hash      |128   |128     ||128  |Message Digest 4 (MD4)
8003h|Hash      |128   |128     ||128  |Message Digest 5 (MD5)
8008h|Hash      |288   |288     |288   |SSL3 SHAMD5
8005h|Hash      |0     IO      IO     |Message Authentication
                                        Code
2400h|Signature|1024  |1024    |512   |RSA Signature
a400h|Exchange |512   |512     |512   |RSA Key Exchange
8009h|Hash      |0     IO      IO     IHugo's MAC (HMAC)
Number of bits for the increment length of signature key - 512
Number of bits for the increment length of exchange key - 512
Provider name: Microsoft Base Cryptographic Provider v1.0
Provider version: 2.0
Provider type: RSA Full (Signature and Key Exchange)
Provider implementation type: Software
                 |Length info (bits)|
AlgId | Type     ||Def  | Max   [Min  |Name
6602h|Encrypt   |40    |56      |40    |RSA Data Security's RC2
6801h|Encrypt   |40    |56      |40    |RSA Data Security's RC4
6601h|Encrypt   |56    |56      |56    |Data Encryption Standard
                                        (DES)
8004h|Hash      |160   ||160     |160   |Secure Hash Algorithm
                                        (SHA-1)
8001h|Hash      |128   |128     |128   IMessage Digest 2 (MD2)
8002h|Hash      |128   ||128     |128   IMessage Digest 4 (MD4)
8003h|Hash      |128   ||128     |128   |Message Digest 5 (MD5)
8008h|Hash      |288   |288     |288   |SSL3SHAMD5
8005h|Hash      IO     IO      IO     |Message Authentication
                                        Code
2400h|Signature|512   |16384   |384   |RSA Signature
a400h|Exchange |512   |1024    |384   |RSA Key Exchange
8009h|Hash      IO     IO      IO     |Hugo's MAC (HMAC)
Number of bits for the increment length of signature key - 8
Number of bits for the increment length of exchange key - 8
```

(см. след. стр.)

```
Provider name: Microsoft Base DSS and Diffie-Hellman
   Cryptographic Provider
Provider version: 1.0
Provider type: DSS Signature with Diffie-Hellman Key
   Exchange
Provider implementation type: Software
                |Length info (bits)|
AlgId|Type      |Def   |Max    | Min|Name
660ch|Encrypt   |40    |40     |40  |CYLINK Message Encryption
                                     Algorithm
6602h|Encrypt   |40    | 56    |40  |RSA Data Security's RC2
6801h|Encrypt   |40    |56     |40  |RSA Data Security's RC4
6601h|Encrypt   | 56   |56     |56  |Data Encryption Standard
                                     (DES)
8004h|Hash      |160   |160    |160 |Secure Hash Algorithm
                                     (SHA-1)
8003h|Hash      ||128  |128    |128 |Message Digest 5 (HD5)
2200h|Signature|1024  |1024   |512 IDigital Signature
                                     Algorithm
aa01h|Exchange  |512   |1024   |512 I Diffie-Hellman Key
                                     Exchange Algorithm
aa02h|Exchange  |512   |1024   |512 |Diffie-Hellman Ephemeral
                                     Algorithm
Number of bits for the increment length of signature key - 64
Number of bits for the increment length of exchange key - 64
Provider name: Microsoft Base DSS Cryptographic Provider
Provider version: 1.0
Provider type: DSS Signature
Provider implementation type: Software
                |Length info (bits)|
AlgId|Type      |Def   |Max    |Min  |Name
8004h|Hash      |160   |160    |160  ISecure Hash Algorithm
                                      (SHA-1)
8003h|Hash      |128   |128    |128  |Message Digest 5 (MD5)
2200h|Signature|1024  |1024   |512  |Digital Signature
                                      Algorithm
Number of bits for the increment length of signature key - 64
Number of bits for the increment length of exchange key - 64
Provider name: Microsoft DH SChannel Cryptographic Provider
Provider version: 1.0
Provider type: Diffie-Hellman SChannel
Provider implementation type: Software
                [Length info (bits)|
AlgId|Type      |Def   |Max    |Min  |Name
660ch|Encrypt   |40    |40     |40   ICYLINK Message Encryption
                                      Algorithm
```

(см. след. стр.)

```
6602h|Encrypt  |40   |128   |40   |RSA Data Security's RC2
6801h|Encrypt  |40   |128   |40   |RSA Data Security's RC4
6601h|Encrypt  |56   |56    |56   |Data Encryption Standard
                                    (DES)
6609h|Encrypt  |112  |112   |112  |Two Key Triple DES
6603h|Encrypt  |168  |168   |168  |Three Key Triple DES
8004h|Hash     |160  |160   |160  |Secure Hash Algorithm
                                    (SHA-1)
8003h|Hash     |128  |128   |128  |Message Digest 5 (MD5)
2200h|Signature|1024 |1024  |1024 |Digital Signature
                                    Algorithm
aa01h|Exchange |512  |4096  |512  |Diffie-Hellman Key
                                    Exchange Algorithm
aa02h|Exchange |512  |4096  |512  |Diffie-Hellman Ephemeral
                                    Algorithm
Number of bits for the increment length of signature key - 64
Number of bits for the increment length of exchange key - 64
Provider name: Microsoft Enhanced Cryptographic Provider
   v1.0
Provider version: 2.0
Provider type: RSA Full (Signature and Key Exchange)
Provider implementation type: Software
              |Length info (bits)|
AlgId|Type     |Def  |Max   |Min  |Name
6602h|Encrypt  |128  |128   |40   |RSA Data Security's RC2
6801h|Encrypt  |128  |128   |40   |RSA Data Security's RC4
6601h|Encrypt  |56   |56    |56   |Data Encryption Standard
                                    (DES)
6609h]Encrypt  |112  |112   |112  |Two Key Triple DES
6603h|Encrypt  |168  |168   |168  |Three Key Triple DES
8004h|Hash     |160  |160   |160  |Secure Hash Algorithm
                                    (SHA-1)
8001h|Hash     |126  |128   |128  |Message Digest 2 (MD2)
8002h|Hash     |128  |128   |128  |Message Digest 4 (MD4)
8003h|Hash     |128  |128   |128  |Message Digest 5 (MD5)
8008h|Hash     |288  |288   |288  |SSL3 SHAM05
8005h|Hash     |0    |0     |0    |Message Authentication
                                    Code
2400h|Signature|1024 |16384 |384  |RSA Signature
a400h|Exchange |1024 |16384 |384  |RSA Key Exchange
8009h|Hash     |0    |0     |0    |Hugo's MAC (HMAC)
Number of bits for the increment length of signature key - 8
Number of bits for the increment length of exchange key - 8
Provider name: Microsoft Enhanced DSS and Diffie-Hellman
Cryptographic Provider
Provider version: 1.0
```

(см. след. стр.)

```
Provider type: DSS Signature with Diffie-Hellman Key Exchange
Provider implementation type: Software
                |Length info (bits)|
AlgId|Type      |Def   | Max   |Min  |Name
660ch|Encrypt   |40    |40     |40   |CYLINK Message Encryption
                                      Algorithm
6602h|Encrypt   |40    |128    |40   |RSA Data Security's RC2
6801h|Encrypt   |40    |128    |40   |RSA Data Security's RC4
6601h|Encrypt   |56    |56     |56   |Data Encryption Standard
                                      (DES)
6609h|Encrypt   |112   |112    |112  |Two Key Triple DES
6603h|Encrypt   |168   |168    |168  |Three Key Triple DES
8004h|Hash      |160   |160    |160  |Secure Hash Algorithm
                                      (SHA-1)
8003h|Hash      |128   |128    |128  |Message Digest 5 (MD5)
2200h|Signature|1024   |1024   |512  |Digital Signature
                                      Algorithm
aa01h|Exchange  |512   |4096   |512  |Diffie-Hellman Key
                                      Exchange Algorithm
aa02h|Exchange  |512   |4096   |512  |Diffie-HellmanEphemeral
                                      Algorithm
Number of bits for the increment length of signature key - 64
Number of bits for the increment length of exchange key - 64
Provider name: Microsoft RSA SChannel Cryptographic Provider
Provider version: 2.0
Provider type: RSA SChannel
Provider implementation type: Software
                |Length info (bits)|
AlgId|Type      |Def   |Max    |Hin  |Name
6602h|Encrypt   |128   |128    |40   |RSA Data Security's RC2
6801h|Encrypt   |128   |128    |40   |RSA Data Security's RC4
6601h|Encrypt   |56    |56     |56   |Data Encryption Standard
                                      (DES)
6609h|Encrypt   |112   |112    |112  |Two Key Triple DES
6603h|Encrypt   |168   |168    |168  |Three Key Triple DES
8004h|Hash      |160   |160    |160  |Secure Hash Algorithm
                                      (SHA-1)
8003h|Hash      |128   |128    |128  |Message Digest 5 (MD5)
8008h|Hash      |288   |288    |288  |SSL3 SHAMD5
8005h|Hash      |0     |0      |0    |Message Authentication
                                      Code
2400h|Signature|1024   |16384  |384  |RSA Signature
a400h|Exchange  |1024  |16384  |384  |RSA Key Exchange
8009h|Hash      |0     |0      |0    |Hugo's MAC (HMAC)
Number of bits for the increment length of signature key - 8
Number of bits for the increment length of exchange key - 8
Provider name: Microsoft Strong Cryptographic Provider
```

(см, след. стр.)

```
Provider version: 2.0
Provider type: RSA Full (Signature and Key Exchange)
Provider implementation type: Software
                |Length info (bits)|
AlgId | Type    |Def  I Max    |Min   |Name
6602h|Encrypt   |40   |128   | 40    |RSA Data Security's RC2
6801h|Encrypt   |40   |128   |  40   |RSA Data Security's RC4
6601h|Encrypt   |56   I 56     |56    |Data Encryption Standard
                                      (DES)
6609h|Encrypt   |112  |112     |112   |Two Key Triple DES
6603h|Encrypt   |168  |168     |168   |Three Key Triple DES
8004h|Hash      |160  |160     |160   |Secure Hash Algorithm
                                      (SHA-1)
8001h|Hash      |128  ||128    |128   |Message Digest 2 (MD2)
8002h|Hash      |128  |128     |128   |Message Digest 4 (MD4)
8003h|Hash      |128  |128     |128   |Message Digest 5 (MD5)
8008h|Hash      |288   |288  | 288   |SSL3 SHAMD5
8005h | Hash    |0    |0       |0     |Message Authentication
                                      Code
2400h|Signature|512   |16384| 384     |RSA Signature
a400h | Exchange |512 |16384  |384    | | RSA Key Exchange
8009h|Hash      |0    |0       |0     |Hugo's MAC (HMAC)
Number of bits for the increment length of signature key - 8
Number of bits for the increment length of exchange key - 8
Provider name: Schlumberger Cryptographic Service Provider
Provider version: 5.0
Provider type: RSA Full (Signature and Key Exchange)
Provider implementation type: Type 11
                |Length info (bits) |
AlgId |Type     |Def  I Max    |Min   |Name
6602h | Encrypt |40   |128     |40    |RSA Data Security's RC2
6801h|Encrypt   |40   |128     |40    |RSA Data Security's RC4
6601h | Encrypt  | 56  | 56    |56    |Data Encryption Standard
                                      (DES)
6609h|Encrypt   |112  |112     |112   |Two Key Triple DES
6603h|Encrypt   |168  |168     |168   IThree Key Triple DES
8004h|Hash      |160  |160     |160   jSecure Hash Algorithm
                                      (SHA-1)
8001h|Hash      |128  || 128   |128   |Message Digest 2 (MD2)
8002h|Hash      |128  || 128   |128   |Message Digest 4 (MD4)
8003h|Hash      |128  || 128   |128   |Message Digest 5 (MD5)
8008h|Hash      |288  |288     |288   |SSL3 SHAMD5
8005h|Hash      IO   IO        IO     |Message Authentication
                                      Code
2400h|Signature|1024  |1024    |1024  |RSA Signature
a400h| Exchange |1024 |1024    |1024  |RSA Key Exchange
8009h|Hash      IO   IO        !0     |Hugo's MAC (HMAC)
```

(см. след. стр.)

Из результатов работы программы видно, что «enhanced» криптопровайдеры Microsoft поддерживают четыре алгоритма симметричного шифрования с длиной ключа свыше 56 бит: RC2 -- длина ключа от 40 до 128 бит, RC4 — длина ключа от 40 до 128 бит, Triple DES с двойным ключом — длина ключа 112 бит и Triple DES с тройным ключом — длина ключа 168 бит. Такие возможности есть и у криптопроваидеров «Gemplus GemSAFE Card CSP v1.0» и «Schlumberger Cryptographic Service Provider». Но в действительности для реализации процедур шифрования криптопровайдеры Gemplus и Schlumberger вызывают или «Microsoft Strong Cryptographic Provider» или «Microsoft Base Cryptographic Provider v1.0». Таким образом, если у вас установлена ОС Windows 2000 и не установлен пакет «Microsoft Windows 2000 High Encryption Pack», то криптопровайдеры Gemplus и Schlumberger будут поддерживать тс же алгоритмы, которые поддерживает криптопровайдер «Microsoft Base Cryptographic Provider v1.0».

Иногда очень полезными оказываются данные о том, в какой библиотеке реализован данный криптопровайдер, но функции CryptoAPI не позволяют ее получить. Полагаем, что при изучении материалов предыдущего параграфа, вы просмотрели ключи реестра со списком криптопровайдеров и соответствующих им библиотек. Здесь только отметим, что не все криптопровайдеры, показанные в листинге 2-3, реализованы в различных библиотеках. Так криптопровайдеры «Microsoft Enhanced Cryptographic Provider v1.0», «Microsoft Strong Cryptographic Provider» и «Microsoft RSA SChannel Cryptographic Provider» реализуются одной библиотекой rsaenh.dll, а криптопровайдеры «Microsoft Enhanced DSS and Diffie-Hellman Cryptographic Provider» и «Microsoft DH SChannel Cryptographic Provider» — в библиотеке dssenh.dll. Напомним, что для получения полной информации необходимо обратиться к ключу реестра HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider.

Использование CryptoAPI для обмена защищенными сообщениями

В предыдущем параграфе было показано, как получить полную информацию о возможностях криптопроваидеров, установленных на вашем рабочем месте. Теперь мы расскажем о том, как использовать CryptoAPl 1.0 для реализации различных схем обмена криптографически защищенными данными.

Разумеется, схем обмена защищенными данными очень много, и описать их все в данной книге мы не Б состоянии, да и этом и нет необходимости. Поэтому поставим задачу более узко: на примерах формирования нескольких видов *криптографических сообщений* (cryptographic message) показать возможности и принципы работы с интерфейсом CryptoAPI 1.0. Под криптографическим сообщением мы понимаем шифртекст (если сообщение шифруется), а также всю совокупность криптографической информации, отправляемой получателю.

Два пользователя, одного из которых будем называть *отправитель* (originator), а другого *получатель* (recipient), участвуют в обмене сообщениями. Необходимо обеспечить *конфиденциальность* (privacy), *целостность* (integrity) и *аутентичность* (authentication). Это означает, что получатель должен быть *с высокой степенью вероятности* уверен, что он получил именно то сообщение, которое ему предназначалось (целостность), что послал это сообщение именно отправитель (аутентичность) и что никто кроме него не может восстановить открытый текст данного сообщения (конфиденциальность). Рассмотрим, какие из этих задач и как можно решить средствами интерфейса CryptoAPI 1.0.

Отправитель и получатель могут разделять общий «секрет», например заранее сгенерированный общий ключ или пароль, на основе которого генерируется общий ключ. В этом случае, как правило, реализуется полностью симметричная схема шифрования, которая обеспечивает и конфиденциальность, и целостность, и аутентичность. В разделе «Использование CryptoAPI 1.0 для реализации схемы симметричного шифрования» рассмотрен пример реализации подобной схемы, в том числе и вопрос генерации ключа из пароля.

Если отправитель и получатель не имеют общего «секрета», то для обеспечения конфиденциальности необходимо использовать несимметричные ключевые схемы. В этом случае, чтобы послать зашифрованное сообщение, отправитель должен получить открытый ключ ключевой пары обмена получателя. В разделе «Использование CryptoAPI 1.0 для реализации схем несимметричного шифрования» рассмотрен пример реализации схемы шифрования с использованием несимметричного обмена ключей.

Для решения проблемы целостности в случае несимметричной ключевой схемы необходимо использовать цифровую подпись.

Для этого отправителю следует применить свою ключевую пару цифровой подписи, а для проверки подписи отправить свой открытый ключ вместе с сообщением. В разделе «Использование CryptoAPI 1.0 для реализации схемы цифровой подписи» рассмотрен пример реализации подобной схемы. Мы не стали объединять несимметричное шифрование и цифровую подпись в одном примере из-за его громоздкости, хотя часто требуется отправить зашифрованное сообщение вместе с цифровой подписью. Объединить эти возможности в одной программе нетрудно, если вы разберетесь в приведенных далее примерах.

В этой главе мы покажем три учебных примера использования CryptoAPI 1.0 для обмена криптографическими сообщениями. Для наглядности все этапы обмена сообщениями, а также работа отправителя и получателя объединены в одной программе. Сообщения для шифрования и подписи представлены в файле, зашифрованное сообщение или подпись также сохраняются в файле. *Эти примеры неучитывают реальных условий передачи данных в распределенных сетях. Именно поэтому в каждой главе мы оговариваем требования, при которых будут выполняться условия конфиденциальности и (или) целостности и аутентичности.*

Использование CryptoAPI 1.0 для реализации схемы симметричного шифрования

В этом разделе мы расскажем о реализации схемы симметричного шифрования, использующей в качестве ключевого материала пароль. Исходный текст программы находится в файле Chapter2/CrMsgPswd/CrMsgPswd.c.

Проблема генерации криптографических ключей из осмысленной информации (пароля) является весьма актуальной для практического использования механизмов криптографической защиты,

С точки зрения криптографического анализа задача формулируется следующим образом. Необходимо из осмысленной информации переменной длины с низкой энтропией (сильно неравновероятные последовательности) получить однозначным детерминированным образом ключевую последовательность заданной длины. При этом необходимо учитывать, что пароль имеет переменную длину, как правило, меньшую, чем длина криптографического ключа.

Пароль, вообще говоря, не очень хороший ключевой материал. Часто в качестве пароля используют значимые слова, и в этом случае он более или менее хорошо поддается словарной атаке — перебору всех осмысленных n-буквенных сочетаний. Как правило, перебор по различным словарям имеет трудоемкость порядка $10^5 - 10^6$ элементарных операций типа «опробование одного пароля». Следовательно, механизм генерации ключа *из* пароля должен занимать длительное время — в этом случае однократная генерация ключа из пароля, проводимая легальным пользователем, занимает хотя длительное, но практически незаметное для пользователя время, а перебор указанного числа парольных комбинаций по словарю выходит за рамки реального времени.

Если предполагаемая мощность словаря $N_{сл}$, требуемое время сохранности ключа, полученного из пароля, — T, то время t_0, необходимое на генерацию ключа из пароля, рассчитывается по формуле

$$t_0 = 2T/(N_{сл}).$$

Предполагается, что перебор в среднем происходит до половины словаря. Таким образом, если T = 30 суток = $2{,}6 * 10^6$ секунд, а $N_{сл} = 10^6$, то t_0 составит 5,2 секунды. При этом делается предположение, что пароль обязательно найдется в словаре (это объясняет парадокс, заключающийся в том, что при малом словаре растет величина t_0 — перебор по малому словарю происходит быстро, но при этом понижается вероятность присутствия в нем искомого пароля).

Однако, даже если применяется хаотичный набор символов, все равно количество печатных символов крайне ограничено и не позволяет добиться высокой степени стойкости на разумной длине пароля. *Разумеется, если высокая стойкость шифрования информации является для вас определяющей, то вместо пароля необходимо использовать случайную бинарную последовательность длины, равную длине используемого ключа шифрования. Ее можно сгенерировать посредством функции* ***CryptGenRandom*** *или взять с другого доступного вам датчика случайных чисел (ДСЧ).*

Так как же сгенерировать ключ шифрования из пароля, чтобы максимально затруднить возможному злоумышленнику задачу дешифрования? Способы генерации ключа из пароля подробно рассмотрены в [35]. Мы не будем здесь дублировать материалы этого источника, а перечислим лишь основные момсн-

ты, на основе которых реализована функция генерации в нашей программе.

Итак, использование пароля напрямую в качестве ключа по уже упомянутым причинам невозможно. Следовательно, необходимо применение размешивающего преобразования, отображающего компактную область паролей в распределенную область пространства ключей. Так, «близкие» пароли «морс» и «море» должны отображаться в «далекие» ключи во избежания применения методов дифференциального криптоанализа (использующего знания о том, что шифртексты зашифрованы на близких ключах). Наилучшим кандидатом на размешивающие преобразования, отображающие текст произвольной длины в векторы фиксированной длины, являются хеш-функции. В качестве базовых функций при преобразовании ключа используются различные алгоритмы хеш-функций (MD2, MD5, SHA-1, HMAC-SHA-1). Кроме того, имеется несколько способов улучшения качества *схем шифрования, основанных на пароле* (password-based encryption).

Первый способ — введение *модификатора ключа* (salt). Это случайная бинарная последовательность, которая «замешивается» в функцию преобразования вместе с паролем. Введение модификатора ключа усложняет словарную атаку, основанную на переборе заранее вычисленных хеш-функций от фиксированного набора паролей (подробно об этом в [24]). Однако, если злоумышленник обладает необходимыми вычислительными ресурсами, то он может пронести словарную атаку и без предвычисленных хеш-функций, поскольку модификатор ключа является открытой информацией. Использования только модификатора ключа в этом случае недостаточно. В настоящее время минимальная рекомендуемая длина модификатора ключа составляет 8 байт.

Второй способ - введение *счетчика итераций* (iteration count). Счетчик итераций — это то количество раз, которое должна повториться функция преобразования, участвующая в генерации ключа. Рекомендуемое значение счетчика итераций — не менее 1000. Такое количество итераций практически не скажется на скорости вычислений для легальных пользователей, однако злоумышленнику при переборе паролей, потребуются огромные вычислительные мощности.

Функция генерации ключа из пароля (password-based key derivation function) ***CryptPBKDFunction*** используемая в програм-

ме, показана в листинге 2-4. Процедура не является реализацией функций PBKDF1 или PBKDF2, описанных в [35], а лишь показывает простейший вариант использования всех перечисленных способов с помощью функций CryptoAPI 1.0.

Листинг 2-4. Текст процедуры CryptPBKDFunction

```
// См. комментарий 1
BOOL
CryptPBKDFunction(
   HCRYPTPROV   hCryptProv,
   ALG_ID aiCryptAlg,
   DWORD     dwKeyFlags,
   HCRYPTKEY *phPBKey,
   PDATA_BLOB   pPassword,
   PDATA_BLOB   pSalt,
   ALG_ID aiHashAlg,
   DWORD     dwHashCount
   )
{
   BOOL      Result=TRUE;
   HCRYPTHASH    hHash=0;
   DWORD     dwDataLen=0, dwCount=0;
   DWORD     dwKeyLen=0;
   DATA_BLOB HashVal={0,NULL}, IV={0,NULL};
   // Создаем объект хеш-функции
   if (!CryptCreateHash(hCryptProv,aiHashAlg,0,0,&hHash)) <
      printf("Error:
         CryptCreateHash=0x%X.\n",GetLastError());
      Result=FALSE;
      goto ReleaseResource;
   }
// См. комментарий 2
   // Если модификатор ключа отсутствует, то вырабатываем его
   if (!pSalt->pbData) {
      // Выделяем память для модификатора клоча
      pSalt->pbData=LocalAlloc(LMEM_ZEROINIT,
         pSalt->cbData);
      if (!pSalt->pbData) {
         printf("Error: Out of memory.\n");
         Result=FALSE;
         goto ReleaseResource;
      }
      // Генерируем случайное значение модификатора ключа
      if  (!CryptGenRandom(hCryptProv,pSalt->cbData,
         pSalt->pbData)) {
         printf("Error:
```

(см. след. стр.)

```
            CryptGenRandom=0x%X.\n",GetLastError());
         Result=FALSE;
         goto ReleaseResource;
      }
   }
// См. комментарий 3
   // Хешируем значение пароля
   if (!CryptHashData(hHash,pPassword->pbData,
      pPassword->cbData,0)) {
      printf("Error:
         CryptHashData=0x%X.\n",GetLastError());
      Result=FALSE;
      goto  ReleaseResource;
   }
// См. комментарий 4
   // Цикл хеширования модификатора ключа
   for (dwCount=1; dwCount <= dwHashCount; dwCount++) {
      // Хешируем модификатор ключа
      if (!CryptHashData(hHash,pSalt->pbData,
         pSalt->cbData,0)) {
         printf("Error:
            CryptHashData=0x%X.\n",GetLastError());
         Result=FALSE;
         goto ReleaseResource;
      }
   }
// См. комментарий 5
   // На основании хеш-функции вырабатываем ключ
   if (!CryptDeriveKey(hCryptProv,aiCryptAlg,
      hHash,dwKeyFlags,phPBKey)) {
      printf("Error:
         CryptDeriveKey=0x%X.\n",GetLastError());
      Result=FALSE;
      goto ReleaseResource;
   }
// См. комментарий 6
   // Получаем размер ключа
   dwDataLen=sizeof(DWORD);
   if (!CryptGetKeyParam(*phPBKey,KP_KEYLEN,
      (PBYTE)&dwKeyLen,&dwDataLen,0)) {
      printf("Error:
         CryptGetKeyParam=0x%X.\n",GetLastError());
      Result=FALSE;
      goto ReleaseResource;
   }
   dwKeyLen >>= 3;
```

(см. след. стр.)

```
   // Получаем размер значения хеш-функции
   if  (!CryptGetHashParam(hHash,HP_HASHVAL,HashVal.pbData,
      &HashVal.cbData,0)) {
      printf("Error:
         CryptGetHashParam=0x%X.\n",GetLastError());
      Result=FALSE;
      goto ReleaseResource;
   }
   if (dwKeyLen< HashVal.cbData) {
// См. комментарий 7
      // Выделяем память для значения хеш-функции
      HashVal.pbData=LocalAlloc(LMEM_ZEROINIT,
         HashVal.cbData);
      if (!HashVal.pbData) {
         printf("Error: Out of memory.\n");
         Result=FALSE;
         goto ReleaseResource;
      }
      // Получаем значение хеш-функции
      if  (!CryptGetHashParam(
               hHash,
               HP_HASHVAL,
               HashVal.pbData,
               &HashVal.cbData,
               0» {
         printf("Error:
            CryptGetHashParam=0x%X.\n",GetLastError());
         Result=FALSE;
         goto ReleaseResource;
      }
      // Получаем размер блока шифрования
      dwDataLen=sizeof(DWORD);
      if  (!CryptGetKeyParam(
               *phPBKey,
               KP_BLOCKLEN,
               (PBYTE)&IV.cbData,
               &dwDataLen,
               0)) {
         printf("Error:
            CryptGetKeyParam=0x%X.\n",GetLastError());
         Result=FALSE;
         goto ReleaseResource;
      }
      IV.cbData >>= 3;
      // Выделяем память для синхропосылки
      IV.pbData=LocalAlloc(LMEM_ZEROINIT,IV.cbData);
```

(см. след. стр.)

```
        if (!IV.pbData) {
            printf("Error: Out of memory.\n");
            Result=FALSE;
            goto ReleaseResource;
        }
        // Копируем значение синхропосылки
        CopyMemory(
            IV.pbData,
            HashVal.pbData+dwKeyLen,
            (HashVal.cbData >= (dwKeyLen+IV.cbData)) ?
                        IV.cbData : (dwKeyLen+IV.cbData)-
                            HashVal.cbData);
        // Устанавливаем синхропосылку
        dwDataLen=sizeof(DWORD);
        if (!CryptSetKeyParam(*phPBKey, KP_IV, IV.pbData, 0)) {
            printf("Error:
                CryptGetKeyParam=0x%X.\n",GetLastError());
            Result=FALSE;
            goto ReleaseResource;
        }
    }
ReleaseResource:
    // Освобождаем выделенные ресурсы
    . . .
    return Result;
}
```

Теперь прокомментируем показанный фрагмент.

1. Параметры функции CryptPBKDFunction:

   hCryptProv

   [in] Дескриптор криптопровайдера.

   aiCryptAlg

   [in] Идентификатор алгоритма, генерируемого ключа,

   dwKey Flags

   [in] Набор флагов, используемых при генерации ключа. Флаги передаются в функцию ***CryptDeriveKey*** (см. Приложение 2).

   phPBKey

   [in/out] Указатель на дескриптор ключа. Через этот указатель функция возвращает дескриптор, сгенерированного ключа.

   pPassword

[in] Указатель на структуру PDATA_BLOB, содержащую указатель пароль (pPassword->pbData) и его размер (pPassword->cbData).

p*Salt*

[in/out] Указатель на структуру PDATA_BLOB, содержащую указатель на модификатор ключа (*pSalt->pbData*) и его размер (*pSalt->cbData*). Член структуры *pbData*, может быть равным NULL. Тогда функция сама генерирует случайное значение модификатора ключа и возвращает указатель в *pSalt->pbData*.

a*iHashAlg*

[m] Идентификатор алгоритма хеш-функции, используемой в функции генерации ключа.

d*wHashCount*

[in] Значение счетчика итераций, используемого в функции генерации ключа.

При успешном завершении функция возвращает TRUE, в противном случае возвращается FALSE. Если возвращается величина FALSE, соответствующий код ошибки может быть получен через функцию ***GetLastError***.

2. Перед тем как начать хеширование с помощью созданного объекта хеш-функции, проверяем указатель па модификатор ключа. Если указатель нулевой, то выделяем память, размера *pSalt->cbData*. Затем используем функцию ***CryptGenRandom*** для генерации случайной последовательности.
3. Сначала хешируем буфер, содержащий значение пароля.
4. Затем в том же объекте хеш-функции хешируем буфер, содержащий значение модификатора ключа. Примем хеширование модификатора ключа именно тем преобразованием, которое повторяется в соответствии со значением счетчика итераций. В нашем случае счетчик итераций — это параметр *dwHashCount*. Поэтому повторяем хеширование буфера *pSalt->pbData dwHashCount* еще раз.
5. Для получения ключа на основании хеш-функции используем процедуру ***CryptDeriveKey***. Алгоритм нового ключа определяется параметром *aiCryptAlg*, а длина ключа старшим словом параметра *dwKeyFlags*.
6. Криптопровайдеры Microsoft при получении ключа из хеш-функции в качестве ключа длины N бит используют пер-

вые N бит от значения хеш-функции. Поэтому зададим оставшееся значение хеш-функции в качестве синхропосылки к ключу. Для этого получим и сравним длины ключа и хеш-функции.

7. Если на ключ использовано не все значение хеш-функции, то устанавливаем оставшиеся биты хеш-функции в качестве синхропосылки.

На основании рассмотренной нами функции получения ключа и реализована программа обмена криптографическими сообщениями, с использованием пароля. Программа показана в листинге 2-5. Пароль, который используют отправитель и получатель, определен константой CRYPT_PASSWORD. Для обеспечения целостности и аутентичности рассчитываем по алгоритмам CBC-MAC или HMAC секретный хеш и включаем в криптографическое сообщение. Ключ для секретной хеш-функции также генерируется из пароля с использованием функции ***CryptPBKDFunction*** Для простоты считаем, что участники обмена заранее условились обо всех криптографических параметрах и в *сообщение* включаются только модификатор ключа и счетчик итераций для ключа шифрования и ключа секретной хеш-функции.

В этом и *всех* остальных примерах для работы с блоками данных широко используется структура DATA_BLOB (см. файл WinCrypt.h). Она содержит указатель па блок данных типа PBYTE (член структуры *pbData*) и размер данных типа DWORD (член структуры *cbData*), поэтому ее очень удобно использовать для хранения и манипуляции буферами данных с нефиксированным размером. Для сохранения в файле и считывания из файла структур DATA_BLOB в программах используются функции ***WriteFileBlob*** и ***ReadFileBlob***. Они принимают в качестве параметров указатель на файл и указатель на структуру.

Листинг 2-5. Текст программы CrMsgPswd

```
// См. комментарий 1
// Определение имени и типа рабочего провайдера
ttdefine   PROV_NAME            MS_DEF_PROV
#define PROV_TYPE               PROV_RSA_FULL
// Определение алгоритмов, используемых при получении ключа
// из пароля
«define KDF_HASH_ALG            CALG_SHA
#define KDF_SALT_LEN            12
#define KDF_HASH_COUNT          1000
```

(см. след. стр.)

```
// Определение алгоритмов шифрования и секретного хеша
#define CRYPT_ALG                CALG_RC2
#define CRYPT_CIPHER_MODE        CRYPT_MODE_CFB
«define BLOCK_SIZE               160
#define BUFFER_SIZE              (BLOCK_SIZE+16)
#define MAC_ALG                  CALG_HMAC
#define HMAC_HASH_ALG            CALG_MD5
// Определение длины ключа шифрования и флагов
«define CRYPT_KEY_LEN            56
«define CRYPT_RC2_EFFECTIVE_LEN  CRYPT_KEY_LEN
«define CRYPT_KEY_FLAGS          0
// Определение строки для шифрования и строки пароля
#define CRYPT_MSG                "The data that is to be
                                 encrypted and MAC'ed."
ttdefine CRYPT_PASSWORD    L"Test password"
// Определение имен используемых файлов
«define ORIGINATOR_PLAINTEXT_FILE   "Plaintext.org"
#define CIPHERTEXT_FILE             "Ciphertext.msg"
#define RECIPIENT_PLAINTEXT_FILE "Plaintext.rcp"
void main(void) {
   BOOL      Result=TRUE;
   FILE      *hOriginator=NULL,  *hCipher=NULL,
             *hRecipient=NULL;
   DWORD      Final=0;
   fpos_t fSavePos=0,  fSetPos=0;
   DATA_BLOB OriginatorMsg={strlen((char*)CRYPT_MSG)+1,
      (PBYTE)CRYPT_MSG};
   DATA_BLOB Password={wcslen((wchar_t*)CRYPT_PASSWORD)+2,
      (PBYTE)CRYPT_PASSWORD};
   DATA_BLOB Salt={KDF_SALT_LEN,NULL};
   DATA_BLOB MACVal={0,NULL};
   DATA_BLOB MACValChecked={0,NULL};    HMAC_INFO HMACInfo=
                                {HMAC_HASH_ALG,NULL,0,NULL,0};
   ALG_ID aiKDFHashAlg=KDF_HASH_ALG;
   DWORD         dwKDFHashCount=KDF_HASH_COUNT;
   DWORD         dwCipherMode=CRYPT_CIPHER_MODE;
   DWORD         dwEffectiveLen=CRYPT_RC2_EFFECTIVE_LEN;
   DWORD         dwDataLen=0;
   DWORDLONG     dwlEncryptDataLen=0,  dwlReadDataLen=0;
   HCRYPTPROV    hCryptProv=0;
   HCRYPTHASH    hHash=0,  hMAC=0;
   HCRYPTKEY     hCryptKey=0,  hMACKey=0;
   DWORD         dwKeyFlags=(CRYPT_KEY_LEN << 16)  |
                                CRYPT_KEY_FLAGS;
   BYTE          pbBuffer[BUFFER_SIZE];
   DWORD         dwCount;
```

(см. след. **стр.)**

```
// Первый этап
// Отправитель на основании пароля вырабатывает ключи
// шифрования и аутентификации и отправляет зашифрованное
// сообщение с аутентифицирующим хешем получателю
// См. комментарий 2
   // Открываем файл с открытым текстом отправителя
   if  ((hOriginator=fopen(ORIGINATOR_PLAINTEXT_FILE,
         "rb"))==NULL) {
      if  ((hOriginator=fopen(ORIGINATOR_PLAINTEXT_FILE,
         "wb"))==NULL) {
         printf("Error: Opening %s
            file.\n",ORIGINATOR_PLAINTEXT_FILE);
         Result=FALSE;
         goto  FirstPhaseReleaseResource;
      }
      fwrite(OriginatorMsg.pbData, OriginatorMsg.cbData, 1,
            hOriginator);
      if (ferror(hOriginator)) {
         printf("Error: Writing data to file.\n");
         Result=FALSE;
         goto  FirstPhaseReleaseResource;
      }
      fclose(hOriginator); hOriginator=0;
      if  ((hOriginator=fopen(ORIGINATOR_PLAINTEXT_FILE,
         "rb"))==NULL) {
         printf("Error: Opening %s
            file.\n",ORIGINATOR_PLAINTEXT_FILE);
         Result=FALSE;
         goto  FirstPhaseReleaseResource;
      }
   }
   // Открываем файл для записи шифртекста
   if((hCipher=fopen(CIPHERTEXT_FILE,"wb"))==NULL) {
      printf<"Error: Opening %s file.\n",CIPHERTEXT_FILE);
      Result=FALSE;
      goto   FirstPhaseReleaseResource;
   }
   // Открываем дескриптор ключевого контейнера отправителя
   if  (!CryptAcquireContext(
            &hCryptProv,NULL,PROV_NAME,PROV_TYPE,
               CRYPT_VERIFYCONTEXT)) {
      printf("Error: CryptAcquireContext=0x%X.\n",
            GetLastError());
      Result=FALSE;
      goto  FirstPhaseReleaseResource;
   }
```

(см. след. стр.)

```
// См. комментарий 3
   // На основании пароля и дополнительных параметров
   // создаем ключ шифрования
   if (!CryptPBKDFunction(
         hCryptProv,
         CRYPT_ALG,
         dwKeyFlags,
         &hCryptKey,
         &Password,
         &Salt,
         aiKDFHashAlg,
         dwKDFHashCount)) {
      Result=FALSE;
      goto FirstPhaseReleaseResource;
   }
// См. комментарий 4
   // Устанавливаем эффективную длину для ключа RC2
   if (CRYPT_ALG == CALG_RC2) {
      if (!CryptSetKeyParam(
               hCryptKey,KP_EFFECTIVE_KEYLEN,
                  (PBYTE)&dwEffectiveLen,0)) {
         printf("Error: CryptSetKeyParam=0x%X.\n",
            GetLastErrorO);
         Result=FALSE;
         goto  FirstPhaseReleaseResource;
      }
   I
   // Устанавливаем режим шифрования ключа шифрования
   //  сообщения
   if (!CryptSetKeyParam(hCryptKey,KP_MODE,
      (PBYTE)&dwCipherMode,0)) {
      printf("Error: CryptSetKeyParam=0x%X.\n",
            GetLastErrorO);
      Result=FALSE;
      goto FirstPhaseReleaseResource;
   }
// См, комментарий 5
   // Сохраняем в файле значение модификатора ключа
   // для ключа шифрования
   if (!WriteFileBlob(hCipher,&Salt)) {
      printf("Error: Writing data to file.\n");
      Result=FALSE;
      goto  FirstPhaseReleaseResource;
   }
   // Сохраняем в файле значение счетчика хеш-функций
```

(см. след. стр.)

```
    // для ключа шифрования
    fwrite(&dwKDFHashCount, 1, sizeof(DWORD), hCipher);
    if (ferror(hCipher)) {
       printf("Error: Writing data to file.\n");
       Result=FALSE;
       goto FirstPhaseReleaseResource;
    }
    LocalFree(Salt.pbData); Salt.pbData=NULL;
// См. комментарий 6
    // На основании пароля и дополнительных параметров
    // создаем ключ секретного хеша
    if (!CryptPBKDFunction(
          hCryptProv,
          CRYPT_ALG,
          dwKeyFlags,
          &hMACKey,
          &Password,
          &Salt,
          aiKDFHashAlg,
          dwKDFHashCount)) {
       Result=FALSE;
       goto FirstPhaseReleaseResource;
    }
// См. комментарий 7
    // Вырабатываем секретный хеш
    // Создаем объект хеш-функции для алгоритма MAC_ALG
    if (!CryptCreateHash(hCryptProv, MAC_ALG,
       hMACKey, 0, &hMAC)) {
       printf("Error: CryptCreateHash=0x%X.\n",
          GetLastError());
       Result=FALSE;
       goto FirstPhaseReleaseResource;
    }
    if (MAC_ALG == CALG_HMAC) {
       // Устанавливаем параметры алгоритма HMAC
       if (!CryptSetHashParam(hMAC, HP_HMAC_INFO,
          (PBYTE)&HMACInfo, 0)) {
          printf("Error: CryptSetHashParam=0x%X.\n",
             GetLastError0);
          Result=FALSE;
          goto FirstPhaseReleaseResource;
       }
    }
    // Сохраняем позицию в файле для записи длины
    // зашифрованных данных
    fgetpos(hCipher, &fSavePos);
```

(см, след. стр.)

```
   if (ferror(hOriginator)) {
      printf("Error: Get file position.\n");
      Result=FALSE;
      goto  FirstPhaseReleaseResource;
   }
   // Устанавливаем позицию в файле для записи
   //  зашифрованных данных
   fseek(hCipher,sizeof(DWORDLONG),SEEK_CUR);
   if (ferror(hCipher)) {
      printf("Error: Set file position.\n");
      Result=FALSE;
      goto  FirstPhaseReleaseResource;
   }
// См. комментарий 8
   // Зашифровываем данные из файла отправителя
   do {
      // Читаем из файла отправителя данные размером
      // BLOCK_SIZE байт
      dwCount = fread(pbBuffer, 1, BLOCK_SIZE,
         hOriginator);
      if(ferror(hOriginator)) {
      printf("Error: Reading data from file.\n");
         Result=FALSE;
         goto  FirstPhaseReleaseResource;
      }
      Final=feof(hOriginator);
      // Зашифровываем данные и рассчитываем значение
      // секретного хеша
      if (!CryptEncrypt(
               hCryptKey,
               hMAC,
               Final,
               0,
               pbBuffer,
               &dwCount,
               BUFFER_SIZE)) {
         printf("Error: CryptEncrypt=0x%X.\n",
            GetLastError());
         Result=FALSE;
         goto  FirstPhaseReleaseResource;
      }
      // Записываем зашифрованные данные в файл
      fwrite(pbBuffer, 1, dwCount, hCipher);
      if(ferror(hCipher)) {
         printf("Error: Writing data to file.\n");
         Result=FALSE;
```

(см. след. стр.)

```
            goto FirstPhaseReleaseResource;
         }
         dwlEncryptDataLen+=dwCount;
      } while(!Final);
// См. комментарий 9
      // Запрашиваем размер значения хеш-функции
      if (!CryptGetHashParam(hMAC,HP_HASHVAL,NULL,
         &MACVal.cbData,0)) {
         printf("Error: CryptGetHashParam=0x%X.\n",
            GetLastError());
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
// Выделяем память для значения хеш-функции
      MACVal.pbData=LocalAlloc(LMEM_ZEROINIT,MACVal.cbData);
      if (!MACVal.pbData) {
         printf("Error: Out of memory.\n");
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      // Запрашиваем значение хеш-функции
      if (!CryptGetHashParam(hMAC,HP_HASHVAL,MACVal.pbData,
            &MACVal.cbData,0)) {
         printf("Error: CryptGetHashParam=0x%X.\n",
            GetLastError());
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      // Сохраняем в файле значение модификатора ключа
      // для секретного хеша
      if (!WriteFileBlob(hCipher,&Salt)) {
         printf("Error: Writing data to file.\n");
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      // Сохраняем в файле значение счетчика хеш-функций
      // для секретного хеша
      fwrite(&dwKDFHashCount,1,sizeof(DWORD),hCipher);
      if (ferror(hCipher)) {
         printf("Error: Writing data to file.\n");
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      // Сохраняем в файле значение секретного хеша
      if (!WriteFileBlob(hCipher,&MACVal)) {
         printf("Error: Writing data to file.\n");
```

(см. след. стр.)

```
      Result=FALSE;
      goto FirstPhaseReleaseResource;
   }
   // Устанавливаем позицию в файле для сохранения
   // длины зашифрованных данных
   fsetpos(hCipher, &fSavePos);
   if (ferror(hCipher)) {
      printf("Error: Set file position.\n");
      Result=FALSE;
      goto FirstPhaseReleaseResource;
   }
   // Сохраняем в файле длину зашифрованных данных
   fwrite(&dwlEncryptDataLen, 1, sizeof(DWORDLONG),
      hCipher);
   if (ferror(hCipher)) {
      printf("Error: Writing data to file.\n");
      Result=FALSE;
      goto FirstPhaseReleaseResource;
   }
// См, комментарий 10
FirstPhaseReleaseResource:
   // Освобождаем открытые на первом этапе ресурсы
   // и обнуляем переменные
   . . .
   if (!Result) return;

// Второй этап
// Получатель на основании пароля вырабатывает ключи
// шифрования и аутентификации, зашифровывает
// сообщение и проверяет его целостность
// См. комментарий 11
   // Открываем файл с шифртекстом
   if ((hCipher=fopen(CIPHERTEXT_FILE,"rb"))==NULL) {
      printf("Error: Opening %s file.\n",CIPHERTEXT_FILE);
      Result=FALSE;
      goto SecondPhaseReleaseResource;
   }
   // Открываем файл для записи открытого текста получателя
   if ((hRecipient=fopen(RECIPIENT_PLAINTEXT_FILE,
      "wb"))==NULL)
      printf("Error: Opening %s file.\n",
         RECIPIENT_PLAINTEXT_FILE);
      Result=FALSE;
      goto SecondPhaseReleaseResource;
   }
   // Открываем дескриптор ключевого контейнера получателя
```

(см. след. стр.)

```
   if ( ! CryptAcquireContext(
            &hCryptProv, NULL, PROV_NAME, PROV_TYPE,
               CRYPT_VERIFYCONTEXT)) {
      Result=FALSE;
      goto  SecondPhaseReleaseResource;
   }
   // Считываем значение модификатора ключа для ключа
   // шифрования сообщения
   if  (!ReadFileBlob(hCipher,&Salt)) {
      printf("Error: Reading data from file.\n");
      Result=FALSE;
      goto  SecondPhaseReleaseResource;
   }
   // Считываем значение счетчика хеш-функций
   // для ключа шифрования сообщения
   fread(&dwKDFHashCount, 1, sizeof(DWORD),hCipher);
   if (ferror(hCipher) || feof(hCipher)) {
      printf("Error: Reading data from file.\n");
      Result=FALSE;
      goto SecondPhaseReleaseResource;
   }
// 0.  комментарий 12
   // На основании пароля и дополнительных параметров
   // создаем ключ шифрования
   if  (!CryptPBKDFunction(
         hCryptProv,
         CRYPT_ALG,
         dwKeyFlags,
         &hCryptKey,
         &Password,
         &Salt,
         aiKDFHashAlg,
         dwKDFHashCount)) {
      Result=FALSE;
      goto  SecondPhaseReleaseResource;
   }
   if  (CRYPT_ALG == CALG_RC2)  {
      if  (!CryptSetKeyParam(
               hCryptKey, KP_EFFECTIVE_KEYLEN,
                  (PBYTE)&dwEffectiveLen, 0)) {
         printf( "Error: CryptSetKeyParam=0x%X.\n",
            GetLastError());
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
   }
```

(*см.след. стр.*)

```
// Устанавливаем режим шифрования ключа шифрования
// сообщения
if  (!CryptSetKeyParam(hCryptKey,KP_MODE,
    (PBYTE)&dwCipherMode,0)) {
    printf("Error: CryptSetHashParam=0x%X.\n",
       GetLastError());
    Result=FALSE;
    goto  SecondPhaseReleaseResource;
}
LocalFree(Salt.pbData); Salt.pbData=NULL;
// Считываем длину зашифрованных данных
fread(&dwlEncryptDataLen,1,sizeof(DWORDLONG),hCipher);
if(ferror(hCipher) || feof(hCipher)) {
    printf("Error: Reading data from file.\n");
    Result=FALSE;
    goto SecondPhaseReleaseResource;
}
// Сохраняем позицию в файле для считывания
// зашифрованных данных
fgetpos(hCipher, &fSavePos);
if (ferror(hCipher)) {
    printf("Error: Get file position.\n");
    Result=FALSE;
    goto   SecondPhaseReleaseResource;
}
// Устанавливаем позицию в файле для считывания
// параметров секретного хеша
fSetPos=fSavePos+dwlEncryptDataLen;
fsetpos(hCipher, &fSetPos);
if (ferror(hCipher)) {
    printf("Error: Set file position.\n");
    Result=FALSE;
    goto  SecondPhaseReleaseResource;
}
// Считываем значение модификатора ключа для ключа
// секретного хеша
if (!ReadFileBlob(hCipher,&Salt)) {
    printf("Error: Reading data from file.\n");
    Result=FALSE;
    goto  SecondPhaseReleaseResource;
}
// Считываем значение счетчика хеш-функций для ключа
// секретного хеша
fread(&dwKDFHashCount,1,sizeof(DWORD),hCipher);
if(ferror(hCipher) || feof(hCipher)) {
    printf("Error: Reading data from file.\n");
```

(см. след. стр.)

```
      Result=FALSE;
      goto SecondPhaseReleaseResource;
   }
// См. комментарий 13
   // На основании пароля и дополнительных параметров
   // создаем ключ секретного хеша
   if (!CryptPBKDFunction(
         hCryptProv,
         CRYPT_ALG,
         dwKeyFlags,
         &hMACKey,
         &Password,
         &Salt,
         aiKDFHashAlg,
         dwKDFHashCount)) {
      Result=FALSE;
      goto SecondPhaseReleaseResource;
   }
   // Вырабатываем секретный хеш
   // Создаем объект хеш-функции для алгоритма MAC_ALG
   if (!CryptCreateHash(hCryptProv,MAC_ALG,
      hMACKey,0,&hMAC)) {
      printf("Error: CryptCreateHash=0x%X.\n",
         GetLastError());
      Result=FALSE;
      goto SecondPhaseReleaseResource;
   }
   if (MAC_ALG == CALG_HMAC) {
      // Устанавливаем параметры алгоритма HMAC
      if (!CryptSetHashParam(hMAC,HP_HMAC_INFO,
         (PBYTE)&HMACInfo,0)) {
         printf("Error: CryptSetHashParam=0x%X.\n",
            GetLastErrorO);
         Result=FALSE;
         goto SecondPhaseReleaseResource;
      }
   }
   // Считываем значение хеш-функции вычисленной
   // отправителем
   if (!ReadFileBlob(hCipher,&MACVal)) {
      printf("Error: Reading data from file.\n");
      Result=FALSE;
      goto SecondPhaseReleaseResou rce;
   }
   // Сравниваем длину значений хеш-функции отправителя
   // и получателя
```

(см. след. стр.)

```
   if (MACVal.cbData != MACValChecked.cbData) {
      printf("Ciphertext file is corrupt!\n");
      Result=FALSE;
      goto SecondPhaseReleaseResource;
   }
   // Устанавливаем позицию в файле для считывания
   // зашифрованных данных
   fsetpos(hCipher, &fSavePos);
   if (ferror(hCipher)) {
      printf("Error: Set file position.\n");
      Result=FALSE;
      goto SecondPhaseReleaseResource;
   }
// См. комментарий 14
   // Расшифровываем файл
   do {
      // Считываем из файла блок размером BLOCK_SIZE байт
      dwCount = fread(pbBuffer,1,BLOCK_SIZE,hCipher);
      if(ferror(hCipher)) {
         printf("Error: Reading data from file.\n");
         Result=FALSE;
         goto SecondPhaseReleaseResource;
      }
      // Расшифровываем данные и рассчитываем значение
      // секретного хеша
      dwlReadDataLen+=dwCount;
      if (dwlReadDataLen > dwlEncryptDataLen) {
         Final=TRUE;
         dwDataLen=dwCount-(DWORD)(dwlReadDataLen-
            dwlEncryptDataLen);
      }
      else
      dwDataLen=dwCount;
      if(!CryptDecrypt(
               hCryptKey,
               hMAC,
               Final,
               0,
               pbBuffer,
               &dwDataLen)) {
         printf("Error: CryptDecrypt=0x%X.\n",
            GetLastError());
         Result=FALSE;
         goto SecondPhaseReleaseResource;
      }
      // Записываем данные в файл получателя сообщения
```

(см. след. стр.)

```
        fwrite(pbBuffer,1,dwDataLen,hRecipient);
        if(ferror(hRecipient)) {
           printf("Error: Writing data to file.\n");
           Result=FALSE;
           goto SecondPhaseReleaseResource;
        }
     } while(!Final);
// Запрашиваем размер значения хеш-функции
     if (!CryptGetHashParam(hMAC,HP_HASHVAL,NULL,
        &MACValChecked.cbData,0)) {
        printf("Error: CryptGetHashParam=0x%X.\n",
           GetLastError());
        Result=FALSE;
        goto SecondPhaseReleaseResource;
     }
     // Выделяем память для рассчитываемой хеш-функции
     MACValChecked.pbData=LocalAlloc(LMEM_ZEROINIT,
        MACValChecked.cbData);
     if (!MACValChecked.pbData) {
        printf("Error: Out of memory.\n");
        Result=FALSE;
        goto SecondPhaseReleaseResource;
     }
     // Считываем значение хеш-функции
     if (!CryptGetHashParam(
              hMAC,HP_HASHVAL,MACValChecked.pbData,
                 &MACValChecked.cbData,0)) {
        printf("Error: CryptGetHashParam=0x%X.\n",
           GetLastError());
        Result=FALSE;
        goto SecondPhaseReleaseResource;
     }
// См. комментарий 15
     // Сравниваем размеры хеш-функций
     if (MACVal.cbData != MACValChecked.cbData) {
        printf("File MAC failed to validate!\n");
        Result=FALSE;
        goto SecondPhaseReleaseResource;
     }
     // Сравниваем значение хеш-функций отправителя
     // и получателя
     if (memcmp(MACVal.pbData,MACValChecked.pbData,
        MACValChecked.cbData)) {
        printf("File HAC failed to validate!\n");
        Result=FALSE;
        goto SecondPhaseReleaseResource;
```

(см. след. стр.)

```
    }
    printf("File MAC validated Ok!\n");
SecondPhaseReleaseResource:
    // Освобождаем открытые на втором этапе ресурсы
    // и обнуляем переменную
    . . .
    return;
}
```

Теперь прокомментируем показанный фрагмент.

1. Определение констант, используемых в программе. Помимо криптографических параметров, здесь определены следующие константы:
   * CRYPT_MSG — строка с открытым текстом. Эта строка используется, если отправитель заранее не создал файл с открытым текстом;
   * CRYPT_PASSWORD - строка пароля;
   * ORIGINATOR_PLAINTEXT_FILE - имя файла, в котором находится открытый текст;
   * CIPHERTEXT_FILE — имя файла, в который помещается зашифрованное сообщение, значение секретного хеша и дополнительные параметры;
   * RECIPIENT_PLAINTEXT_FILE - имя файла, в который получатель помещает расшифрованное сообщение.
2. Начинается первый этап — этап формирования отправителем криптографического сообщения. Отправитель открывает файл ORIGINATOR_PLAINTEXT_FILE с открытым текстом. Если он отсутствует, то создается новый файл и в нем сохраняется строка CRYPT_MSG.
3. Генерируем ключ шифрования сообщения. Параметрами генерации являются:
   * CRYPT_ALG — алгоритм ключа шифрования;
   * *dwKeyFlags* — старшее слово этой переменной, в котором содержится длина ключа, инициализирована константой CRYPT_KEY_LEN;
   * *Password* — структура типа DATA_BLOB, инициализирована указателем на пароль и его длиной;
   * *Salt* — структура типа DATA_BLOB. Инициализирован только параметр Salt.cbData константой KDF_SALT_LEN, поэтому модификатор ключа будет сгенерирован

в функции ***CryptPBKDFunction*** и указатель будет возвращен в параметре *Salt.pbData*;

* *aiKDFHashAlg* – переменная, идентифицирующая алгоритм хеш-функции, используемой при генерации ключа из пароля, инициализирована константой KDF_HASH_ALG;
* dwKDFHashCount - переменная, идентифицирующая счетчик итераций, инициализирована константой KDF_HASH_COUNT.

Дескриптор созданного ключа шифрования возвращается в переменной *hCryptKey*.

4. Через вызов функции ***CryptSetKeyParam*** (см. Приложение 2) устанавливаем параметры ключа шифрования.

   Если используется алгоритм RC2, то устанавливаем значение эффективной длины для ключа шифрования RC2. Влияние значения эффективной длины ключа на стойкость шифрования по алгоритму RC2 полностью изложено в описании алгоритма (см. [25]).

   Здесь отметим главное. Реальная стойкость шифрования по алгоритму RC2 эквивалентна наименьшему из значений длины ключа и эффективной длины ключа. В криптопровайдерах Microsoft, в том числе и «enchanced» криптопровайдерах, которые *поддерживают шифрование* по алгоритму RC2 с длиной ключа до 128 бит, по умолчанию значение эффективной длины ключа равно 40 битам. Это означает, что при генерации ключа RC2 с помощью функций ***CryptGenKey*** или ***CryptDeriveKey*** (см. Приложение 2), даже если вы задали максимальную длину ключа 128 бит, эффективная длина ключа раина 40 битам и реальная стойкость шифрования на этом ключе будет эквивалентна 40 битам. Для изменения значения эффективной длины ключа необходимо вызвать функцию ***CryptSetKeyParam*** и задать для параметра KP_EFFECTIVE_KEYLEN значение не меньше длины самого ключа.

   В нашей программе значение эффективной длины ключа устанавливается переменной *dwEffectiveLen* инициализированной константой CRYPT_RC2_EFFECTIVE_LEN, которая эквивалента *константе* CRYPT_KEY_LEN. Это означает, что в нашей программе значение эффективной длины ключа для алгоритма RC2 равно длине самого ключа,

Второй устанавливаемый параметр — режима шифрования. Задастся для всех алгоритмов шифрования. Вообще говоря, в «боевых» программах рекомендуется использовать режим простой замены с зацеплением (CRYPT_MODE_CBC), поскольку он обладает наилучшими характеристиками. Этот же режим устанавливается по умолчанию при генерации ключа. Но поскольку рассматриваются все возможности использования функций CryptoAPI, то и режим шифрования можно изменять через константу CRYPT_CIPHER_MODE.

5. Первое, что мы сохраняем в файле с криптографическим сообщением, — параметры, необходимые для генерации ключа шифрования из пароля. С помощью функции ***WriteFileBlob*** сохраняем модификатор ключа шифрования, а затем записываем счетчик итераций. Как уже отмечалось, мы считаем, что алгоритм, длина ключа и режим шифрования сторонами согласованы заранее.

6. Генерируем ключ секретного хеша. Параметры генерации аналогичны использованнным при генерации ключа шифрования. Модификатор ключа секретного хеша также генерируется в функции ***CryptPBKDFunction*** и указатель будет возвращен в параметре *Salt.pbData*.

   Дескриптор созданного ключа возвращается в переменной *hMACKey*.

7. На основании сгенерированного ключа создаем объект хеш-функции. Объект секретной хеш-функции создастся аналогично обычной хеш-функции, через вызов функции ***CryptCreateHash*** (см. Приложение 2). Отличие заключается в том, что устанавливается дескриптор секретного ключа *hKey*. В нашем случае в функцию передается дескриптор *hMACKey*.

   Если используется алгоритм секретного хеша HMAC, потребуется выполнить дополнительное действие. Дело в том, что для алгоритма HMAC (см. [29]) требуется определить базовый алгоритм хеш-функции, который он будет использовать. В настоящее время в качестве базовых, как правило, применяют SHA-1 или MD5, тогда алгоритм именуется HMAC-SHA-1 или HMAC-MD5 соответственно. Кроме того, у алгоритма имеются параметры *ipad* (inner pad) и *opad* (outer pad), которые также разрешается изменять. Для установки этих параметров после создания объекта хеш-функции необходимо вызвать функцию ***CryptSetHashParam*** (см. Приложение 2) с параметром HP_HMAC_INFO и ука-

зателем на структуру HMAC_INFO. В структуре HMAC_INFO (см. файл WinCrypt.h) следует установить алгоритм базового хеша, в нашей программе он определен константой HMAC_HASH_ALG. Параметры *InnerString* и *OuterString* можно не инициализировать, тогда будут использоваться значения по умолчанию, эти значения описаны в [29] и в справочнике Microsoft Platform Software Development Kit (SDK).

8. Выполняется цикл зашифрования открытого текста отправителя.

   Данные считываются из файла в буфер размером BUFFER_SIZE байт блоками размером BLOCK_SIZE байт. Размер BUFFER_SIZE установлен больше, чем BLOCK_SIZE, поскольку в блочных режимах шифрования (режим простой замены, режим простой замены с зацеплением) последний блок данных *шифруется* с *использованием дополнения* (padding). В криптопровайдерах Microsoft используется режим дополнения PKCS5_PADDING, описанный в [35]. При этом максимальный размер дополнения равен размеру блока шифрования.

   Затем буфер передается на шифрование. Кроме дескриптора ключа и размеров данных и буфера шифрования, в функцию шифрования ***CryptEncrypt*** (см. Приложение 2) передается дескриптор объекта секретной хеш-функции. В этом случае функция ***CryptEncrypt*** перед зашифрованием хеширует открытый текст.

   С выхода функции шифрования шифртекст записывается в файл криптографического сообщения. Кроме того, увеличивается переменная *dwlEncryptDataLen*, которая накапливает размер зашифрованных данных. Еще раз отметим, что размер шифртекста в блочных режимах шифрования больше, чем исходный открытый текст. Поскольку максимальный размер открытого текста определяется максимальным размером файла, то используем переменную типа DWORDLONG.

9. Завершаем формирование криптографического сообщения.

   Устанавливаем размер хеш-функции, выделяем память и считываем в буфер *MACVal.pbData* значение секретной хеш-функции. После этого сохраняем в файле параметры генерации ключа секретного хеша (модификатор ключа, счетчик итераций) и значение самого хеша.

Последнее, что мы записываем в файл, — размер шифртекста. Поскольку текущий указатель указывает на конец файла, то предварительно смещаем указатель па заранее сохраненную для этого позицию.

10. Файл с криптографическим сообщение сформирован. В итоге сформированный файл содержит:
    * размер и значение модификатора ключа, значение счетчика итераций для генерации ключа шифрования;
    * размер шифртекта, сам шифртекст;
    * размер и значение модификатора ключа, значение счетчика итераций для генерации ключа секретной хеш-функции;
    * размер и значение секретной хеш-функции.
11. Здесь начинается второй этап работы. Получатель принимает криптографическое сообщение от отправителя. Он располагает паролем и некоторыми заранее согласованными параметрами шифрования и секретной хеш-функции. Из файла с криптографическим сообщением помимо шифртекста он получает параметры, перечисленные в комментарии 10.
12. Считав из файла модификатор ключа и счетчик итераций для генерации ключа шифрования, получатель, как и отправитель, вызывает функцию ***CryptPBKDFunction***Только в данном случае структура Salt инициализирована указателем на буфер и размером модификатора ключа.

    Дескриптор созданного получателем ключа шифрования возвращается в переменной *hCryptKey*.
13. Считав из файла модификатор ключа и счетчик итераций для генерации ключа секретной хеш-функций, получатель, аналогично отправителю, вызывает функцию ***CryptPBKDFunction.*** Только в данном случае структура Salt инициализирована указателем на буфер и размером модификатора ключа.

    Дескриптор созданного получателем ключа возвращается в переменной *hMACKey*.
14. Выполняется цикл расшифрования шифртекста.

    Считываем шифртекст блоками BLOCK_SIZE байт. Поскольку файл криптографического сообщения не заканчивается шифртскстом, то па основе размера шифртекста и накопленной длины расшифрованных данных определяем размер блока данных, передаваемых на расшифрование.

Кроме дескриптора ключа и размеров данных в функцию расшифрования *CryptDecrypt* (см. Приложение 2) передается дескриптор объекта секретной хеш-функции. В этом случае функция *CryptEncrypt* после расшифрования хеширует открытый текст.

Расшифрованные данные записываем в файл с открытым текстом получателя.

15. Сравниваем значения считанной из файла и вычисленной секретных хеш-функций. Сначала сравним размеры хеш-функций: если они не совпадают, то контроль целостности не пройден. Далее сравниваем непосредственно значения хеш-функций: если значения совпадают, то контроль целостности пройден успешно.

Таким образом, формат данного криптографического сообщения обеспечивает конфиденциальность с помощью шифрования, а также целостность и аутентичность с помощью секретной хеш-функции. Разумеется, условия эти выполняются только в случае доверительных отношений получателя и отправителя. Если вы используете представленную схему просто для шифрования файлов на пароле, то есть сами являетесь и отправителем и получателем, то результат оправдает каши надежды. Кроме того, необходимо быть уверенным, что пароль не был перехвачен программными или техническими средствами (при помощи программной закладки или средств визуального наблюдения). При многосторонних отношениях участники обмена должны быть уверены, что рабочий ключ (в данном случае пароль) по тем или иным причинам не стал доступным третьему лицу.

Использование CryptoAPI 1.0 для реализации схем несимметричного шифрования

Теперь рассмотрим задачи реализации схем несимметричного шифрования. В данном параграфе мы расскажем о программе, реализующей две схемы: несимметричного шифрования по алгоритму RSA и несимметричного обмена ключами по алгоритму Диффи — Хеллмана. Именно эти две схемы применяются чаще всего, кроме того, именно они поддерживаются в настоящее время криптопровайдерами Microsoft. За основу для реализации примера мы взяли стандарт Cryptographic Message Syntax из [30]. Поскольку мы используем Crypto API 1.0, то вместо сертификатов в нашей программе реализуется обмен

открытыми ключами ключевых пар обмена и структура самого криптографического сообщения очень упрощена. Несмотря на это, в примере отработаны все необходимые этапы для реализации обмена криптографическими сообщениями на основе алгоритмов RSA и Диффи — Хеллмана. Исходный текст программы находится в файле Chapter2/CrMsgKeyX/CrMsgKeyX.c на Web-сайте издательства «Русская Редакция».

Хотелось бы, чтобы, прежде чем приступать к изучению материалов данного параграфа, читатель ознакомился с описаниями алгоритмов RSA и Диффи — Хеллмана, поскольку мы подробно расскажем лишь о реализации обмена на этих алгоритмах в рамках интерфейса CryptoAPI 1.0. Их описание имеется во многих источниках, рекомендуем вам обратиться к [33], [34J или [24|.

Прежде всего, рассмотрим процедуру генерации ключевых пар, которая применяется и в этом, и в следующем параграфе. Процедура носит название ***CryptAcquireContextEx***. Из названия видно, что она является расширением функции ***CryptAcquireContext*** (см. Приложение 2). Действительно, она открывает или создает ключевой контейнер, если такового еще нет, и возвращает дескриптор этого ключевого контекста. При этом устанавливается наличие в контейнере соответствующей ключевой пары и в случае необходимости ее генерация. Текст процедуры ***CryptAcquireContextEx*** показан в листинге 2-6.

Листинг 2-6. Текст процедуры CryptAcquireContextEx

```
// См. комментарий 1
BOOL
CryptAcquireContextEx(
    HCRYPTPROV   *phProv,
    LPTSTR pszContainer,
    LPTSTR pszProvider,
    DWORD     dwProvType,
    DWORD     dwFlags,
    DWORD     dwKeySpec,
    DWORD     dwKeyFlags
    )
{
    BOOL            Result=TRUE;
    DWORD         dwError;
    HCRYPTPROV  hCryptProv=0;
    HCRYPTKEY    hKey;
    DWORD       dwDataLen=0, dwKeyLen=0;
```

(си. *след. стр.)*

```
// См, комментарий 2
   // Пытаемся открыть ключевой контейнер
   if (!CryptAcquireContext(
         &hCryptProv,
         pszContainer,
         pszProvider,
         dwProvType,
         dwFlags)) {
      dwError=GetLastError();
      if ( dwError!=NTE_BAD_KEYSET) {
         printf( "Error: CryptAcquireContext=0x%X.\n",
            dwError);
         Result=FALSE;
         goto  ReleaseResource;
      }
      else {
         // Если контейнер не существует, создаем новый
         if ( !CryptAcquireContext(
               &hCryptProv,
               pszContainer,
               pszProvider,
               dwProvType,
               dwFlags | CRYPT_NEWKEYSET)) {
            printf("Error: CryptAcquireContext=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto  ReleaseResource;
         }
      }
   }
// См. комментарий 3
   if (dwKeySpec) {
      // Пытаемся открыть ключевую пару, заданную
      // параметром dwKeySpec
      if (!CryptGetUserKey(hCryptProv,dwKeySpec, &hKey)) {
         // Если ключевая пара в контейнере отсутствует,
         // то создаем ее
         if ( !CryptGenKey(hCryptProv,dwKeySpec,
            dwKeyFlags,&hKey)) {
            printf( "Error: CryptGenKey=0x%X.\n",
               GetLastErrorO);
            Result=FALSE;
         }
         goto  ReleaseResource;
      }
```

(см. след. стр.)

```
// См, комментарий 4
      // Если удалось открыть существующую ключевую пару,
      // то сравниваем длину ключа с определенной в старшем
      // слове параметра dwKeyFlags
      dwDataLen=sizeof(DWORD);
      if  (!CryptGetKeyParam(hKey,KP_KEYLEN,
          (PBYTE)&dwKeyLen,&dwDataLen,0)) {
          printf("Error: CryptGetKeyParam=0x%X.\n",
              GetLastError());
          Result=FALSE;
          goto ReleaseResource;
      }
      if (dwKeyLen != (dwKeyFlags & KEY LENGTH_MASK) >> 16)
{
          CryptDestroyKey(hKey); hKey=0;
          // Если длина ключа ие соответствует
          // запрашиваемой, то создаем новую ключевую пару
          if  (!CryptGenKey(hCryptProv,dwKeySpec,
              dwKeyFlags,&hKey)) {
              printf("Error: CryptGenKey=0x%X.\n",
                  GetLastErrorO);
              Result=FALSE;
              goto ReleaseResource;
          }
      }
   }
ReleaseResource:
   // Освобождаем дескриптор ключа
   if (hKey) CryptDestroyKey(hKey);
   if (!Result) {
      // Освобождаем ключевой контекст
      if (hCryptProv) CryptReleaseContext(hCryptProv,0);
hCryptProv=0;
   }
   // Возвращаем дескриптор ключевого контейнера
   *phProv=hCryptProv;
   return Result;
}
```

Теперь прокомментируем показанный **фрагмент.**

1. Параметры функции CryptAcquireContextEx:

   phProv

   [in/out] Указатель на дескриптор криптопровайдера.

   pszContainer

   [in] **Имя** ключевого контейнера.

pszProvider

[in] Имя криптопровайдера.

dwProvType

[in] Значение типа запрашиваемого криптопровайдера.

dwFlags

[in] Значение флагов, передаваемых в функцию ***CryptAcquireContext*** (см. Приложение 2).

dwKeySpec

[in] Идентификатор ключевой пары (AT_SIGNATURE или AT_KEYEXCHANGE). Функция проверит наличие в контейнере ключевой пары, соответствующей идентификатору, и в случае ее отсутствия сгенерирует новую.

dwKeyFlags

[in] Ключевые флаги. Передаются в функцию ***CryptGenKey*** (см. Приложение 2), если необходимо генерировать новую ключевую пару. В старшем слове флаги должны в явном виде содержать значение запрашиваемой длины ключа. Это необходимо. если в контейнере уже существует требуемая ключевая пара. Функция сравнит запрашиваемую длину ключа с длиной ключа и существующей ключевой паре и в случае несовпадения сгенерирует новую.

При успешном завершении функция возвращает TRUE, в противном случае возвращается FALSE. Если возвращается величина FALSE, соответствующий код ошибки может быть получен через функцию ***GetLastError***.

2. Пытаемся открыть ключевой контейнер. Вызываем функцию ***CryptAcquire Context*** с переданными параметрами *pszContainer*, *pszProvider*, *dwProvType* и *dwFlags*. Если возвращается ошибка NTE_BAD_KEYSET, то ключевой контейнер, определенный параметром pszContainer, отсутствует. Для создания нового ключевого контейнера снова вызываем функцию ***CryptAcquireContext***, при этом к параметру *dwFlags* через операцию поразрядного ИЛИ присоединяем флаг создания нового контейнера CRYPT_NEWKEYSET.
3. Если параметр *dwKeySpec* не нулевой, то необходимо проверить наличие соответствующей ключевой пары. Для получения дескриптора ключевой пары, соответствующей идентификатору *dwKeySpec*, вызываем функцию ***CryptGetUserKey*** (см. Приложение 2).

4. Если функция возвращает ошибку, то запрашиваемая ключевая пара в контейнере отсутствует. В этом случае для генерации новой ключевой пары вызываем функцию ***CryptGenKey*** с параметрами *dwKeySpec* и *dwKeyFlags*.
5. Если запрашиваемая ключевая пара присутствует в контейнере, то необходимо сравнить длину ключа с запрашиваемой. Запрашиваемая длина ключа содержится и старшем слове параметра *dwKeyFlags*. Длину ключа существующей ключевой пары получаем через запрос функции ***CryptGetKeyParam*** (см. Приложение 2) с параметром KP_KEYLEN.
6. Если полученные длины не совпадают, то генерируем новую ключевую пару с необходимыми параметрами.

Именно из-за сравнения длины ключей параметр dwKeyFlags должен содержать значение запрашиваемой длины ключа в явном виде. Старшее слово флагов может быть и нулевым, тогда генерируется ключ с длиной, используемой по умолчанию. Но в различных криптопровайдерах длина, задаваемая по умолчанию, различна. Для установки ее пришлось бы сгенерировать новую ключевую пару с нулевым старшим словом флагов и после этого запросить длину ключа или выполнить последовательное перечисление алгоритмов криптопровайдера (см. раздел «Получение информации о криптопровайдерах, установленных в системе»). И то, и другое довольно громоздко и в данном случае не актуально, поэтому мы не стали реализовывать эту возможность.

Теперь рассмотрим общие принципы формирования криптографических сообщений в рамках несимметричных ключевых схем и особенности реализации алгоритмов RSA и Диффи — Хеллмана. Напомним, что эти принципы полностью изложены в [30].

Тип криптографического сообщения, которое формируется в программе, носит название *конвертированные данные* (Enveloped-data). Криптографическое сообщение этого тина содержит *зашифрованное сообщение* (encryptedcontent) и *зашифрованные ключи шифрования сообщения* (encrypted content-encryption keys) для одного или нескольких получателей. Комбинация зашифрованного содержание сообщения и один зашифрованный ключ шифрования сообщения для получателя и есть *цифровой конверт* (digital envelope) для этого получателя. Для простоты в нашем примере показано формирование криптографического сообщение только для одного получателя. Ясно,

что несимметричные алгоритмы применяются здесь для формирования зашифрованного ключа шифрования сообщения и именно в этой части отличаются форматы сообщения для алгоритмов RSA и Диффи — Хеллмана.

В предыдущем примере предполагается, что отправитель и получатель заранее согласовали параметры алгоритма шифрования, который они используют. Для текущего примера мы не делаем подобных допущений и включим в сообщение всю необходимую информацию об используемых алгоритмах шифрования. С этой целью необходимо определить структуру, описывающую алгоритм *шифрования* со всеми *сопутствующими* параметрами, и процедуры записи этой структуры в файл и считывания се из файла криптографического сообщения. В нашем примере структурой, идентифицирующей алгоритм шифрования, является структура CIPHER_ALGORITHM_IDENTIFIER (см. листинг 2-7). Как видно из описания структуры, она содержит полный набор параметров, описывающих алгоритм шифрования: идентификатор алгоритма шифрования, режим шифрования, длину ключа, параметр IV (initialize vector) типа DATA_BLOB, а также длину и указатель на буфер синхропосылки. Кроме того, определены процедуры записи структуры CIPHER_ALGORITHM_IDENTIFIER в файл ***WriteFileAID*** и считывания из файла ***ReadFileAID***. В качестве параметров обе функции принимают указатель на файл и указатель па обрабатываемую структуру.

В нашем примере все этапы, которые необходимо выполнить отправителю и получателю для отправки и приема сообщения в рамках схем несимметричного обмена ключами, объединены в одну программу. Однако для удобства мы будем рассматривать и комментировать каждый этак отдельно. Определение констант и структур, используемых в программе, приведено в листинге 2-7. Их *назначение* и *использование* мы поясним, комментируя этапы.

Листинг 2-7. Определение констант, структур и переменных программы CrMsgKeyX

```
// Определение имени и типа рабочего криптопровайдера
#define PROV_NAME                    MS_DEF_PROV
ttdefine   PROV_TYPE                 PROV_RSA_FULL
// Определение длины ключа обмена и его флагов
#define EXCHANGE_KEY_LEN             512
ttdefine   EXCHANGE_KEY_FLAGS        0
// Определение идентификатора алгоритма шифрования
```

(см. след. стр.)

```
// сообщения, длины ключа, режима шифрования и флагов
ttdeflne CONTENT_ENCRYPT_ALG            CALG_RC2
#define  CONTENT_ENCRYPT_MODE           CRYPT_MODE_CBC
#define  CONTENT_ENCRYPT_LEN            56
#define  CONTENT_ENCRYPT_FLAGS      CRYPT_EXPORTABLE
// Определение идентификатора алгоритма, используемого
// для шифрования ключа сообщения, длины ключа, режима
// шифрования и флагов
«define  KEY_ENCRYPT_ALG                CALG_RC2
#define  KEY_ENCRYPT_MODE               CRYPT_MODE_CBC
#define  KEY_ENCRYPT_LEN                56
#define  KEY_ENCRYPT_FLAGS              0
// Определение схемы шифрования для алгоритма RSA
«define  RSA_ENCRYPT_FLAGS              0
// Размер блока считываемых из файла данных
// и рабочего буфера
#define  BLOCK_SIZE                     160
«define  BUFFER_SIZE                    (BLOCK_SIZE+16)
// Определение сообщения, используемого по умолчанию
#define  CRYPT_MSG      "The data that is to be encrypted."
// Определение имени файла с открытым ключом получателя
#define  RECIPIENT_PUBKEY_FILE      "PublicKey.rcp"
// Определение имени файла с открытым текстом
#define  ORIGINATOR_PLAINTEXT_FILE  "Plaintext.org"
// Определение имени файла с криптографическим сообщением
#define  CIPHERTEXT_FILE            "Ciphertext.msg"
// Определение имени файла с текстом, полученным после
// обработки получателем криптографического сообщения
#define  RECIPIENT_PLAINTEXT_FILE   "Plaintext.rcp"
// Определение имени криптографического контейнера
// получателя
#define  RECIPIENT_CONTEXT           "Recipient"
// Определение структуры, идентифицирующей алгоритм
// шифрования со всеми необходимыми параметрами
typedef struct _CIPHER_ALGORITHM_IDENTIFIER_ {
   ALG_ID       aiAlgorithm;     // Идентификатор алгоритма
   DWORD        dwCipherMode;    // Режим шифрования
   DWORD        dwKeyLen;        // Длина ключа
   DATA_BLOB    IV;              // Синхропосылка
} CIPHER_ALGORITHM_IDENTIFIER, *PCIPHER_ALGORITHM_IDENTIFIER;
// Определение структур криптографического сообщения
typedef struct _KEY_TRANS_RECIPIENT_INFO_ {
   DATA_BLOB EncryptedKey;
} KEY_TRANS_RECIPIENT_INFO, *PKEY_TRANS_RECIPIENT_INFO;
typedef struct _KEY_AGREE_RECIPIENT_INFO_ <
   DATA_BLOB                        OriginatorPublicKey;
```

(см. след. стр.)

```
   CIPHER_ALGORITHM_IDENTIFIER  KeyEncryptionAID;
   DATA_BLOB
RecipientEncryptedKey;
} KEY_AGREE_RECIPIENT_INFO, *PKEY_AGREE_RECIPIENT_INFO;
typedef union _RECIPIENT_INFO_ {
   KEY_TRANS_RECIPIENT_INFO  KeyTransRecipientInfo;
   KEY_AGREE_RECIPIENT_INFO  KeyAgreeRecipientInfo;
} RECIPIENT_INFO, *PRECIPIENT_INFO;
typedef struct _ENCRYPTED_CONTENT_INFO_ {
   CIPHER_ALGORITHM_IDENTIFIER  ContentEncryptionAID;
} ENCRYPTED_CONTENT_INFO, *PENCRYPTED_CONTENT_INFO;
typedef struct _ENVELOPED_DATA_ {
   ALG_ID                    aiKeyXAlg;
   RECIPIENT_INFO            RecipientInfo;
   ENCRYPTED_CONTENT_INFO    EncryptedContentInfo;
} ENVELOPED_DATA, *PENVELOPED_DATA;
// Определение и инициализация переменных
   BOOL                  Result=TRUE;
   FILE                  *hOriginator=NULL, *hCipher=NULL,
                         *hRecipient=NULL;
   HMODULE               hDll=0;
   DWORD                 Final=0;
   fpos_t                fSavePos=0, fSetPos=0;
   DATA_BLOB
OriginatorMsg={strlen((char*)CRYPT_MSG)+1,(PBYTE)CRYPT_MSG};
   ENVELOPED_DATA      EnvelopedData;
   DWORD               dwDataLen=0;
   DWORDLONG           dwlEncryptDataLen=0, dwlReadDataLen=0;
   DATA_BLOB           RecipientPubKeyBlob={0,NULL};
   HCRYPTPROV          hOriginatorProv=0, hRecipientProv=0;
   HCRYPTKEY           hContentKey=0, hRecipientKeyX=0;
   HCRYPTKEY           hOriginatorKeyX=0; hAgreedKey=0;
   DWORD               dwExportKeyFlags=0;
   DWORD               dwKeyXFlags=(EXCHANGE_KEY_LEN << 16) |

      EXCHANGE_KEY_FLAGS;
   DWORD
dwContentEncryptFlags=(CONTENT_ENCRYPT_LEN << 16) |

      CONTENT_ENCRYPT_FLAGS;
   DWORD          dwKeyEncryptFlags=(KEY_ENCRYPT_LEN << 16) |

      KEY_ENCRYPT_FLAGS;
   DWORD               dwRC2EffectiveLen=0;
   BYTE                   pbBuffer[BUFFER_SIZE];
   DWORD               dwCount=0;
```

Программа состоит из трех этапов. Каждый мы рассмотрим в следующей последовательности: определение задач, которые необходимо выполнить на данном этапе, решение задач, фрагмент программы примера и необходимые комментарии. Фрагменты программы, выполняющие этапы формирования сообщения отправителем и приема сообщения получателем, мы решили не приводить полностью. Покажем только фрагменты формирования зашифрованного ключа шифрования сообщения и импортирования этого ключа получателем, поскольку здесь и используются механизмы несимметричной криптографии. Остальные части этапов выполняют работу с симметричными ключами и отвечают за запись и считывание информации в файл, об этом мы рассказали в предыдущем параграфе.

Этап первый.

Определим главные задачи. Для того чтобы получатель имел возможность принимать криптографические сообщения в рамках схем несимметричного обмена, он должен иметь пару ключей (секретный и соответствующий ему открытый). Открытый ключ этой пары получатель должен сделать доступным тем, от кого он хочет получать зашифрованные сообщения.

Действия, выполняемые на первом этапе (выполняются получателем).

1. Генерируем ключевую пару обмена. Алгоритм ключевой пары зависит от используемого криптопровайдера и определяется константами PROV_NAME и PROV_TYPE. Информацию об алгоритмах и их параметрах, которые поддерживают различные криптопровайдеры, вы можете получить, используя программу EnumCSP (см. раздел «Получение информации о криптопровайдерах, установленных в системе»).
2. Экспортируем открытый ключ, сгенерированной ключевой пары обмена.
3. Сохраняем открытый ключ в файле с именем, определенным константой RECIPIENT_PUBKEY_FILE.

Фрагмент программы примера, реализующей первый этап, показан в листинге 2-8.

Листинг 2-8. Фрагмент программы CrMsgKeyX

```
// Первый этап
// Получатель вырабатывает ключевую пару обмена
// и посылает ее отправителю сообщения.
// См. комментарий 1
```

(см. след. стр.)

```
    // Открываем дескриптор ключевого контейнера получателя
    if  (!CryptAcquireContextEx(
            &hRecipientProv,
            RECIPIENT_CONTEXT,
            PROV_NAME,
            PROV_TYPE,
            0,
            AT_KEYEXCHANGE,
            dwKeyXFlags)) {
        flesult=FALSE;
        goto  FirstPhaseReleaseResource;
    }
    // Открываем дескриптор ключевой пары получателя
    if  (!CryptGetUserKey(
            rtRecipientProv,
            AT_KEYEXCHANGE,
            &hRecipientKeyX)) {
        printf("Error:  CryptGetUserKey=0x%X.\n",
            GetLastErrorO);
        Result=FALSE;
        goto  FirstPhaseReleaseResource;
    }
// См.  комментарий 2
    // Определяем идентификатор алгоритма ключевой пары
    dwDataLen=sizeof(DWORD);
    if  (!CryptGetKeyParam(
            hRecipientKeyX,
            KP_ALGID,
            (PBYTE)&EnvelopedData.aiKeyXAlg,
            &dwDataLen,0)) {
        printf("Error:  CryptGetKeyParam=0x%X.\n",
            GetLastErrorO);
        Result=FALSE;
        goto  FirstPhaseReleaseResource;
    }
    // Если используется алгоритм обмена ключами
    // Диффи-Хеллмана,  то  при  экспорте  используем  формат
    // блоба версии 3
    if  (EnvelopedData.aiKeyXAlg == CALG_DH_SF)
        dwExportKeyFlags=CRYPT_BLOB_VER3;
    // Определяем размер памяти для экспорта открытого
    //  ключа  получателя
    if (!CryptExportKey(
            hRecipientKeyX,
            0,
            PUBLICKEYBLOB,
```

(см. след. стр.)

```
            dwExportKeyFlags,
            NULL,
            &RecipientPubKeyBlob.cbData)) {
         printf("Error: CryptExportKey=0x%X.\n",
            GetLastError());
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      // Выделяем память для экспорта открытого ключа
      // получателя
      RecipientPubKeyBlob.pbData=LocalAlloc(LMEM_ZEROINIT,
         RecipientPubKeyBlob.cbData);
      if (!RecipientPubKeyBlob.pbData) {
         printf("Error: Out of memory.\n");
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      // Экспортируем открытый ключ получателя
      if (!CryptExportKey(
            hRecipientKeyX,
            0,
            PUBLICKEYBLOB,
            dwExportKeyFlags,
            RecipientPubKeyBlob.pbData,
            &RecipientPubKeyBlob.cbData)) {
         printf("Error: CryptExportKey=0x%X.\n",
            GetLastError());
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      // Открываем файл для сохранения открытого ключа
      // получателя
      if ((hRecipient=fopen(RECIPIENT_PUBKEY_FILE,
         "wb"))==NULL) {
         printf("Error: Opening %s file.\n",
            RECIPIENT_PUBKEY_FILE);
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      // Сохраняем открытый ключ получателя
      if (!WriteFileBlob(hRecipient,&RecipientPubKeyBlob)) {
         printf("Error: Writing data to file.\n");
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
FirstPhaseReleaseResource:
```

(см. след. стр.)

```
// Освобождаем открытые на первом этапе ресурсы
// и обнуляем переменные
. . .
if (!Result) return;
```

Теперь прокомментируем показанный фрагмент.

1. Для создания (или открытия) криптографического контейнера получателя используем функцию CryptAcquireContext-*Ex* (см. листинг 2-6). Имя криптографического контейнера определяется константой RECIPIENT_CONTEXT. Длина ключа, который будет сгенерирован функцией, определяется константой EXCHANGE_KEY_LEN (см. листинг 2-7).
2. Для экспорта открытого ключа используем функцию ***CryptExportKey*** (см. Приложение 2) с параметром PUBLICKEYBLOB. Однако при экспорте ключа RSA и Диффи — Хеллмана возникают различия. Дело в том, что обычный PUBLICKEYBLOB для ключа Диффи — Хеллмана содержит только значение открытого ключа, в то время как в общем случае необходимы также значения констант P и G. Для этого при экспорте ключа Диффи - Хсллмана, необходимо использовать PUBLICKEYBLOB версии 3. Поэтому с помощью функции ***CryptGetKeyParam*** (см. Приложение 2) определяем идентификатор алгоритма, если зто идентификатор CALG_DH_SF (*Diffie-Hellman* Store and Forward), то устанавливаем переменную флагов экспорта *dwExportKeyFlags* идентификатором третьей версии блоба CRYPT_BLOB_VER3 и экспортируем открытый ключ.

Надо отметить, что флаг CRYPT_BLOB_VER3 поддерживается, начиная с Windows 2000. Однако возможна реализация экспорта ключа Диффи — Хсллмана и параметров алгоритма и без использования третьей версии *ключевого* блоба. Для этого надо воспользоваться функцией ***CryptGetKeyParam с*** параметрами KP_P и KP_G (см. Приложение 2). Это позволит получить значения параметров, связанных с открытым ключом, а затем сохранить их вместе с блобом данного ключа. Этот вариант — устаревший и довольно неудобный, поэтому мы и предпочли воспользоваться способом поновее,

Этап второй.

Определим главные задачи. Имея открытый ключ получателя, отправитель должен сформировать криптографическое сообщение, содержащее зашифрованное на симметричном ключе содержание сообщения и зашифрованный па открытом ключе получателя ключ шифрования сообщения.

Действия, выполняемые на втором этапе (выполняются отправителем).

1. Генерируем случайный *ключ шифрования сообщения* (content-encryption key). Для этого используем алгоритм шифрования, его параметры определяются константами:
   * CONTENT_ENCRYPT_ALG - алгоритм шифрования;
   * CONTENT_ENCRYPT_MODE - режим шифрования;
   * CONTENT_ENCRYPT_LEN — длина ключа шифрования.
2. Для алгоритма RSA ключ шифрования сообщения шифруется на открытом ключе получателя. Такая схема называется *транспортировка ключа* (key transport).

   Для алгоритма Диффи — Хеллмана на основе открытого ключа получателя и секретного ключа отправителя формируется *парный симметричный ключ* (pairwise symmetric key), и ключ шифрования сообщения шифруется на этом симметричном ключе. Такая схема называется *согласование ключа* (key agreement). Алгоритм парного ключа (еще называемого ключом шифрования ключа шифрования сообщения) и его параметры определяются константами:
   * KEY_ENCRYPT_ALG — алгоритм шифрования;
   * KEY_ENCRYPT_MODE - режим шифрования;
   * KEY_ENCRYPT_LEN — длина ключа шифрования.
3. Зашифрованный ключ шифрования сообщения и сопутствующая информация собирается в структуру RECIPIENT_INFO. Состав структуры RECIPIENT_INFO зависит от используемой схемы. В программе для схемы RSA определена структура KEY_TRANS_RECIPIENT_INFO, а для схемы Диффи — Хеллмана структура KEY_AGREE_RECIPIENT_INFO.
4. Содержание сообщения шифруется на ключе шифрования сообщения.
5. Структура RECIPIENT_INFO и зашифрованное содержание сообщения собираются в структуру ENVELOPED_DATA.

Прежде чем рассматривать пример формирование зашифрованного ключа шифрования сообщения, разъясним важный момент в работе отправителя. После того как отправитель считает открытый ключ получателя и определит алгоритм шифрования сообщения и длину ключа, ему следует открыть дескриптор рабочего криптопровайдера. Последний должен поддерживать

алгоритм обмена, указанный в открытом ключе получателя, а также выбранный алгоритм шифрования и рабочую длину ключа. Для вызова криптопровайдера с заданными характеристиками в функциях CryptoAPI 2.0 используется процедура ***I_CryptGetDefaultCryptProvForEncrypt*** из библиотеки Crypt32.dll.

```
HCRYPTPROV
WINAPI
I_CryptGetDefaultCryptProvForEncrypt(
    ALG_ID      aiKeyXAlg,
    AI.G_ID     aiEncryptAlg,
    DWORD       dwKeyLen
);
```

Из прототипа функции видно, что она принимает в качестве параметров идентификатор алгоритма обмена (*aiKeyXAlg*), идентификатор алгоритма шифрования *(aiEncryptAlg)* и рабочую длину ключа *(dwKeyLen)*. Функция возвращает дескриптор ключевого контекста. Поскольку мы не передали никакой информации о *ключевом контейнере*, то открытый ключевой контекст является проверочным (CRYPT_VERIFYCONTEXT). Этого вполне достаточно, так как отправитель генерирует только временную ключевую пару, которая не сохраняется, и нам нет необходимости открывать ключевой контейнер.

Адрес функции ***I_CryptGetDefaultCryptProvForEncrypt*** определяем через вызов функции ***Loadlibrary*** для библиотеки Crypt32.dll и через запрос адреса вызовом ***GetProcAddress***. Адрес сохраняем в переменной *pfnCryptGetDefaultCryptProvForEncrypt*.

Фрагмент программы, реализующей формирование зашифрованного ключа шифрования сообщения, показан в листинге 2-9.

Листинг 2-9. Фрагмент второго этапа программы CrMsgKeyX

```
switch (EnvelopedData.aiKeyXAlg) {
    case CALG_RSA_KEYX:
// См. комментарий 1
        // Импортируем открытый ключ получателя
        // в криптопровайдер
        if(!CryptImportKey(
                hOriginatorProv,
                RecipientPubKeyBlob.pbData,
                RecipientPubKeyBlob.cbData,
                0,
                0,
                &hRecipientKeyX)) {
```

(см. след. стр.)

```
            printf("Error: CryptImportKey=0x%X.\n",
               GetLastError());
            goto SecondPhaseReleaseResource;
         }
// См. комментарий 2
         // Определяем размер памяти для экспорта ключа
         // шифрования сообщения на открытом ключе получателя
         if(!CryptExportKey(
               hContentKey,
               hRecipientKeyX,
               SIMPLEBLOB,
               RSA_ENCRYPT_FLAGS,
               NULL,
               &EnvelopedData.RecipientInfo.
               KeyTransRecipientInfo.
               EncryptedKey.cbData)) {
            printf("Error: CryptExportKey=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
         // Выделяем память под SIMPLEBLOB
         EnvelopedData.RecipientInfo.KeyTransRecipientInfo.
            EncryptedKey.pbData=
         LocalAlloc(
            LMEM_ZEROINIT,
            EnvelopedData.RecipientInfo.KeyTransRecipientInfo.
            EncryptedKey.cbData);
      if(!EnvelopedData.RecipientInfo.KeyTransRecipientInfo.
            EncryptedKey.pbData) {
            printf("Error: Out of memory.\n");
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
// См. комментарий 3
         // Экспортируем ключ шифрования сообщения на открытом
         // ключе получателя
         if(!CryptExportKey(
               hContentKey,
               hRecipientKeyX,
               SIMPLEBLOB,
               RSA_ENCRYPT_FLAGS,
               EnvelopedData.RecipientInfo.
               KeyTransRecipientInfo.EncryptedKey.pbData,
               &EnvelopedData.RecipientInfo.
               KeyTransRecipientlnfo.
```

(см. след. стр.)

```
            EncryptedKey.cbData)) {
            printf("Error: CryptExportKey=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
            // Сохраняем в файле экспортированный ключ
            // шифрования сообщения
         if (!WriteFileBlob(
                  hCipher, &EnvelopedData.RecipientInfo.
                  KeyTransRecipientInfo.

               EncryptedKey)) {
            printf("Error: Writing data to file.\n");
            Result=FALSE;
            goto   SecondPhaseReleaseResource;
         >
            break;
   case CALG_DH_SF:
// См. комментарий 4
         // Устанавливаем параметры шифрования для ключа
         // шифрования ключа сообщения
         EnvelopedData.RecipientInfo.KeyAgreeRecipientInfo.
               KeyEncryptionAID.aiAlgorithm=KEY_ENCRYPT_ALG;
         EnvelopedData.RecipientInfo.KeyAgreeRecipientInfo.
               KeyEncryptionAID.dwCipherMode=KEY_ENCRYPT_MODE;
         EnvelopedData.RecipientInfo.KeyAgreeRecipientInfo.
               KeyEncryptionAID.dwKeyLen=KEY_ENCRYPT_LEN;
// См. комментарий 5
         // Проводим предварительную генерацию ключа
         // Диффи-Хеллмана (тип ephemerial)
         if (!CryptGenKey(
                  hOriginatorProv,
                  CALG_DH_EPHEM,
                  CRYPT_PREGEN,
                  &hOriginatorKeyX)) {
            printf("Error: CryptGenKey=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto   SecondPhaseReleaseResource;
         }
         // Устанавливаем параметры (P, G) клича
         // Диффи-Хеллмана
         if (!CryptSetKeyParam(
                  hOriginatorKeyX,
                  KP_PUB_PARAMS,
```

(см. *след. стр.*)

```
                (PBYTE)&RecipientPubKeyBlob,
                0)) {
            printf("Error: CryptSetKeyParam=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
         // Проводим окончательную генерацию ключа
         // Диффи-Хеллмана
         if (!CryptSetKeyParam(hOriginatorKeyX,KP_X,NULL,0)) {
            printf("Error: CryptSetKeyParam=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
// См. комментарий 6
         // Импортируем открытый ключ получателя
         if (!CryptImportKey(
                hOriginatorProv,
                RecipientPubKeyBlob.pbData,
                RecipientPubKeyBlob.cbData,
                hOriginatorKeyX,
                0,
                &hAgreedKey)) {
            printf("Error: CryptImportKey=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
         // Устанавливаем идентификатор алгоритма шифрования
         // ключа
         if (!CryptSetKeyParam(
                hAgreedKey,
                KP_ALGID,
                (LPBYTE)&EnvelopedData.RecipientInfo.
                KeyAgreeRecipientInfo.KeyEncryptionAID.
                aiAlgorithm, dwKeyEncryptFlags)) {
            printf("Error: CryptSetKeyParam=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
         // Если алгоритм шифрования ключа RC2, то
         // устанавливаем эффективную длину ключа,
         // эквивалентную длине ключа
         if (EnvelopedData.RecipientInfo.
```

(см. след. стр.)

```
            KeyAgreeRecipientInfo.
            KeyEncryptionAID.aiAlgorithm == CALG_RC2) {
            dwRC2EffectiveLen=EnvelopedData.RecipientInfo.
            KeyAgreeRecipientInfo.KeyEncryptionAID.dwKeyLen;
            if  (!CryptSetKeyParam(
                     hAgreedKey,
                     KP_EFFECTIVE_KEYLEN,
                     (PBYTE)&dwRC2EffectiveLen,
                     0)) {
               printf("Error: CryptSetKeyParam=0x%X.\n",
                  GetLastError());
               Result=FALSE;
               goto FirstPhaseReleaseResource;
            }
         }
         // Устанавливаем режим шифрования ключа
         if  (!CryptSetKeyParam(
                  hAgreedKey,
                  KP_MODE,
                  (LPBYTE)&EnvelopedData.RecipientInfo.
                  KeyAgreeRecipientlnfo.
                  KeyEncryptionAID.dwCipherMode,
                  0)) {
            printf( "Error: CryptSetKeyParam=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
// См. комментарий 7
         // Определяем размер памяти для экспорта открытого
         // ключа получателя
         if  (!CryptExportKey(
                  hOriginatorKeyX,
                  0,
                  PUBLICKEYBLOB,
                  0,
                  NULL,
                  &EnvelopedData.RecipientInfo.
                  KeyAgreeRecipientlnfo.
                  OriginatorPublicKey.cbData)) {
            printf("Error: CryptExportKey=0x%X.\n",
               GetLastError0);
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
         // Выделяем блок памяти для экспорта открытого клича
```

(см. след. стр.)

```
        // получателя
        EnvelopedData.RecipientInfo.
        KeyAgreeRecipientInfo.OriginatorPublicKey.pbData=
           LocalAlloc(
              LMEM_ZEROINIT,
              EnvelopedData.RecipientInfo.
              KeyAgreeRecipientlnfo.
              OriginatorPublicKey.cbData);
        if(!EnvelopedData.RecipientInfo.
           KeyAgreeRecipientInfo.
              OriginatorPublicKey.pbData) {
           printf("Error: Out of memory.\n");
           Result=FALSE;
           goto SecondPhaseReleaseResource;
        }
        // Экспортируем открытый ключ получателя
        if (!CryptExportKey(
                 hOriginatorKeyX,
                 0,
                 PUBLICKEYBLOB,
                 0,
                 EnvelopedData.RecipientInfo.
                 KeyAgreeRecipientlnfo.
                 OriginatorPublicKey.pbData,
                 &EnvelopedData.RecipientInfo.
                 KeyAgreeRecipientlnfo.
                 OriginatorPublicKey.cbData)) {
           printf("Error: CryptExportKey=0x%X.\n",
              GetLastError());
           Result=FALSE;
           goto SecondPhaseReleaseResource;
        }
// См, комментарий 8
        // Определяем размер памяти для экспорта ключа
        // шифрования сообщения
        if (!CryptExportKey(
                 hContentKey,
                 hAgreedKey,
                 SYMMETRICWRAPKEYBLOB,
                 0,
                 NULL,
                 &EnvelopedData.RecipientInfo.
                 KeyAgreeRecipientInfo.
                 RecipientEncryptedKey.cbData)) {
           printf("Error: CryptExportKey=0x%X.\n",
              GetLastErrorO);
```

(см. след. стр.)

```
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
         // Выделяем блок памяти для экспорта ключа шифрования
         // сообщения
         EnvelopedData.RecipientInfo.KeyAgreeRecipientInfo.
               RecipientEncryptedKey.pbData=
            LocalAlloc(
               LMEM_ZEROINIT,
               EnvelopedData.RecipientInfo.
               KeyAgreeRecipientlnfo.
               RecipientEncryptedKey.cbData);
         if (!EnvelopedData.RecipientInfo.
            KeyAgreeRecipientlnfo.
            RecipientEncryptedKey.pbData) {
            printf("Error: Out of memory.\n");
            Result=FALSE;
            goto SecondPhaseReleaseResource;
            }
         // Экспортируем ключ шифрования сообщения
         if (!CryptExportKey(
                  hContentKey,
                  hAgreedKey,
                  SYMMETRICWRAPKEYBLOB,
                  0,
                  EnvelopedData.RecipientInfo.
                  KeyAgreeRecipientlnfo.
                  RecipientEncryptedKey.pbData,
                  &EnvelopedData.RecipientInfo.
                  KeyAgreeRecipientlnfo.
                  RecipientEncryptedKey.cbData)) {
            printf("Error: CryptExportKey=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto SecondPhaseReleaseResource;
         }
// См. комментарий 9
         // Сохраняем в файле экспортированный открытый ключ
         // отправителя
         if (!WriteFileBlob(
                  hCipher,
                  &EnvelopedData.RecipientInfo.
                  KeyAgreeRecipientInfo.OriginatorPublicKey)) {
            printf("Error: Writing data to file.\n");
            Result=FALSE;
            goto SecondPhaseReleaseResource;
```

(*см. след. стр.*)

```
        }
        // Сохраняем в файле идентификатор алгоритма ключа
        // шифрования ключа шифрования сообщения
        if  (!WriteFileAID(
                hCipher,
                &EnvelopedData.RecipientInfo.
                KeyAgreeRecipientInfo.KeyEncryptionAID)) {
           printf("Error: Writing data to file.\n");
           Result=FALSE;
           goto SecondPhaseReleaseResource;
        f
        // Сохраняем в файле экспортированный ключ шифрования
        // сообщения
        if  (!WriteFileBlob(
              hCipher,
              &EnvelopedData.RecipientInfo.
              KeyAgreeRecipientInfo.RecipientEncryptedKey)) {
        printf("Error: Writing data to file.\n");
           Result=FALSE;
           goto SecondPhaseReleaseResource;
        }
        break;
    default:
        printf("Error: Unknown key exchange algorithm.\n");
        Result=FALSE;
        goto SecondPhaseReleaseResource;
}
```

Теперь прокомментируем показанный фрагмент.

1. В соответствии со считанным из блоба открытого ключа получателя происходит формирование зашифрованного ключа шифрования сообщения. Если применяется алгоритм RSA, то первый шаг — импорт открытого ключа получателя в криптографический контекст получателя. Для этого используем функцию ***CryptImportKey*** (см. Приложение 2), передав ей в качестве параметров указатель на блоб и его длину. Функция возвращает дескриптор открытого ключа в переменной *hRecipientKeyX*.

1. Теперь, имея дескриптор открытого ключа получателя, можем зашифровать на нем ключ шифрования сообщения. Для шифрования по алгоритму RSA симметричного ключа на открытом ключе используется функция ***CryptExportKey*** (см. Приложение 2) с параметром SIMPLEBLOB. Экспорт осуществляется по обычной схеме: определение необходимого размера памяти для блоба, выделение необходимой памяти

и экспорт в буфер выделенной памяти (см. Приложение 2, раздел «Получение данных неизвестного размера»). Используемые функцией ***CryptExportKey*** схемы шифрования по алгоритму RSA подробно описаны в [33]. В соответствии с этим документом в настоящее время имеются две схемы шифрования по алгоритму RSA. Это старая схема RSAES-PKCS1-v1_5, которая используется функцией ***CryptExport-Key*** по умолчанию. И новая схема RSAES-OAEP, которая и рекомендуется к использованию. Однако имейте ц виду, что она поддерживается, начиная с Windows NT SP6. В программе используемая схема устанавливается константой RSA_ENCRYPT_FLAGS. Для применения новой схемы шифрования необходимо установить константу равной CRYPT_OAEP.

3. Как уже отмечалось, и случае алгоритма RSA вся информация о ключе шифрования ключа сообщения объединяется в структуру KEY_TRANS_RECIPIENT_INFO. Структура KEY_TRANS_RECIPIENT_INFO содержит единственный параметр типа DATA_BLOB *EncryptedKey*. Этот параметр сохраняет при экспорте указателя на буфер и размер зашифрованного ключа шифрования ключа сообщения.
4. С помощью функции ***WriteFileBlok*** сохраняем блоб в файле криптографического сообщения.

   В случае использования алгоритма Диффи — Хеллмана вся информация о ключе шифрования ключа сообщения объединяется в более сложную структуру KEY_AGREE_RECIPIENT_INFO. Начинаем работу по схеме Диффи — Хеллмана с инициализации параметра этой структуры KeyEncryptionAID — идентификатора алгоритма шифрования для парного симметричного ключа, на котором будет зашифрован ключ шифрования сообщения.
5. Теперь, согласно последовательности действий, которую мы изложили для алгоритма обмена ключами Диффи — Хеллмана, необходимо на основании открытого ключа получателя и секретного ключа отправителя получить парный симметричный ключ. Надо сказать, что существуют две модели реализации алгоритма Диффи — Хеллмана.

   Самая распространенная модель — Ephemeral-Static Diffie-Hellman. В рамках этой модели отправитель генерирует *временную* (ephemera!) ключевую пару обмена. Разумеется, при вычислении открытого ключа отправитель использует параметры P и G, которые предоставлены получателем. На

основании своего секретного и открытого ключа отправителя получатель вычисляет парный симметричный ключ. Ключ шифрования сообщения шифруется на этом ключе и вместе с открытым ключом отправителя включается в криптографическое сообщение. После этого временная ключевая пара отправителем уничтожается.

Другая модель — Static-Static Diffie-Hellman. В этом случае и у получателя, и у отправителя имеются статические ключевые пары, которые они применяют для обмена криптографическими сообщениями. Разумеется, что в этих ключевых парах используются одинаковые параметры алгоритма P и G. Кроме того, поскольку пары статические, то парный симметричный ключ всегда одинаков, что неприемлемо. Поэтому при реализации этой модели необходимо использовать модификаторы ключа.

В нашей программе реализуем модель Ephemeral-Static. Для этого при генерации ключевой пары Диффи — Хеллмана в функцию ***CryptGenKey***(см. Приложение 2) передается идентификатор алгоритма CALG_DH_EPHEM (Diffie-Hellman Ephemeral). Он как раз и предназначен для генерации временной ключевой пары. В отличие от ключевой пары, которая генерировалась получателем на первом этане CALG_DH_SF (Diffie-Hellman Store and Forward), временная ключевая пара не сохраняется в ключевом контейнере, а существует только в оперативной памяти, и после уничтожения дескриптора ключевой пары функцией ***CryptDestroyKey*** (см. Приложение 2) она также уничтожается. Кроме того, при генерации ключевой пары применяем ключ CRYPT_PREGEN. Если устанавливается этот флаг, то функция ***CryptGenKey*** сгенерирует только секретный ключ, а открытый ключ пока не вычислен. Таким образом, вы получаете возможность установить для ключа необходимые параметры P и G. Для установки параметров используем функцию ***CryptSetKeyParam***(см. Приложение 2) с параметром KP_PUB_PARAMS. В качестве указателя на параметр передаем указатель на структуру *RecipientPubKeyBlob* типа DATA_BLOB. Эта структура содержит указатель на буфер с блобом открытого ключа получателя и размер блоба. Параметр KP_PUB_PARAMS позволяет использовать для установки параметров Диффи — Хеллмана блоб открытого ключа третьей версии, применение которого мы обсуждали на первом этапе. При использовании старого способа пара-

метры передаются отдельно и функцию ***CryptSetKeyParam*** вызывают для параметров KP_P и KP_G также отдельно. После установки параметров для окончательной генерации ключевой пары опять вызывается функция ***CryptSetKeyParam*** с параметром KP_X, при этом указатель на параметр должен быть равен NULL. Таким образом, переменная *hOriginatorKeyX* содержит дескриптор временной ключевой пары отправителя.

6. Теперь на основе полученного дескриптора ключевой пары и блоба открытого ключа получателя вычисляем парный симметричный ключ. Для этого импортируем открытый ключ получателя через функцию ***CryptImportKey*** (см. Приложение 2), а в качестве ключа импорта устанавливаем *hOriginatorKeyX*. В результате переменная *hAgreedKey* будет содержать дескриптор *согласованного* (agreed) парного симметричного ключа. Однако, перед тем как применять этот ключ, необходимо установить его параметры. Прежде всего, у ключа пока нет даже алгоритма, для его установки вызываем ***CryptSetKeyParam.*** После этого ключ сформирован окончательно, и его можно использовать. Подробно формирование парного симметричного ключа для различных алгоритмов изложено в [31J. Теперь следует назначить параметры в соответствии с идентификатором алгоритма шифрования. Установка параметра KP_EFFECTIVE_KEYLEN для ключа RC2 обсуждается в комментарии 4 листинга 2-5.
7. После вычисления парного симметричного ключа надо сформировать параметры, необходимые для получателя. Сначала экспортируем открытый ключ отправителя, для этого вызываем функцию ***CryptExportKey*** (см. Приложение 2) с параметром PUBLICKEYBLOB. Сейчас нам не надо экспортировать параметры P и G, поэтому мы не устанавливаем никаких флагов экспорта. Блоб открытого ключа сохраняем в параметре *Originator Pub lic* ***Key*** структуры KEY_AGREE_RECIPIENT_INFO.
8. Затем необходимо зашифровать и экспортировать ключ шифрования сообщения, описываемый дескриптором *hContentKey*. Для этого также используется функция ***CryptExportKey*** с параметром SYMMETRICWRAPKEYBLOB, предназначенным для шифрования симметричного ключа на другом симметричном ключе. В качестве ключа экспорта, естественно, используем дескриптор *hAgreedKey*. Блоб ключа

шифрования сообщения сохраняем в параметре *RecipientEncryptedKey*структуры KEY_AGREE_RECIPIENT_INFO. Методы шифрования, применяемые при экспорте на различных симметричных ключах изложены в [30] и [31].

9. Сохраняем последовательно в файле криптографического сообщения все параметры структуры KEY_AGREE_RECIPIENT_INFO: сначала открытый ключ отправителя, затем идентификатор алгоритма шифрования парного симметричного ключа и, наконец, зашифрованный ключ шифрования сообщения.

Этап третий.

Определим главные задачи. Получив криптографическое сообщение от отправителя, получатель, используя свой секретный ключ, расшифровывает ключ шифрования сообщения и с его помощью расшифровывает содержание сообщения.

Действия, выполняемые на третьем этапе (выполняются получателем).

1. Считываем из криптографического сообщения информацию и заполняем структуру RECIPIENT_INFO.
2. Для алгоритма RSA: используя свой секретный ключ, получатель расшифровывает ключ шифрования сообщения.
3. Для алгоритма Диффи — Хеллмана: на основе открытого ключа отправителя и своего секретного ключа получатель формирует парный симметричный ключ и расшифровывает ключ шифрования сообщения.
4. Содержание сообщения расшифровывается на ключе шифрования сообщения.

Фрагмент программы, реализующей импорт зашифрованного ключа шифрования сообщения, показан в листинге 2-10.

Листинг 2-10. Фрагмент третьего этапа программы CrMsgKeyX

```
switch (EnvelopedData.aiKeyXAlg) {
   case CALG_RSA_KEYX:
// См. комментарий 1
      // Считываем зашифрованный ключ шифрования сообщения
      if (!ReadFileBlob(
               hCipher,
               &EnvelopedData.RecipientInfo.
               KeyTransRecipientInfo.EncryptedKey)) {
         printf("Error: Reading data from file.\n");
```

(см. след. стр.)

```
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
         }
// См. комментарий 2
         // Импортируем в криптопровайдер ключ шифрования
         // сообщения
         if (!CryptImportKey(
                  hRecipientProv,
                  EnvelopedData.RecipientInfo.
                  KeyTransRecipientInfo.EncryptedKey.pbData,
                  EnvelopedData.RecipientInfo.
                  KeyTransRecipientInfo.EncryptedKey.cbData,
                  0,
                  RSA_ENCRYPT_FLAGS,
                  &hContentKey)) {
            printf("Error: CryptImportKey=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
         }
         break;
      case CALG_DH_SF:
// См, комментарий 3
         // Считываем открытый ключ отправителя
         if (!ReadFileBlob(
                  hCipher,
                  &EnvelopedData.RecipientInfo.
                  KeyAgreeRecipientInfo.OriginatorPublicKey)) {
            printf("Error: Reading data from file.\n");
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
         }
         // Считываем идентификатор алгоритма ключа шифрования
         // ключа шифрования сообщения
         if (!ReadFileAID(
                  hCipher,
                  &EnvelopedData.RecipientInfo.
                  KeyAgreeRecipientInfo.KeyEncryptionAID)) {
            printf("Error: Reading data from file.\n");
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
         )
         // Считываем зашифрованный ключ шифрования сообщения
         if (!ReadFileBlob(
                  hCipher,
                  &EnvelopedData.RecipientInfo.
```

(см. след. стр.)

```
                KeyAgreeRecipientInfo.
                RecipientEncryptedKey)) {
            printf("Error: Reading data from file.\n");
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
        }
// См. комментарий 4
        // Получаем дескриптор ключевой пары обмена
        if (!CryptGetUserKey(
                hRecipientProv,
                AT_KEYEXCHANGE,
                &hRecipientKeyX)) {
            printf("Error: CryptGetUserKey=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
        }
        // Импортируем открытый ключ получателя
        if (!CryptImportKey(
                hRecipientProv,
                EnvelopedData.RecipientInfo.
                KeyAgreeRecipientInfo.
                OriginatorPublicKey.pbData,
                EnvelopedData.RecipientInfo.
                KeyAgreeRecipientInfo.
                OriginatorPublicKey.cbData,
                hRecipientKeyX,
                0,
                &hAgreedKey)) {
            printf("Error: CryptImportKey=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
        }
// См. комментарий 5
        dwKeyEncryptFlags=EnvelopedData.RecipientInfo.
        KeyAgreeRecipientInfo.KeyEncryptionAID.dwKeyLen << 16;
        // Устанавливаем алгоритм ключа шифрования ключа
        // шифрования сообщения
        if (!CryptSetKeyParam(
                hAgreedKey,
                KP_ALGID,
                (PBYTE)&EnvelopedData.RecipientInfo.
                KeyAgreeRecipientInfo.KeyEncryptionAID.
                aiAlgorithm,
                dwKeyEncryptFlags)) {
```

(см. след. стр.)

```
            printf("Error: CryptSetKeyParam=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
         }
         // Если алгоритм шифрования ключа RC2, то
         // устанавливаем эффективную длину ключа,
         // эквивалентную длине ключа
         if (EnvelopedData.RecipientInfo.
            KeyAgreeRecipientlnfo.
            KeyEncryptionAID.aiAlgorithm == CALG_RC2) {
            dwRC2EffectiveLen=EnvelopedData.RecipientInfo.
            KeyAgreeRecipientInfo.KeyEncryptionAID.dwKeyLen;
            if (!CryptSetKeyParam(
                     hAgreedKey,
                     KP_EFFECTIVE_KEYLEN,
                     (PBYTE)&dwRC2EffectiveLen,
                     0)) {
               printf("Error: CryptSetKeyParam=0x%X.\n",
                  GetLastErrorO);
               Result=FALSE;
               goto SecondPhaseReleaseResource;
            }
         ;
         // Устанавливаем режим ключа шифрования ключа
         // шифрования сообщения
         if (!CryptSetKeyParam(
                  hAgreedKey,
                  KP_MODE,
                  (PBYTE)&EnvelopedData.RecipientInfo.
                  KeyAgreeRecipientInfo.KeyEncryptionAID.
                  dwCipherMode,
                  0)) {
            printf("Error: CryptSetKeyParam=0x%X.\n",
               GetLastErrorO);
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
         }
// См. комментарий 6
         // Импортируем ключ шифрования сообщения
         if (!CryptImportKey(
                  hRecipientProv,
                  EnvelopedData.RecipientInfo.
                  KeyAgreeRecipientInfo.
                  RecipientEncryptedKey.pbData,
                  EnvelopedData.RecipientInfo.
```

(см. след. стр.)

```
                KeyAgreeRecipientInfo,
                RecipientEncryptedKey.cbData,
                hAgreedKey,
                0,
                &hContentKey)) {
            printf("Error: CryptImportKey=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto ThirdPhaseReleaseResource;
        }
        break;
    }
```

Теперь прокомментируем показанный фрагмент.

1. Для алгоритма RSA считываем зашифрованный ключ шифрования сообщения. Сохраняем размер и указатель на буфер блобав параметре *EncryptedKey*структуры KEY_TRANS_RECIPIENT_INFO.
2. Расшифровываем ключ шифрования сообщения на секретном ключе получателя. Для этого используем функцию ***CryptImportKey*** (см. Приложение 2). В функцию не надо передавать дескриптор ключа импорта, поскольку дескриптор ключевого контейнера однозначно идентифицирует секретный ключ обмена. Дескриптор ключа шифрования сообщения возвращается в переменной *hContentKey*.
3. Для алгоритма Диффи — Хеллмана необходимо считать несколько параметров. Первым считываем открытый ключ отправителя и сохраняем его в параметре *OriginatorPublicKey* структуры KEY_AGREE_RECIPIENT_INFO. Затем считываем и заполняем структуру *Key Encryption**AID*** — идентификатор алгоритма шифрованиядля парного симметричного ключа. Последним считываем зашифрованный ключ шифрования сообщения и сохраняем его в параметре *Recipient-EncryptedKey*структуры KEY_AGREE_RECTPIENT_INFO.
4. На основе открытого ключа отправителя и своего секретного ключа вырабатываем парный симметричный ключ. Для этого с помощью функции ***CryptImportKey*** импортируем открытый ключ в ключевой контейнер получателя. В отличие от импорта открытого ключа RSA, при импорте ключа Диффи — Хеллмана необходимо указать ключ импорта, поскольку кроме основной ключевой пары обмена в контейнере может быть создано несколько *временных* (ephemeral), и надо явно указать, какая пара используется. Для

получения дескриптора ключевой пары обмена используем функцию ***CryptGetUserKey*** (см. Приложение 2) с параметром AT_KEYEXCHANGE. В результате импорта открытого ключа отправителя, вырабатывается парный симметричный ключ, дескриптор которого находится в переменной *hAgreedKey*.

5. Как уже отмечалось и комментарии 6 ко второму этапу, для того чтобы использовать полученный дескриптор ключа, необходимо установить идентификатор алгоритма и другие параметры шифрования, переданные в криптографическом сообщении.
6. Расшифровываем ключ шифрования сообщения на парном симметричном ключе. Для этого с помощью функции ***CryptImportKey*** импортируем зашифрованный ключ шифрования сообщения в ключевой контейнер получателя. В качестве ключа импорта указываем дескриптор *hAgreedKey*. После этого дескриптор ключа шифрования сообщения содержится в переменной *hContentKey*.

Таким образом, формат данного криптографического сообщения обеспечивает конфиденциальность с помощью симметричного шифрования и несимметричных механизмов обмена ключами. Рассмотрим требования, которые необходимо выполнить, чтобы условие конфиденциальности действительно выполнялось.

Первая проблема, которую необходимо решать при реализации любой несимметричной схемы, — защита от так называемой атаки *«человек посередине»* (man in the middle). Этот вид атаки предполагает, что у злоумышленника есть возможность разомкнуть канал связи и вклиниться между отправителем и получателем. поэтому атака и получила такое название. Злоумышленник перехватывает открытый ключ получателя, заменяет его на открытый ключ своей ключевой пары обмена и посылает отправителю. Далее он перехватывает криптографические сообщения, расшифровывает их на своем секретном ключе, перешифровывает с использованием перехваченного открытого ключа и отправляет получателю. Использование сертификатов открытых ключей и доверенных центров сертификации, безусловно, решает эту проблему. Если сертификаты не используются, то доставку открытого ключа от получателя к отправителю следует производить по каналам, в которых обеспечивается целостность передаваемых данных. Это может быть переда-

ча на внешних магнитных носителях (например, дискетах), доставка которых гарантируется комплексом организационных мероприятий, либо передача по специальным сетям, которые гарантируют невозможность нарушения их физической целостности. Поскольку нет необходимости обеспечивать конфиденциальность открытого ключа, передача его от получателя к отправителю в неизменном виде — вполне решаемая задача.

Вторая проблема, — выбор рабочей длины ключа. В настоящее время рекомендуемые значения длины ключа для RSA — не менее 2048 бит, а для Диффи — Хеллмана — 1024 бита. Однако проблема реальной стойкости несимметричных схем сложна, и рассказ о пей выходит за рамки нашей книги. Материалы по этому вопросу публикуются постоянно. Для алгоритмов RSA и Диффи — Хеллмана рекомендуем посмотреть [33] и [32].

Использование CryptoAPI 1.0 для реализации схемы цифровой подписи

В предыдущем параграфе мы познакомили вас с тем, как обеспечивается в рамках концепции несимметричного распределения ключей конфиденциальность при передаче данных. Для решения задач целостности и аутентичности необходимо использовать механизм электронной цифровой подписи. Сейчас на конкретном примере мы покажем процедуру формирования отправителем подписи к отправляемому сообщению. Как и в предыдущих примерах, текст сообщения содержится в файле, собственно, как и электронная подпись со всеми необходимыми параметрами. Исходный текст программы хранится в файле Chapter2/SignMsg/SignMsg.c.

Криптопровайдеры Microsoft поддерживают два алгоритма цифровой подписи: RSA и DSA. Кроме того, криптопровайдерами Microsoft поддерживаются четыре алгоритма хеш-функций, используемых в подписи: MD2, MD4, MD5 и SHA-1. Надо, однако, помнить, что если для алгоритма цифровой подписи RSA подходят все хеш-функции, то стандарт DSA использует только хеш-функции с алгоритмом SHA-1.

В файл криптографического сообщения, которое формируется в примере, мы не включаем само сообщение, а только подпись и необходимые параметры. Считаем, что сообщение передастся самостоятельно. Таким образом, файл криптографического сообщения включает идентификатор алгоритма хеш-функции, открытый ключ пары цифровой подписи отправителя и

значение цифровой подписи. Мы не включаем явно идентификатор алгоритма подписи, поскольку он содержится в блобе открытого ключа отправителя.

Фрагмент программы показан в листинге 2-11. Используемые в программе функции ***CryptAcquireContextEx***, ***WriteFileBlob*** и ***ReadFileBlob*** мы описали в предыдущих параграфах.

Листинг 2-11. Фрагмент программы SignMsg

```
// См. комментарий 1
// Определение имени и типа рабочего провайдера
#define PROV_NAME              MS_DEF_PROV
«define PROV_TYPE              PROV_RSA_FULL
// Определение идентификатора, используемого алгоритма
//  хеширования
#define HASH_ALG               CALG_SHA
// Определение длины ключа подписи и флагов
#define SIGN_KEY_LEN           512
#define SIGN_KEY_FLAGS         0
// Определение строки для подписи и строк описания
#define SIGN_MSG               "The data that is to be
                               hashed and signed."
#define SIGN_DESCRIPTION       NULL
«define  VERIFY_DESCRIPTIONNULL
// Определение флагов подписи и проверки
#define SIGN_FLAGS             0
«define VERIFY_FLAGS           0
// Определение имен файлов сообщения и цифровой подписи
#define MESSAGE_FILE           "Message.txt"
#define SIGNATURE_FILE         "Message.sig"
// Определение имени ключевого контейнера отправителя
ftdefine ORIGINATOR_CONTEXT    "Originator"
#define BUFFER_SIZE            160
// См. комментарий 2
// Определение прототипа функции I_CryptGetDefaultCryptProv
// из библиотеки Crypt32.dll
typedef HCRYPTPROV (WINAPI
   *PFN_I_CRYPT_GET_DEFAULT_CRYPT_PROV)(
                  ALG_ID Algid
                  );
PFN_I_CRYPT_GET_DEFAULT_CRYPT_PROV
   pfnCryptGetDefaultCryptProv=NULL;
void main(void) {
   // Определение и инициализация переменных
   BOOL            Result=TRUE;
   FILE            *hMessage=NULL;
```

(см. след. стр.)

```
   FILE            *hSignature=NULL;
   fpos_t       fSavePos=0, fSetPos=0;
   HCRYPTPROV   hCryptProv=0;
   HCRYPTKEY    hSigKey=0;
   HCRYPTHASH   hHash=0;
   DWORD        dwKeyFlags=(SIGN_KEY_LEN << 16) |
                SIGN_KEY_FLAGS;
   ALG_ID       aiHashAlg=HASH_ALG;
   DATA_BLOB    PubKeyBlob={0,NULL};
   DATA_BLOB    Signature={0,NULL};
   DATA_BLOB
SignMsg={strlen((char*)SIGN_MSG)+1,(PBYTE)SIGN_MSG};
   BYTE            pbBuffer[BUFFER_SIZE];
   DWORD           dwCount;
   HMODULE         hDll;
// Первый этап
// Отправитель генерирует (или открывает) ключ подписи,
// вырабатывает цифровую подпись файла сообщения
// и отправляет ее получателю.
   // Открываем файл с сообщением
   if((hMessage=fopen(MESSAGE_FILE,"rb"))==NULL) {
      if((hMessage=fopen(MESSAGE_FILE,"wb"))==NULL) {
         printf("Error: Opening %s file.\n",MESSAGE_FILE);
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      fwrite(SignMsg.pbData, SignMsg.cbData, 1, hMessage);
      if(ferror(hMessage)) {
         printf("Error: Writing data to file.\n");
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
      fclose(hMessage); hMessage=0;
      if((hMessage=fopen(MESSAGE_FILE,"rb"))==NULL) {
         printf("Error: Opening %s file.\n",MESSAGE_FILE);
         Result=FALSE;
         goto FirstPhaseReleaseResource;
      }
   }
   // Открываем файл для сохранения цифровой подписи
   if((hSignature=fopen(SIGNATURE_FILE,"wb"))==NULL) {
      printf("Error: Opening %s file.\n",SIGNATURE_FILE);
      Result=FALSE;
      goto FirstPhaseReleaseResource;
   }
// См. комментарий 3
```

(см. след. стр.)

```
    // Получаем дескриптор ключевого контекста отправителя
    if (!CryptAcquireContextEx(
              &hCryptProv,
              ORIGINATOR_CONTEXT,
              PROV_NAME,
              PROV_TYPE,
              0,
              AT_SIGNATURE,
              dwKeyFlags)) {
       Result=FALSE;
       goto FirstphaseReleaseResource;
    }
    // Получаем дескриптора ключа подписи
    if(!CryptGetUserKey(
          hCryptProv,
          AT_SIGNATURE,
          &hSigKey)) {
       printf("Error: CryptGetUserKey=0x%X.\n",
          GetLastError());
       Result=FALSE;
       goto FirstPhaseReleaseResource;
    }
// См. комментарий 4
    // Определяем необходимый размера памяти для
    // экспорта открытого клоча пары подписи
    if(!CryptExportKey(
              hSigKey,
              0,
              PUBLICKEYBLOB,
              0,
              NULL,
              &PubKeyBlob.cbData)) {
       printf("Error: CryptExportKey=0x%X.\n",
          GetLastErrorO);
       Result=FALSE;
       goto FirstPhaseReleaseResource;
    }
    // Выделяем память для открытого ключа подписи
    PubKeyBlob.pbData=LocalAlloc(LMEM_ZEROINIT,
       PubKeyBlob.cbData);
    if(!PubKeyBlob.pbData) {
       printf("Error: Out of memory.\n");
       Result=FALSE;
       goto FirstPhaseReleaseResource;
    }
    // Экспортируем открытый ключ пары подписи
```

(см. след. стр.)

```
    if(!CryptExportKey(
              hSigKey,
              0,
              PUBLICKEYBLOB,
              0,
              PubKeyBlob.pbData,
              &PubKeyBlob.cbData)) {
       printf(" Error:  CryptExportKey=0x%X.\n",
          GetLastError());
       Result=FALSE;
       goto  FirstPhaseReleaseResource;
    }
// См. комментарий 5
    // Создаем объект хеш-функции
    if(!CryptCreateHash(
          hCryptProv,
          HASH_ALG,
          0,
          0,
          &hHash)) {
       printf("Error:  CryptCreateHash=0x%X.\n",
          GetLastError());
       Result=FALSE;
       goto    FirstPhaseReleaseResource;
    }
    do {
       // Читаем из файла сообщения данные размером
       // BUFFER_SIZE байт
       dwCount=fread(pbBuffer,1,BUFFER_SIZE,hMessage);
       if(ferror(hMessage)) {
       printf("Error: Reading data from file.\n");
          Result=FALSE;
          goto  FirstPhaseReleaseResource;
       }
       // Хешируем буфер данных
       if(!CryptHashData(
             hHash,
             pbBuffer,
             dwCount,
             0)) {
          printf("Error:  CryptHashData=0x%X.\n",
             GetLastError());
          Result=FALSE;
          goto FirstPhaseReleaseResource;
       }
    } while(!feof(hMessage));
```

(см. след. стр.)

```
// См. комментарий 6
   // Определяем неоходимый размер памяти для
   // буфера цифровой подписи
   if(!CryptSignHash(
            hHash,
            AT_SIGNATURE,
            SIGN_DESCRIPTION,
            SIGN_FLAGS,
            NULL,
            &Signature.cbData)) {
      printf("Error: CryptSignHash=0x%X.\n",
         GetLastError());
      Result=FALSE;
      goto FirstPhaseReleaseResource;
   }
   // Выделяем память под буфер цифровой подписи

Signature.pbData=LocalAlloc(LMEM_ZEROINIT,Signature.cbData);
   if(!Signature.pbData) {
      printf("Error: Out of memory.\n");
      Result=FALSE;
      goto FirstPhaseReleaseResource;
   }
   // Вычисляем цифровую подпись и записываем значение
   // подписи в выделенный буфер
   if(!CryptSignHash(
            hHash,
            AT_SIGNATURE,
            SIGN_DESCRIPTION,
            SIGN_FLAGS,
            Signature.pbData,
            &Signature.cbData)) {
      printf("Error: CryptSignHash=0x%X.\n",
         GetLastError());
      Result=FALSE;
      goto FirstPhaseReleaseResource;
   }
// См. комментарий 7
   // Сохраняем в файле алгоритм хеш-функции
   fwrite(&aiHashAlg,1,sizeof(ALG_ID),hSignature);
   if(ferror(hSignature)) {
      printf("Error: Writing data to file.\n");
      Result=FALSE;
      goto   FirstPhaseReleaseResource;
   }
   // Сохраняем в файле открытый ключ подписи отправителя
```

(см. след. стр.)

```
    if (!WriteFileBlob(hSignature,&PubKeyBlob)) {
       printf("Error: Writing data to file.\n");
       Result=FALSE;
       goto FirstPhaseReleaseResource;
    }
    // Сохраняем в файле значение цифровой подписи
    if (!WriteFileBlob(hSignature,&Signature)) {
       printf("Error: Writing data to file.\n");
       Result=FALSE;
       goto FirstPhaseReleaseResource;
    }
FirstPhaseReleaseResource:
    // Освобождаем открытые на первом этапе ресурсы
    // и обнуляем переменные
    . . .
    if (!Result) return;

// Второй этап
// Получатель, используя открытый ключ отправителя,
// проверяет цифровую подпись сообщения.
// См. комментарий 8
    // Открываем файл с сообщением
    if((hMessage=fopen(MESSAGE_FILE,"rb"))==NULL) {
       printf("Error: Opening %s file.\n",MESSAGE_FILE);
       Result=FALSE;
       goto SecondPhaseReleaseResource;
    }
    // Открываем файл с цифрой подписью
    if((hSignature=fopen(SIGNATURE_FILE,"rb"))==NULL) {
       printf("Error: Opening %s file.\n",SIGNATURE_FILE);
       Result=FALSE;
       goto SecondPhaseReleaseResource;
    }
    // Считываем из файла алгоритм хеш-функции
    fread(&aiHashAlg,1,sizeof(DWORD),hSignature);
    if(ferror(hSignature) || feof(hSignature)) {
       printfC'Error: Reading data from file.\n");
       Result=FALSE;
       goto SecondPhaseReleaseResource;
    }
    // Считываем из файла открытый ключ отправителя
    if (!ReadFileBlob(hSignature,&PubKeyBlob)) {
       printfC'Error: Reading data from file.\n");
       Result=FALSE;
       goto SecondPhaseReleaseResource;
    }
```

(см. след. стр.)

```
    // Считываем из файла значение цифровой подписи
    if (!ReadFileBlob(hSignature,&Signature)) {
       printf("Error: Reading data from file.\n");
       Result=FALSE;
       goto SecondPhaseReleaseResource;
    }
// См, комментарий 9
    // Получаем дескриптор библиотеки Crypt32.dll
    hDll=LoadLibrary("Crypt32.dll");
    if (!hDll) {
       printf("Error:  LoadLibrary=0x%X.\n", GetLastError0);
       Result=FALSE;
       goto SecondPhaseReleaseResource;
    }
    // Устанавливаем адрес процедуры
    //  I_CryptGetDefaultCryptProv
    pfnCryptGetDefaultCryptProv=
       (PFN_I_CRYPT_GET_DEFAULT_CRYPT_PROV)GetProcAddress(
                                                  hDll,
"I_CryptGetDefaultCryptProv");
    if (!pfnCryptGetDefaultCryptProv) {
       printf("Error: GetProcAddress=0x%X.\n",
          GetLastError0);
       Result=FALSE;
       goto SecondPhaseReleaseResource;
    }
    // Открываем дескриптор ключевого контекста получателя
    hCryptProv=pfnCryptGetDefaultCryptProv(
                     ((BLOBHEADER*)PubKeyBlob.pbData)-
                     >aiKeyAlg);
    if (!hCryptProv) {
       printf("Error: Get default provider!\n");
       Result=FALSE;
       goto SecondPhaseReleaseResource;
    }
// См. комментарий 10
    // Импортируем открытый ключ отправителя в криптопровайдер
    if(!CryptImportKey(
              hCryptProv,
              PubKeyBlob.pbData,
              PubKeyBlob.cbData,
              0,
              0,
              &hSigKey)) {
       printf("Error: CryptImportKey=0x%X.\n",
```

(см. след. стр.)

```
            GetLastError());
         Result=FALSE;
         goto  SecondPhaseReleaseResource;
      }
// См. комментарий 11
      // Создаем объект хеш-функции
      if(!CryptCreateHash(
                hCryptProv,
                aiHashAlg,
                0,
                0,
                &hHash)) {
         printf("Error:  CryptCreateHash=0x%X.\n",
            GetLastError());
         Result=FALSE;
         goto  SecondPhaseReleaseResource;
      }
      do {
         // Читаем из файла отправителя данные размером
         // BUFFER_SIZE байт
         dwCount=fread(pbBuffer,1,BUFFER_SIZE,hMessage);
         if(ferror(hMessage)) {
         printf("Error:  Reading data from file.\n");
            Result=FALSE;
            goto  SecondPhaseReleaseResource;
         }
         // Хешируем буфер данных
         if(!CryptHashData(
                   hHash,
                   pbBuffer,
                   dwCount,
                   0» {
            printf("Error:  CryptHashData=0x%X.\n",
               GetLastError());
            Result=FALSE;
            goto  SecondPhaseReleaseResource;
         }
      } while(!feof(hMessage));
// См. комментарий 12
      // Проверяем цифровую подпись
      if(!CryptVerifySignature(
                hHash,
                Signature.pbData,
                Signature.cbData,
                hSlgKey,
                VERIFY_DESCRIPTION,
```

(см, след. стр.)

```
            VERIFY_FLAGS)) {
        if(GetLastError() == NTE_BAD_SIGNATURE) {
            printf("File signature failed to validate!\n");
        }
        else {
            printf("Error: CryptVerifySignature=0x%X.\n",
                GetLastError());
            Result=FALSE;
        }
    }
    else {
        printf("File signature validated Ok!\n");
    }
SecondPhaseReleaseResource:
    // Освобождаем открытые на втором этапе ресурсы
    // и обнуляем переменные
. . .
    return;
};
```

Теперь прокомментируем показанный фрагмент.

1. Определение констант, используемых в программе. Помимо криптографических параметров, здесь определены следующие константы:
   * SIGN_MSG — строка с текстом сообщения. Эта строка используется, если отправитель заранее не создал подписываемый файл;
   * MESSAGE_FILE — строка, содержащая имя файла с подписываемым сообщением;
   * SIGNATURE_FILE — строка, содержащая имя файла, в котором сохраняются подпись файла и другие параметры криптграфического сообщения.
2. Определение прототипа функции *I_CryptGetDefaultCryptProv*. Эта функция экспортируется библиотекой Crypt32.dll. Процедура используется функциями CryptoAPI 2.0 для определения криптопровайдера, применяемого по умолчания для алгоритма подписи, который определяется параметром *Algid*. В нашем примере процедура используется получателем сообщения для определения криптопровайдера, который будет применяться для проверки цифровой подписи.
3. Генерируем ключевую пару подписи. Алгоритм ключевой пары зависит от используемого криптопровайдера, который

определяется константами PROV_NAME и PROV_TYPE. Информацию об алгоритмах цифровой подписи и их параметрах, которые поддерживают различные криптопровайдерьг, вы можете получить, используя программу EnumCSP (см. раздел «Получение информации о криптопровайдерах, установленных в системе»). Длина генерируемого ключа определяется константой SIGN_KEY_LEN.

4. Экспортируем открытый ключ ЭЦП отправителя. Для этого используем функцию ***CryptExportKey*** (см. Приложение 2) с параметром PUBLICKEYBLOB. Блоб открытого ключа сохраняем в структуре *PubKeyBlob* типа DATA_BLOB.
5. Создаем объект хеш-функции и хешируем содержимое файла MESSAGE_FILE. Считывание и хеширование производится блоками размером BUFFER_SIZE байт. Алгоритм, создаваемой хеш-функции, определяется константой HASH_ALG.
6. Вычисляем цифровую подпись для объекта хеш-функции. Для этого используется функция ***CryptSignHash*** (см. Приложение 1). Получение значения цифровой подписи производится так же, как и при получении других данных неизвестного размера (См. Приложение 2, раздел «Получение данных неизвестного размера»). В первый раз вызываем функцию для определения необходимого размера буфера, при этом указатель на буфер равен NULL. Затем выделяем необходимую память и снова вызываем функцию ***CryptSignHash*** с указателем на буфер для копирования цифровой подписи. Остановимся подробней на некоторых параметрах функции и значениях, которые передаются в процедуру:
   * *dwKeySpec* — параметр идентифицирует секретный ключ, на котором будет производиться подпись. Дело в том, что для алгоритма RSA нет разницы между ключами подписи и ключевого обмена (см. [33]). Поэтому если вы хотите произвести подпись на ключе обмена и ключевая пара обмена в контейнере существует, то необходимо передать в функцию идентификатор AT_KEYEXCHANGE. Но криптоповайдеры, поддерживающие пару алгоритмов Диффи — Хеллмана и DSA, не обладают в настоящее время возможностью подписи на ключе обмена;
   * *sDescription* — параметр определяется, как описатель подписываемого сообщения. Функция построена так, что если при вызове функции CryptSignHash указан параметр *sDescription*, то его необходимо передать и в функ-

цию проверки *CryptVerifySignature* (см. Приложение 2), иначе подпись не пройдет проверку. Поддерживают этот параметр только криптопройдеры, реализующие подпись по алгоритму RSA. Они просто еще раз вызывают внутреннюю функцию хеширования ***CPHashData*** криптопровайдера и передают и нее строку *sDescription* (в виде Unicode-строки) и ее длину. И только после этого значение хеш-функции подписывается. В нашем примере строка, передаваемая в качестве параметра *sDescription* в функцию подписи, определяется константой SIGN_DESCRIPTION;

* *dwFlags* — в настоящее время определено одно значение флагов CRYPT_NOHASHOID. Флаг относится только к криптопровайдерам, реализующим подпись по алгоритму RSA. Этот флаг определяет то, каким образом будет форматироваться значение хеш-функии перед подписью. В двух словах мы постараемся объяснить, что это означает. По умолчанию значение хеш-функции перед подписью кодируется по алгоритму EMSA-PKCS1-v1_5 (Encoding methom for signatures with appendix PKCS#1 v1.5), описанному в [33]. Один из шагов этого алгоритма заключается в *кодировании* (encoding) значения хеш-функции и идентификатора *(OID) хеш-функции в ASN.1* структуру *DigestInfo*. Если вы устанавливаете флаг CRYPT_NOHASHOID, то подписывается только значение хеш-функции. В примере флаги передаваемые функции подписи определяются константой SIGN_FLAGS.

7. Сохраняем в файле все необходимы для проверки целостности сообщения параметры: алгоритм, используемого хеша, блоб открытого ключа отправителя, значение цифровой подписи.
8. Получатель начинает свой этап с открытия файлов сообщения и цифровой подписи и считывания алгоритма хеш-функции, блоба открытого ключа отправителя, значения цифровой подписи.
9. Из полученного открытого ключа отправителя мы можем определить алгоритм используемой цифровой подписи. Как уже говорилось, для получения дескриптора криптопровайдера, поддерживающего полученный алгоритм, используется функция ***I_CryptGetDefaultCryptPro*** Адрес функции оп-

ределяем через вызов функции ***Loadlibrary*** для библиотеки Crypt32.dll и через запрос адреса вызовом ***GetProcAddress***. Адрес сохраняем в переменной *pfnCryptGetDefaultCryptProv*.

10. Для использования открытого ключа отправителя необходимо импортировать его к ключевой контейнер получателя. Используем для этого функцию ***CryptImportKey*** (см. Приложение 2). Таким образом, в переменной *hSigKey* сохраняется дескриптор открытого ключа проверки подписи.
11. В соответствии с идентификатором алгоритма хеша, полученным от отправителя, создаем объект хеш-функции. Затем производим хеширование содержания файла сообщения. Так же как и на первом этапе, считывание и хеширование выполняют блоки размером BUFFER_SIZE байт.
12. Получатель проверяет цифровую подпись сообщения. Для этого используется функция ***CryptVerifySignature*** (см. Приложение 2). Применение параметров *sDescription* и *dwFlags* рассмотрены в комментарии 6. Значения этих параметров, которые передаются в функцию, определяются константами VERIFY_DESCRIPTION и VERIFY_FLAGS соответственно. Отметим только одну особенность, связанную с параметром *dwFlags*. Как сказано в комментарии 6, если установлен флаг CRYPT_NOHASHOID, то формат подписываемого хеша будет отличаться от обычного. Но при проверке подписи вам не надо указывать этот флаг. Криптопровайдеры Microsoft, использующие подпись RSA, после расшифровки подписи на открытом ключе (подпись RSA — это просто зашифрованное значение хеша) сами разбирают формат и сравнивают со значением хеш-функции, определенной параметром *hHash*.

Таким образом, формат данного криптографического сообщения обеспечивает целостность и аутентичность с помощью механизма цифровой подписи. Требования, необходимые для того, чтобы эти условия выполнялись, аналогичны тем, которые мы рассмотрели в предыдущем разделе. А именно — необходимо, чтобы открытый ключ отправителя доставлялся получателю по каналам, которые обеспечивают целостность данных. В примере этапы доставки открытого ключа и сообщения для простоты объединены.

Заключение

В этой главе мы рассмотрели некоторые вопросы применения CryptoAPI 1.0. Безусловно, в примерах мы опустили тонкости применения криптоинтерфейса, однако все базовые операции рассмотрели достаточно подробно.

Еще раз отметим, что именно интерфейс CryptoAPI 1.0 является ядром прикладной криптографии в операционных системах Microsoft. Разумеется, коммерческие приложения очень редко напрямую вызывают базовые криптографические функции. Они используют интерфейс CryptoAPI 2.0, в котором уже реализованы стандарты для поддержки инфраструктуры открытых ключей. Однако, по нашему опыту, использование некоторых из них не всегда возможно. Например, стандарт криптографических сообщений PKCS#7, поддерживаемый CryptoAPI 2.0, создаст крайне громоздкие сообщения. Авторам известны ситуации, когда испытания показывали невозможность использования стандартных криптографических сообщений для рассылки (при больших объемах и скорости передачи сообщений, которые необходимо подписывать и зашифровывать). В этих случаях возникает необходимость создавать собственные криптографические сообщения и проектировать более оптимальные ключевые схемы. Именно тогда и возникает необходимость использования базовых криптографических функций CryptoAPI 1.0.

Корректное использование криптографических функций, реализованных и CryptoAPI, *возможно* только при выполнении ряда условий. В первую очередь — это необходимость сохранения в тайне ключей подписи и шифрования (в частности, обеспечение сохранности ключевых контейнеров). Далее, необходимо обеспечивать качественную реализацию датчиков случайных чисел (качество датчика существенно для реализации подписи сообщений). Важную роль играет также процедура доставки открытого ключа шифрования и подписи, обеспечивающая его целостность.

Глава 3

Основы разработки криптопровайдеров Microsoft средствами Microsoft CSPDK

В предыдущей главе мы рассмотрели вопросы использования базовых криптографических функций Microsoft CryptAPI для решения проблем криптографической защиты данных. В этой главе рассказано о ядре интерфейса CryptoAPI — провайдере криптографических услуг (Cryptographic Service Provider), или просто криптопровайдерс.

Надеемся, вы внимательно прочитали раздел «Принципы реализации интерфейса вызовов CryptoAPI 1.0» предыдущей главы и разобрались в примере восстановленного кода функции ***CryptAcquireContextA.*** Если нет, то вернитесь к этому параграфу, поскольку он очень важен для понимания следующих разделов.

Как вы уже поняли, криптопровайдер представляет собой обычную библиотеку (DLL), которая экспортирует определенный набор функций и соответствующим образом зарегистрирована в реестре. Поэтому программист, используя описания функций и правила регистрации, которые доступны в справочниках Microsoft (MSDN Library или Platform SDK Documentation), может самостоятельно написать *основу* (для создания полной библиотеки необходимы также глубокие знания в области криптографии) собственной библиотеки криптопровайдера и правильно се зарегистрировать.

Отметим, что при этом разработчик столкнется с одной проблемой, разрешить которую документация Microsoft ему не поможет. Проблема заключается в следующем: криптопровайдер должен быть подписан цифровой подписью компании Microsoft, так как без нее система не загрузит вашу библиотеку. Когда криптопровайдер будет готов, можно, конечно, отправить со-

ответствующий запрос в Microsoft. Но ведь подпись необходима и во время разработки *криптопровайдера*. Авторы не сомневаются, что квалифицированные программисты решат и эту проблему, однако трудностей можно избежать, выбрав более удобный путь для создания собственного криптопровайдера.

Компания Microsoft создала для разработчиков криптопровайдеров специальный программный инструментарий — Microsoft Cryptographic Service Provider Development Kit (CSPDK). Этот набор включает в себя все необходимые заголовочные файлы, исходные тексты программ регистрации и тестирования библиотек, исходные текст простейшего криптопровайдера и набор системных библиотек advapi32.dll для различных операционных платформ, позволяющих загружать криптопровайдер, подписанный тестовой цифровой подписью. Утилита для создания тестовой цифровой подписи также включена в состав CSPDK. Загрузить инструментарий можно с download-сервера Microsoft. Архив CSPDK версии 2.0 находится по адресу http://download.microsoft.com/download/win2000pro/Utility/V2.0/W98NT42KMe/EN-US/cspdk.exe.

В этой главе мы расскажем об интерфейсе CryptoSPI (Cryptographic System Program Interface), который поддерживают криптопровайдеры Microsoft, и особенностях регистрации новых библиотек. Кроме того, рассмотрим проблему контроля целостности криптопровайдера — вопрос — который был обойден в разделе «Принципы реализации интерфейса вызовов CryptoAPI 1.0» предыдущей главы, поскольку связан непосредственно с разработкой криптопровайдера и использованием CSPDK. Все эти вопросы будут обсуждаться во взаимосвязи с составом и назначением компонента Microsoft CSPDK.

Эта глава полезна всем, кто программирует в интерфейсе CryptoAPI, а не только тем, кто хотел бы создать собственный криптопровадер.

Состав Microsoft CSPDK

Как уже говорилось, CSPDK содержит необходимые для создания криптопровайдера заголовочные файлы, исходные тексты программ регистрации и тестирования библиотек, исходный текст простейшего криптопровайдера и набор системных библиотек advapi32.dll. В таблице 3-1 приведен полный перечень компонентов CSPDK.

Таблица 3-1. Компоненты Microsoft CSPDK

Компонент	Описание
\Csp\Csp.c	Исходный текст тестового криптопровайдера.
\Csp\Csp.def	Файл экспортируемых функций.
\Csp\Csp.rc	Файл ресурсов.
\Csp\Csp.sig	Файл, содержащий подпись библиотеки. По умолчанию — нулевой-
\Csp\Resource.h	Заголовочный файл ресурсов.
\Csp\AutoReg.cpp	Исходный текст процедур регистрации криптопровайдера.
\Csp.dll	Пример тестового криптопровайдера.
\Csp.sig	Файл, содержащий подпись тестового криптопровайдера.
\CspSign.exe	Утилита создания тестовой цифровой подписи криптопровайдера.
\TestCsp\TestCsp.c	Исходный текст программы тестирования криптопровайдера.
\TestCsp.exe	Программа тестирования криптопровайдера.
\CspInstl\CspInstl.c	Исходный текст программы регистрации криптопровайдера.
\CspInstl.exe	Программа регистрации криптопровайдера.
\SdkInc\Cspdk.h	Заголовочный файл, необходимый для разработки библиотеки.
\SdkInc\WinCrypt.h	Заголовочный файл CryptoAPI.
\Win98\advapi32.dl_	Специальный вариант библиотеки advapi32.dll для Windows 98.
\WinME\advapi32.dl_	Специальный вариант библиотеки advapi32.dll для Windows ME.
\NT4\SP3\advapi32.dl_	Специальные варианты библиотеки advapi32.dll для Windows NT 4.0.
\NT4\SP4\advapi32.dl_	
\NT4\SP6\advapi32.dl_	
\Win2k\advapi32.dl_	Специальные варианты библиотеки advapi32.dll для Windows 2000.
\Win2k\SP1\advapi32.dl_	
\Win2k\SP2\advapi32.dl_	
\WinCE\Csp\Csp.cpp	Файлы исходных текстов тестового криптопровайдера для Windows CE.
\WinCE\Csp\Csp.def	
\WinCE\Csp\makefile	
\WinCE\Csp\sources	
\WinCE\Csp.dll	Пример тестового криптопровайдера.
\WinCE\CspInstl\ CspInstl.cpp	Файлы исходных текстов программы регистрации криптопровайдера для Windows CE.
\WinCE\CspInstl\makefile	
\WinCE\CspInstl\sources	

(см. след. стр.)

Компонент	Описание
\WinCE\CspInstl.exe	Программа регистрации криптопровайдера.
\WinCE\TestCsp\TestCsp.cpp	Файлы исходных текстов программы тестирования криптопровайдера для WindowsCE,
\WinCE\TestCsp\makefile	
\WinCE\TestCsp\sources	
\WinCE\TestCsp.exe	Программа тестирования криптопровайдера.
\WinCE\Inc\Csp.h	Заголовочный файл, необходимый для разработки библиотеки под Windows CE.
\WinCE\Coredll.dl_	Специальные вариант библиотеки Coredll.dll для Windows CE.
\Docs\SCardCsp.doc	Описание дополнительных возможностей, необходимых криптопровайдеру для работы со смарт-картами. Рассмотрение данного вопроса выходит за рамки книги, поэтому вопросов создания и регистрации криптопровайдеров для работы со смарт-картами авторы не касаются,
\readme.txt	Основной источник вдохновения при написании этой главы.

Как видно из описания компонентов *CSPDKЮ*, комплект содержит практически все необходимое для разработки собственной библиотеки криптопровайдера.

Контроль целостности библиотеки

Прежде всего CSPDK помогает программисту решить вопрос контроля целостности создаваемой библиотеки криптопровайдера. Поскольку соответствующие проверки реализованы в системной библиотеке advapi.32.dll, то для решения этой задачи в состав CSPDK включен набор измененных модулей advapi32.dll для различных операционных платформ (таблица 3-1) и утилита для создания электронной подписи разрабатываемых модулей.

Для описания работы контроля целостности вернемся к разделу «*Принципы* реализации интерфейса *вызовов* CryptoAPI 1.0» главы 2. При обсуждении этого примера ранее был обойден вопрос о проверке подписи криптопровайдера. Ниже мы приводим фрагмент исходного текста из файла Chapter3/CpContext/CpContext.c, который мы не рассматривали в предыдущей главе.

Листинг 3-1. Фрагмент процедуры OwnCryptAcquireContextA

```
. . .
// См. комментарий 1
// Определение прототипа функции CProvVerifyImage
BOOL
WINAPI
CProvVerifyImage(LPCSTR  lpszImage,  BYTE  *pSigData )
{
   return TRUE;
>
// Определение прототипа функции CheckSignatureInFile
typedef
BOOL
(WINAPI * CHECK_SIGNATURE_IN_FILE_FN) (
   LPWSTR
   );
CHECK_SIGNATURE_IN_FILE_FN  pfnCheckSignatureInFile=NULL;
. . .
// См. комментарий 2
// Проверяем наличие данной библиотеке в кэше
   if (!CSPInCacheCheck(pszDllName,&hProvDll)) {
      // Если библиотека отсутствует, то загружаем ее ...
      hProvDll=LoadLibrary(pszDllName);
      if (!hProvDll) {
         Result=FALSE;
         dwCryptError=NTE_PROVIDER_DLL_FAIL;
         goto ReleaseResource;
      }
      // Проверяем цифровую подпись библиотеки
   #if (_WIN32_WINNT >= 0x0500)
// См. комментарий 3
      hAdvapiDll=LoadLibrary("advapi32.dll");
      if (!hAdvapiDll) {
         Result=FALSE;
         dwCryptError=NTE_FAIL;
         goto  ReleaseResource;
      )
      pfnCheckSignatureInFile=(CHECK_SIGNATURE_IN_FILE_FN)
         GetProcAddress(
            hAdvapiDll,"SystemFunction035");
      if (!pfnCheckSignatureInFile) {
         Result=FALSE;
         dwCryptError=NTE_FAIL;
         goto  ReleaseResource;
      }
      pwszDllName=LocalAlloc(LMEM_ZEROINIT,
```

(*см. след. стр.*)

```
            lstrlen(pszDllName)*2+2);
        if  (!MultiByteToWideChar(
                CP_ACP,
                MB_PRECOMPOSED,
                pszDllName,
                lstrlen(pszDllName),
                pwszDllName,
                lstrlen(pszDllName))) {
            Result=FALSE;
            dwCryptError=GetLastError();
            goto  ReleaseResource;
        }
        if (!pfnCheckSignatureInFile(pwszDllName)) {
            Result=FALSE;
            dwCryptError=NTE_BAD_SIGNATURE;
            goto ReleaseResource;
        }
        // Освобождаем выделенные ресурсы
        LocalFree(pwszDllName);  pwszDllName=NULL;
    #else
// См  комментарий 4
        // По открытому ключу считываем цифровую подпись
        // библиотеки
        dwRegError=RegQueryValueEx(
                  hkResult,
                  "Signature",
                  NULL,
                  &dwRegType,
                  NULL,
                  &cbData);
        if (dwRegError != ERROR_SUCCESS)  {
            Result=FALSE;
            dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
            goto ReleaseResource;
        }
        pbSignature=LocalAlloc(LMEM_ZEROINIT,cbData);
        if (!pbSignature) {
            Result=FALSE;
            dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
            goto  ReleaseResource;
        }
        dwRegError=RegQueryValueEx(
                  hkResult,
                  "Signature",
                  NULL,
                  &dwRegType,
```

(см. след. стр.)

```
                pbSignature,
                &cbData);
        if (dwRegError != ERROR_SUCCESS) {
            Result=FALSE;
            dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
            goto ReleaseResource;
        }
        if (!CProvVerifyImage(pszDllName,pbSignature)) {
            Result=FALSE;
            dwCryptError=NTE_BAD_SIGNATURE;
            goto ReleaseResource;
        }
    #endif
        // ... и помещаем дескриптор в кэш
        AddHandleToCSPCache(hProvDll);
    }
```

Теперь прокомментируем показанный фрагмент.

1. Определение прототипов функций проверки цифровой подписи для различных операционных платформ.

   Функция ***CProvVerifyImag***используется в Windows NT 4.0. В качестве параметров ей передаются имя контролируемой библиотеки и буфер, содержащий цифровую подпись. Как видно, в параметрах не передается длина буфера подписи, поскольку размер буфера 136 байт, так как используется цифровая подпись RSA с ключом длиной 1024 бита плюс 2 двойных слова *дополнения* (padding).

   В примере определена пустая функция ***CProvVerifyImage,*** так как в Windows NT 4.0 модуль advapi32.dll не экспортирует функцию проверки подписи библиотеки.

   Функция ***CheckSignatureInFile***реализована в Windows 2000 и Windows XP. В качестве ее параметра передается имя контролируемой библиотеки в виде Unicode-строки. Цифровая подпись в отличие от Windows NT 4.0 содержится непосредственно в самом файле. Данные подписи хранятся в ресурсах файла и составляют 144 байта, где 128 байт занимает сама подпись, 8 байт — служебная информация и 8 байт — дополнение. Ресурс, в котором хранятся данные подписи, имеет шестнадцатеричный номер 0x29A. Если перевести его в десятичную форму, то получится номер, который если не пугает, то несколько удивляет ... однако пусть читатель сам делает выводы о чувстве юмора программистов Microsoft.

   В примере используется указатель на функцию, поскольку в дальнейшем он будет приравнен к указателю на реальную

функцию *CheckSignatureInFile* из модуля advapi32.dll. В Windows 2000 и XP такая возможность есть, так как функция проверки подписи библиотеки является экспортируемой.

2. На момент рассмотрения имя библиотеки криптопровайдера считано из реестра и проверяется содержание внутреннего кэша. Если кэш не содержит запрашиваемого модуля, то для загрузки библиотеки вызывается функция *LoadLibrary*. Дальнейший код процедуры *CryptAcquireContextA* отличается для Windows NT 4.0 и Windows 2000/XP; причины объясняются в комментарии 1.
3. Для Windows 2000/XP, как уже отмечалось, имеется возможность получить адрес реальной функции проверки цифровой подписи. Для этого через вызов процедуры *LoadLibrary* получаем дескриптор модуля advapi32.dll и вызываем функцию *GetProcAddress*, передавая ее в качестве названия функции ее имя — «*SystemFunction035*». Именно под таким именем экспортируется процедура *CheckSignatureInFile*. Затем создаем Unicode-строку с именем библиотеки криптопровайдера и передаем ее в качестве параметра функции проверки подписи.

   Б программе примера есть определенные несоответствия действительности. Конечно, в реальной функции *CryptAcquireContextA* не вызываются процедуры *LoadLibrary* и *GetProcAddress* для получения адреса функции *CheckSignatureInFile*. К сожалению, в рассматриваемом примере экспортируемую функцию *SystemFunction035* тоже нельзя вызвать напрямую, поскольку библиотека advapi32.1ib из Microsoft Platform SDK не содержит точки входа для этой функции и программа просто не сможет скомпилироваться. И еще одна условность: для конвертирования ASCII-строки в Unicode в реальности используется процедура *RttCreateUnicodeStringFromAscii* из ядра ОС NtOsKrnl.exe, а не *MultiByteToWideChar*. Но это не принципиально в нашем случае.
4. В Windows NT 4.0 процесс вызова функции проверки выглядит по-другому. По открытому ключу реестра считывается значение с именем «Signature», в котором и должна находиться цифровая подпись криптопровайдера. И затем буфер со считанной подписью и имя файла передаются в процедуру *CProvVerifyImage*. Как уже отмечалось, эта функция не экспортируется и в примере нельзя реализовать вызов реальной функции проверки подписи.

В модулях advapi32.dll, поставляемых с CSPDK, реализация вызовов процедур проверки подписи абсолютно аналогична описанной. Однако реализация непосредственно процедур проверки подписи отличаются от оригинальных библиотек. В процедуру добавлен тестовый открытый ключ RSA. Если подпись, проверенная оригинальным ключом, не верна, то проверка выполняется на дополнительном тестовом ключе. Тестовую подпись, соответствующую дополнительному открытому ключу, создаст утилита CspSign.exe, поставляемая в составе CSPDK.

На самом деле в оригинальных библиотеках advapi32.dll проверка проводится не на одном открытом ключе. Один из ключей называется просто *_KEY,* а другой — *_NSAKEY.* Для того чтобы функция проверки целостности завершилась успешно, достаточно того, чтобы подпись соответствовала любому из двух ключей. Зачем проверка выполняется на двух ключах? Имеет ли ключ *_NSAKEY* отношение к АНБ (см. Приложение 1), авторы судить не берутся. Таким образом, в модулях advapi32.dll, поставляемых с CSPDK, проверка происходит сразу на трех ключах.

Замену оригинальной библиотеки на библиотеку из состава CSPDK можно осуществить несколькими способами. Не забудьте, что модули в составе CSPDK архивированы, для восстановления модулей воспользуйтесь утилитой expand.exe.

Если есть такая возможность, загрузитесь в другую операционную систему, переименуйте оригинальную версию библиотеки (например, назовите ее advapi32.dl_, чтобы не потерять), скопируйте соответствующий вариант advapi32.dll в системный каталог и загрузите рабочую систему. Другой вариант: переименовать библиотеку advapi32.dll прямо в рабочей операционной системе, скопировать в системный каталог нового модуля и перезагрузить. Но этот вариант не работает в Windows 2000 и XP, подсистема WFP (Windows File Protection) восстановит переименованную библиотеку. Если у Вас нет возможности загрузить альтернативную операционную систему, то загрузитесь в режиме Safe Mode, в нем WFP отключается.

Функции криптопровайдера

Итак, криптопровайдер представляет собой динамическую библиотеку (DLL), которая экспортирует набор из 23 обязательных функций с именами, определенными в массиве *FunctionNames* и двух необязательных функций с именами, определенными в массиве *OptionalFunctionNames.* Набор этих функций

составляет интерфейс Microsoft CryptoSPI (Cryptographic System Program Interface). Приведем краткое описание функций CryptoSPI.

В таблице 3-2 описаны функции управления криптопровайдера, в таблице 3-3 — функции работы с ключами и обмена криптографическими данными, в таблице 3-4 — функции шифрования и расшифрования, в таблице 3-5 — функции хеширования и цифровой подписи и в таблице 3-6 — функции дублирования объектов ключа и хеш-функции (необязательные).

Таблица 3-2. Функции управления криптопровайдера

Функция	Краткое описание
CPAcquire Context	Используется для получения дескриптора определенного ключевого контейнера в криптопровайдере.
CryptGetProvParam	Возвращает параметры криптопровайдера.
CryptReleaseContext	Используется для освобождения дескриптора криптопровайдера, созданного *CPAcquire Context.*
CryptSetProvParam	Устанавливает параметры криптопровайдера.

Таблица 3-3. Функции работы с ключами и обмена криптографическими данными

Функция	Краткое описание
CPDeriveKey	Создает сессионные криптографические ключи из ключевого материала.
CPDestroyKey	Освобождает дескриптор ключа.
CPExportKey	Используется для экспорта криптографических ключей и ключевых пар из ключевого контейнера криптопровайдера.
CPGenKey	Генерирует случайные сессионные ключи и ключевые пары.
CPGenRandom	Вырабатывает случайную последовательность и сохраняет ее в буфер.
CPGetKeyParam	Возвращает параметры ключа.
CPGetUserKey	Возвращает дескриптор одной из постоянных ключевых пар.
CPImportKey	Используется для импорта криптографического ключа из ключевого блоба в контейнер криптопровайдера.
CPSetKeyParam	Устанавливает параметры ключа.

Таблица 3-4. Функции шифрования и расшифрования

Функция	Краткое описание
CPDecrypt	Используется для расшифрования данных.
CPEncrypt	Используется для зашифрования данных.

Таблица 3-5. Функции хеширования и цифровой подписи

Функция	Краткое описание
CPCreateHash	Используется для инициализации хеширования потока данных.
CPDestroyHash	Уничтожает объект хеш-функции.
CPGetHashParam	Возвращает параметры объекта хеш-функции,
CPHashData	Используется для добавления данных к объекту хеш-функции.
CPHashSessionKey	Используется для добавления к объекту хеш-функции значения сессионного ключа.
CPSetHashParam	Устанавливает параметры объекта хеш-функции.
CPSignHash	Вычисляет значение ЭЦП от значения хеша, определенного дескриптором объекта хеширования.
CPVerifySignature	Осуществляет проверку подписи, соответствующей объекту хеширования.

Таблица 3-6. Функции дублирования объектов ключа и хеш-функции (необязательные)

Функция	Краткое описание
CPDuplicateKey	Делает точную копию ключа, его параметров и внутреннего состояния.
CPDuplicateHash	Делает точную копию объекта хеш-функции.

Следует напомнить, что если хотя бы одной функции из набора обязательных функций не окажется в таблице экспорта библиотеки, она не будет опознана в качестве криптопровайдера.

Как видно из описания, функции криптопровайдера полностью соответствуют большинству функций CryptoAPI. Не будем приводить здесь все прототипы указанных функций, поскольку они практически полностью копируют соответствующие прототипы функций CryptoAPI. Очевидное отличие заключается в том, что в процедурах работы с объектами ключей и хеш-функций передается еще и дескриптор криптопровайдера.

Остановимся подробней лишь па функции *CPAcquireContext*, прототип которой существенно отличается от вызывающей се функции *CryptAcquireContext*

```
BOOL WINAPI CPAcquireContext(
    HCRYPTPHOV *phProv,
    LPCSTR pszContainer,
    DWORD dwFlags
    PVTableProvStruc pVTable
);
```

Параметры *phProv*, *pszContainer* и *dwFlags* соответствуют описанным для функции ***CryptAcquireContext*** (см. Приложение 2). Рассмотрим процесс формирование структуры *pVTable*, показанный в примере CpContext.c (см. на сайте издательства «Русская Редакция»: www.rusedit.ru).

```
typedef struct _VTableProvStruc {
    DWORD Version;
    FARPROC FuncVerifyImage;
    FARPROC FuncReturnhWnd;
    DWORD dwProvType;
    BYTE* pbContextInfo;
    DWORD cbContextInfo;
    LPSTR pszProvName;
} VTableProvStruc, *PVTableProvStruc;
```

Параметры

Version

Версия структуры. В различных операционных системах используются различные версии. В Windows NT 4.0 используется версия 1, состоящая только из трех параметров: *Version*, *FuncVerifyImage* и *FuncReturnhWnd*. Версия 2 используется в Windows 98. Она состоит уже из шести параметров. Версия 3, показанная в примере, используется, только начиная с Windows 2000.

FuncVerifyImage

Указатель на функцию проверки цифровой подписи. Эта функция передается к криптопровайдер для того, чтобы разработчик имел возможность вставить в кол библиотеки проверку целостности дополнительных модулей, используемых криптопровайдером и имеющих соответствующую подпись.

В примере этот параметр приравнивается к указателю на функцию ***CProvVerifyImage,*** однако в разных операционных систем эта функция реализуется разными путями. В Windows NT 4.0 этот параметр содержит адрес именно функции ***CprovVerifyImage,*** с помощью которой операционная система проверяется целостность криптопровайдера. Начиная с Windows 2000, функция проверки претерпела изменения. Теперь в процедуру ***CProv Verify Image*** можно *не передавать указатель* на буфер с цифровой подписью, если подпись содержится в файле. В этом случае функция просто конвертирует имя в Unicode-строку и вызовет ***CheckSignatureInFile***. Если же указатель на буфер не нулевой, то ***CProv VerifyImage*** вызывает функцию ***NewVerify-***

Image, которая работает аналогично Windows NT 4.0. Функции ***CProvVerifyImage*** и ***NewVerifyImage*** в Windows 2000 не экспортируются.

FuncReturnh Wnd

Указатель на функцию, которая возвращает дескриптор окна приложения для взаимодействия с пользователем. Если такое взаимодействие в криптопровайдере не предусматривается, то данный параметр просто игнорируется. Функция ***FuncReturnh - Wnd*** имеет следующий тип:

*typedef void (*CRYPT_RETURN_HWND)(HWND *phWnd).*

Процедура просто копирует в передаваемый ей указатель глобальную переменную *hWnd*. По умолчанию эта переменная равна нулю. Установить необходимое значение дескриптора окна можно через вызов функции ***CryptSetProvParam*** (см. Приложение 2) с параметром PP_CLIENT_HWND.

dwProvType

Значение типа, запрашиваемого криптопровайдера.

pbContextInfo

Указатель на буфер пользовательского контекста. Пользовательский контекст устанавливается приложением через функцию ***CryptSetProvParam*** (см. Приложение 2) с параметром PP_CONTEXT_INFO.

cbContextInfo

Размер буфера пользовательского контекста.

pszProvName

Указатель на строку, содержащую имя криптопровайдера.

Применение CSPDK

В предыдущих параграфах были рассмотрены общесистемные вопросы работы и структуры криптопровайдеров. Теперь остановимся непосредственно па вопросе использования CSPDK для создания собственного криптопровайдера.

Пример простейшего криптопровайдера, который можно использовать как основу для работы, находится в каталоге \Csp (см. таблицу 3-1). Файлы, необходимые для создания криптопровайдера, — Csp.c, Csp.def, Csp.rc, Resource.h и AutoReg.cpp. Кроме того, требуется скопировать файл \SdkInc\Cspdk.h в каталог заголовочных файлов. При компилировании иногда возникает проблема — переопределение структуры *VTableProv-*

Struc. Это происходит из-за того, что эта структура уже может быть определена в файле WinCrypt.h. Чтобы избежать этой проблемы, стоит воспользоваться файлом \SdkInc\WinCrypt.h, который поставляется с CSPDK, в нем *VTableProvStruc*не определяется. Или же просто закомментировать определение данной структуры в файле Cspdk.h.

Кроме того, что криптопровайдер должен экспортировать правильный набор функций, он должен быть правильно зарегистрирован. В файле Csp.def помимо функций, относящихся непосредственно к интерфейсу CryptoSPI, экспортируются еще две функции ***DllRegisterServer*** и ***DllUnregisterServer***. Вообще-то, они используются в операционных системах Windows для установки и *удаления COM-объектов*. Но Microsoft часто обращается к этим процедурам для размещения кода регистрации и обычных библиотек. В состав операционных систем Windows входит утилита *RegSvr32.exe*, которая позволяет вызвать данные процедуры прямо из командной строки. Командная строка RegSvT32 DllName.dll вызывает функцию ***DllRegisterServer*** из библиотеки DllName.dll, а командная строка RegSvr32 /u DllName.dll позволяет вызвать функцию ***DllUnregisterServer.***

Установка и удаление криптопровайдера обычно заключается в создании необходимых для его регистрации ключей реестра. В тестовом криптопровайдере эти функции реализованы в файле AutoReg.cpp. Процедура ***DllRegisterServer,*** реализованная в файле, поддерживает обе модели регистрации криптопровайдера как с цифровой подписью в реестре, так и с подписью в самой библиотеке. Рассмотрим фрагмент кода функции *DllRegisterServer* из примера тестового криптопровайдера.

Листинг 3-2. Фрагмент процедуры DllRegisterServer

```
// См. комментарий 1
// See if we're self-signed.  On NT5, CSP images can carry
// their own signatures.
hSigResource = FindResource(
                    hThisDll,
                    MAKEINTRESOURCE(CRYPT_SIG_RESOURCE_NUMBER),
                    RT_RCDATA);
// См. комментарий 2
// Install the file signature.
//
ZeroMemory(&osVer, sizeof(OSVERSIONINFO));
osVer.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
```

(см. след. стр.)

```
fStatus = GetVersionEx(&osVer);
// ?BUGBUG? - This works on Windows Millenium, too.
if (fStatus
&& (VER_PLATFORM_WIN32_NT == osVer.dwPlatformId)
&& (5 <= osVer.dwMajorVersion)
&& (NULL != hSigResource))
{
    //
    // Signature in file flag is sufficient.
    //
    dwStatus = 0;
    nStatus = RegSetValueEx(
                        hMyCsp,
                        TEXT("SigInFile"),
                        0,
                        REG_DWORD,
                        (LPBYTE)&dwStatus,
                        sizeof(DWORD));
                        if (ERROR_SUCCESS != nStatus)
    {
        hReturnStatus = HRESULT_FROM_WIN32(nStatus);
        goto ErrorExit;
    }
}
else
{
// См. комментарий 3
    // We have to install a signature entry.
    // Try various techniques until one works.
    //
    for (dwIndex = 0; !fSignatureFound; dwIndex += 1)
    {
        switch (dwIndex)
        {
        //
        // Look for an external *.sig file and load that into
        // the registry.
        //
        case 0:
            tcscpy(szFileExt, TEXT("sig"));
            hSigFile = CreateFile(
                                szModulePath,
                                GENERIC_READ,
                                FILE_SHARE_READ,
                                NULL,
                                OPEN_EXISTING,
```

(см. след. стр.)

```
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
    if <INVALID_HANDLE_VALUE == hSigFile)
       continue;
    dwSigLength = GetFileSize(hSigFile, NULL);
    if ((dwSigLength > sizeof(pbSignature))
       || (dwSigLength < 72))     // Accept a 512-bit
                                  // signature
    {
       hReturnStatus = NTE_BAD_SIGNATURE;
       goto ErrorExit;
    }
    fStatus = ReadFile(
                   hSigFile,
                   pbSignature,
                   sizeof(pbSignature),
                   &dwSigLength,
                   NULL);
    if (!fStatus)
    {
       hReturnStatus =
          HRESULT_FROM_WIN32(GetLastError());
       goto ErrorExit;
    }
    fStatus  =  CloseHandle(hSigFile);
    hSigFile = NULL;
    if (!fStatus)
    {
       hReturnStatus =
          HRESULT_FROM_WIN32(GetLastError());
       goto ErrorExit;
    }
    fSignatureFound = TRUE;
    break;
//
// Other cases may be added in the future.
//
default:
    hReturnStatus = NTE_BAD_SIGNATURE;
    goto ErrorExit;
}
if (fSignatureFound)
{
    for (dwIndex = 0; dwIndex < dwSigLength;
       dwIndex += 1)
    {
```

(см. след. стр.)

```
            if (0 != pbSignature[dwIndex])
            break;
        }
        if (dwIndex >= dwSigLength)
        fSignatureFound = FALSE;
    }
}
// См. комментарий 4
    // We've found a signature somewhere! Install it.
    //
    nStatus = RegSetValueEx(
                        hMyCsp,
                        TEXT("Signature"),
                        0,
                        REG_BINARY,
                        pbSignature,
                        dwSigLength);
    if (ERROR_SUCCESS != nStatus)
    {
        hReturnStatus = HRESULT_FROM_WIN32(nStatus);
        goto ErrorExit;
    }
}
nStatus = RegCloseKey(hMyCsp);
hMyCsp = NULL;
if (ERROR_SUCCESS != nStatus)
{
    hReturnStatus = HRESULT_FROM_WIN32(nStatus);
    goto ErrorExit;
}
```

Теперь прокомментируем показанный фрагмент.

1. К этому моменту уже создан ключ криптопровайдера, установлены значения *ImagePath* и *Type*.

   Вызовом процедуры FindResource функция ***DllRegisterServer*** пытается обнаружить в регистрируемой библиотеке бинарный ресурс с номером Ox29A, в котором должна храниться цифровая подпись криптопровайдера.

2. Определяется версия операционной системы. Если это Windows 2000 или старше и ресурс с цифровой подписью находится в файле, то в ключе криптопровайдера устанавливается значение *SigInFile* типа DWORD и регистрация цифровой подписи библиотеки криптопровайдера на этом заканчивается.

Вы, вероятно, обратили внимание, что процедура не проверяет корректность бинарной информации, которая хранится в найденном ресурсе. Было бы неплохо проконтролировать размер данных ресурса, как это делается в случае с подписью в файле. Если читатель захочет модифицировать код, то он сможет найти описание структуры ресурса *InFileSignatureResource* в файле Cspdk.h, однако будьте осторожны.

```
typedef struct {
    DWORD dwVersion;
    DWORD dwCrcOffset;
    BYTE rgbSignature[88]; // 1024-bit key, plus 2 DWORDs
        of padding.
} InFileSignatureResource;
```

Здесь приведено определение, взятое из заголовочного файла. Ошибка в описании очевидна, размер массива подписи должен быть не 88 байт, а 0x88=136 байт. Если вы решите воспользоваться описанием структуры, исправьте это место.

3. Если ресурс в библиотеке не найден или операционная система не поддерживает такой тип проверки цифровой подписи, то функция пытается найти цифровую подпись в файле. Открывается файл с именем библиотеки и расширением .sig. Если файл обнаружен, то контролируется его размер и массив подписи считывается в буфер. В приведенном примере другой способ получения подписи файла не рассматривается. Разработчик, если это необходимо, может дополнить код другими вариантами получения массива цифровой подписи.
4. Если цифровая подпись тем или иным путем успешно получена, то она помещается в ключ криптопровайдера под именем Signature.

Кроме возможности регистрации библиотеки через вызов утилиты RegSvr32.exe, CSPDK позволяет воспользоваться отдельной программой регистрации криптопровайдера. Исходный текст программы приведен в файле \CspInstl\CspInstl.c (см. на сайте издательства «Русская Редакция»: www.rusedit.ru). Однако эта программа поддерживает только старый тип размещения цифровой подписи, поэтому для се работы необходим файл с подписью.

Каким бы образом вы не регистрировали свой криптопровайдер, перед регистрацией вам необходимо сгенерировать циф-

ровую подпись библиотеки. Для создания тестовой подписи CSPDK предоставляет утилиту CspSign.exe, которая предназначена для создания и проверки цифровой подписи библиотеки. Причем эти операции можно выполнять для цифровой подписи как размещенной в отдельном файле, так и сохраняемой в ресурсе самого криптопровайдера.

Если вы полагаете, что корректно скомпилировали, подписали и зарегистрировали свою библиотеку, запустите утилиту TestCsp.exc, ее исходный текст имеется в файле \TestCsp\TestCsp.c. Если вы не корректировали еще пример криптопровайдера или программу тестирования, то в случае успеха на экране должен появиться следующий результат (листинг 3-3).

Листинг 3-3. Фрагмент процедуры DllRegisterServer

```
Calling CryptAcquireContext - SUCCEED
Calling CryptGenKey - SUCCEED
Calling CryptDestroyKey - SUCCEED
Calling CryptGenKey - SUCCEED
Calling CryptSetKeyParam - SUCCEED
Calling CryptGetKeyParam - Access violation
Calling CryptSetProvParam - SUCCEED
Calling CryptGetProvParam - Access violation
Calling CryptGenRandom - SUCCEED
Calling CryptGetUserKey - SUCCEED
Calling CryptGenKey - SUCCEED
Calling CryptExportKey - Access violation
Calling CryptImportKey - SUCCEED
Calling CryptCreateHash - SUCCEED
Calling CryptSetHashParam - SUCCEED
Calling CryptGetHashParam - Access violation
Calling CryptHashData - SUCCEED
Calling CryptHashSessionKey - SUCCEED
Calling CryptEncrypt - Access violation
Calling CryptDecrypt - Access violation
Calling CryptDeriveKey - SUCCEED
Calling CryptSignHash - Access violation
Calling CryptVerifySignature - SUCCEED
Calling CryptDestroyHash - SUCCEED
SUCCEED
```

Не беспокойтесь о том, что некоторые функции возвращают ошибку. В качестве параметров они требуют указатель на буфер данных и указатель на переменную, содержащую размер этих данных. Программа тестирования при вызове всех функций передает тестовые параметры, которые просто являются

значениями от одного до десяти. Тестовый криптопровайдер пытается записать данные по такому указателю, и происходит исключение, а обработчик исключений возвращает ошибку ERROR_INVALID_PARAMETER.

Заключение

В заключение этой главы хочется пожелать успехов тем из вас, кто решит заняться разработкой криптопровайдеров. Отметим еще раз, что для успешной разработки любой криптографической библиотеки, в том числе и криптопровайдера, необходимо обладать глубокими знаниями не только в программировании, но и в области криптографии. Если Вас устраивает криптографическая составляющая существующих криптопровайдеров, то Вы можете выбрать способ, которым написаны криптопровайдеры Gemplus и Schlumberger. В этих библиотеках реализовано лишь хранение и обработка криптографической информации, хранящейся на смарт-картах, а для выполнения криптографических операций вызывается один из криптопровайдеров Microsoft.

Глава 4

Системные вопросы реализации средств криптографической защиты информации

Способы и особенности реализации криптографических подсистем

Механизмы криптографической защиты информации в компьютерной системе (КС) не являются «вещью в себе», их используют для решения конкретных задач. Из глобальных целей безопасности, таких, как обеспечение конфиденциальности, целостности и доступности, средства криптографической защиты позволяют обеспечить первые две. Для обеспечения конфиденциальности используются алгоритмы шифрования, для обеспечения целостности — алгоритмы хеширования и ЭЦП. В этой главе мы рассмотрим вопросы целевого использования криптографических алгоритмов, т.е. системно-концептуальное использование криптографии,

В настоящее время применяются два способа шифрования: предварительное и динамическое («прозрачное»). (Без существенных ограничений можно выводы, касающиеся шифрования, распространить и на алгоритмы контроля целостности и ЭЦП.)

Предварительное шифрование заключается в следующем: файл шифруется некой программой (субъектом КС), а затем расшифровывается тем же или иным субъектом (для расшифрования можно применять ту же или другую, специально предназначенную для расшифрования, программу). Далее расшифрованный массив данных (файл) непосредственно используется прикладной программой пользователя. Этот способ, хотя и применяется достаточно широко, имеет ряд принципиальных недостатков:

* необходимость дополнительного ресурса для работы с зашифрованным объектом (дискового пространства — в случае расшифрования в файл с другим именем);
* наличие потенциальной возможности доступа со стороны активных субъектов КС *к* расшифрованному файлу (во время его существования);
* необходимость гарантированного уничтожения расшифрованного файла после его использования.

Поэтому в последнее время широко применяется альтернативный метод — *динамическое шифрование.* Сущность его такова: зашифрование всего файла выполняется аналогично предварительному шифрованию, затем посредством специальных механизмов, *обеспечивающих* модификацию функций ПО КС, выполняют обращение к объектам. При этом расшифрованию подвергается только та часть зашифрованного объекта, которая в текущий момент времени используется прикладной программой. Перед записью прикладная программа шифрует записываемую часть объекта.

Данный способ позволяет оптимально использовать вычислительные ресурсы КС, поскольку расшифровывается только та часть объекта, которая непосредственно нужна прикладной программе. Кроме того, на внешних носителях информация всегда хранится в зашифрованном виде, что гарантирует ее безопасность. Таким образом, динамическое шифрование целесообразно применять для защиты удаленных или распределенных объектов КС.

Кроме того, его применяют для защиты группового массива данных — каталога или логического диска. Этот способ позволяет группировать защищенную информацию в одной или нескольких *единицах хранения,* что упрощает доступ к ней.

Кратко рассмотрим схему динамического шифрования в сетевой среде. При необходимости обращения к удаленным файлам на локальном компьютере активизируется сетевое программное обеспечение. Оно переопределяет *функции* работы с файловой системой ОС и создает, с точки зрения *рабочей* станции, единое простра] ство дан ных локального компьютера, удаленных компьютеров и файл-серверов.

Поскольку для работы с файлами требуются функции установленной на рабочей станции ОС, сетевое программное обеспечение модифицирует эти функции так, что обращение к ним со

стороны прикладного уровня КС выполняется обычным образом. Это позволяет обеспечить нормальную работу прикладного и пользовательского уровня программного обеспечения рабочей станции КС.

Криптографические функции для работы с файлами КС встраиваются в цепочку обработки файловых операций (рис. 4-1).

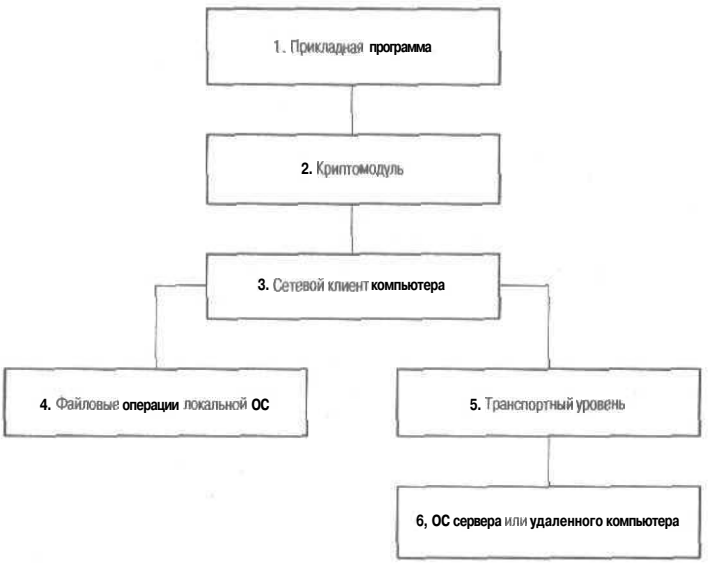

Рис. 4-1. Динамическое файловое шифрование

Необходимо заметить, что модули 1–4 физически локализованы в оперативной памяти рабочей станции КС.

Детализируем перечень обрабатываемых криптографическим модулем (2) функций работы с файлами. В него входят 5 основных функций:

* создание файла;
* открытие файла;
* закрытие файла;
* чтение из открытого файла;
* запись в открытый файл.

Рассмотрим две основные группы потенциальных действий злоумышленника:

1. обращение к зашифрованному сетевому ресурсу с рабочего места, не имеющего ключа расшифрования;
2. перехват информации в канале связи.

Первая угроза блокируется, поскольку шифрование информации происходит только в оперативной памяти рабочей станции КС, и запись, а также считывание информации с диска файл-сервера или рабочей станции ведется только в шифрованном виде.

По той же причине блокируются и вторая угроза, и обмен по транспортной системе «рабочая станция — сервер» выполняется на 3—5 уровнях, когда зашифрование уже закончено или расшифрование еще не произведено.

> Примечание Данный метод защиты при условии инвариантности к прикладному программному обеспечению рабочей станции считается оптимальным (обеспечивает минимальную вероятность доступа к незашифрованной информации) по сравнению с другими криптографическими механизмами.

Модификацией **описанного метода является принцип** *прикладного криптосервера.*

В этом случае выделяется активный аппаратный компонент КС (как правило, выделенный локальный компьютер), который имеет общий ресурс со всеми субъектами, требующими исполнения криптографических функций. При создании файла, принадлежащего общему ресурсу, и записи *к* него автоматически происходит его зашифрование или фиксация целостности.

Кроме того, в прикладном криптосервере можно реализовать функцию изоляции защищенного объекта-файла, возможность его перемещения в выделенный групповой массив (каталог исходящих файлов). Процесс обратного преобразования (или проверки целостности) выполняется аналогичным образом в других выделенных массивах.

Для субъекта рабочей станции этот процесс выглядит как автоматическое зашифрование (или интеграция цифровой подписи в файл) при записи в некоторый, заранее указанный каталог и появление зашифрованного файла в другом каталоге.

Этот способ широко применяется для криптографической защиты электронных документов в гетерогенной КС и для сопряжения разнородных прикладных систем с телекоммуникационными.

Криптографическая защита транспортного уровня

При анализе проблемы защиты транспортного уровня КС необходимо учитывать свойство, следующее из иерархической модели взаимодействия открытых систем: передача информации от верхних уровней представления (файлов) к нижним уровням (пакетам) полностью и без изменения сохраняет содержательную часть информации. Отсюда следует, что из-за шифрования файла, сопряженного с файловыми операциями (рассмотрено ранее), информационные части пакета, полученного из зашифрованного файла, проходят на транспортном уровне уже в зашифрованном виде. И наоборот, процедуры шифрования, локализованные на транспортном уровне, получают и передают информацию в верхние уровни представления в открытом виде.

Этот механизм позволяет определить область применения шифрования транспортного уровня:

1. при расположении кабельной системы КС на территории, доступной для злоумышленника;
2. когда не удается обеспечить взаимодействие прикладных задач пользователя с системой динамического файлового шифрования;
3. при отсутствии необходимости шифрования локальных ресурсов.

Криптозащиту транспортного уровня (рис. 4-2) можно реализовать программно при встраивании в информационные потоки сетевых программных средств и аппаратно, и на стыке рабочей станции и сетевых средств (*уровень 4* па рис. 4-2) или рабочей станции и кабельной системы *(уровень 6* на рис. 2; в этом случае можно говорить о наложении криптосредств). В зарубежной литературе подобного рада аппаратура именуется *криптобоксом* или *модулем безопасности* (Security Application Module, SAM). Налагаемые криптосредства обеспечивают практически полную независимость от прикладного и системного программного наполнения КС.

Криптографическая защита транспортного уровня, как было отмечено, прозрачно пропускает информацию прикладного уровня, а значит, не защищает от опасностей, характерных для уровня взаимодействия операционной среды и прикладного взаимодействия (влияние прикладных программ на зашифрованные файлы и на программы шифрования).

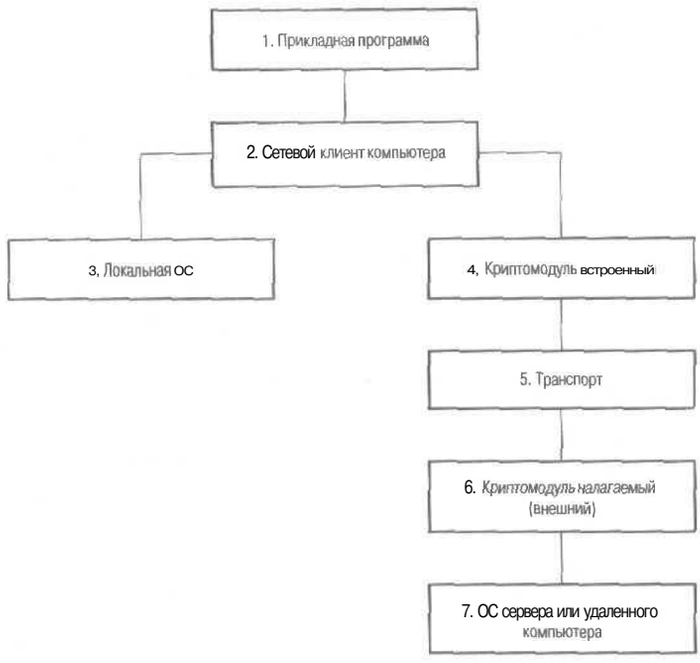

Рис. 4-2. Шифрование транспортного уровня

Криптографическая защита на прикладном уровне КС

Криптографическая защита информации на прикладном уровне считается оптимальным вариантом защиты информации с точки зрения гибкости защиты, но наиболее сложной в части программно-технической реализации.

Криптографическая защита информации на прикладном уровне (или криптографическая защита прикладного уровня) предусматривает такой порядок проектирования, реализации и использования криптографических средств, при котором входная и выходная информация, и возможно, ключевые параметры принадлежат потокам и объектам прикладного уровня (в модели ISO взаимодействия открытых систем).

Информация, расположенная на нижних иерархических уровнях модели ISO (далее для краткости используется термин «нижние уровни») относительно объекта прикладного уровни представляет собой подобъекты данного объекта, рассматри-

ваемые, как правило, изолированно друг от друга. В связи с этим на нижних уровнях (сетевом и др.) невозможно достоверно распознавать, а следовательно, и защищать криптографическими методами объекты сложной структуры типа «электронный документ» или «поле базы данных*. На нижних уровнях данные объекты представляются последовательностью вметающих подобъектов типа «пакет».

Кроме того, только на прикладном уровне возможна персонализация объекта, т.е. однозначное сопоставление созданного объекта породившему его субъекту (субъектом прикладного уровня является, как правило, прикладная программа, управляемая пользователем). Субъекты нижних уровней снабжают порождаемую ими последовательность подобъектов (подобъектов объекта прикладного уровня) некими дополнительными атрибутами — в основном адресом (информацией, характеризующей субъект нижнего уровня), который является лишь опосредованной характеристикой породившего информацию субъекта прикладного уровня.

В то же время необходимо отметить свойство наследования логической защиты верхнего уровня для нижнего. Поясним данное свойство примером.

Предположим, что на прикладном уровне зашифровано поле базы данных. При передаче информации по сети происходит его преобразование на нижний сетевой уровень. При этом поле, передаваемое в зашифрованном виде, разобщено (в смысле выполнения операции декомпозиции объекта на подобъекты) на последовательность пакетов. Информационное поле каждого из этих пакетов также зашифровано, передается по транспортной системе локальной или глобальной телекоммуникационной сети в виде датаграмм с зашифрованным информационным полем (адрес при этом не закрыт, поскольку у субъекта нижнего уровня, который произвел декомпозицию, нет информации о функции преобразования объекта).

Следовательно, криптографическая защита объекта прикладного уровня действительна и для всех нижних уровней.

Из указанного свойства следует утверждение: при защите информации на прикладном уровне такие процедуры, как пересдача, разборка на пакеты, маршрутизация и обратная сборка, не могут нанести ущерба конфиденциальности информации.

Упомянув о понятии конфиденциальности, нельзя не отметить, что две классические задачи криптографической защиты: за-

щита конфиденциальности и защита целостности — инвариантны относительно любого уровня модели ISO (с учетом свойства наследования). Защита прикладного уровня также в основном решает две указанные задачи отдельно или в совокупности.

Особенностью существования субъектов-программ прикладного уровня, а также порождаемых ими объектов является отсутствие стандартизованных форматов представления объектов. Более того, можно утверждать, что такая стандартизация возможна лишь для отдельных структурных компонентов субъектов и объектов прикладного уровня (например, типизация данных в транслируемых и интерпретируемых языках программирования, форматы результирующего хранения для текстовых процессоров и др.). Субъекты и объекты прикладных систем создаются пользователем, и априорно задать их структуру невозможно.

Известны два способа построения СКЗИ на прикладном уровне.

Первый способ (налагаемые СКЗИ) подразумевает реализацию функций криптографической защиты в отдельном субъекте-программе. Например, после подготовки электронного документа активизируется программа цифровой подписи для данного файла. Этот способ также называется абонентской защитой, поскольку программа активизируется оконечным пользователем-абонентом и локализуется в пределах рабочего места пользователя.

Второй способ (встроенные СКЗИ) предполагает вызов функций субъекта СКЗИ непосредственно из программы порождения защищаемых объектов и связан с внедрением криптографических функций в прикладную программу.

При сравнении этих способов видно, что первый отличается простотой реализации и применения, но требует соблюдения двух важных условий.

Во-первых, субъект СКЗИ необходимо реализовать в той же операционной среде либо операционной среде, связанной потоками информации с той, в которой существует прикладной субъект. Во-вторых, и прикладной субъект, и СКЗИ должны воспринимать объекты КС, т.е. декомпозиция компьютерной системы на объекты должна быть общей для обоих субъектов.

Все это предполагает раздельную реализацию СКЗИ в операционной среде и связь по данным. С другой стороны, в совре-

менных системах обработки и передачи информации достаточно сложно произвести пространственно-временную локализацию порождения конечного защищаемого объекта.

Поэтому сейчас все более широко используется второй способ (встраивание СКЗИ), для чего разрабатываются различные технические решения.

Схематично оба способа сравниваются в таблице 4-1,

Таблица 4-1. Сравнение двух способов построения СКЗИ на прикладном уровне

Характеристика	Налагаемые СКЗИ	Встроенные СКЗИ
Сопряжение с прикладной подсистемой	На этапе эксплуатации	На этапе проектирования и разработки
Зависимость от прикладной системы	Низкая	Высокая
Локализация защищаемого объекта	Внешняя (относительно защитного модуля и прикладной программы)	Внутренняя (защита внутреннего объекта прикладной программы)
Операционная зависимость	Полная	Низкая

Сейчас мы более подробно рассмотрим реализацию криптографической защиты в выделенном субъекте. В данном случае возможны несколько вариантов реализации варианта защиты:

1. локальная реализация в виде выделенной прикладной программы;
2. распределенная реализация но технологии «создание и запись в защищенной области».

Первую группу методов в свою очередь составляют:

* локальная реализация в базовой КС:
* локальная реализация в «гостевой» КС;
* локальная реализация по принципу «копирование в защищенный объект хранения».

Третий метод первой группы и второй вариант, как правило, реализуются в виде локального или распределенного прикладного криптосервера (см. ранее).

Для встраивания криптографических функций в прикладную систему можно выделить следующие направления:

1. встраивание функций по технологии «открытый интерфейс»;

2. встраивание функций по технологии «криптографический сервер»;
3. встраивание функций на основе интерпретируемого языка прикладного средства.

Основная проблема при этом — корректное использование вызываемых функций.

Достаточно перспективной считается реализация криптографических функций на прикладном уровне при помощи интерпретируемых языков, когда с ассоциированными объектами прикладного субъекта, требующими выполнения криптографических преобразований, производятся операции с использованием функций, реализованных в самом субъекте. Как правило, механизмы преобразования внутренних объектов реализованы на базе интерпретируемого языка. Преимущество данного способа – замкнутость относительно воздействия других субъектов, отсутствие необходимости использования внешнего субъекта, встроенных механизмов корректной реализации потоков информации в рамках субъекта прикладного уровня, а также потоков уровня межсубъектного взаимодействия. Основной недостаток — низкое быстродействие.

Практически *идеальным* языком программирования криптографических функций в субъекте прикладного уровня считается JAVA. Он отличается развитыми встроенными средствами работы с объектами прикладного уровня, а также широкими возможностями для реализации криптографических преобразований (элементарные логические и арифметические операции с числами, работа с матрицами и т.д.). Однако необходимо обратить внимание на проблему реализации программного датчика случайных чисел, безусловно, необходимого ряду СКЗИ (в частности, для реализации цифровой подписи).

Особенности сертификации и стандартизации криптографических средств

Процесс синтеза и анализа СКЗИ отличается сложностью и трудоемкостью, поскольку необходим всесторонний учет влияния перечисленных выше факторов на надежность реализации СКЗИ. В связи с этим практически во всех странах, обладающих развитыми криптографическими технологиями, разработка СКЗИ относится к сфере государственного регулирования. Государственное регулирование включает, как правило, *лицензирование* деятельности, связанной с разработкой и эксп-

луатацией криптографических средств, *сертификацию* СКЗИ и *стандартизацию* алгоритмов криптографических преобразований.

В Росси в настоящее время организационно-правовые и научно-технические проблемы синтеза и анализа СКЗИ находятся в компетенции Федерального агентства правительственной связи и информации (ФАПСИ).

Правовая сторона разработки и использования СКЗИ регламентируется в основном указом Президента Российской Федерации от 03.04.95 № 334 с учетом принятых ранее законодательных и нормативных актов РФ.

Кроме того, в этой сфере действуют законы: «О федеральных органах правительственной связи и информации», «О государственной тайне», «Об информации, информатизации и защите информации», «О сертификации продукции и услуг».

Порядок *сертификации* СКЗИ установлен «Системой сертификации средств криптографической защиты информации РОСС.RU.0001.030001» Госстандарта России.

Стандартизация алгоритмов криптографических преобразований включает всесторонние исследования и публикацию в виде стандартов элементов криптографических процедур с целью использования разработчиками СКЗИ апробированных, криптографически стойких преобразований, обеспечения возможности совместной работы различных СКЗИ, а также возможности тестирования и проверки соответствия реализации СКЗИ заданному стандартом алгоритму.

В России приняты и действуют следующие стандарты — алгоритм криптографического преобразования 28147-89, алгоритмы хеширования, выработки и проверки цифровой подписи Р34.10.94 и Р34.11.94 (подробно рассмотрены ранее).

Из зарубежных стандартов широко известны и применяются алгоритмы шифрования DES, RC2, RC4, алгоритмы хеширования MD2, MD4 и MD5, алгоритмы простановки и проверки цифровой подписи DSS и RSA.

Заключение

Подводя итоги, можно указать примерные направления и перспективы развития информационных технологий в рамках защищенных КС, опирающиеся на применение криптографических механизмов.

1. Полное управление системой криптографической защиты разрешено только администратору сети или администраторам СКЗИ. Реализация процедур изоляции ключей пользователей от администратора программными мерами: существование ключей только внутри программ и их уничтожение сразу после использования, хранение ключей в объектах КС (контейнерах), недоступных обычным пользователям.
2. Работа администратора с выделенного удаленного компьютера с реализацией принципов централизованного управления СКЗИ.
3. Администрирование сетевых криптографически защищенных ресурсов преимущественно без участия владельцев индивидуальных ключей. Возможность установки защиты самим пользователем.
4. Локальная интероперабельность — реализация защитных механизмов СКЗИ для некоторого подмножества файлово-совместимых ОС на рабочей станции и любого ПО на сервере.
5. Существование индивидуальных защищенных данных пользователя (эта возможность задастся дополнительными атрибутами защиты и правом порождения ключа для защиты локальных данных, возможностью отключения администратора СКЗИ от личного криптографически защищенного объекта, возможностью создания объекта индивидуального доступа, правом допустить к защищенному объекту другого пользователя).
6. Использование открытого интерфейса криптографических функций. Организация универсальных структур данных под переменные длины ключей, цифровых подписей и имитовставок.
7. Реализация процедуры единого выхода во внешнюю сеть через почтовый (транспортный) СКЗИ-шлюз либо шифрование объектов транспортной системы КС.
8. Снабжение клиента пользовательским интерфейсом (через функции открытого интерфейса), посредством которого он может встраивать в свои приложения (например, клиенты СУБД) криптографические функции.

На основании изложенных выше положений можно сделать вывод о том, что перспективные технологии применения криптографической защиты развиваются в направлении применения открытых интерфейсов криптографических функций, а также использования собственных вычислительных ресурсов прикладных средств.

Глава 5

Реализации криптографических алгоритмов в рамках криптопровайдера

Программная реализация алгоритмов шифрования

Эта глава посвящена особенностям программной реализации шифрующих преобразований. Вы узнаете о реализации алгоритма шифрования на языках низкого и высокого уровней, о способах решения задачи скоростной оптимизации и проверки правильности реализации алгоритма. В качестве базового приводится алгоритм блочного шифрования, соответствующий действующему в настоящее время стандарту шифрования ГОСТ 28147-89 (далее - ГОСТ).

Программная (или программно-аппаратная) реализация шифрующих преобразований занимает весьма важное и значимое место при реализации элементарных криптографических функций (примитивов) в составе криптпровайдсра. Кроме защиты собственноданных, шифрование применяется для защиты ключевого контейнера, при построении надежной хеш-функции, для реализации датчика случайных чисел и т.д. В качестве основного примера, иллюстрирующего базовые подходы к реализации и оптимизации, мы рассмотрим алгоритм ГОСТ 28147-89.

С качественной точки зрения метод синтеза стойких шифров описан Клодом Шенноном в работе «Теория связи в секретных системах» (см. сборник; Клод Шеннон «Избранные работы по теории информации и кибернетике». М., Мир, 1963) . Суть его изложена далее.

1. Пусть имеется алгоритм шифрования S с известной стойкостью, равной t элементарных операций.

1. Пусть также имеется размешивающее преобразование T (под размешивающим преобразованием Шеннон понимает некоторое отображение векторного пространства на себя, при котором каждая или почти каждая его компактная область в отображении распределяется в большую, некомпактную с точки зрения метрики, область).
3. Преобразования S и T обратимы (взаимно однозначны); обратимость S следует из определения шифра.
4. Шифрующее отображение описывается в этом случае итеративно:

   E(A)=T(S(T(S(.....(T(S(A))) ...)))) k раз

   k раз применяется суперпозиция T(S(A));

   A — символ открытого текста.

Поясним понятие размешивающего преобразования:

Рассмотри множество двоичных векторов длины 3 (таблица 5-1):

Таблица 5-1. Линейное пространство бинарных **векторов** длины 3

№	вектор
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

Рассмотрим преобразование вида x + a(mod 2),

где a = (1 1 1).

Легко заметить, что это преобразование не является размешивающим (фактически это инверсия компонентов вектора x) — действительно — компактная область 1 2 3 отобразится также в компактную область линейного пространства.

(0 0 1) + (1 1 1) = (1 1 0) 6

(0 1 0) + (1 1 1) = (1 0 1)5

(0 1 1) + (1 1 1) = (1 0 0) 4

Рассматривая вектора как числа и применив подстановку:

0 1 2 3 4 5 6 7

7 3 1 5 4 2 3 0

мы видим, что данное преобразование с большим основанием можно назвать размешивающим, т.к. компактная область 0 1 2 отображается в 7 3 1.

В качестве размешивающих преобразований целесообразно использовать нелинейные преобразования (реализация подстановок, умножение и др.).

Очевидно, что итеративность предложенного метода создаст потенциальную трудность — невозможность достижения высоких скоростей. Эту проблему можно решить, если выполнять преобразования над большими блоками данных (открытые тексты и шифртексты должны быть векторами большой размерности). В связи *с* этим идеи Шеннона, сформулированные *к* 1945 году, в полной мере удалось реализовать только при использовании микропроцессорной техники.

Впервые подход К. Шеннона был реализован в одном из самых известных из алгоритмов шифрования — DES (Data Encryption Standart). Длина вектора, участвующего в процедурах шифрования, составляет 64 бита. Мощность множества возможных ключей DES равна 2^{56}, что равно примерно 10^{18}, реальная стойкость DES (минимальная по всем известным методам дешифрования) оценивается несколько меньшим значением.

В 1989 году в СССР па тех же принципах был разработан алгоритм шифрования ГОСТ 28147-89, объем его ключевого множества равен 2^{256}, или примерно 10^{75}, реальная стойкость оценивается примерно так же, и размер блока 64 бита.

Если в алгоритме DES четко прослеживаются влияния аппаратных реализаций криптографических алгоритмов (преобразования отдельных битов), то ГОСТ содержит операции над 32-разрядным векторами, т.е. ГОСТ создан *с* учетом некоторой перспективы в развитии вычислительных платформ.

Описание алгоритма ГОСТ и его реализация

Стандарт шифрования СССР (позже — Российской Федерации) описан в документе «Алгоритм криптографического преобразования ГОСТ 28147-89». Для детального описания алгоритма и его программной реализации, о чем пойдет речь в этом разделе, требуется ввести некоторые обозначения.

В рассматриваемом описании элементы данных, участвующие в преобразовании (элементы открытого текста или ключа) состоят из нескольких компонентов: $X = (X_0, X_1 \ldots X_{n-1}) = X_0 \| X_1 \| \ldots \| X_{n-1}$.

Процедура объединения нескольких элементов данных в один называется *конкатенацией* данных и обозначается символом ||.

Данные могут интерпретироваться как бинарный вектор длины n:

$X = (x_0, x_1, ..., x_{n-1}) = x_0 + 2^1 \cdot x_1 + ... + 2^{n-1} \cdot x_{n-1}$.

Если над векторами выполняется некоторая логическая операция, то предполагается, что она выполняется над соответствующими битами. $A \bullet B = (a_0 \bullet b_0, a_1 \bullet b_1, ..., a_{n-1} \bullet b_{n-1})$, где и = $|A| = |B|$, т.е. размерность векторов А и В, символом «•» обозначается произвольная логическая операция; в ГОСТе операция «сумма по модулю *2»,* задаваемая инструкцией XOR процессора Intel или, в другой интерпретации, операция *«исключающего или»,*

Далее необходимо указать, что описание алгоритма имеет иерархическую структуру, а именно, описаны три алгоритма нижнего уровня, называемые *циклами.*

В свою очередь, каждый из базовых циклов представляет собой многократное повторение одной процедуры, называемой далее итерацией.

В ГОСТ ключевая информация состоит из двух структур данных. Помимо собственно *ключа,* являющегося параметром линейного преобразования имеется *таблица замен* — параметр нелинейной компоненты (размешивающего преобразования).

Ключ представляет собой массив из восьми 32-битных векторов:

$K = \{K_i\}_{0 \le i \le 7}$.

В ГОСТ элементы ключа используются как 32-разрядные целые числа без знака (*unsigned long* в нотации языка С). Таким образом, размер ключа составляет 32x8 = 256 бит, или 32 байта.

Таблица замен представляется матрицей 8x16, содержащей 4-битовые элементы, которые можно представить в виде чисел от 0 до 15. Строки *таблицы замен* называются *узлами замен,* они должны содержать различные значения, то есть каждый *узел замен* должен содержать 16 различных чисел от 0 до 15 и произвольном порядке. Таким образом, он представляет собой подстановку степени $2^4 = 16$. Таблица замен обозначается символом Я:

$Я = (H_{ij})_{\substack{0 \le i \le 7 \\ 0 \le j \le 15}}, 0 \le H_{ij} \le 15$.

Таким образом, общий объем таблицы замен равен: 8 узлов (подстановок) x 16 элементов/узел × 4 бита/элемент = 512 бит, или 64 байта.

Одна итерация криптографического преобразования определяет преобразование 64-битового блока данных (далее рассматриваемого как пара 32-битных векторов). Параметром этого преобразования является 32-битовый блок, в качестве которого используется какой-либо элемент ключа. Схема одной итерации показана на рис. 5-1. Такой рисунок обычно называют криптографической схемой преобразования или, кратко, криптосхемой.

Исходные данные для одной итерации:

N — преобразуемый 64-битовый блок данных, в ходе выполнения шага его младшая (N_1) и старшая (N_2) части обрабатываются как отдельные 32-битовые (4-байтные) целые числа. Таким образом, можно записать $N = (N_1//N_2)$.

X — 32-битовый элемент ключа;

Сложение с ключом. Младшая половина преобразуемого блока складывается по модулю 2^{32} с используемым на шаге элементом ключа, результат передается на следующий шаг;

Поблочная замена. 32-битовое значение, полученное на предыдущем шаге, интерпретируется как массив из восьми 4-битовых блоков (полубайт): $S = (S_0,S_1,S_2,S_3,S_4,S_5,S_6,S_7)$.

Далее значение каждого из восьми блоков заменяется на новое, которое выбирается по таблице согласно определению подстановки: значение блока S_i заменяется на $S_{i-й}$ по порядку элемент i-го узла замены (т.е. i-й строки таблицы замен).

Циклический сдвиг на 11 бит влево. Результат предыдущего шага сдвигается циклически на 11 бит в сторону старших разрядов и передается на следующий шаг. На схеме алгоритма символом Q_{11} обозначена функция циклического сдвига своего аргумента на 11 бит в сторону старших разрядов.

Побитовое сложение. Значение, полученное на предыдущем шаге, побитно складывается по модулю 2 со старшей половиной преобразуемого блока.

Сдвиг по цепочке. Младшая часть преобразуемого блока сдвигается на место старшей, а на ее место помещается результат выполнения предыдущего шага.

Полученное значение преобразуемого блока возвращается как результат выполнения алгоритма основного шага преобразования.

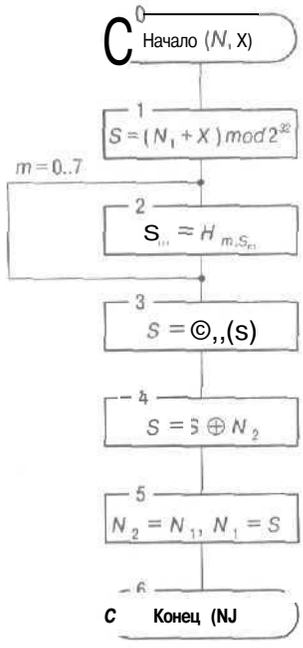

Рис. 5-1. Схема одной итерации алгоритма ГОСТ 28147-89

Здесь необходимо пояснение. Дело в том, что реализация подстановок на полубайтах не оптимальна для существующих процессоров.

Покажем, что существует алгоритм получения «развернутой» подстановки на байтах. Действительно, рассмотрим байт B как конкатенацию полубайтов.

B = $b_1 \| b_2$, b_1 и b_2 принимают значение от 0 до F (в шестнадцатеричной записи). Для реализации узла замены используются две подстановки

P, = (p_{10}, p_{11}, ..., p_{1F}) и p_2 = (p_{20}, p_{21}, ... p_{2F}), тогда **подстановка** p_{12} = ($p_{10}\|p_{20}$, $p_{10}\|p_{21}$, ..., $p_{10}\|p_{2F}$, $p_{11}\|p_{20}$, $p_{11}\|p_{21}$, ..., $p_{11}\|p_{2F}$, ..., $p_{1F}\|p_{20}$, ..., $p_{1F}\|p_{2F}$) реализует отображение целого байта, тождественное независимому применению подстановок p_1 и p_2. Степень подстановки p_{12} равна 256.

Рассмотрим фрагмент кода, который реализует указанное развертывание.

Пусть

char compuz[8][16];

массив, содержащий 8 узлов замены (подстановки, как было указано, имеют степень 16).

Тогда преобразование пары подстановок (compuz[0] и compuz[1]) в развернутый узел fulluz [256] задается операторами

```
for(i=0;i<16;i++)
for(j=0;j<16;j++) fulluz[i*16+j]=
   (compuz[0][i]<<4)^compuz[1][j];
```

Далее везде (если не указано особо) мы считаем, что узлы замены расширены и размер каждого — 256 байт (таким образом, совокупность компактных узлов замены по описанному алгоритму преобразуется в 1024 байта расширенных подстановок). Эта операция одновременно является и мерой оптимизации при реализации шифра.

Обратное преобразование развернутого узла замены в исходный можно выполнить следующей парой циклов (предполагаем, что развернутый узел находится в массиве pod):

```
for(i=0;i<16;i++)
compuz[1][i]=pod[i]&0x0F;
for(i=0;i<16;i++)
compuz[0][i]=(pod[i*16]&0xF0)>>4;
```

Приведем теперь реализацию одной итерации ГОСТ на языке ассемблера процессора Intel (предполагается, что процессор не менее, чем 386). Полагаем, что N_1 и N_2 размещаются в регистрах eax и edx соответственно. Пусть файл uzv.dat содержит развернутые узлы замены (4x256 байт), лежащие последовательно от метки uz0.

Включим файл данных, содержащий узел замены.

include uzv.dat

Таблица 5-2. Реализация одной итерации алгоритма ГОСТ

Номер инструкции	Команда	Выполняемые действия
1	gost MACRO	Начало макроса
2	mov ecx,eax	Сохранение N_1
3	add eax,dword ptr cur_key[I]	Сумма по mod 2^{32} с i-m элементом ключа (размером 32 бита)
4	mov bx, offsetuz0	Подготовка к реализации подстановки на расширенном узле замены
5	xlat	Реализации первой подстановки, результат получен в регистре a]

(см. след. стр.)

Номер инструкции	Команда	Выполняемые действия
6	ror eax,8	Перемещение результата вправо На байт, получение в al исходного значения для реализации следующей подстановки
7	inc bh	Переход к следующему узлу замены
8	xlat	
9	ror eax,8	
10	inc bh	
11	xlat	
12	ror eax,8	
13	inc bh	
14	xlat	
15	rol eax,3	Сдвиг на дополнительные 3 байта (до необходимых 11)
16	xor eax,edx	
17	raov edx,ecx	Сдвиг по цепочке
18	ENDM	Окончание макроса

Поясним этот фрагмент кода. Команда xlat процессора Intel представляет собой реализацию подстановки степени 256 (в описании команд процессора она названа «заменой по таблице»), при ее выполнении байт, содержащийся л регистре al, заменяется на байт, расположенный по адресу bx+al *(Ъх должен содержать адрес подстановки)*. Команда ror eax,8, следующая после первых трех команд xlat, позволяет «выдвигать» в регистр al последовательно байты для их преобразования. После четвертой команды xlat содержимое регистра eax уже сдвинуто на байт (8 бит), значит, для получения необходимого по описанию сдвига на 11 требуется дополнительный сдвиг еще на 3 бита. Поскольку развернутые узлы замены расположены в памяти последовательно, то для перехода к следующему узлу надо увеличить содержимое регистра bx на 256, для чего и прибавляется единица к старшему байту регистра bx (*команда* inc bh).

Далее показана реализация одной итерации па языке С:

```
void elem_gost( unsigned long *aa, unsigned long *bb,
    unsigned long *key)
{
unsigned char r[4];

 *(unsigned long *)r=*aa+*key;
*  r=*(pod+ *  r );
 *(r+1)=*(pod+256+*(r+1));
 *(r+2)=*(pod+512+*(r+2));
```

```
  *(r+3)=*(pod+768+*(r+3));
*bb^=(*(unsigned long *)r<<11)|((*(unsigned
    long *)r>>21)&0x7FF);
}
```

В этой процедуре переменная aa соответствует K , а bh — N_2, развернутый узел замены помещен в глобальную переменную pod и, как и ранее, представляет собой 4 последовательно расположенные подстановки степени 256. Key — указатель на массив unsigned long, содержащий ключ криптографического преобразования.

Базовые циклы криптографических преобразований

Базовые циклы алгоритма ГОСТ построены из итераций криптографического преобразования, рассмотренных в предыдущем разделе.

Цикл зашифрования данных использует элементы ключа в следующей последовательности (напомним, что элемент ключа — 4-байтовое целое число, причем в ключе таких элементов 8) (рис. 5-2).

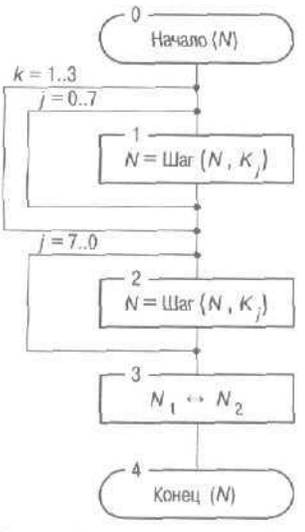

Рис. 5-2. Схема цикла зашифрования

Цикл зашифрования на языке Ассемблера (в виде макроса)

```
przam proc near
REPT 3
```

```
I=0
REPT 8
gost
I=I+4
ENDM
ENDM
I=28
REPT 8
gost
I=I-4
ENDH
xchg eax,edx
ret
przam ENDP
```

$K_0,K_1,\ldots,K_7,K_0,K_1,\ldots,K_7,K_0,K_1,\ldots,K_7,K_7,\ldots,K_1,K_0$ или, иначе говоря, 32-битные элементы ключа, ныбираются 3 раза в прямой и один раз в обратной последовательности (обозначение 32-3).

Цикл расшифрования данных:

$K_0,K_1,\ldots,K_7,K_7,K_6,\ldots,K_2,K_1,K_0,K_7,K_6,\ldots,K_1,K_0,K_7,K_6,\ldots,K_1,K_0$, т.е. элементы ключа используются один раз в прямой и три раза в обратной последовательности (обозначение 32-Р).

Здесь уместно заметить, что ГОСТ предназначен для режима контроля целостности, или, согласно терминологии этой книги (см. «Введение»), режима имитозащиты данных или выработки имтовставки.

Цикл выработки имитовставки:

$K_0,K_1,K_2,K_3,K_4,K_5,K_6,K_7,K_0,K_1,K_2,K_3,K_4,K_5,K_6,K_7$.

Цикл расшифрования

```
przam_1 proc near
I=0
REPT 8
gost
I=I+4
ENDM
REPT 3
I=28
REPT 8
gost
I=I-4
ENDH
ENDM
xchg eax,edx
ret
przam_1 ENDP
```

Цикл расшифрования должен выполняться в порядке, обратном циклу зашифрования, то есть в результате последовательного применения этих двух циклов к произвольному блоку данных должен получиться исходный блок (рис. 5-3).

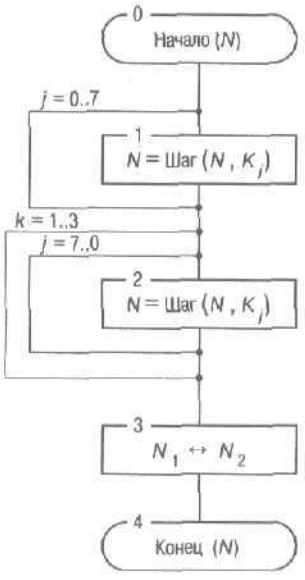

Рис. 5-3. Схема цикла расшифрования

При реализации криптографических преобразований и, особенно, при оптимизации тех или иных криптографических процедур необходимо проверить соответствие программной или аппаратной реализации описанию криптографического алгоритма. Как правило, для решения этой задачи используется система тестовых примеров.

Приведем тестовый пример, который позволит проверять правильность реализации ГОСТ.

Пусть содержание подстановок (описанный выше массив compuz) соответствует данным таблицы 5-3 (первый столбец — номер узла замены, остальные — шестнадцатеричный номер соответствующего элемента подстановки).

Тогда при указании исходного блока в виде двух чисел

0x55555555 0xAAAAAAAA (unsigned long)

и ключа

Таблица 5-3. Тестовые значения узлов замены

0	0xc	0x9	0xf	0xe	0x8	0x1	0x3	0xa	0x2	0x7	0x4	0xd	0x6	0x0	0xb	0x5
1	0x4	0x2	0xf	0x5	0x9	0x1	0x0	0x8	0xe	0x3	0xb	0xc	0xd	0x7	0xa	0x6
2	0xe	0x9	0xb	0x2	0x5	0xf	0x7	0x1	0x0	0xd	0xc	0x6	0xa	0x4	0x3	0x8
3	0xd	0x8	0xe	0xc	0x7	0x3	0x9	0xa	0x1	0x5	0x2	0x4	0x6	0xf	0x0	0xb
4	0x8	0xf	0x6	0xb	0x1	0x9	0xc	0x5	0xd	0x3	0x7	0xa	0x0	0xe	0x2	0x4
5	0x3	0xe	0x5	0x9	0x6	0x8	0x0	0xd	0xa	0xb	0x7	0xc	0x2	0x1	0xf	0x4
6	0xc	0x6	0x5	0x2	0xb	0x0	0x9	0xd	0x3	0xe	0x7	0xa	0xf	0x4	0x1	0x8
7	0x9	0xb	0xc	0x0	0x3	0x6	0x7	0x5	0x4	0x8	0xe	0xf	0x1	0xa	0x2	0xd

```
unsigned long Key[8]=
{
 0x11111111, 0x22222222, 0x33333333, 0x44444444,
 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF
};
```

зашифрованный текст должен получиться таким:

0x3113A05E, 0xC4F6F857 (два числа unsigned long)

Цикл зашифрования, реализованный на языке C, выглядит следующим образом (s — указатель на массив из двух блоков открытого текста, предназначенного для зашифрования, k — указатель на массив ключа).

```
void pz(unsigned long *s,unsigned long *k)
{
void elem_gost(unsigned long *, unsigned long *,unsigned
long *);
unsigned long cur;
elem_gost(s,s+1,k );
elem_gost(s+1,s,k+1);
elem_gost(s,s+1,k+2);
elem_gost(s+1,s,k+3);
elem_gost(s,s+1,k+4);
elem_gost(s+1,s,k+5);
elem_gost(s,s+1,k+6);
elem_gost(s+1,s,k+7);
//... Далее указанный блок повторяется еще два раза
//... Затем производится обратная выборка ключа
elem_gost(s,s+1,k+7);
elem_gost(s+1,s,k+6);
elem_gost(s,s+1,k+5);
elem_gost(s+1,s,k+4);
elem_gost(s,s+1,k+3);
elem_gost(s+1,s,k+2);
elem_gost(s,s+1,k+1);
elem_gost(s+1,s,k );
cur=*s; *s=*(s+1); *(s+1)=cur;
}
```

Цикл выработки имитовставки вдвое короче циклов шифрования, порядок использования ключевых элементов в нем такой же, как и в первых 16-ти шагах цикла зашифрования (рис. 5-4).

```
imit proc near
REPT 2
I=0
REPT 8
gost
```

```
I=I+4
ENDM
ENDM
ret
imit ENDP
```

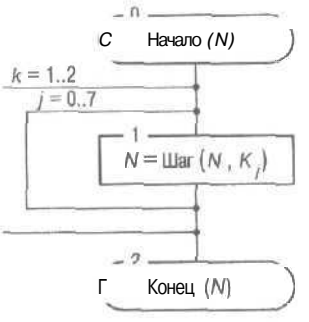

Рис. 5-4. Схема цикла выработки имитовставки

Схемы базовых циклов показаны на рис. 5-2 —5-4. В каждой из них принимается в качестве аргумента и возвращается в качестве результата 64-битный блок данных, обозначенный на схемах *N*. Символ Шаг(*N*, *X*) обозначает выполнение основного шага криптопреобразования для блока *N* с использованием ключевого элемента *X*. Между циклами шифрования и вычисления имитовставки есть еще одно отличие — в конце базовых циклов шифрования старшая и младшая часть блока результата меняются местами, это необходимо для обратимости, а для цикла вычисления имитовставки этого не делается.

Основные режимы шифрования

ГОСТ 28147-89 предусматривает три режима шифрования данных:

- простая замена;
- гаммирование;
- гаммирование с обратной связью.

В любом из этих режимов данные обрабатываются блоками по 64 бита, на которые разбивается массив данных, подвергаемый криптографическому преобразованию. Однако в двух режимах гаммирования предусмотрена возможность обработки неполного блока данных размером меньше 8 байт, что существенно при шифровании массивов данных с произвольным размером.

Прежде чем перейти к рассмотрению конкретных алгоритмов криптографических преобразований, необходимо пояснить обозначения, используемые на схемах в следующих разделах:

$T_о$, $T_ш$ — массивы соответственно открытых и зашифрованных данных;

T'', $T^{Ш}$, — i-тые по порядку 64-битные блоки соответственно открытых и зашифрованных данных:

$T_О = (T_1^О, T_2^О, ..., T_n^О)$, $T_Ш \equiv (T_1^Ш, T_2^Ш, ..., T_n^Ш)$, $i = 1, ..., n$.

Последний блок может быть неполным:

$|T_i^О| = |T_i^Ш| = 64$ при $1 \le i < n$, $1 \le |T_n^О| = |T_n^Ш| \le 64$;

n — число 64-битных блоков в массиве данных;

$Ц_Х$ — функция преобразования 64-битного блока данных по алгоритму базового цикла «Х».

Теперь пора познакомить вас с основными режимами шифрования, О них и пойдет речь в следующих разделах.

Простая замена

Зашифрование в режиме простой замены заключается в применении цикла зашифрования к блокам открытых данных, расшифрование — применение цикла расшифрования к блокам зашифрованных данных (рис. 5-5). Это наиболее простой из режимов, 64-битовые блоки данных обрабатываются в нем независимо друг от друга. Схемы алгоритмов зашифрования и расшифрования в режиме простой замены показаны на рис. 5-5 и 5-6 соответственно.

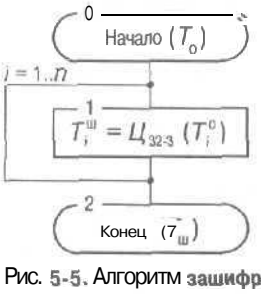

Рис. 5-5. Алгоритм зашифрования данных в режиме простой замены

Размер массива открытых или зашифрованных данных, подвергающийся соответственно зашифрованию или расшифрованию, должен быть кратен 64 битам: $|T_о| = |T_ш| = 64 \cdot n$, после

выполнения операции размер полученного массива данных не изменяется.

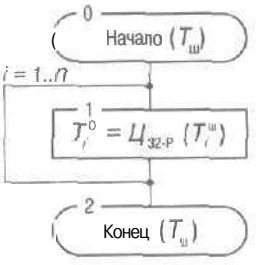

Рис. 5-6. Алгоритм **расшифрования** данных в режиме простой **замены**

Гаммирование

Режим гаммирования позволяет получить из блочного шифра поточный, т.е. такой шифр, который позволяет получить последовательность с хорошими статистическим свойствами и использоиать ее для наложения на открытый текст, применив некоторую групповую операцию (как правило, коммутативную типа XOR «исключающее или»).

Гаммирование — это наложение (снятие) на открытые (зашифрованные) данные криптографической гаммы, то есть последовательности элементов данных, вырабатываемых с помощью некоторого криптографического алгоритма, для получения зашифрованных (открытых) данных. Для наложения гаммы при зашифровании и се снятия при расшифровании в ГОСТе используется операция побитного сложения по модулю 2 (хог).

Теперь опишем режим гаммирования. Гамма для этого режима получается следующим образом: с помощью рекуррентного преобразования вырабатываются 64-битные блоки данных, которые далее подвергаются преобразованию по циклу зашифрования в режиме простой замены, в результате получаются блоки гаммы. Благодаря тому, что наложение и снятие гаммы осуществляется при помощи одной и той же операции побитового «исключающего или», алгоритмы зашифрования и расшифрования в режиме гаммирования идентичны, их общая схема показана на рис. 5-7.

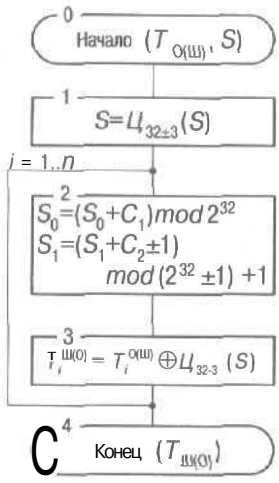

Рис. 5-7. Алгоритм **зашифрования (расшифрования)** данных **в режиме гаммирования**

Функция, используемая для выработки гаммы, является рекуррентной:

$W_{i+1} = f(W_i)$, где W_i — элементы рекуррентной последовательности, f — функция преобразования. Элемент W_0 является параметром алгоритма для режимов гаммирования, на схемах он обозначен как *S,* и называется в криптографии *синхропосылкой,* а в ГОСТ — *начальным заполнением* одного из регистров. Разработчики ГОСТ решили использовать для инициализации не непосредственно синхропосылку, а результат ее преобразования по циклу зашифрования: $W_0 = Ц_{32\,3}(S)$.

Таким образом, последовательность элементов гаммы для использования в режиме гаммирования однозначно определяется ключевыми данными и синхропосылкой. Естественно, для обратимости процедуры шифрования в процессах зашифрования и расшифрования должна использоваться одна и та же синхропосылка. Из требования уникальности (или однократности использования) гаммы (см. главу 1, где данный вопрос рассматривается подробно), невыполнение которого приводит к снижению стойкости шифра, следует, что для шифрования двух различных массивов данных на одном ключе необходимо обеспечить использование различных синхропосылок.

Упомянутая рекуррентная функция имеет следующий вид:

* в 64-битовом блоке старшая и младшая части обрабатываются независимо друг от друга:

$$\Omega_i = (\Omega_i^0, \Omega_i^1),\ |\Omega_i^0| = |\Omega_i^1| = 32,\ \Omega_{i+1}^0 = \hat{f}(\Omega_i^0),\ \Omega_{i+1}^1 = \tilde{f}(\Omega_i^1)$$

* рекуррентные соотношения для старшей и младшей частей следующие:

$$\Omega_{i+1}^0 = (\Omega_i^0 + C_1) mod 2^{32}, \text{ где } C_1 = 1010101_{16}$$

$$\Omega_{i+1}^1 = (\Omega_i^1 + C_2 - 1) mod (2^{32} - 1) + 1, \text{ где } C_2 = 1010104_{16}$$

Нижний индекс в записи числа означает его систему счисления.

Второе выражение нуждается в пояснении, так как в тексте ГОСТа приведено другое выражение:

$\Omega_{i+1}^1 = (\Omega_i^1 + C_2) mod (2^{32} - 1)$, с тем же значением константы C_2. Но далее в тексте стандарта приводится комментарий, из которого следует, что под операцией взятия остатка по модулю $2^{32} - 1$ понимается некая специфическая операция. Отличие заключается в том, что согласно ГОСТ $(2^{32} - 1) mod (2^{32} - 1) = (2^{32} - 1)$, а не 0. На самом деле это упрощает реализацию формулы, а математически корректное выражение для нее приведено выше.

Схема алгоритма шифрования в режиме гаммирования показана на рис. 5-7, ниже изложены пояснения к схеме.

Определяем исходные данные для основного тага криптопреобразования:

* $T_{o(m)}$ — массив открытых (зашифрованных) данных произвольного размера, подвергаемый процедуре зашифрования (расшифрования), по ходу процедуры массив подвергается преобразованию порциями по 64 бита;
* S — синхропосылка, 64-битный элемент данных, необходимый для инициализации генератора гаммы;
* начальное преобразование синхропосылки, выполняемое для устранения статистических закономерностей. Используется как начальное состояние рекуррентной функции.

Приведем процедуру реализации шифрования в режиме гаммирования на языке С [процедура gamm, sp — синхропосылка, din — входной массив данных, dour. — выходной массив данных (после наложения гаммы), k — ключ].

```
void gamm(unsigned long *sp,unsigned long *din,unsigned
long *dout,int n,unsigned long *k)
{
unsigned long e,d,a[2];
a[0]=*(sp++);
```

```
a[1]=*sp;
pz(a,k);
d=a[0];
e=a[1];
while(n->0)
 {
 a[0]=d+=0x01010101;
a[1]=e=e+0x01010104+((((e>>16)&0xffff)+0x0101)+
   (((e&0xffff)+0x0104)>>16)>>16);
pz(a,k);
*(dout++)=a[0]^*(din++);
*(dout++)=a[1]^*(din++);
}
}
```

pz — функция зашифрования в режиме простой замены.

Для проверки соответствия реализации описанному выше алгоритму используйте следующий тестовый пример:

```
unsigned long Key01[8]={
0x00000000,0x00000000,0x00000000,0x00000000,
0xFFFFFFFF,0xFFFFFFFF,0xFFFFFFFF,0xFFFFFFFF};
unsigned long
Plaintext01[4]={0xCCCCCCCC,0x33333333,0x33333333,
   0xCCCCCCCC};
unsigned long InitVector01[2]={0x11111111,0x22222222};
unsigned long xtext[4];
```

При вызове функции gamm с приведенными параметрами

```
gamm(InitVector01,Plaintext01,xtext,2,Key01);
```

в результате в массиве xtext должно получиться

```
936a448d 7c85df99
ad763abc 185e2719
```

Итак, при гаммировании 64-битный элемент, выработанный рекуррентной функцией, подвергается процедуре зашифрования по циклу 32-3, результат используется как элемент гаммы для зашифрования (расшифрования) очередного блока открытых (зашифрованных) данных того же размера.

Результат работы алгоритма — зашифрованный (расшифрованный) массив данных.

Режим гаммирования имеет особенность, которая была отмечена в первой главе. В этом режиме биты массива данных шифруются независимо друг от друга. Таким образом, каждый бит шифртекста зависит от соответствующего бита открытого текста и, естественно, от порядкового номера бита в массиве:

$t_i^{ш} = t_i^{о} \oplus \gamma_i = f(t_i^{о}, i)$.

Из этого следует, что изменение бита шифртекста на противоположное значение приведет к аналогичному изменению бита открытого текста на противоположный:

$$\overline{t_i}^{ш} = t_i^{ш} \oplus 1 = (t_i^{о} \oplus \gamma_i) \oplus 1 = (t_i^{о} \oplus 1) \oplus \gamma_i = \overline{t_i}^{о} \oplus \gamma_i ,$$

где обозначает инвертированное по отношению к £ значение бита($\overline{0} = 1$, $\overline{1} = 0$).

Данное свойство дает злоумышленнику, имеющему доступ к зашифрованному тексту, возможность целенаправленно изменять открытый текст, полученный после расшифрования, не обладая при этом секретным ключом. Однако отмеченное свойство режима гаммирования не должно рассматриваться как его недостаток.

Гаммирование с обратной связью

Данный режим похож на режим гаммирования и отличается от него только способом выработки элементов гаммы — очередной элемент гаммы вырабатывается как результат преобразования по циклу 32-3 предыдущего блока зашифрованных данных, а для зашифрования первого блока массива данных элемент гаммы вырабатывается как результат преобразования по тому же циклу синхропосылки. Этим достигается так называемое зацепление блоков — каждый блок шифртекста в этом режиме зависит от соответствующего и всех предыдущих блоков открытого текста. Поэтому данный режим иногда называется *гаммированием с зацеплением блоков* или *гаммированием с зацеплением*. На стойкость шифра зацепление блоков не оказывает существенного влияния.

Схема алгоритмов зашифрования и расшифрования в режиме гаммирования с обратной связью показана па рис. 5-8.

Рассмотрим пример реализации процедуры гаммирования с обратной связью.

```
// Зашифрование данных в режиме гаммирования с обратной связью
// Возвращаемое значение:
// нет
// Параметры:
// Key - ключ для шифрования
// InData - исходные открытые данные
// Size - длина исходных данных в байтах
// Salt - синхропосылка
// OutData - полученные зашифрованные данные
void GammaEncrypt
```

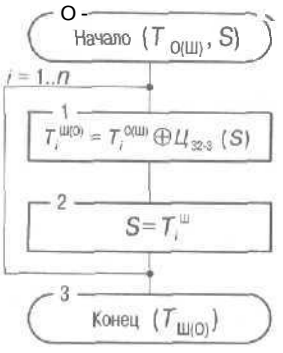

Рис. 5-8. Алгоритм зашифрования (расшифрования) данных в режиме гаммирования с обратной связью

Для тестирования данного режима можно использовать следующие данные:

```
static unsigned long Key01[8]=
{
0x00000001, 0x00000001, 0x00000001, 0x00000001,
0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF};
static unsigned long
InitVector01[2]={0xC3A7802A, 0x47E3A8FF};
static unsigned long
Plaintext01[4]={0x55555555, 0xAAAAAAAA, 0x0, 0xCCCCCCCC};
GammaEncrypt(Key01, Plaintext01, 16, InitVector01, out_d, out_s);
```

При вызове функции GammaEncrypt в переменной out_d получим значения:

```
0x807f0572  0x99e6fa4
0xbb42cb56  0x7e336363
```

Шифрование в режиме гаммирования с обратной связью исключает целенаправленное влияние искажений шифртекста на соответствующий открытый текст. Для сравнения запишем функции расшифрования блока для обоих упомянутых режимов:

$T^{G} = T_i^{Ш} \oplus \Gamma_i$, гаммирование;

$T'' - T_i^{Ш} \oplus Ц_{32\text{-}3}(T_{i-1}^{Ш})$, гаммирование с обратной связью.

Если в режиме обычного гаммирования изменения в определенных битах шифротекста влияют только на соответствующие биты открытого текста, то в режиме гаммирования с обратной связью ситуация несколько сложнее. Как видно из соответствующего уравнения, при расшифровании блока данных в режи-

ме гаммирования с обратной связью блок открытых данных зависит от соответствующего и предыдущего блоков зашифрованных данных. Поэтому, если внести искажения в зашифрованный блок, то после расшифрования искаженными окажутся два блока открытых данных — текущий искаженный и следующий за ним, причем искажения в первом случае будут носить тот же характер, что и в режиме гаммирования, а во втором случае — как в режиме простой замены. Другими словами, в соответствующем блоке открытых данных искаженными окажутся те же самые биты, что и в блоке шифрованных данных, а в следующем блоке открытых данных все биты независимо друг от друга с вероятностью 0,5 изменят свои значения.

Выработка имитовставки к массиву данных

Рассматривая режимы шифрования, мы отметили влияние искажений в шифртексте на расшифрованный текст. При расшифровании в режиме простой замены соответствующий блок открытых данных оказывается искаженным непредсказуемым образом, а при расшифровании блока в режиме гаммирования изменения предсказуемы. В режиме гаммирования с обратной связью искаженными оказываются два блока, один предсказуемым, а другой непредсказуемым образом. Необходимо учесть, что непредсказуемые изменения в расшифрованном блоке данных удается обнаружить только и случае *избыточности* этих данных, причем чем больше степень избыточности, тем вероятнее обнаружение искажения. Большая избыточность характерна, например, для текстов па естественных и искусственных языках, в этом случае факт искажения обнаруживается с высокой вероятностью. Однако в других случаях, например при искажении звуковых или графических объектов, мы получим другой образ, искажения в котором могут быть и не обнаружены.

Для решения задачи обнаружения искажений *к* зашифрованном массиве данных с заданной вероятностью в ГОСТ предусмотрен дополнительный режим криптографического преобразования — выработка имитовставки. Имитовставка — это контрольная комбинация, зависящая от открытых данных и секретной ключевой информации. Она предназначена для обнаружения всех случайных или преднамеренных изменений в массиве информации.

Для потенциального злоумышленника, если он не владеет ключевой информацией, практически неразрешимы две задачи:

* вычисление имитовставки для заданного открытого массива информации;
* подбор открытых данных под заданную имитовставку.

Схема алгоритма выработки имитовставки показана па рис. 5-9. В качестве имитовставки берется часть блока, полученного на выходе — как правило, 32 младших бита. При выборе размера имитовставки надо учитывать, что вероятность успешного навязывания ложных данных (при неизвестном злоумышленнику ключе) составляет порядка 2^{-L} на одну попытку навязывания (L — длина контрольной комбинации в битах). При использовании имитовставки размером 32 бита эта вероятность равна 2^{-32} (величина порядка 10^{-9}).

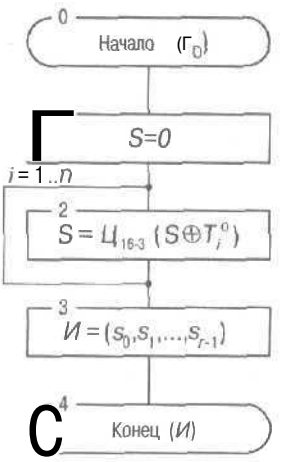

Рис. 5-9. Алгоритм выработки имитовставки для массива данных

Оптимизация ГОСТ на языке высокого уровня

Реализация криптографических алгоритмах на процессороориентированных языках (см. приведенный ранее пример реализации ГОСТ на Ассемблере Intel80386) при хороших скоростных характеристиках тем не менее не позволяет добиться переносимости реализаций для различных аппаратных платформ и операционных сред. Поэтому сейчас речь пойдет о процессе оптимизации реализаций базового цикла шифрования на языке высокого уровня.

Основная идея оптимизации — использование предвычислений, а именно — объединение нелинейного преобразования в единый блок подстановки и сдвига.

Преобразование циклического сдвига, выполняемое на 32-битном векторе, можно представить в виде подстановки R степени 2^{32}. При этом верхняя строка подстановки — вектор x_i, а нижняя — результат воздействия циклического сдвига $S(x)$ на вектор x_j, $j = 0, 1, ..., 2^{32}-1$.

Рассмотрим подстановки P1, ..., P4 степени 256, представляющие собой развернутые узлы замены (см. выше). Во введенной подстановке R выделим элементы верхней строки вида:

$z_{1i} = P1(y_j)\|t\|t\|t,$

$z_{2i} = t\|P2(y_i)\|t\|t,$

...,

$z_{4i} = t\|t\|t\|P4(y_i), t=0.$

Используя тождество $(x1\|0) \text{ xor } (0\|x2) = (x1\|x2)$ и рассматривая суммарное преобразование y. в z_{ki} и затем в $R(z_{ki})$, получим искомый метод оптимизации.

Таким образом, получаем преобразование 32-битных векторов на входе нелинейного преобразования в 32-битные блоки, где результат воздействия подстановки подвергнут циклическому сдвигу в рамках 4-байтного вектора.

Приведем процедуру получения некой «суммарной» таблицы нелинейного преобразования из развернутых узлов замены размером 4 x 256 байт.

```
void MakeLongtUz(char *pod, unsigned long PB _)
{
 int i,j;
 for(i=0;i<1024;i++) PB__[i]=pod[i];
 for(i=0;i<4;i++)
 for(j=0; j<256; j++) PB__[i*256+j]<<=(8*i);
 for(i=0;i<1024;i++)PB__[i]=(PB [i]<<11) | (PB__[i]>>21);
}
```

Переменная pod содержит исходные узлы, а переменная PB__ (unsigned long PB _[1024]) — развернутый и сдвинутый узел замены.

В этом случае одна итерация запишется следующим образом:

```
#define ONE(i, key, NN1, NN2, uz) \
i \
 *(unsigned long *)r=NN1+key[i]; \
NN2=NN2^uz[r[0]]^(uz+256)[r[1]]^(uz+512)\
[r[2]]^(uz+768)[r[3]]; \
}
```

где uz = PB___, а переменная r описана как unsigned char r[4].

Предлагаемая реализация позволяет существенно повысить скорость шифрования и вычисления имитовставок, особенно для реализаций ГОСТ, предназначенных для Windows NT/2000.

Основы программной реализации алгоритмов асимметричной криптографии

Теперь поговорим о реализации базовых процедур, используемых для вычисления и проверки электронной цифровой подписи, а также для генерации открытых ключей. При описании процедур использованы работы ведущего российского криптографа М. Грунтовича.

Будем называть числа, двоичное представление которых имеет больше разрядов, чем используется процессором для представления целых чисел, длинными числами.

В процедурах генерации и проверки подписей используются следующие операции по модулю длинного простого числа: сложение двух длинных чисел, вычитание двух длинных чисел. умножение двух длинных чисел, возведение одного длинного числа в степень другого длинного числа. Однако мы начнем с организации хранения длинных чисел л памяти.

Внутреннее представление длинных чисел в памяти компьютера

Обозначим через m длину Б битах машинного слова, которое используется в качестве минимального элемента памяти. Конкретный выбор зависит от того, какой максимальной длины элемент данных вы можете умножать без потери разрядов. Для процессоров Intel, начиная с модели 80386, m = 32. Для процессора Digital Alpha можно использовать m = 64.

В языке С удобно представлять 32-битные массивы данных типом *unsigned long*. Для m = 64 единообразия пока нет. Некоторые компиляторы поддерживают тип данных *unsigned long long*, другие — *unsigned longlong*,в Microsoft Visual C++ и Borland Builder можно использовать *unsigned___int64*. В любом случае везде далее в текстах программ мы применяем тип с именем DIGIT для обозначения беззнакового типа данных длины m бит, а m — как символьную константу с подходящим значением.

Булем хранить длинные числа, с которыми работает программа, в массивах m-разрядных слов. Опишем, каким образом про-

извольное неотрицательное целое число X превращается в последовательность m-разрядных слов.

Фиксируем в качестве основания системы счисления число $b = 2^m$. Как известно, любое неотрицательное целое число $X < b^n$ можно разложить по степеням b:

$$X = \sum_{i=0}^{n-1} X_i b^i,$$

где X_i — некоторые неотрицательные целые числа меньше b. Число X сохраняется в памяти компьютера как массив m-битовых слов с элементами X[i] = X_i. Числа X_i называются *цифрами* числа X, а число X называется n-значным длинным числом.

Таким образом, число X представляется в программе как массив:

```
DIGIT X[n];   /* n - константа */
```

Помимо базового типа DIGIT нам также понадобится тип данных в 2 раза длиннее (т.е. беззнаковое целое длины 2т бит). Назовем его TWODIGIT. Он пригодится при реализации арифметических операций для того, чтобы не терять старшие биты результата.

Для 2т-битного числа T обозначим LODIGIT(T) число типа DIGIT, состоящее из младших m разрядов, а HIDIGIT(T) — число типа DIGIT, состоящее из старших m разрядов числа T. На языке C это реализуется двумя макроопределениями:

```
#define LODIGIT(T) ((DIGIT)(T))
#define HIDIGIT(T) ((DIGIT)((T)>>m))
```

Для двух m-битных чисел A и B обозначим MAKELONG(A,B) 2m-битное число, чьи младшие биты совпадают с числом A, а старшие — с числом B. Соответствующий макрос:

```
#define MAKELONG(A,B) ((A)|((TWODIGIT)(B)<<m))
```

Кроме этих макроопределений мы в дальнейшем будем использовать несколько элементарных функций для работы с длинными числами, реализация которых послужит простым упражнением для читателя.

Функции Assign и AssignDigit выполняют операции присвоения A := B и A := x, соответственно, где A и B — n-значные длинные числа, x — цифра:

```
void Assign(
    DIGIT At],
    const DIGIT B[],
    int n);
```

```
void AssignDigit(
    DIGIT A[],
    DIGIT x,
    intn);
```

Функция Zero обнуляет число A длины n цифр, например, как вызов функции AssignDigit (A, 0, n).

```
void ZeroC
    DIGIT A[].
    intn);
```

Функция IsZero проверяет, равно ли число A длины n цифр нулю. Она возвращает ненулевое значение +1, если это так, и 0, если A ≠ 0.

```
int IsZero(
    DIGIT At],
    intn);
```

Мы используем также константу MAX_SIZE, определяющую максимальную длину длинных чисел.

Для рассматриваемого ГОСТ Р 34.10-94 MAX_SIZE*m = 512 или 1024.

Сложение и вычитание

Сложение длинных чисел

1. d:=0.
2. Для i = 0 ... n-1 выполнить шаги 3—5.
3. $T := A_i + B_i + d$
4. $C_i := LODIGIT(T)$.
5. $d := HIDIGIT(T)$.
6. Конец.

Здесь и далее:

* T — число длины 2т бит (TWODIGIT), необходимое, так как результат операции над m-битными числами имеет длину, большую т;
* d — число длины т бит (DIGIT), используемое для хранения бита переноса разряда.

Алгоритм сложения реализуется следующей функцией:

```
/* Сложение длинных чисел длины n цифр C=A+B.
   В качестве результата возвращает бит переноса d */
DIGIT Add(
    DIGIT C[],    /* результат */
```

```
   const DIGIT A[],   /* первое слагаемое */
   const DIGIT B[],   /* второе слагаемое */
   intn) /* длина слагаемых */
{
TWODIGIT T;
DIGIT (1=0;
int i;
for (1=0; i<n; i++)
{
   T = (TWODIGIT)A[i]+B[i]+d;
   C[i] ≈ LODIGIT (T);
   d = HIDIGIT (T);
)
return d;
}
```

Вычитание длинных чисел

1. d := 0.
2. Для i = 0 ... n-1 выполнить шаги 3–6.
3. $T := A_i - B_i - d$.
4. $C_i := LODIGIT(T)$.
5. Если HIDIGIT (T) = 0, то d := 0.
6. Иначе d := 1.
7. Конец.

Программная реализация этого алгоритма такова:

```
/* Вычитание длинных чисел длины n цифр C=A-B.
   В качестве результата возвращает заем старшего разряда d */
DIGIT Sub(
   DIGIT C[],   /* результат */
   const DIGIT A[],   /* уменьшаемое */
   const DIGIT B[],   /* вычитаемое */
   intn) /* длина чисел */
{
TWODIGIT T;
DIGIT d=0;
int i;
for (i=0; i<n; i++)
{
   T = (TWODIGIT)A[i]-B[i]-d;
   C[i] = LODIGIT (T);
   d = HIDIGIT (T);
   d = (0-d);
}
return d;
}
```

Кроме описанных выше операций сложения и вычитания к простейшим операциям, используемым далее, можно отнести операцию сравнения. Приводимая ниже функция Cmp выполняет сравнение двух длинных чисел. Она возвращает 1, если первое число больше второго, -1, если второе больше первого, и 0 при равенстве операндов. Вот ее текст.

```
/* Сравнение длинных чисел длины n цифр A==B */
int Cmp(
    const DIGIT A[],   /* первое число */
    const DIGIT B[],   /* второе число */
    int n) /* длина чисел */
{
int t;
for (i=n-1; (i>=0)&&(A[i]==B[i]); i-);
if (i<0) return 0;
if (A[i]>B[i]) return +1;
return -1;
}
```

Умножение

С умножением дело обстоит не так просто, как с операциями сложения и вычитания. Дело в том, что известный «школьный* алгоритм умножения «в столбик» не самый быстрый. Он показан в следующем разделе.

Умножение «в столбик» длинных чисел C=A×B длины n цифр

1. C := 0; (достаточно обнулить n младших цифр).
2. Для i = 0 ... n-1 выполнить шаги 3—8.
3. d := 0.
4. Для j = 0 ... n-1 выполнить шаги 5—7.
5. $T := C_{i+j} + A_i \times B_j + d$.
6. $C_{i+j} := LODIGIT(T)$.
7. d := HIDIGIT (T).
8. C_{i+n} :-d.
9. Конец.

Примечание Обратите внимание, что результат операции умножения в общем случае в 2 раза длиннее сомножителей, то есть имеет длину 2n цифр,

```
/* Умножение длинных чисел "в столбик" */
void Mul(
    DIGIT C[],   /* результат длины 2n цифр*/
    const DIGIT A[],   /* первый сомножитель */
    const DIGIT B[],   /* второй сомножитель */
```

```
    int n) /* длина сомножителей */
{
int i, j;
DIGIT d;
TWODIGIT T;

Zero (C, n);
for (i=0; i<n; i++)
{
        d = 0;
        for (j=0; j<n; j++)
        {
            T = (TWODIGIT)A[i]*B[j]+C[i+j]+d;
            C[i+j] = LODIGIT (T);
            d = HIDIGIT (T);
        }
        C[i+j] = d;
  }
}
```

Легко заметить, что описанный алгоритм требует выполнения n^2 операций умножения и еще некоторого количества операций сложения.

Умножение на *цифру*

Далее при реализации деления длинных чисел нам понадобится функция умножения длинного n-значного числа на цифру. В принципе, можно просто обратиться к алгоритму умножения, предварительно обобщив его на операнды различной длины. Мы поступим по-другому.

Умножение числа A длины n цифр на цифру x: C - Dxx

1. T := 0.
2. Для i = 0...n-1 выполнить 3—4.
3. $T := T/b + A_i \times x$.
4. C := LODIGIT(T).
5. C_n := HIDIGIT(T).
6. Конец.

При этом получается число C длины n+1. Даже если старшая цифра результата равна нулю, то мы требуем, чтобы для нее выделялась память.

```
/* Умножение длинного числа на цифру C = AЧx */
void  ShortMul(
    DIGIT  C[],    /* результат длины n+1 цифра */
```

```
    const DIGIT A[],   /* сомножитель длины л цифр */
    DIGIT x,  /* сомножитель длины 1 цифра */
    intn) /* длина A */
{
    TWODIGIT T = 0;
    int i;
    for (i=0;   i<n;   i++)
    {
        T = (TWODIGIT)A[i]*x + (TWODIGIT)HIDIGIT(T);
        C[i] = LODIGIT(T);
    }
    C[n] = HIDIGIT(T);
}
```

Возведение в квадрат

При рассмотрении алгоритмов возведения длинного числа в длинную степень нам придется выполнять операцию возведения в квадрат длинного числа. Один из способов ускорения этой процедуры — ускорение работы алгоритма возведения во вторую степень.

Делать это можно следующими способами;

- умножать операнд на себя самого, используя при этом один из приведенных выше алгоритмов умножения;
- применить «метод треугольника».

Возведение в квадрат умножением

Первый метод прост, действуем по определению: $C = A^2 = AxA$.

```
/* Возведение в квадрат умножением C = AxA */
ffdefine Square(C, A, n) Mul(C, A, A, n)
```

При этом требуется выполнить n^2 элементарных операций умножения и некоторое количества операций сложения.

Метод «треугольника» возведения в квадрат

Представим основание степени в b-ичной системе счисления

$$X = \sum_{i=0}^{\alpha-1} X_i b^i ,$$

Очевидно, что

$$X^2 = \sum_{i=0}^{n-1} X_i^2 b^{2i} + 2\sum_{i<j} X_i X_j b^{i+j} ,$$

Вычисление квадрата длинного числа A длины n цифр: $C = A^2$

```
C := 0;
для i = 0…n-1 выполнить 3-11;
для j = 0…i-1 выполнить 4-6;
T := T + 2×A_i×A + C_{i+j};
C_{i+j} := LODIGIT (T);
T := T/b;
T := T + A_i^2 + C_{2i};
C_{2i} := LODIGIT (T);
T := T/b + C_{2i+1};
C_{2i+1} := LODIGIT (T);
C_{2i+2} := HIDIGIT (T)
конец.
```

```
/* Вычисление квадрата длинного числа C = A² методом
    треугольника" */
void SquareTri(
   DIGIT C[],   /* результат длины 2n цифр */
   const DIGIT A[],   /* основание длины n цифр */
   int n) /* длина основания */
{
   TWODIGIT t, f;
   int i, j;

   Zero (C, 2*n);
   for (i=0; i<n; i++)
   {
      f = 0;
      for ( J=0; j<i; j++)
      \
         t = (TWODIGIT)A[i]*(TWODIGIT)A[j];
         f += (((TWODIGIT)LODIGIT(t))<<1)+C[i+j];
         Cti+J] = LODIGIT (f);
         f = HIDIGIT (f);
         f += (((TWODIGIT)HIDIGIT(t))<<1);
      }
      t = (TWODIGIT)A[i]*(TWODIGIT)A[i];
      f += LODIGIT(t)+C[i+i];
      C[i+i] = LODIGIT (f);
      f = HIDIGIT(f)+C[i+i+1]+HIDIGIT(t);
      C[i+i+1] = LODIGIT (f);
      C[i+i+2] = HIDIGIT (f);
   }
}
```

Вычисление остатка

Итак, теперь вы знаете, как умножать числа, но в алгоритмах ГОСТ Р 34.10-94 требуется вычислять произведение двух чисел по модулю простого числа. Для его реализации необходимо реализовать вычисление остатка от деления 2n-значного числа на n-значное. Эта операция называется редукцией (приведением) по модулю. Выполнять ее можно двумя способами. Первый заключается в том, что выполняют процедуру деления и берут остаток. Второй метод носит имя П. Монтгомери, который предложил модифицировать саму операцию редукции.

Деление

Алгоритм деления «лесенкой», знакомый нам из начальной школы, помогает и здесь. Просто он достаточно точно формализован. Заодно он позволяет попутно получить остаток. Приведенный здесь алгоритм построен на базе аналогичных алгоритмов. Однако для его выполнения нам потребуется деление длинного n-значного числа на короткое. Эту операцию можно выполнять с помощью алгоритма, описанного в следующем разделе.

Деление C = A/x длинного числа A дайны n цифр на цифру x с вычислением остатка r = A mod x

1. T := 0.
2. Для i = n-1...0 выполнить шаги 3—5.
3. T := T x Ь + A_i.
4. C_i := T mod x.
5. T := T/x.
6. r := T.
7. Конец.

```
/* Деление C = A/x и вычисление остатка r = A mod x (x-цифра)*/
void ShortDiv{
    DIGIT  *C, /* результат */
    const DIGIT  A[],   /* делимое */
    DIGIT  x,  /* делитель (1 цифра) */
    DIGIT  *pr,   /* остаток */
    int n) /* длина делимого */
<
TWODIGIT T = 0;
int i;
if (x==0) return  ;      /* если x == 0, то конец */
    for (i=n-1; i>=0; i-)
```

```
{
   T = MAKELONG (A[i], T);
   lf(C  !=  NULL)
      C[i] = LODISIT (T/x);
   T %= x;
}
if (pr  !=  NULL)  *pr  =  LODIGIT (T);    Л остаток */
}
```

Теперь рассмотрим алгоритм «длинного» деления. Отметим, что он никак не обрабатывает ситуацию, когда делитель равен нулю. Просто «конец» и все. В нашем контексте ситуация «деление на ноль» может возникнуть только по ошибке аппаратуры или при некорректном подборе параметров. Последнее обнаруживается при отладке, а сбой в любом случае приведет к неверному результату.

Деление «лесенкой» Q = [A/B] длинного числа A длины t цифр на B длины n цифр с вычислением остатка R = A mod B

1. R := 0; Q := 0.
2. Если B = 0, то конец.
3. Если t<n, то R := A; конец.
4. Если n=1, то $Q := [A/B_0]$; $R := A \bmod B_0$; конец.
5. $d := [b/B_{n-1}]$; // нормализующий множитель.
6. V := B x d; // нормализация.
7. U := A × d.
8. Для i = t-1 ... n выполнить шаги 9—16.
9. Если $U_i = V_{n-1}$, то $Q_{i-n} := b-1$.
10. Иначе $Q_{i-n} := (U_i \times b + U_{i-1})/V_{t-1}$.
11. Пока $Q_{i-n} \times (V_{n-1} \times b + V_{n-2}) > U_i \times b^2 + U_{i-1} \times b + U_{i-2}$ выполнить шаг 12.
12. $Q_{i-n} := Q_{i-n}-1$.
13. $U := U - Q_{i-n} \times V \times b^{i-n}$.
14. Если U < 0, то выполнить шаги 15—16.
15. $U := U+V \times b^{i-n}$.
16. $Q_{i-n} := Q_{i-n}-1$.
17. R := U/d; // денормализация.
18. Конец.

Примечание Заметим, что умножение на степени числа b означает всего лишь сдвиг второго сомножителя, а в силу нашего представления длинных чисел в памяти это означает смещение индекса в соответствующем массиве,

Определим предварительно символическую константу MAXDIGIT, равную b - 1, которая понадобится, например, для вычисления нормализующего сомножителя:

```
#define MAXDIGIT ((DIGIT)(-1))
```

Теперь приведем программную реализацию алгоритма деления.

```
/* Деление длинных чисел с получением частного и остатка */
void  Div(
    const  DIGIT  A[],   /* делимое */
    const  DIGIT  B[],   /* делитель */
    DIGIT  *Q, /* частное */
    DIGIT  *R, /* остаток */
    intt,  /* размер чисел A и Q в словах */
    intn)  /* размер чисел B и R в словах */
{
DIGIT q;     /* промежуточные результаты */
DIGIT U[MAX_SIZE*2+2];    /* нормализованное делимое */
DIGIT V[MAX_SIZE+1];     /* нормализованный делитель */
DIGIT W[MAX_SIZE+1];      /* промежуточные результаты */
TWODIGIT T;               /* промежуточные результаты */
int i, j, k;              /* индексные переменные */
DIGIT d;                 /* нормализующий множитель */
if (R!=NULL) Zero (R,n);  /* R := 0 */
if (Q!=NULL) Zero (Q,t);  /* Q := 0 */
for (i=n-1;  (i>=0)&&(B[i]==0); i-); /* анализ делителя */
n = i+1;     /* новый размер делителя */
if (n==0)  return;      /* "Деление на ноль" */
for (k=t-1;  (k>=0)&&(A[k]==0); k-); /* анализ делимого */
t = k+1;     /* новый размер делимого */
if(n > t)
{        /* B > A */
      if (R!=NULL) Assign (R,A,t); /* R := A */
      return;
}
else if (n==1)     /* размер делителя 1 */
      {
          ShortDiv (Q, A, B[0], R, t); /* упрощенный
                                          алгоритм */
          return;
      }
/* Нормализация */
d = (DIGIT)(((TWODIGIT)MAXDIGIT+1)/((TWODIGIT)B[n-1]+1));
ShortMul (V. B, d, n); /* V = B*d */
ShortMul (U, A, d, t); /* U = A*d */
```

```
U[t+1] = 0;
/* Основной цикл */
for (j=t; j>=n; j-)
{
      T = MAKELONG (U[j-1], U[j]);
      if (U[j]==V[n-1]) q=MAXDIGIT;
      else q=(DIGIT)(T/V[n-1]);
      т x= V[n-1];
      if ((TWODIGIT)V[n-2]*q>MAKELONG ((DIGIT)T, U[j-2]))
      while ((TWODIGIT)V[n-2]*q >
           MAKELONG(U[j-2],(DIGIT)(T-(TWODIGIT)q*V[n-1])))
         q-;
      ShortMul (W, V, q, n); /* W = V*q */
      Sub (U+j-n, U+j-n, W, n+1);
      if(U[j+1])     /* U < 0 */
      {
          U[j] += Add (U+j-n, U+j-n, V, n);
          q-;
      }
      if (Q!=NULL) Q[j-n] = q;/* цифра частного */
}
if (R!=NULL)
      ShortDiv (R, U, d, NULL, n); /* денормализация
                                      остатка */
}
```

Как видно, алгоритм достаточно сложен. И хотя время его работы есть величина порядка n^2, практически вычислять остаток таким образом неэффективно. Его можно использовать там, где требуется выполнить несколько операций, и выигрыш описанного далее метода Монтгомери практически отсутствует.

Модулярное умножение

Операцию умножения по модулю можно выполнять следующим образом. Если A и B < P, то C = (A * B) (mod P) вычисляется так:

D := A * B;

C := D mod P.

```
/* Умножение длинных чисел по модулю: C=(A*B) mod P */
void MulMod(
   DIGIT C[],   /* результат */
   const DIGIT  A[],   /* первый сомножитель */
   const DIGIT  B[],   /* второй сомножитель */
   const DIGIT  P[],   /* модуль */
   intn)  /* длина чисел */
{
```

```
DIGIT D[MAX_SIZE*2];
Mul (D, A, B, n);
Div (D, P, NULL, C, 2*n, n);
}
```

В данном случае опять обходимся без динамического выделения памяти, поскольку знаем заранее максимальную длину операндов.

Аналогичным образом выполняется вычисление квадрата по модулю.

C = A^2 (mod P) вычисляется так:

D:= A^2;

C:= D mod P.

```
/* Вычисление квадрата длинного числа по модулю: C=A^2 mod P */
void SquareMod(
    DIGIT C[],    /* результат */
    const DIGIT At],    /* основание степени */
    const DIGIT P[],    /* модуль */
    intn) /* длина чисел */
{
DIGIT D[MAX_SIZE*2];
SquareTri (D, A, n);
Div (D, P, NULL, C, 2*n, n);
>
```

Операция Монтгомери

Однако поступать описанным выше образом при вычислении произведений и квадратов по простому модулю в силу сложной операции деления неэффективно, поэтому предлагается слегка модифицировать операцию умножения на множестве чисел {0,1...P-1}. Определим новую операцию умножения следующим образом:

K o L=(mod P), где $E = b^n$ (mod P).

Желающие могут убедиться, что множество целых чисел {0,..., P - 1} с операциями «+» и «o» является полем с единицей E. Обозначим его MF(P). Это поле изоморфно полю GF(P). Изоморфизм

f: GF(P) a MF(P)

задается формулой:

f(K) = K x E (mod P).

Обратное отображение f^{-1}: MF(P) a GF(P)

$f^{-1}(S) = S$ o 1.

Для операции Монтгомери существует оптимизированный алгоритм, не требующий деления в поле GF(P), которое бы существенно замедлило вычисления.

Вычисление произведения Монтгомери S = KoL (mod P); при этом z = $-P_0^{-1}$ (mod b)

1. S:=0.
2. Для i = 0...n-1 выполнить шаги 3—5.
3. S := S + K × L_i.
4. e := S_0 × z (mod b).
5. S := (S + e × P) / b.
6. Если S > P, то S := S - P.
7. Конец.

Следует заметить, что результат сложения S на шаге 5 может иметь значение длины n + 1 цифра.

Число z можно вычислить на первом шаге алгоритма, но поскольку оно зависит только от модуля P, а тот не изменяется при выполнении, скажем, алгоритма возведения в степень, то удобно заранее подготовить этот параметр.

Как видно из алгоритма, выигрыш в том, что на каждом шаге выполняется умножение для вычисления очередного значения e, а не деление, как в алгоритме деления. Кроме того, деления на b выполнять не надо: оно равносильно смещению индекса. В программе, показанной далее, вы увидите, как это делается. Не требуется также модификация очередной цифры частного, которая то и дело возникала при делении. В общем, при той же теоретической сложности порядка n^2 операция Монтгомери практически выполняется быстрее обычного модулярного умножения.

```
/* Умножение Монтгомери длинных чисел: S=(KoL) mod P */
void MulMont(
    DIGIT S[],    /* результат */
    const DIGIT K[],   /* первый сомножитель */
    const DIGIT L[],   /* второй сомножитель */
    const DIGIT P[],   /* модуль */
    DIGIT z, /* -P0^-1 (mod b) */
    intn) /* длина чисел */
{
    intj, i;
    DIGIT T[MAX_SIZE+1];
    TWODIGIT ulTmp;
```

```
    DIGIT uiTmp, e;
    Zero (T, n+1);        /* T = 0 */
    for (i=0; i<n; i++)
    {       Л T += K * L[i] */
        ulTmp = 0;
        uiTmp = L[i];
        for (j=0; j<n; j++)
        {
            ulTmp = (TWODIGIT)K[j] * uiTmp +
                (TWODIGIT)HIDIGIT (ulTmp) + (TWODIGIT)T[j];
            T[j] = LODIGIT (ulTmp);
        }
        T[n] += HIDIGIT (ulTmp);
        e = T[0]*z;
            /, т += е * P и сдвиг влево */
        ulTmp = (TWODIGIT)e*P[0] + (TWODIGIT)T[0];
        for (j=1; j<n; j++)
        {
            ulTmp = (TWODIGIT)e * P[j] +
                (TWODIGIT)HIDIGIT (ulTmp) + (TWODIGIT)T[j];
            T[j-1] = LODIGIT (ulTmp);
        }
        UlTmp = (TWODIGIT)HIDIGIT (ulTmp) + (TWODIGIT)T[n];
        T[n-1] = LODIGIT (ulTmp);
        T[n] = HIDIGIT (ulTmp);
    }
    if (T[n])     /* S i P */
        Sub (T, T, P, n);
    while (Cmp (T, P, n) >=0) /* S i P */
        Sub (T, T, P, n);
    Assign (S, T, n);
}
```

Для корректного использования операции Монтгомери при выполнении модулярного возведения в степень необходимы функции, реализующие изоморфизмы f и f^{-1}, определенные выше.

```
/* Вычисление S = f(A) */
void GF2HF(
    DIGIT S[],   /* результат */
    const DIGIT A[],   /* аргумент */
    const DIGIT P[],   /* модуль */
    intn) /* длина чисел */
{

DIGIT D[MAX_SIZE*2];
```

```
    Zero (D, n);
    Assign (&D[n], A, n);
    Div (D, P, NULL, S, 2*n, n);
}

/* Вычисление A = f^-1(S) */
void MF2GF(
    DIGIT A[],   /* результат */
    const DIGIT S[],   /* аргумент */
    const DIGIT P[],   /* модуль */
    DIGIT z, /* -P0^-1 (mod b) */
    int n) /* длина чисел */
{
    DIGIT T[MAX_SIZE];
    AssignDigit (T, 1, n);
    MulMont (A, S, T. P, z, n);
}
```

Последнее, что необходимо определить для того, чтобы использовать алгоритм Монтгомери — это нахождение параметра z. Так получилось, что нам придется решать задачу нахождения мультипликативного обратного для *произвольного* числа по модулю P.

Возведение в степень

Решим задачу вычисления значения $C = A^B \pmod P$, где P — простое число длины n цифр; A и C — числа меньшие P; B — число длины s цифр.

Бинарный алгоритм возведения в степень

Известный всем из школы алгоритм возведения в степень B путем B - 1 умножения очень неэффективен. При малых значениях B он еще приемлем, но при B порядка 2^{256} его сложность превышает всякие разумные пределы. Однако можно воспользоваться алгоритмом, который называется бинарным алгоритмом (иногда его также называют схемой Горнера или дихотомическим алгоритмом).

Идея его построения состоит в следующем. Число B можно разложить но степеням 2 следующим образом:

$$B = \sum_{i=0}^{m\le=1} 2^i \cdot B_i \text{, где } B_i = 0 \text{ или } 1.$$

Нетрудно видеть, что тогда

$$A^B = A^{B_0} \cdot \left(A^{B_1} \cdot \left(\ldots \cdot \left(A^{B_{ms-1}} \right)^2 \ldots \right)^2 \right)^2$$

Бинарный алгоритм возведения в степень $C = A^B \pmod P$

1. $C := 1$.
2. Для i = ms−1...0 выполнить шаги 3–4.
3. $C := C^2 \pmod P$.
4. Если $B_i = 1$, то $C := C \times A \pmod P$.
5. Коней.

Этот алгоритм реализуется следующей функцией:

```
/* Бинарный алгоритм возведения в степень по модулю:
   C = A^B (mod P) -/
void PowerModBinary(
   DIGIT C[],    /* результат */
   const DIGIT A[],    /* основание степени */
   const DIGIT B[],    /* показатель степени */
   const DIGIT P[],    /* модуль */
   ints,  /* длина показателя */
   intn) /* длина модуля */
{
tnt i,j;
AssignDigit (C, 1, n);
for (i=s-1; i>=0; i-)
       for (j=m-1; j>=0; j-)
       {
       SquareMod (C, C, P, n);
       if((B[i]>>j)&1)
              MulMod (C, C, A, P, n);
       }
}
```

Теперь покажем, как использовать операцию Монтгомери для вычисления степени по модулю. При этом бинарный алгоритм возведения слегка видоизменяется. Идея такова:

$C = A^B \pmod P = f^{-1}(f(A)^B) \pmod P$. Поскольку f— изоморфизм полей, то эта формула верна.

Бинарный алгоритм возведения в степень $C = A^B \pmod P$ с использованием операции Монтгомери

1. $z := -P_0^{-1} \pmod b$.
2. $C := f(1)$.
3. $D := f(A)$.

4. Для i = ms−1...0 выполнить шаги 3–4.
5. C := CoC (mod P).
6. Если B =1, то C := CoD (mod P).
7. C := f^{-1}(C).
8. **Конец.**

Программно этот алгоритм реализуется следующим образом,

```
void PowerModBinaryUsingMont(
   DIGIT C[],   /* результат */
   const DIGIT A[],   /* основание степени */
   const DIGIT B[],   /* показатель степени */
   const DIGIT P[],   /* модуль */
   ints, /* длина показателя */
   intn) /* длина модуля */
{
int i,j;
DIGIT z;
DIGIT D[MAX_SIZE];
z = Findz(P[0]);
AssignDigit (C, 1, n);
GF2MF(C,C,P,n);
GF2MF(D,A,P,n);
for (i=s-1; i>=0; i-)
   for (j=m-1; j>=0; j-)
{
      SquareMont (C, C, P, z, n);
      if(B[i]&(1<<j))
         MulMont (C. C, 0, P, z, n);
   }
HF2GF(C, C, P, z, n);
}
```

Мы как будто замедлили вычисления, введя дополнительные операции вычисления f(1), f(A), f^{-1}(C). Однако при этом удается ускорить вычисление произведений, количество которых больше, так что в сумме значительно выигрываем.

В дальнейшем мы не будем приводить тексты программ, использующих операцию Монтгомери. Для простоты используются функции MulMod и SquareMod. При желании вы сами сможете сделать необходимые изменения способом, показанным ранее.

Вычисление степени с помощью заранее заготовленной таблицы

Опять воспользуемся формулой и запишем степень A^B следующим образом:

$$A^B = \prod_{i=0}^{ms-1} \left(A^{2^i}\right)^{B_i},$$

где все произведения вычисляются по модулю P. Если показатель степени A известен заранее или редко изменяется, то можно заготовить таблицу степеней вида $Tab[i] = A^{2^i}$ и при конкретных вычислениях просто перемножать те ее элементы, которые потребуются. Опишем это точнее.

Предварительные вычисления для табличного алгоритма возведения в степень $Tab_i = A^{2^i} \pmod P$

1. $Tab_0 := A$.
2. Для i = 1...ms−1 выполнить шаг 3.
3. $Tab_i := Tab_{i-1}^2 \pmod P$.
4. Конец.

Табличный алгоритм возведения в степень $C = A^B \pmod P$ с использованием предварительно вычисленных значений $Tab_i = A^{2^i} \pmod P$

1. C := 1.
2. **Для i = 0...ms−1 выполнить шаг 3.**
3. Если $B_i \neq 0$, то $C := C \times Tab_i \pmod P$.
4. Конец.

```
/* Предварительные вычисления табличного алгоритма
   возведения в степень по модулю: Tab_i = A^(2^i) (mod P) */void
PreparePowerModTabbed(
   DIGIT Tab[], /* таблица длины msn цифр */
   const DIGIT A[],   /* основание степени */
   const DIGIT P[],   /* модуль */
   ints,  /* длина показателя */
   intn)  /* длина модуля */
{
int i;
Assign (Tab, A, n);
for (i=1; i<m*s; i++)
   SquareMod (&Tab[i*n], &Tab[(i-1)*n], P, n);
}
/* Табличный алгоритм возведения в степень по модулю:
   C = A^B (mod P) */
void PowerModTabbed(
   DIGIT C[],   /* результат */
   const DIGIT Tab[],/* таблица степеней основания */
```

```
    const DIGIT B[],   /* показатель степени */
    const DIGIT P[],   /* модуль */
    int s,  /* длина показателя */
    int n)  /* длина модуля */
{
int i,j;
AssignDigit (C, 1, n);
for (i=0; i<s; i++)
   for (j=0; j<m; j++)
      if(B[i]&((DIGIT)1<<j))
         MulMod (C, C, &Tab[i*n], P, n);
}
```

Одновременное вычисление произведения степеней

Приведенные выше алгоритмы возведения в степень с использованием заранее подготовленных таблиц вполне применимы при вычислении значения подписи и в первом возведении при проверке ЭЦП. Однако возможно вычислять произведение нескольких степеней параллельно. Вот как это делается.

Пусть нам необходимо вычислить произведение $C = \prod_{j=1}^{t} A_j^{B_j} \pmod P$. Воспользовавшись формулой, получим

$$\prod_j A_j^{B_j} = \prod_j A_j^{B_{j0}} \cdot \left(\prod_j A_j^{B_{j1}} \cdot \left(\ldots \cdot \left(\prod_j A_j^{B_{j,m-1}} \right)^2 \ldots \right)^2 \right)^2$$

При этом B_{ij} обозначает i-й двоичный разряд числа B_j. Если предварительно заготовить таблицу значений произведений вида $\prod_{j=1}^{t} A_j^{u_j}$ для всех наборов $\{u_j\}(j=1...t)$, $u_j = 0$ или 1, то дальнейшие вычисления заключаются только в возведении во вторую степень и умножении на подходящие элементы таблицы. Совсем как в бинарном алгоритме возведения в степень.

Бинарный алгоритм одновременного вычисления произведения степеней $C = \prod_{j=1}^{t} A_j^{P} \pmod P$

1. $C := 1$.
2. $Tab_0 := 1$; $Tab_1 := A_1$; // Tab - таблица из 2^t длинных чисел.
3. Для j = 1...t-1 выполнить шаги 4—5.
4. Для i = $0...2^j-1$ выполнить шаг 5.

5. $Tab_{i+2^j} := Tab_i \times A_{j+1} \pmod P$.
6. Для t = ms— 1...0 выполнить шаги 7—11.
7. $C := C^2 \pmod P$.
8. d := 0.
9. Для j = l...t выполнить шаг9.
10. **d** := $d+B_{jl}\times 2^{j-1}$.
11. Если $d\neq 0$, то $C := C \times Tab_d \pmod P$.
12. Конец.

Оценим скорость работы этого алгоритма. Подготовка таблицы требует $2\times(2^{t-1} - 1)$ операций умножения. Основной цикл — ms квадратов и в среднем ms((t-1)/t) умножений. Итого требуется в среднем 2^t-2+ms((2t-1)/t) операций модулярного умножения.

Если m s = 256, результаты расчета трудоемкости приведены н таблице. Для сравнения даны результаты с использованием t обращений к k-арному алгоритму (330t + t-i = 331t-l умножение).

Таблица 5-4. Трудоемкость вычисления **степеней**

t	Среднее число умножений	
	Алгоритм одновременного вычисления	**k-ый алгоритм**
2	**386**	**661**
3	**432,7**	**992**
4	**462**	**1323**
5	**490,8**	**1654**

Выигрыш несомненный. В чем же дело? При возведении в степень мы на каждом шаге цикла по битам показателя вычисляем квадрат и умножаем на основание. И так для каждого из возведений. Алгоритм же одновременного вычисления произведения делает это только один раз.

Вот функция, реализующая данный алгоритм для t = 2.

```
/* Бинарный алгоритм одновременного вычисления
   произведения степеней C = A1^B1 * A2^B2 (mod P) */
void SimPowerModBinary(
   DIGIT C[],   /* результат */
   const DIGIT A1[],  /* основание степени */
   const DIGIT B1[],  /* показатель степени */
   const DIGIT A2[],  /* основание степени */
   const DIGIT B2[],  /* показатель степени */
   const DIGIT P[],   /* модуль */
   ints,  /* длина показателя */
   intn) /* длина модуля */
{
```

```
int i,j;
DIGIT Tab[4*MAX_SIZE];
DIGIT b1,b2,d;
AssignDigit (Tab,1,n);
Assign (&Tab[n],A1,n);
Assign (&Tab[2*n],A2,n);
MulMod (&Tab[3*n], &Tab[n],A2,P,n);
AssignDigit (C, 1, n);
for (1=8-1; i>=0; i-)
   for (j=m-1; j>=0; j-)
   {
      SquareMod (C, C, P, n);
      b1 = (B1[i]>>j)&1;
      b2 = (B2[i]>>j)&1;
      d = b1+b2*2;
      if(d)
         MulMod (C, C,&Tab[d], P, n);
   }
I
```

Именно такой алгоритм необходимо применять при проверке ЭЦП. Действительно, в этом случае как раз необходимо вычислять произведение двух степеней по модулю P.

Если набор чисел A_j фиксирован, то таблицу Tab можно заготовить заранее. Это позволит сократить время вычислений.

Приведенный алгоритм удается адаптировать и для *вычисления степени* при фиксированном основании (именно это требуется при вычислении подписи). Для этого представим показатель степени как $B = \sum_j B_j 2^j$, где B_j — числа длины t бит.

Тогда $C = A^B = \prod_j \left(A^{2^j}\right)^{B_j}$, и можно применить алгоритм одновременного вычисления произведения степеней. Причем, если A фиксировано, то таблицу значений A^{2^j} можно заготовить заранее.

В качестве упражнения рекомендуем вам создать функцию, реализующую такой алгоритм, и заменить ею функцию PowerModTabbedkary в реализации алгоритма вычисления подписи.

Вычисление обратного по модулю

При проверке ЭЦП необходимо вычислить величину $v = (h(M_1))^{q-2} \pmod q$. Это можно сделать по крайней мере двумя способами: возведением в степень и вычислением обратного с помощью расширенного алгоритма Евклида.

Примечание Действительно, в теории чисел известна *малая теорема Ферма,* которая утверждает, что если p - простое число, a - целое число, не делящееся на p, то $a^{p-1} \equiv 1 \pmod p$.

Отсюда следует, что $a^{p-2} \equiv a^{-1} \pmod p$.

При этом a - 1 есть такое целое число, что $aa^{-1} \equiv 1 \pmod p$. Это число называется мультипликативным обратным a по модулю p.

С другой стороны, поскольку a и p взаимно просты, то существуют такие целые числа x и y, что

$ax + py = 1$.

Если перейти к сравнениям mod p, то мы получим

$a x \equiv 1 \pmod p$.

Откуда

$x = a^{-1} \pmod p$

и значит

$x \equiv a^{p-2} \pmod p$.

Для нахождения таких чисел x и y можно использовать так называемый расширенный алгоритм Евклида.

Расширенный алгоритм Евклида, вычисления d = НОД(a,b) и чисел x и y, таких, что a x + b y = d

1. Если Ь = 0, то d := a; x := 1; y := 0.
2. $x_2 := 1$; $x_1 := 0$; $y_2 := 0$; $y_1 := 1$; a_1;=a; b_1;=b.
3. Пока b > 0 выполнить 4-5.
4. q := ла$_1$/b_1ы; г := $a_1 - qb_1$; x := $x_2 - qx_1$; y := y, - qy_1.
5. $a_1 := b_1$; b_1:= г; $x_2 := x_1$; $x_1 := x$; $y_2 := y_1$; $y_1 := y$.
6. d := a_1; x := x_2; y := y_2.
7. Конец.

Программную реализацию функции вычисления мультипликативного обратного с использованием расширенного алгоритма Евклида мы оставим читателям в качестве упражнения. Скажем только, что с его помощью можно достичь существенного ускорения.

Ниже приведен текст функции, которая вычисляет значение мультипликативного обратного возведением в степень,

```
/* Вычисление обратного C = A^-1 (mod B) */
void Inverse(
    DIGIT CП,    /* значение обратного */
    const DIGIT A[],    /* аргумент */
```

```
    const DIGIT B[],    /* модуль */
    int n) /* длина операндов */
{
    DIGIT B1[MAX_SIZE];
    Assign (B1, B, n);
    B1[0] -= 2;
    PowerModkary (C, A, B1, B, n, n);
}
```

У нас остался «долг» при реализации операции Монтгомери. Мы не рассмотрели функцию, вычисляющую параметр $z = -P_0^{-1} \pmod b$. Малая теорема Ферма в этом случае не годится, поскольку модуль в данном случае не простой (напомним, $b = 2^m$). Но можно воспользоваться ее обобщением — *теоремой Эйлера*.

Примечание Теорема Эйлера гласит:
если a и c – взаимно простые целые числа, то $a^{f(c)} = 1 \pmod c$.
(f – функция Эйлера).

Таким образом, $a^{f(c)-1} = a^{-1} \pmod c$.

В нашем случае модулем является число $b = 2^m$. Поэтому $f(b) = 2^{m-1}$ и $f(b) - 1 = 2^{m-1} - 1$ — число, двоичная запись которого состоит из m - 1 единичного разряда: $11...11_2$. Вы уже знаете возведение в степень и можете воспользоваться бинарным алгоритмом для вычисления значения $a^{-1} \bmod b$. Заметим при этом, что $a^{-1} \bmod b$ существует только при a взаимно простом с числом b, т.е. при *нечетном* a.

Алгоритм вычисления $z = -a^{-1} \pmod{2^m}$

1. Если a четно, то z=0; конец.
2. d := a.
3. Повторить т-2 раза шаги 4—5.
4. $d := d^2 \pmod{2^m}$.
5. $d := d \times a \pmod{2^m}$.
6. Конец.

```
/* Вычисление z = a^-1(mod 2^m) */
DIGIT Findz (DIGIT a)
{
    int i;
    DIGIT d;
    if  ((a&1)==0)
        return 0;
    for (i=0,d=a; i<m-2; i++)
    {
        d *= d;
```

```
        d *= a;
    }
    return (0-d);
}
```

Здесь мы предполагаем, что m — длина машинного слова, и приведение к модулю 2^m выполняется при умножении автоматически. Если это не так, необходимо выполнить эту редукцию перед возвратом результата.

Вычисление электронной цифровой подписи по ГОСТ Р 34.10-94

Поскольку для вычисления подписи требуется узнать только степень фиксированного основания, то логично воспользоваться функцией PowerModTabbedkary. При этом необходимо заранее заготовить таблицу степеней числа A mod p.

Договоримся, что функция вычисления подписи Sign возвращает значение 0 в случае успеха и значение 1, если необходимо повторить выработку случайного числа k.

```
/* Вычисление подписи по алгоритму ГОСТ Р 34.10-94 */
int Sign34_10_94 (
    DIGIT s[],    /* значение части s подписи */
    DIGIT r_[],   /* значение части r' подписи */
    DIGIT h[],    /* значение хеш-функции */
    const DIGIT k[],    /* случайное число */
    const DIGIT x[],    /* секретный ключ подписи */
    const DIGIT a[],    /* значение основания a */
    DIGIT *Tab,   /* таблица степеней a */
    const DIGIT p[],    /* модуль p */
    const DIGIT q[],    /* модуль q */
    int n,  /* длина модуля p */
    int t)  /* длина секретного ключа x */
{
    DIGIT r[MAX_SIZE], h1[MAX_SIZE];
    Div (h, q, NULL, h1, t, t);    /* h1 = h (mod q) */
    if (IsZero (h1, t))            /* если h1=0, то h1=1 */
        AssignDigit (h1, 1, t);
    PowerModTabbedkary (r, Tab, k, p, t, n); /* r = a* (mod p) */
    Div (r, q, NULL, r_, n, t);    /* r' = r (mod q) */
    if (IsZero (r_, t))            /* если r'=0, повторить */
        return 1;
    MulMod (s, k, h1, q, t);       /* s = k*h (mod q) */
    MulMod (r, x, r_, q, t);
    if( Add (s, s, r, t) || Cmp (s, q, t)>=0)/*s+=x*r'(mod q)*/
        Sub (s, s, q, t);
```

```
    if (IsZero (s, t))          /* если s=0, повторить */
        return 1;
    return 0;
}
```

Проверка электронной цифровой подписи по ГОСТ Р 34.10-94

При проверке подписи необходимо вычислять произведение двух степеней по модулю p и обратное по модулю q, для чего воспользуемся функциями SimPowerModBinary и Inverse, соответственно.

Договоримся, что функция проверки подписи CheckSign возвращает значение 0 в случае успеха и ненулевое значение, если подпись не верна.

```
/* Проверка подписи по алгоритму ГОСТ Р 34.10-94 */
int CheckSign34_10_94 (
    DIGIT s[],   /* значение части s подписи */
    DIGIT r_[],  /* значение части r' подписи */
    DIGIT h[],   /* значение хеш-функции */
    const DIGIT y[],   /* открытый ключ подписи */
    const DIGIT a[],   /* значение основания a */
    const DIGIT p[],   /* модуль p */
    const DIGIT q[],   /* модуль q */
    intn,  /* длина модуля p */
    intt)  /* длина секретного ключа x */
{
    DIGIT tmp[MAX_SIZE], h1[MAX_SIZE], z1[MAX_SIZE],
        z2[MAX_SIZE];
    DIGIT u[MAX_SIZE], v[MAX_SIZE];
    if (Cmp (s, q, t)>=0 ||   /* если s >= q */
        IsZero (s, t) ||/* or s == 0 */
        Cmp (r_, q, t)>=0 ||   /* or r' >= q */
        IsZero (r_,t))      /* or r' == 0 */
        return 1;
    Div (h, q, NULL, h1, t, t);/* h1 = h (mod q) */
    if (IsZero (h1, t))    /* если h1=0, то h1=1 */
        AssignDigit (h1, 1, t);
    Inverse (v, h1, q, t); /* v = h^(q-2) (mod q) +/
    MulMod (z1, s, v, q, t);   /* z1 = s*v (mod q) */
    Sub (z2, q, r_, t);     /* z2 = q-r' */
    MulMod (z2, z2, v, q, t); /* z2 *= v (mod q) */
        /* tmp = a^(z1)*y^(z2) (mod p) */
    SimPowerModBinary (tmp, a, z1, y, z2, p, t, n);
    Div (z2, q, NULL, u, n, t); /* u = z2 (mod q) */
    return Cmp (r_, u, t); /* r' == u ? */
}
```

Реализация алгоритма хеширования

Рассмотрим описанный в стандарте алгоритм хеширования, используемый в процедурах выработки и проверки ЭЦП. Структурируем описание в виде программы на языке С. Функция hash_0 вычисляет в выходном буфере h хеш-значение входного вектора inbuf длины len.

Процедура вычисления хеш-функции для буфера рассматривает содержимое буфера как бинарное число, причем первый байт содержит 8 самых младших бит (x7,x6,x5,x4,x3,x2,x1,xO) этого числа, второй — следующие 8 бит (x15,x14,x13,x12,x11,x10,x9,x8) и т.д.

Поэтому приведенное в приложении А к стандарту тестовое сообщение M=73657479 6220...6920 739696854 представляет собой записанный в шестнадцатеричном виде обычный текст: «This is message, length=32 bytes», а тестовое сообщение M в следующем пункте А.3.2 — текст: «Suppose the original message has length = 50 bytes». Именно эти тексты, оформленные в файлы надо предъявлять этой и другим программам для проверки на соответствие стандарту.

```
BOOL hash_0(unsigned char *inbuf,int len,unsigned char *h)
{
int i,nl,seti,end;
unsigned char m[34];
WORD l[17],s[17],vl[17];
// проверка на короткий буфер
if(len<32) return FALSE;
memmove(h,start_h,32);
// загрузка стартового вектора хеширования
for(i=0;i<16;i++) { l[i]=0; s[i]=0; vl[i]=0;}
/* этап 1 вычисления хеша, пп. 1.3 и 1.4 "Описания...P34.11" */
vl[0]=256;
/* заготовка в vl[] "длинного" представления числа 256
   для п 3.3 */
seti=0;end=0;
while(end==0) /* извлечение из буфера по 256 бит(32 байта) */
   {
// в nl число прочитанных байтов
      if(len-seti>32)
         <
         for(i=0;i<32;i++)
            m[i]=inbuf[seti+i];nl=32;seti=seti+32;
   }
else {
```

```
    for(i=0; i<len-seti;i++) m[i]=inbuf[seti+i];
        nl=len-seti;end=1;}
    for(i=nl;i<32;i++) m[i]=0;
// дополнение нулями при неполном блоке
    nl*=8; // теперь в nl число прочитанных бит
    if(nl<256) { vl[0]=nl; vl[1]=0; }
/* корректировка "длинного" представления числа бит для
пункта 2.2 этапа 2 "Описания..." */
Add(l,16,vl,16,1);
Add(s,16,(WORD *)m,16,s);
// вызов функции "шагового" хеша
    S hash(m,h); // пункт 2.5 или 3.2
    }
// финишная обработка
s_hash((unsigned char*)l,h); // пункт 2.6 этапа 2
s_hash((unsigned char*)s,h); // пункт 2.7 этапа 2
return 0;
}
```

Легко видеть, что основная процедура использует процедуру шагового хеша s_hash, входным аргументом которой является блок длиной 32 байта, а выходом — хеш-значение (также 32 байта).

```
// Сменные параметры алгоритма хеширования:
// стартовый вектор хеширования и заполнения узлов замены
// алгоритма ГОСТ 28147, приведенные здесь, взяты
// из ПРИЛОЖЕНИЯ А ГОСТ Р 34.11-94
 char start_h[32]=
{0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
 0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
 0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
 0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0};
```

Процедура суммирования двух длинных чисел,

```
void Add( WORD *U, int lU, WORD *V, int lV, WORD *W )
{
 WORD *u, *v, k;
 DWORD s;
 int lu, lv, j;
 if( lU >= lV )
// первое локальное слагаемое всегда более длинное
 u = U, lu = lU, v = V, lv = lV;
   else v = U, lv = lU, u = V, lu = lV;
   for(k=j=0;j<lv;j++)
 {
 *W++ = s = (DWORD) *u++ + *v++ + k;
 k = (s>>16) ? 1 : 0;
```

```
// в k "перенос" из суммы двух предыдущих слов
 }
 while( j++ < la )
 {
// Протаскивание "переноса" по более длинному слагаемому
 *W++ = s = (DWORD) *u++ + k;
 k = (s>>16) ? 1 : 0;
 }
 *W = k;
}
```

Рассмотрим функцию вычисления шагового хеша.

```
void s_hash(unsigned char *m, unsigned char *h)
{
/* Шаговая функция хеширования со стр.3 стандарта */
    unsigned char s[180], key[128];
    g_key(m, h, key);
    crypt(h,key,s);
    reg(m,s,h);
}
```

Данная функция использует процедуры генерации ключей g_kcy, шифрования блока crypt и реализации регистра сдвига.

```
void crypt(unsigned char *h, unsigned char *key, unsigned
char *s)
{
//Шифрующее преобразование стр.4 ГОСТа Р 34.11-94:
//Si=Ekі(Hi)i=1.....4*/
int i;
unsigned long *k;
for(i = 0; i < 4; i++) {
    k = (unsigned long *)(h+i*8);
    n1 = *k; n2 = *(k+1);
    k = (unsigned long *)(key+32*i);
    pz_z(k);
    k = (unsigned long *)(s+i*8);
    -k = n1; *(k+1) = n2;    }
}
```

Для рассмотрения функции pz_z зададим глобальные переменные. Функция Encrypt соответствует одному циклу шифрования ГОСТ в режиме простой замены, где x — ключ (32 байта). in — массив входных блоков (2 блока по 4 байта), ои - массив выходных блоков (2 блока по 4 байта).

```
unsigned long n1,n2;
/* 32-битовые накопители N1, N2 из ГОСТа 28147-89 */
void pz_z(unsigned long *x)
```

```
{
unsigned long in[2],ou[2];
in[0]=n1;in[1]=n2;
Encrypt(x, in[0],in[1],ou[0],ou[1]);
n1=ou[0];n2=ou[1];
}
```

Рассмотрим ряд *вспомогательных функций*

```
void A(unsigned char *x)
/* Преобразование A(X) = (x1+x2)|x4|x3|x2 */
{
/* из раздела 5.1 - Генерация ключей */
int i;
unsigned char buf[8];
for(i=0;i<8;i++) buf[i]=x[i]^x[i+8];
for(i=0;i<24;i++) x[i]=x[i+8];
for(i=0;i<8;i++) x[i+24]=buf[i];
}
void P(unsigned char *w,unsigned char *key)
{
/* Преобразование P - перестановка байтов */
int i,k;
for(i=0;i<=3;i++)
   for(k=0;k<8;k++) key[i+4*k]=w[8*i+k];
}
void summod2(unsigned char *x,unsigned char *y,unsigned char
*z)
{
Л Обычное сложение векторов по модулю 2, */
int i;
for(i=0;i<32;i++) z[i]=x[i]^y[i];
}
void g_key(unsigned char *m,unsigned char *h,unsigned char *k)
!
/* 5.1 Генерация ключей */
unsigned char c[96],u[32],v[32],w[32];
int i;
for(i=0;i<96;i++) c[i]=0;
   c[33]=c[35]=c[37]=c[39]=c[40]=c[42]=c[44]=c[46]=0xff;
   c[49]=c[50]=c[52]=c[55]=c[56]=c[60]=c[61]=c[63]=0xff;
for(i=0;i<32;i++) { u[i]=h[i]; v[i]=m[i]; }
summod2(u,v,w);
P(w,k);
for(i=0;i<3;i++)
   {
      A(u);
      summod2(u,c+i*32,u);
```

```
        A(v); A(v); summod2(u,v,w); P(w,k+32*(i+1));
    }
}
void s_reg(WORD *r)
/* Обратная связь преобразования пси */
{
/* 5.3 Перемешивающее преобразование */
   r[16]=r[0]^r[1]^r[2]^r[3]^r[12]^r[15];
}
void reg(unsigned char *m,unsigned char *s,unsigned char *h)
{
/* 5,3 Перемешивающее преобразование */
/* x(M,H)=psi^61(H+psi(M+psi^12(S))) */
/* Для понимания этой части программы следует иметь */
/* ввиду, что "сдвиг вправо на 2 байта" в каждом */
/* такте psi не делается, а делается один сдвиг */
/* на 148=2*(12+1+61) байтов в конце функции. */
int i;
for(i=0;i<12;i++) s_reg((WORD*)(s+2*i));
summod2(s+24,m,s+24);
s_reg((WORD*)(s+24));
summod2(s+26,h,s+26);
for(i=0;i<61;i++) s_reg((WORD*)(s+26+2*i));
for(i=0;i<32;i++) h[i]=s[i+148];
}
```

Для проверки соответствия реализации хеш-функции необходимо использовать следующий тестовый пример.

Для процедуры pz_z устанавливается узел **замены,** как показано в таблице 5-5.

Текст для хеширования:

```
unsigned char Text02[50]={
        0x53,0x75,0x70,0x70,0x6F,0x73,0x65,0x20,0x74,
        0x68,0x65,0x20,0x6F,0x72,0x69,0x67,0x69,0x6E,
        0x61,0x6C,0x20,0x6D,0x65,0x73,0x73,0x61,0x67,
        0x65,0x20,0x68,0x61,0x73,0x20,0x6C,0x65,0x6E,
        0x67,0x74,0x68,0x20,0x3D,0x20,0x35,0x30,0x20,
        0x62,0x79,0x74,0x65,0x73};
```

Результат хеширования данного текста должен быть таким:

```
        unsigned long Hash02[8]={
        0x57BA1A47,0x0D770AA6,0x0613763A,0xEAFBC135,
        0xE54DF14E,0xAEB4781F,0x3B89DD57,0x0852F562};
```

Таблица 5-5. **Узлы** замены для хеширования

0xe	0xb	0x1	0xc	0x6	0xd	0xf	0xa	0x2	0x3	0x8	0x1	0x0	0x7
0x5	0x9												
0x4	0xa	0x9	0x2	0xd	0x8	0x0	0xe	0x6	0xb	0x1	0xc	0x7	0xf
0x5	0x3												
0x7	0xd	0xa	0x1	0x0	0x8	0x9	0xf	0xe	0x4	0x6	0xc	0xb	0x2
0x5	0x3												
0x5	0x8	0x1	0xd	0xa	0x3	0x4	0x2	0xe	0xf	0xc	0x7	0x6	0x0
0x9	0xb												
0x4	0xb	0xa	0x0	0x7	0x2	0x1	0xd	0x3	0x6	0x8	0x5	0x9	0xc
0xf	0xe												
0xG	0xc	0x7	0x1	0x5	0xf	0xd	0x8	0x4	0xa	0x9	0xe	0x0	0x3
0xb	0x2												
0x1	0xf	0xd	0x0	0x5	0x7	0xa	0x4	0x9	0x2	0x3	0xc	0x6	0xb
0x8	0xc												
0xd	0xb	0x4	0x1	0x3	0xf	0x5	0x9	0x0	0xa	0xe	0x7	0x6	0x8
0x2	0xc												

Алгоритмы генерации случайных последовательностей и оценка их качества

При синтезе и реализации криптографических модулей весьма важную роль играет выработка и распределение криптографически значимой информации (ключей, синхропосылок и т.п.). Так, в рассмотренных ранее алгоритмах формирования и проверки ЭЦП важную и криптографически значимую роль играет выработка случайного числа, используемого для «индивидуализации» процесса подписи каждого сообщения.

Для решения задач создания (выработки) ключевой информации применяются различного рода датчики случайных чисел, которые делятся на два больших класса — аппаратные и программные.

В аппаратных датчиках источником случайного процесса является шум в электронных приборах. Очевидно, что применение аппаратных датчиков требует наличия специального оборудования.

При создании криптопровайдера весьма актуальна задача построения программного датчика случайных чисел.

Программный датчик случайных чисел (ПДСЧ) предназначен для выработки ключевой и другой криптографически значимой информации, используемой в процессе шифрования, контроля целостности или простановки ЭЦП.

Программно-математически ПДСЧ есть некая процедура, характеризуемая производимой ей выходной последовательностью. Основные требования к вырабатываемой датчиком последовательности $Y_1, Y_2, \ldots, Y_n$ заключаются в равновероятности знаков Y_i и статистической независимости между знаками.

Однако специалист-разработчик должен четко понимать, что для последовательностей, вырабатываемых ПДСЧ, эти требования невыполнимы, например, в силу эффективности вычисления значений Y_i. Поэтому требование статистической независимости следует заменить на иное, имеющее алгоритмический смысл.

В данной главе мы используем понятие алгоритмической сложности восстановления элемента Y_i по Y_j при $i \neq j$ и наоборот. Смысл данного подхода заключается в том, что если бы существовала аналитическая или какая либо иная зависимость между выходами, то, используя её, можно было бы построить эффективно вычислимый критерий статистической зависимости.

Если же трудоёмкость вычисления такого критерия высока, то можно говорить о сложностной (в смысле сложности алгоритма), а следовательно, статистической независимости.

Программный датчик, как всяким конечный автомат, определяется состоянием S(t), где t — цикл работа датчика, функцией перехода Ф(S(t))=S(t + 1) и функцией выхода YY(S(t))=Y(t). Для ПДСЧ возможны следующие проблемы:

- сложность восстановления Y(t) по S(t - 1) — атака из прошлого;
- сложность восстановления S(t) по Y(t) — атака из настоящего;
- сложность восстановления S(t) либо Y(t) по S(t + 1) — атака из будущего.

Требование 1 для ПДСЧ: невозможно осуществить эффективные атаки из прошлого, настоящего и будущего.

Требование 2 для ПДСЧ: период выходной последовательности Y(t) должен быть гарантирован.

Требование 3: величины S(t) следует хранить в защищенном от просмотра виде.

Статистические свойства выходной последовательности ПДСЧ определяются как выбором функций Ф и YY, так и дополнительными средствами контроля.

Требование 4: необходим контроль статистического качества выходной последовательности Y(t).

Далее перечислены требования программиста к ПДСЧ:

- разумная производительность. Нет смысла разрабатывать ПДСЧ, который не будет использоваться вследствие низкой скорости выработки случайных последовательностей;
- датчик должен быть прост в применении;
- необходимо задействовать существующие «строительные блоки», т.е. элементарные криптографические функции.

Рассмотрим теперь архитектуру и основные компоненты ПДСЧ,

Архитектура программного датчика случайных чисел

Как уже говорилось, ПДСЧ моделируется автоматом и в любой момент времени содержит внутреннее состояние, которое используется для генерации псевдослучайных выходных сигналов. Это состояние содержится в тайне и контролирует боль-

шую часть обработки. По аналогии с шифрами такое состояние получило название *ключ ПДСЧ*(key of the PRNG).

Очевидно, что ПДСЧ должен пройти процедуру начальной инициализации случайными данными (это может быть, например, инициированные пользователем события либо информация о внутреннем состоянии компьютера). Такая инициализация может учитывать сохраненное состояние ПДСЧ. В иностранной литературе процедура начальной инициализации называют сбором энтропии.

Далее, для обновления ключа генератору необходимо получить входные сигналы, которые являются случайными.

Часто в этом качестве используются отчеты времени нажатий клавиш или подробности передвижения мыши. Эти входные сигналы далее будут называться *выборками* (samples). В большинстве систем имеется несколько источников генерации выборок, поэтому их классифицируют по источникам поступления.

Затем необходимо выполнить нелинейное перемешивание текущего состояния ПДСЧ и полученных выборок. Производится усложнение ключа ПДСЧ.

Усложнение (reseeding) — это процесс объединения определённым способом существующего ключа и новой выборки (выборок) в новый ключ.

При выключении и последующем включении системы желательно сохранять некоторые данные (например, ключ) в защищенной внешней памяти. Это позволит при следующем запуске стартовать в состоянии, которое противнику трудно угадать. Эти сохранённые данные далее называются *seed-файлами* (the seedfile).

Методы анализа ПДСЧ в реальных приложениях

Переоценка энтропии и угадываемые начальные состояния

Это самая обычная ошибка *в* реальных ПДСЧ. Например, легко взять последовательность выборок, кажущуюся случайной, длиной 128 бит, загрузить её в генератор и начать генерировать выходной сигнал. Если окажется, что эта последовательность имеет лишь 40 случайных бит, то противник сможет выполн ить полный перебор *начального состояния* (starting points).

Неправильное обращение с ключами и seed-файлами

Возможно неправильное обращение с ключами и seed-файлами, например, они могут быть записаны в своп-файл операци-

онной системой или при открытии seed-файла ключ обновляется не при каждом использовании.

Ошибки реализации

Основная мера предосторожности — сделать разумно простым интерфейс, чтобы программист получ ил возможность безопасного использования ПДСЧ в реальном приложении, не вникая в то, как он работает.

Криптоаналитические атаки на механизм генерации ПДСЧ

Между усложнениями механизм генерации выходного сигнала ПДСЧ является потоковым шифром. Как и с другими потоковыми шифрами, весьма вероятно, что шифр, используемый в ПДСЧ, несколько слаб в криптографическом отношении, что делает выходной поток в некоторой степени предсказуемым или по меньшей мере узнаваемым. Поиск слабых мест в этой части ПДСЧ аналогичен поиску слабостей в поточном шифре.

Атаки на файлы промежуточного хранения данных

Это атаки, в которых используется дополнительная информация о внутреннем механизме реализации.

Атака подменой входных сигналов

Противник может построить атаку, навязывая некие «специальные» значения для входа ПДСЧ.

Типовая рекомендация для решения данной проблемы — выборки должны обрабатываются хеш-функцией и смешиваться с существующим ключом ПДСЧ.

Атаки постоянной компрометации

Некоторые ПДСЧ обладают следующим свойством: после компрометации ключа противник всегда может предсказывать выходные сигналы.

Атаки итеративного угадывания

Если выборки смешиваются с ключом по мере поступления, то противник, знающий ключ ПДСЧ, может попытаться угадать следующую «непредсказуемую» выборку, посмотреть на следующий выходной сигнал и проверить свою догадку. Это означает, что ПДСЧ, который смешивает выборки с 32 битами энтропии при генерации каждых нескольких выходных слов, восстановится после компрометации ключа только в том случае, если противник не обнаружит воздействие трёх-четырёх таких выборок на выходные сигналы. Атаки такого рода получили на-

звание *атак итеративного угадывания*. Единственный способ противостоять им — собирать выборки энтропии в пул, отдельный от ключа, и усложнять ключ.

Атаки перебора с возвратами

Некоторые ПДСЧ легко прогонять назад, так же как и вперёд. В качестве примера можно привести RSAREF 2,0 PRNG: противник, скомпрометировавший ключ ПДСЧ после того, как была сгенерирована пара ключей RSA (high-value RSA key pair), может вернуться и узнать эту ключевую пару. В ПДСЧ следует включить механизм ограничения атак такого рода путём ограничения числа выходных байт.

Компрометация ключей, сгенерированных из скомпрометированного ключа

Наихудшее, что может случиться при компрометации ПДСЧ, — компрометация системных ключей, если для их генерации используется данный ПДСЧ. Если генерируемый ключ очень ценен, владелец потерпит значительный урон. Как говорилось ранее, опасность атаки итеративного угадывания заставляет собирать энтропию в пул прежде, чем с её же помощью будет усложнен ключ генератора. При генерации весьма ценно™ ключа рекомендуется иметь некоторое количество дополнительной энтропии. Поэтому пользователь может потребовать явного усложнения ключа генератора. Эта функция не предназначена для частого использования — только для генерации секретов (ключей) высокой ценности.

Типовые компоненты ПДСЧ

Современные проекты ПДСЧ выделяют четыре основных компонента реализации датчика:

1. *накопитель энтропии* (entropy accumulator) — собирает выборки из источников случайности и помещает их в один или несколько пулой;
2. *механизм усложнения* (reseed mechanism) — периодически усложняет ключ генератора, используя энтропию пулов;
3. механизм генерации (generation mechanism) — генерирует выходные сигналы ПДСЧ из ключа;
4. *механизм управления усложнением* (reseed control) — определяет время выполнения усложнения.

Предполагается, что можно накопить достаточно случайных данных, чтобы привести датчик в неугадываемое состояние (без

этого предположения нет смысла проектировать ПДСЧ). Далее будем считать, что есть криптографические механизмы, которые будут генерировать выходные сигналы, неотличимые для противника от случайных (например, генерация отрезков гаммы при помощи алгоритма ГОСТ).

Необходимо отметить два важных момента:

1. выходные сигналы ПДСЧ производятся криптографически. Таким образом, системы, где они используются, *не более безопасны, чем используемый механизм генерации;*
2. ПДСЧ имеет принципиально ограниченную стойкость, выраженную функцией от размера ключа.

Накопление энтропии (entropy accumulation) — это процесс, при котором ПДСЧ получает новое неугадываемое внутреннее состояние. Во время инициализации и для осуществления усложнения во время работы важно успешно накапливать энтропию из выборок. Во избежание атак итеративного угадывания и учитывая регулярность усложнения важно правильно оценивать количество энтропии, набранное до конкретного момента времени. Механизм накопления энтропии должен также сопротивляться атакам *подмены входного сигнала* (chosen-input attack), когда противник, контролирующий несколько выборок, по не все, не сможет заставить ПДСЧ потерять энтропию из неизвестных выборок.

Так, в проекте Yarrow энтропия накапливается в двух пулах (т.н. быстром и медленном).

Быстрый пул обеспечивает частые усложнения ключа генератора. Это гарантирует, что компрометация ключа имеет невысокую продолжительность.

Медленный пул обеспечивает редкие, но существенные нелинейные усложнения. Это должно гарантировать безопасное усложнение даже в том случае, когда оценки исходной энтропии сильно завышены. Входные выборки посылаются в быстрый и медленный пул поочередно.

Ниже перечислены требования для *компонента — накопителя энтропии* (the entropy accumulation component).

1. Необходимо добиться накопления почти всей энтропии из выборок до размера пула.
2. Противник не должен иметь возможность выбирать выборки для уничтожения влияния на пул тех из них, которые он не знает.

3. Противник не должен иметь возможность привести пул в любого рода «слабое» состояние, из которого он не сможет успешно собирать энтропию.
4. Противник, который может выбирать, какие биты в каких выборках ему известны, но всё ещё вынужденный допускать n неизвестных бит, не должен иметь возможность сузить число вероятных состояний пула значительно меньше 2^n.

Заметим, что последнее условие очень важное. Оно требует использования хеш-функции.

Оценка энтропии — это процесс определения количества усилий, необходимых противнику для отгадывания текущего содержимого пулов.

Важный принцип — группировать выборки по источникам и оценивать вклад энтропии каждого источника отдельно. Для этого отдельно оценивается энтропия каждой выборки, а затем складываются оценки всех выборок, пришедших из одного источника.

При «питании» ПДСЧ из одного источника, который, как кажется, даёт высокослучайные данные, возможна ситуация, когда ПДСЧ окажется уязвимым для атаки итеративного угадывания.

Одиночному быстрому источнику позволяется инициировать частое усложнение из быстрого пула, но не из медленного. Это гарантирует то, что усложнение происходит часто, но если оценки энтропии из лучшего источника не точны, в конце концов, произойдёт усложнение из медленного пула на основе оценок энтропии другого источника. Напомним, что выборки из каждого источника чередуются между пулами.

Необходимо быть осторожным при определении вероятностных характеристик источников случайности. Они не должны быть близко связаны или иметь какие-либо существенные корреляции.

Энтропия каждой выборки измеряется тремя способами:

* программист дает оценку выборки при написании программы сбора данных из этого источника. Например, программист может создать выборку с оценкой ее качеств длиной в 20 байт;
* для каждого источника используется особое статистическое устройство оценки энтропии выборки. Эта проверка направлена на выявление ненормальных ситуаций, в которых выборки имеют очень низкую энтропию;

* для получения максимальной оценки энтропии в выборке существует максимальная «плотность» выборки. Это длина выборки в битах, умноженная на некоторую константу, меньшую единицы. В настоящее время используется множитель 0,5.

В качестве энтропии выборки задается наименьшая из этих оценок.

Специфические статистические проверки зависят от природы источника и отличаются в разных реализациях. Более подробно о них рассказано далее.

Выходной сигнал ПДСЧ должен быть таким, чтобы не знающий ключа генератора противник не смог отличить выходной сигнал генератора от случайной последовательности.

Механизм генерации должен быть:

* стойким к криптоаналитическим атакам;
* *производительным* (efficient);
* стойким к атакам *перебором с возвратами* (backtracking) после компрометации ключа;
* способным генерировать очень длинную последовательность сигналов без усложнения.

Механизм усложнения «соединяет» накопитель энтропии с механизмом генерации. Когда механизм контроля усложнения определяет, что усложнение необходимо, механизм усложнения должен обновить ключ, используемый механизмом генерации, данными от одного или двух пулов, поддерживаемых накопителем энтропии, таким образом, чтобы ключ был неизвестен противнику после усложнения, если ключ либо пулы неизвестны ему перед усложнением.

Возможно, что при построении ПДСЧ потребуется сделать процесс усложнения достаточно ресурсоемким, чтобы затруднить атаки на основе угадывания неизвестных входных выборок. Для генерирования нового ключа при усложнении из быстрого пула используются текущий ключ и хеш-значение всех сигналов — входных для быстрого пула с момента последнего усложнения (или момента запуска генератора). После этого все оценки энтропии для быстрого пула обнуляются. Для генерирования нового ключа при усложнение из медленного пула используются текущий ключ, хеш всех сигналов — входных для быстрого пула с момента последнего усложнения (или момента запуска генератора) и хеш аналогичных сигналов для мед-

ленного пула. После этого псе оценки энтропии для обоих пулов обнуляются.

Модуль управления усложнением должен учитывать определенные факторы. Частое ус.ложнение желательно, но оно делает более вероятной атаку итеративного угадывания. Нечастое — даёт противнику, скомпрометировавшему ключ, больше информации. В модуле управления усложнением соблюдается компромисс этих условий. По мере поступления выборок в каждый пул сохраняются оценки энтропии для каждого источника, Когда такая оценка для любого источника в быстром пуле преодолевает пороговое значение, инициируется усложнение из быстрого пула (во многих системах это должно происходить по много раз в час). Когда оценки для любых k из п источников в медленном пуле превышают пороговые значения (более высокие), инициируется усложнение из медленного пула. Этот процесс гораздо медленнее,

Общая конструкция ПДСЧ

Дадим общее описание ПДС Ч с использованием произвольных блочного шифра и хеш-функции. Если оба данных алгоритма надежны и генератор получает достаточную исходную энтропию, то мы получаем стойкий (по указанным выше критериям) ПДСЧ.

Итак, необходимы два алгоритма со следующими свойствами:

1. односторонняя хеш-функция h(x) с хеш-значением длиной m бит;
2. блочный шифр E(x, K) с размером ключа k бит и размером блока открытого текста п бит.

Предполагается, что хеш-функция обладает следующими свойствами:

* невозможностью обнаружения коллизий;
* односторонностью;
* при данном множестве M возможных входных значений выходные значения распределены как |M *наборов* (selections) равномерного распределения no m бит.

Последнее требование подразумевает следующее: даже если противник знает большую часть входного блока хеш-функцию, у пего всё же нет сведений о выходном сигнале, если только он не может перечислить множество возможных входных сигналов.

Блочный шифр должен отвечать следующим свойствам:

* он должен быть стойким к атакам при знании пар «открытый — шифованный текст», даже к тем, которые требуют большого числа открытых сообщений и соответствующих им зашифрованных сообщений;
* он должен иметь хорошие статистические свойства выходных сигналов, даже при высоко шаблонных входных сигналах.

Стойкость в битах получаемого генератора ограничивается $min(m, k)$. На практике этот предел немного не достигается. Причина в том, что если взять m-битное случайное значение и применить хеш-функцию, дающую m бит выходного сигнала, то результат имеет энтропии меньше, чем m бит, из-за случающихся коллизий. Влияние этого фактора невелико, самое большое теряется несколько бит энтропии. Этот небольшой постоянный фактор в дальнейшем игнорируется и считается, что ПДСЧ имеет стойкость $min(m, k)$ бит.

Генератор основан на использовании блочного шифра в режиме счётчика.

Имеется n-битное значение C счётчика. Чтобы сгенерировать n-битный выходной блок, это значение увеличивается на один и зашифровывается блочным шифром:

$C \leftarrow (C+1) \bmod 2^n$

$R \leftarrow E_k(C)$, где R — следующий выходной блок, K — текущий ключ ПДСЧ.

Если в некоторый момент времени ключ компрометируется, не должно раскрываться слишком много сгенерированных до момента компрометации выходных сигналов. Видно, что данный механизм генерации никак не сопротивляется атакам такого рода. По этой причине необходимо вести постоянный подсчёт количества выходных блоков. При достижении некоторого предела *системного параметра безопасности* (system security parameter) $P_б$ — он отвечает требованиям $1 < P_б < 2^{n/3}$ — генерируется k бит выходного сигнала генератора. Они используются как новый ключ:

K ← следующие k бит выходного сигнала ПДСЧ.

Эта операция называется *затвором генератора* (generategate). Заметим, что это не операция усложнения, так как в ключ не вводится новая энтропия. С целью сохранения наибольшей безопасности конструкции максимальное число выходных сигналов генератора между усложнениями ограничивается числом

$\min(2^n, 2^{k/3}P_n)$ n-битных выходных блоков. Первый аргумент функции и предотвращает С от зацикливания, второй практически исключает вероятность того, что К примет одно значение дважды. На практике P_n следует устанавливать гораздо ниже этого числа (например, $P_n = 10$), чтобы минимизировать число выходных сигналов, которые можно узнать при помощи атаки перебора с возвратами.

Алгоритмы начальной инициализации ПДСЧ

Зададим функции KeyboardHook и MouseHook как функции *обратного вызова* (callback routines), которые вызываются каждый раз, когда происходит событие, связанное с мышью или клавиатурой. Они запускаются в пространстве процесса, принимающего событие. Эти функции проверяют причину события и сохраняют данные о нем. Доступ к коду сбора/хранения данных контролируется мьютексом. Процесс, запускающий процедуры слежения, устанавливает также флаг, сигнализирующий об этом, так что он не блокирует сам себя (ошибка состояния, возникающая при блокировании двух потоков выполнения, когда каждый поток ждет освобождения ресурса, используемого другим потоком), если следующий сигнал поступает при обработке предыдущего.

Рассмотрим, какие данные записываются этими функциями (все действия с мьютексами при этом опустим).

Самый простой случай при слежении за передвижениями мыши (функция MouseHook()):

```
...
switch(wParam) /* Message type */
{
...
case WM_MOUSEMOVE: /* Record the mouse location */
case WM_NCMOUSEMOVE:
   .../* Start mutex */
((MOUSEMOVETYPE*)mmGetPtr(mousemove))[mousemoveindex++] =
((MOUSEHOOKSTRUCT*)lParam)->pt;
   /* Dump data if necessary */
if(mousemoveindex >= MOUSEMOVESIZE){
WriteMouseMove(MOUSEMOVESIZE);
mousemoveindex = 0;}
   .../* End mutex */
break;
...
}
...
```

Здесь:

* mousemove — псевдоуказатель, используемый при защищённой работе с памятью посредством библиотеки smf.dll (инициализируется в SetHooks);
* mmGetPtr(mousemove) — функция, возвращающая реальный указатель, соответствующий псевдоуказателю;
* mousemoveindex — номер текущего элемента массива, адресуемого ((MOUSEMOVETYPE*) mmGetPtr(mousemove));
* MOUSEMOVESIZE — размер массива, выделенного иод хранение данных.

Теперь случай слежения за нажатиями кнопок мыши (функция MouseHook()):

```
...
static DWORD then = 0;
static DWORD now = 0;
WORD diff;
LARGE_INTEGER time;
...
switch(wParam) /* Message type */
{
case WM_LBUTTONDOWN:       /*Get timing from button
                           clicks...*/
case WM_RBUTTONDOWN:
case WM_MBUTTONDOWN:
case WM_NCLBUTTONDOWN:
case WM_NCRBUTTONDOWN:
case WM_NCMBUTTONDOWN:
   ,../* Start mutex */
   /*Check time*/
   if(PerfCount==TRUE){
     QueryPerformanceCounter(&time);
     /*Less than the last 32 bits will be significant anyway */
     now = time.LowPart;
   }else{now = GetTickCount(); }
   /*Set timers and collect data*/
   diff = (now - then) & timemask;
   then = now;
   ((MOUSETIMETYPE*)mmGetPtr(mousetime))[mousetimeindex++]
      = diff;
   /* Dump data if necessary */
   if(mousetimeindex >= MOUSETIMESIZE){
      /* Return value is irrelevant */
      WriteMouseTime(MOUSETIMESIZE);
      mousetimeindex = 0;
```

```
    }
    .../* End mutex */
break;
...
}
```

Рассматривая код, можно сделать вывод, что записывается разница (переменная diff) между двумя соседними фиксируемыми событиями.

Способ обработки событий от клавиатуры выглядит аналогично:

```
/* Check time */
if(PerfCount==TRUE){
QueryPerformanceCounter(&time);
     /*Less than the last 32 bits will be significant anyway */
now = time.LowPart;
}else{ now = GetTickCount(); }
    /*Set timers and collect data*/
diff = (now - then) & timemask;
then = now;
((KEYTIMETYPE*)mmGetPtr(keytime))[keytimeindex++] = diff;
    /*Dump data if necessary */
if(keytimeindex >= KEYTIMESIZE){
    /* Return value is irrelevant */
WriteKeyTime(KEYTIMESIZE);
keytimeindex = 0;
}
```

Значения переменных Perf Count и timemask устанавливаются функцией setupCounter(), которая вызывается только функцией SetHooks() после установки подключаемых процедур, т.е. в начале работы. Листинг функции setupCounter() и объявления указанных переменных приведены ниже.

```
/*Timer stuff*/
static BOOL PerfCount = FALSE;
static DWORD timemask = 0;

void setupCounter(void)
/* This function will set PerfCount if there is a
performance counter and will set timemask such that only the
mostly significantly random bits of the timing intervals are
kept */
{
    LARGE_INTEGER li;

    PerfCount = QueryPerformanceFrequency(&li);
    if(PerfCount == TRUE) /* If there is a performance
```

```
                        counter, check frequency */
    {
        if(li.HighPart != 0) /*Extremely high frequency
                              counter*/
        { timemask = 0x0000FFFF; }
        else if(li.LowPart >= 1000000) /*Usual value appears
                    to be 1193180*/{ timemask = 0x0000FFFF; }
        else if(li.LowPart >= 70000)
        {  timemask = 0x00000FFF; }
        else if(li.LowPart >= 1000)
        {  timemask = 0x000000FF; }
        else
        {   PerfCount = FALSE; /* You are better off using the
                              device timer at this point... */
        }
    }
    if(PerfCount==FALSE)/* If not, we have to use
                              GetTickCount() */
    { timemask = 0x0000000F; /* Resolution of about 10ms */ }
}
```

Отмстим, что в процедурах слежения используются два варианта получения отсчетов времени:

1. при помощи функции GetTickCount(), причём этот способ используют только в том случае, если не работает второй;
2. при помощи функции QueryPerformanceCounter(), возвращающей текущее значение *высокоточного счётчика производительности* (the high-resolution performance counter), наличие которого и частоты обновления (от нее зависит маска timemask, определяющей количество учитываемых байт в одной посылке), проверяются с помощью функции QueryPerformanceFrequency() при инициализации в функции setupCounter(). (Подробнее об этом счётчике — в MSDN).

Медленный опрос — процесс получения «расплывчато» случайных данных из системных данных ОС. Вызывается при инициализации ПДСЧ для получения начального состояния. Термин «расплывчато» применен потому, что в настоящее время нет хорошей оценки количества энтропии, которую они содержат.

Полный текст данной функции показан ниже.

```
int m_prngSlowPoll(UINT pollsize)
{
BYTE *buf;
DWORD len;
```

```
prng_error_status  retval;

CHECKSTATE(p);

buf = (BYTE*)malloc(pollsize);
if(buf==NULL) {return PRNG_ERR_LOW_MEMORY;}
len = prng_slow_poll(buf,pollsize); /* OS specific call */
retval = prngInputEntropy(buf,len,SLOWPOLLSOURCE);
trashMemory(buf,pollsize);
free(buf);
return retval;
}
```

Наибольший интерес представляет девятая строка. В ней осуществляется вызов функции prng_slow_poll(), код которой уникален для конкретной ОС. Рассмотрим версию prng_slow_poll() для Windows NT.

```
DWORD prng_slow_poll(BYTE *buf,UINT bufsize)
/* Slow pool returns a ton of data from the performance data
registry key */
{
DWORD len = bufsize;

RegQueryValueEx(HKEY_PERFORMANCE_DATA,"Global",
   NULL,NULL,buf,&len);
RegCloseKey(HKEY_PERFORMANCE_DATA);

return _MIN(bufsize,len);
}
```

Как видно, данные берутся из ключа реестра HKEY_PERFORMANCE_DATA, содержащего технические данные, или *данные производительности* (performance data).

Более подробную информацию можно найти в MSDN.

Совместно с версией prng_slow_poll() для NT также используется функция prng_set_NT_security(void), взятая авторами из стандартных примеров, которые поставляются с MicrosoftVisual C++ 5.0(6.0), с небольшой доработкой. Подробнее — показанном ранее листинге и MSDN.

Теперь же рассмотрим версию prng_slow_poll() для Windows 9x.

```
/* TODO: Code needs to be refined to collect only the more
useful (<- very relative term) data */
DWORD prng_slow_poll(BYTE* buf,UINT bufsize)
/* Copy all the possible data from a ToolHelp32 snapshot and
copy it to a buffer */
/* Will copy a maximum of SPLEN bytes into buf. Returns the
```

```
number of bytes copied */
/* Portions of this code are copied from an example in the
Microsoft Systems Journal
and are thus: Copyright <1995>, Microsoft Systems Journal */
{  BYTE* pos;BYTE* end;
HANDLE hSnapshot,hSubSnapshot;
PROCESSENTRY32 pe32;
THREADENTRY32 te32;
MODULEENTRY32 me32;
HEAPLIST32 hl32;
BOOL fOK;
pos = buf;
end = buf+bufsize;
hSnapshot = NULL;
hSubSnapshot = NULL;
/* Get process data */
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
if (hSnapshot == NULL) {goto cleanup_slow_poll;}
pe32.dwSize = sizeof(PROCESSENTRY32);
fOK = Process32First(hSnapshot,&pe32);
while(fOK==TRUE){
 if(pos+pe32.dwSize-sizeof(DWORD) > end) {goto
   cleanup_slow_poll;}
 memcpy(pos, (BYTE*)&pe32 + sizeof(DWORD),
   pe32.dwSize-sizeof(DWORD));
 pos += pe32.dwSize-sizeof(DWORD);
 /* Get Heap info */
 hSubSnapshot=CreateToolhelp32Snapshot( TH32CS_SNAPHEAPLIST,
   pe32.th32ProcessID );
 if (hSubSnapshot == NULL) {goto cleanup_slow_poll;}
 hl32.dwSize = sizeof(HEAPLIST32);
 fOK = Heap32ListFirst(hSubSnapshot,&hl32);
 while (fOK==TRUE){
 if(pos+hl32.dwSize-sizeof(DWORD) > end) {goto
   cleanup_slow_poll;}
 memcpy(pos,(BYTE*)&hl32 + sizeof(DWORD),
   hl32.dwSize-sizeof(DWORD));
 pos += hl32.dwSize-sizeof(DWORD);
 hl32.dwSize = sizeof(HEAPLIST32);
 fOK = Heap32ListNext(hSubSnapshot,&hl32);
 }
 CloseHandle(hSubSnapshot);
 hSubSnapshot = NULL;
 /* End Get Heap info */
 /* Get Module Info */ /* This data is highly repetative */
 hSubSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPMODULE,
```

```
    pe32.th32ProcessID);
  if (hSubSnapshot == NULL) {goto cleanup_slow_poll;}
  me32.dwSize = sizeof(MODULEENTRY32);
  fOK = Module32First(hSubSnapshot,&me32);
  while(fOK==TRUE){
  if(pos+me32.dwSize-sizeof(DWORD) > end) {goto
    cleanup_slow_poll;}
  memcpy(pos,(BYTE*)&me32 + sizeof(DWORD),
  me32.dwSize-sizeof(DWORD));
  pos += me32.dwSize-sizeof(DWORD);
  me32.dwSize = sizeof(MODULEENTRY32);
  fOK = Module32Next(hSubSnapshot,&me32);
  }
  CloseHandle(hSubSnapshot);
  hSubSnapshot = NULL;
  /* End Get Module Info */
  pe32.dwSize = sizeof(PROCESSENTRY32);
  fOK = Process32Next(hSnapshot,&pe32);
}
CloseHandle(hSnapshot);
hSnapshot = NULL;
/* End Get Process Info */
/* Get thread data */ /* May want to collect this data first
*/
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD,0);
if (hSnapshot == NULL) {goto cleanup_slow_poll;}
te32.dwSize = sizeof(THREADENTRY32);
fOK = Thread32First(hSnapshot,&te32);
while(fOK==TRUE){
  if(pos+te32.dwSize-sizeof(DWORD) > end) {goto
    cleanup_slow_poll;}
  memcpy(pos,(BYTE*)&te32 + sizeof(DWORD),
    te32.dwSize-sizeof(DWORD));
  pos += te32.dwSize-sizeof(DWORD);
  te32.dwSize = sizeof(THREADENTRY32);
  fOK = Thread32Next(hSnapshot,&te32);
};
CloseHandle(hSnapshot);
hSnapshot = NULL;
/* End Get Thread Info */
cleanup_slow_poll:
if(hSnapshot!=NULL) {CloseHandle(hSnapshot);}
if(hSubSnapshot!=NULL) {CloseHandle(hSubSnapshot);}
return  pos-buf;
}
```

Рассмотрим приведенный код, опуская очевидные вещи.

Строка 13:

```
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
```

Функция CreateToolhelp32Snapshot() делает *"моментальный снимок"* (snapshot) состояния объектов в конкретный момент времени - процессов, куч, модулей и нитей, принадлежащих процессам (включаемые компоненты определяются значением первого параметра). Значение параметра TH32CS_SNAPPROCESS указывает, что снимок будет включать *список процессов* (the process **list).** Функция возвращает указатель на созданный снимок.

В строках 15—50 полученный снимок разбирается. Здесь отметим строку 19;

```
memcpy(pos, (BYTE*)&pe32 + sizeof(DWORD), pe32.dwSize-
sizeof(DWORD));
```

где pos — это указатель на место в выходном буфере, на котором была прекращена предыдущая запись (изначально равен переданному указателю на выходной буфер), pe32 — переменная-экземпляр структуры PROCESSENTRY32. Получается, что в выходной буфер просто копируется структура типа PROCESSENTRY32 без первого постоянного члена, равного размеру структуры. Определение данной структуры показано в таблице 5-6.

Таблица 5-6. Структура PROCESSENTRY32

Имя поля	Тип	Описание
DwSize	DWORD	Размер структуры в байтах.
CntUsage	DWORD	Число ссылок (обращений) на процесс. Процесс существует, только если данное число не равно 0. Если оно равно 0, то процесс уничтожается.
Th32ProcessID	DWORD	Идентификатор процесса.
th32DefaultHeapID	ULONG_PTR	Идентификатор кучи по умолчанию для процесса.
th32ModuleID	DWORD	Идентификатор модуля процесса.
cntThreads	DWORD	Число выполняемых нитей, запущенных процессом.
th32ParentProcessID	DWORD	Идентификатор процесса, создавшего данный процесс.
pcPriClassBase	LONG	Приоритет по умолчанию для нитей, созданных процессом.
dwFlags	DWORD	Зарезервировано. Не используется.
szExeFile	TCHAR[]	Имя файла выполняемого процесса и путь к нему.

```
hSubSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST,
pe32.th32ProcessID);
```

Функция CreateToolhelp32Snapshot() уже вам знакома, но вызывается она с параметром TH32CS_SNAPHEAPLIST. Результат: функция возвращает указатель на снимок всех куч указанного процесса (напомним, что эта операция выполняется внутри цикла разбора снимка всех процессов). Далее в строках 23—31 в цикле просматриваются все полученные кучи. Обратите внимание, что строка 27 аналогична строке 19:

```
memcpy(pos, (BYTE*)&hl32 + sizeof(DWORD), hl32.dwSize-
sizeof(DWORD) );
```

Изменилась только переменная, а с ней и используемая структура (таблица 5-7).

Таблица 5-7. Структура HEAPLIST32

Имя поля	Тип	Описание
dwSize	SIZE_T	Размер структуры в байтах.
th32ProcessID	DWORD	Идентификатор процесса.
th32HeapID	ULONG_PTR	Идентификатор кучи в контексте процесса-владельца.
dwFlags	DWORD	Имеет значение HF32_DEFAULT - куча по умолчанию для процесса

Далее в строках 34 — 45 аналогично обрабатываются все модули процесса. Используемая структура MODULEENTRY32 описана в таблице 5-8.

Таблица 5-8. Структура **MODULEENTRY32**

Имя поля	Тип	Описание
dwSize	DWORD	Размер структуры в байтах,
th32ModuleID	DWORD	Идентификатор модуля в контексте процесса-владельца.
th32ProcessID	DWORD	Идентификатор процесса.
GlblcntUsage	**DWORD**	Счётчик глобального использования.
ProccntUsage	DWORD	Счётчик использования модуля в контексте процесса-владельца.
modBaseAdrr	BYTE*	Базовый адрес модуля в контексте процесса-владельца.
modBaseSize	DWORD	Размер модуля в байтах.
hModule	HMODULE	Указатель на модуль в контексте процесса-владельца.
pcPriClassBase	LONG	Приоритет по умолчанию для нитей, созданных процессом.
szModule	TCHAR[]	Строка, содержащая имя модуля.
szExePath	TCHAR[]	Строка, содержащая путь к модулю.

После цикла по процессам следует цикл по *нитям* (thread), внутри которого в выходной буфер записываются структуры типа THREADENTRY32 (таблица 5-9), описывающие каждую отдельную нить.

Таблица **5-9. Структура THREADENTRY32**

Имя поля	Тип	Описание
dwSize	DWORD	Размер структуры в байтах.
cntUsage	DWORD	Число ссылок (обращений) к нити. Нить существует, только если данное число не равно 0. Если оно равно 0, то нить уничтожается.
th32Thread*ID*	DWORD	Идентификатор нити.
th32OwnerProcessID	DWORD	Идентификатор процесса-владельца.
tpBasePri	LONG	Начальный уровень приоритета, доступный нити.
tpDeltaPri	LONG	Знаковое значение, определяющее изменение уровня приоритета, доступное нити.
dwFlags	DWORD	Зарезервировано. Не используется.

Отметим, что функция прекращает работу, как только в результате записи очередной структуры возможен выход за границу указанного массива, т.е. число записанных байт может превысить значение параметра bufsize. Данная проверка осуществляется перед каждой попыткой записи в выходной буфер.

Здесь показан лишь поверхностный обзор версий функции prng_slow_poll() для Windows NT и Windows 9x. В обоих случаях данные, записываемые этими функциями в выходные буферы и впоследствии используемые как содержащие некое количество случайных данных, требуют дополнительного исследования (в смысле установки их случайности, выделения высокоэнтропийных данных и т.д.). В версии для Windows NT также следует подробно изучить функцию prng_set_NT_security().

Оценка качества случайных последовательностей

Итак, теперь вы знаете, что в прикладной криптографии *случайное число* — это то, которое не может быть предсказано противником прежде, чем будет сгенерировано. Если это число находится в диапазоне от 0 до $2^n - 1$, то наблюдатель может предсказать это число с вероятностью не более $1/2^n$. Если последо-

вательно генерируются m случайных чисел, наблюдатель по m - 1-му из них не должен предсказать m-ое с вероятностью более 1/2".

Данный раздел посвящена разработке тестов на случайность генераторов случайных и псевдослучайных последовательностей. Набор используемых алгоритмов тестов создается на основе документа Национального Института Стандартов и Технологий (NIST) «A STATISTICAL TEST SUITE FOR RANDOM AND PSEUDORANDOM NUMBER GENERATORS FOR CRYPTOGRAPHIC APPLICATIONS» NIST Special Publication 800-22 (with revision dated December 2000) Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh. Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, San Vo.

Случайность — это вероятностное свойство, поэтому свойства случайных последовательностей можно охарактеризовать и описать в вероятностных терминах. Вероятный исход при применении статистических тестов к истинно случайной последовательности известен заранее (априори), и его также удается описать в вероятностных терминах. Существует бесконечное число возможных статистических тестов, каждый из которых оценивает присутствие или отсутствие какой-либо характеристики, которая, будучи обнаруженной, свидетельствует о неслучайности исследуемой последовательности.

Статистический тест формулируется как тест с определённой *нуль-гипотезой* (H_0). В качестве нуль-гипотезы используем утверждение о том, что тестируемая последовательность случайна.

С нуль-гипотезой связана альтернативная гипотеза (H_a), в качестве которой используем утверждение, что тестируемая последовательность неслучайна. Для каждого применяемого теста решение (или вывод) означает принятие или отвержение нуль-гипотезы. Утверждение, что генератор производит случайные (или неслучайные) числа, базируется на результатах тестирования произведённой им последовательности.

Для каждого теста следует выбрать подходящую статистику, используемую для определения верности или ложности нуль-гипотезы. Каждая статистика должна иметь распределение возможных значений. Теоретически эталонное распределение статистики под нуль-гипотезу определяется математическими методами. В эталонном распределении определяется *критичес-*

кое значение (обычно, это «далёкое» значение *в* хвостах кривой распределения, где-то в районе 99%). Во время теста по данным (тестируемая последовательность) рассчитывается статистическое тестовое значение. Это значение сравнивается с критическим. Если статистическое тестовое значение преиосходит критическое, нуль-гипотеза случайности признаётся ложной. Иначе она признаётся истинной.

На практике тестирование статистической гипотезы возможно потому, что эталонное распределение и критическое значение рассчитываются в *предположении случайности*. Если фактически предположение случайности истинно для имеющихся данных, то результирующее статистическое тестовое значение, рассчитываемое по этим данным, с очень малой вероятностью (например, 0,01%) превысит критическое значение.

С другой стороны, если рассчитываемое тестовое статистическое значение превышает критическое *значение* (т.е. имеет место маловероятное событие), тогда с точки зрения тестирования статистических гипотез произошло неестественное (выделяемое) событие. Если при расчёте тестового статистического значения оно оказывается большим критического, *делается* вывод о том, что предположение о случайности ложно. В этом случае результат тестирования *статистической* гипотезы таков: H_0 (гипотеза о случайности) — ложна, H_a (гипотеза о неслучайности) — истинна.

Проверка статистической гипотезы — это процедура, имеющая два возможных возвращаемых результата: либо согласие с H_0 (данные случайны), либо согласие с H_a (данные неслучайны). В таблице 5-10 устанавливается соотношение между настоящим состоянием данных (неизвестным для тестера) и выводами, полученными при тестировании.

Таблица 5-10. Соотношение результатов тестирования и возможных выводов

	Выводы	
Истинное положение	Согласие с H_0	Согласие с H_a
Данныеслучайны (H_0 истинна)	Нет ошибки.	Ошибка типа 1.
Данные неслучайны (H_a истинна)	Ошибка типа 2.	Нет ошибки.

Ситуация, когда данные на самом деле случайны, но делается вывод о ложности нуль-гипотезы (т.е. данные признаются неслучайными), маловероятна и называется *ошибкой первого рода*

(ошибка типа 1). Ситуация, когда данные на самом деле неслучайны, но делается вывод о истинности нуль-гипотезы (т.е. данные признаются неслучайными), маловероятна и называется *ошибкой второго рода (ошибка типа 2)*.

Вероятность ошибки первого рода называется *уровнем достоверности* теста, Эту вероятность можно вычислить до теста, она обозначается а. Для теста а — это вероятность того, что тест покажет, что последовательность неслучайна, если на самом деле она случайна. То есть последовательность обладает какой-то неслучайной характеристикой, хотя и произведена «хорошим» генератором. Обычно а устанавливают равной 0,01.

Вероятность ошибки второго рода обозначают р. Для тестов р — это вероятность того, что тест покажет, что последовательность случайна, хотя на самом деле она неслучайна. То есть произведённая «плохим» генератором последовательность обладает какой-то случайной характеристикой. В отличие от а, (3 — не фиксированная константа, она может принимать множество различных значений, так как существует бесконечное число вариантов неслучайных потоков данных, и каждый вариант имеет свою неличину р. Поэтому и расчет р гораздо сложнее а.

Одна из главных задач при разработке тестов — минимизация вероятности р, т.е. минимизация вероятности определения как случайной последовательности, произведённой плохим генератором. Вероятности α и β зависят друг от друга и от длины тестируемой последовательности п: при двух известных значениях третье определяется автоматически. На практике сначала выбирают длину п и значение а. Затем устанавливают критическое значение для взятой статистики так, чтобы получить минимальную р. То есть подходящая длина последовательности выбирается в соответствии с приемлемостью вероятности принятия решения о том, что плохой генератор производит случайную последовательность. Точка среза выбирается так, чтобы минимизировать вероятность ошибочного признания последовательности случайной.

Каждый тест базируется на расчёте статистического значения при помощи некоторой функции от данных. Если тестовое статистическое значение обозначить S, а критическое значение t, то вероятность ошибки первого рода

$P(S > t \mid H_0$ истинна$) = P($признание H_0 ложной $\mid H_0$ истинна$)$

и вероятность ошибки второго рода

$P(S < t \mid H_0$ ложна$) = P($признание H_0 истинной $\mid H_0$ ложна$)$.

Статистика теста используется для расчёта P-value, которое суммирует все сильные доводы относительно нуль-гипотезы. Для теста P-value — это вероятность того, что совершенный генератор случайной последовательности произвел бы последовательность менее случайную, чем исследуемая, по типу неслучайности, рассматриваемому в тесте. Если в результате P-value равно 1, то последовательность считается близкой к случайной. P-value, равное 0, сигнализирует о том, что последовательность абсолютно неслучайная (детерминированная). Для теста можно выбрать уровень достоверности a. Если P-value > a. то нуль-гипотеза истинна, т.е. последовательность случайна, Если P-value < a, то нуль-гипотеза ложна, т.е. последовательность неслучайна. Параметр a определяет вероятность ошибки первого рода. Обычно a выбирается из диапазона [0.001, 0,01].

Значение a = 0,001 показывает, что из 1000 случайных последовательностей ожидается одна «бракованная». Для P-value≥ 0,001 рассматриваемая последовательность является случайной с вероятностью 99,9%. Для P-value < 0,001 рассматриваемая последовательность является неслучайной с вероятностью 99,9%.

Значение a = 0,01 показывает, что из 100 случайных последовательностей ожидается одна «бракованная». Для P-value > 0,01 рассматриваемая последовательность является случайной с вероятностью 99%. Для P-value < 0,01 рассматриваемая последовательность является неслучайной с вероятностью 99%.

Пусть во всех тестах a = 0.01,

В отношении случайной двоичной последовательности делаются следующие *предположения:*

* *монотонность* (uniformity). В любой момент генерации последовательности случайных или псевдослучайных бит, вероятности появления нуля или единицы должны быть равны, т.е. должны равняться 1/2. Для последовательности длиной п ожидаемое число нулей (или единиц) равно п/2;
* *масштабируемость* (scalability). Любой тест, применимый к последовательности, можно также применить к подстроке, выбранной случайным образом. Если последовательность случайна, то и любая её подстрока также должна быть случайной. Следовательно, любая подстрока случайной последовательности должна проходить любой тест на случайность;
* *состоятельность* (consistency). Поведение генератора должно быть одинаковым в любой момент времени относительно любых стартовых значений.

Статистика теста. Эталонное распределение статистики теста

В некоторых тестах в качестве эталонного распределения используется нормированное нормальное распределение или распределение хи-квадрат. Если тестируемая последовательность на самом деле неслучайна, то рассчитываемая статистика теста попадёт в крайние области эталонного распределения. Нормированное нормальное распределение (колоколообразная кривая) применяется для сравнения найденного значения статистики теста для ДСЧ с ожидаемым значение статистики теста в предположении случайности. Статистика теста для нормированного нормального распределения имеет вид

$z = (x - \mu)/\sigma$, где x — значение статистики теста, а μ и σ^2 — ожидаемое значение и дисперсия статистики теста.

χ^2-распределение (левая асимметричная кривая) используется для сравнения степени согласия ожидаемой частоты измеряемых образцов с соответствующими ожидаемыми частотами предполагаемого распределения. Здесь статистика теста имеет вид

$A^2 = \sum \frac{(o_i - e_i)^2}{e_i}$, где o_i и e_i — наблюдаемые и ожидаемые частоты появления эталонов соответственно.

Рассмотрим следующие тесты:

1. монобитный тест на частоту;
2. блочный тест на частоту;
3. тест на серийность;
4. матрично-ранговый тест (тест оценки рангов непересекающихся двоичных матриц 32*32 бита);
5. тест на сжимаемость;
6. тест на вхождение неперекрывающихся шаблонов.

Процесс тестирования состоит из двух этапов:

* сбора статистики по исследуемой последовательности;
* оценкаи полученных результатов.

Последовательность подаётся в виде массива 8-битных байт. Каждый бит значимый.

Качество оценки

Как правило, для получения корректной оценки необходимо выполнение некоторых условий (особых для каждого теста),

например, достаточное количество обработанных бит. Подробнее об этом рассказано в описании соответствующего теста.

Методы оценки качества последовательности в разных тестах построены по разным алгоритмам. Но в результате любого теста рассчитывается число, введенное ранее как P-value, причём последовательность признаётся случайной, если P-value > 0,01. P-value можно использовать для косвенной оценки качества исследуемой последовательности.

Как говорилось выше, по ходу каждого теста рассчитывается так называемая статистика теста. Это значение также можно использовать для косвенной оценки качества в смысле случайности исследуемой последовательности (таблица 5-11).

Таблица 5-11. **Обозначение основных** функций

Обозначение	**Описание**
e	Входная исследуемая последовательность.
e_i	i-й бит исследуемой последовательности.
N	Длина входной последовательности.
Erfc	Функция ошибок дополнительная.
Igamc	Неполная гамма-функция.

Монобитный тест на частоту (тест на монотонность)

Объект теста — доля нулей и единиц в исследуемой последовательности. Цель теста — определить, приближается ли число единиц и нулей в последовательности к ожидаемому для случайной равновероятной последовательности. Тест определяет близость содержания единиц к 1/2 от общего количества. Ожидается, что число нулей и единиц должно быть примерно одинаковым. Все последующие тесты применимы только к последовательности, прошедшей данный тест. Если последовательность признана данным тестом неслучайной, то результаты упоминаемых тестов нельзя признать достоверными.

Описание алгоритма теста

Преобразование к ±1: нули и единицы входной последовательности преобразуются в -1 и +1 и складываются с получением $S_n = X_1 + X_2 + ... + X_n$, где $x_i = 2y_i - 1$, где y_i — бит входной последовательности. Например, если на входе Y = 1011010101, тогда n = 10 и $S_n = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 + (-1) + 1 = 2$.

Рассчитывается статистическое значение: $Sabs = \frac{|Sn|}{\sqrt{n}}$

Рассчитывается P-value: $P - value = erfc(\frac{S\_obs}{\sqrt{2}})$,

где erfc — *дополнительная функция ошибок* (complementary error function).

Если P-value меньше, чем 0,01, то делается вывод о неслучайности исследуемой последовательности. Иначе последовательность считается случайной.

Статистика теста

S_{abs} — абсолютное значение суммы преобразованной последовательности, делённое на квадратный корень от длины последовательности.

Эталонное распределение для этого теста — это *половина нормального* (half normal) (при большой длине последовательности). Если последовательность случайна, то S_{abs} стремится к нулю (находится вблизи от нуля). Чем больше перевес нулей или единиц, тем больше между S_{abs} и нулём.

Условия верности оценки

Входная последовательность должна иметь длину 100 и более бит (т.е. 13 и более байт).

Блочный тест на частоту

Объект теста — доля единиц в M-битных блоках исследуемой последовательности. Цель теста — определить, насколько близка частота появления единиц в M-битных блоках к M/2, ожидаемым для случайной последовательности. Для M = 1 тест реализует монобитный тест па частоту.

Описание алгоритма теста

Последовательность делится на N непересекающихся блоков но M бит. Не вошедшие в блоки биты отбрасываются.

Определяется доля единиц в каждом из M-битных блоков:

$$\pi_i = \frac{\sum_{j=1}^{M} \varepsilon_{(i-1)*M+j}}{M}, for\ 1 \le i \le N$$

Рассчитывается статистика: $\chi^2(obs) = 4\,M * \sum_{i=1}^{N} (\pi_i - 1/2)^2$

Рассчитывается P-value = igamc(N/2, χ^2(obs)), где igamc — *неполная гамма-функция* (the incomplete gamma function).

Если P-value меньше, чем 0,01, то делается вывод о неслучайности исследуемой последовательности. Иначе последовательность считается случайной.

Статистика теста

$v_1(obs)$ — оценка того, как хорошо исследуемая доля единиц в M-битном блоке согласуется с ожидаемой долей ($\frac{1}{2}$).

Эталонное **распределение** статистики **теста**

χ^2-распределение.

Условие верности оценки

Рекомендуемая длина исследуемой последовательности не менее 100 бит. Длила блока должна выбираться исходя из следующих условий:

M > 20 и M < 0,01 длины последовательности, т.е. количество блоков должно быть N < 100.

Тест на серийность

Объект теста — общее число серий в последовательности, где под серией подразумевается непрерываемая последовательность идентичных бит. Серия длиной K содержит ровно K одинаковых бит и ограничена битами противоположного значения. Целью данного теста является сравнение общего количества серий различной длины с аналогичным количеством, ожидаемым для случайной последовательности. В частности, тест определяет, происходит ли переход от пулей к единицам и обратно слишком быстро или слишком медленно.

Описание алгоритма теста

Рассчитывается доля единиц в исследуемой последовательности

$$\pi = \frac{\sum_j \varepsilon_j}{n}$$

Определяется предварительное условие — прохождение теста на частоту: если окажется, что $|\pi - \frac{1}{2}| \geq \tau$, то тест на серийность должен быть прерван (т.е. тест нельзя запускать для последовательности, не прошедшей монобитный тест на частоту). Для данного теста t установлено как $\tau = 2 * n^{-0.5}$.

Рассчитывается статистическое значение $V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1$, где r(k) = O,

если $e_k = e_{k+1}$, и r(k) = 1 в противном случае.

Рассчитывается $P\text{-}value = erfc\left(\frac{|v_1(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right)$

Если P-value меньше, чем 0,01, то делается вывод о неслучайности исследуемой последовательности. Иначе последовательность считается случайной.

Статистика теста

$V_n(obs)$ — общее количество серий (т.е. число серий нулей + число серий единиц), отнесённое к длине последовательности. Большое значение $V_n(obs)$ говорит о том, что осцилляции в последовательности происходят слишком быстро; малое значение — о том, что слишком медленно. (Осцилляция происходит при смене значения с нуля на единицу или наоборот.) Быстрая осцилляция наблюдается, когда происходит много смен, например, последовательность 101010101 осциллирует на каждом бите. Поток с медленной осцилляцией содержит меньше серий, чем ожидается для случайной последовательности. Например, последовательность, содержащая подряд 100 единиц, затем 73 нуля и за ними ещё 127 единиц, будет иметь только 3 серии, в то время как ожидается 150.

Эталонное распределение статистики теста

χ^2-распределение.

Условие верности оценки

Входная последовательность должна иметь длину 100 и более бит (т.е. 13 и более байт).

Матрично-ранговый тест

Объект теста — ранг непересекающихся подматриц исследуемой последовательности. Цель теста — проверка исследуемой последовательности на линейную зависимость между подстроками фиксированной длины.

Описание алгоритма теста

Исследуемая последовательность делится на блоки (матрицы) размером 32*32 бита, т.е. по 128 байт каждый. N — полученное количество матриц.

Определяется ранг R_i каждой из матриц, где $i = 1...N$.

Вычисляются: F_m — количество матриц с рангом 32, F_{m-1} — количество матриц с рангом 31, $F_{m-2} = N - F_m - F_{m-1}$.

Рассчитывается χ^2:

$$\chi^2(obs) = \frac{(F_m - 0.2888N)^2}{0.2888N} + \frac{(F_{m-1} - 0.5776N)^2}{0.5776N} + \frac{(F_{m-2} - 0.1336N)^2}{0.1336N}$$

Рассчитывается P-value:

$$P - value = e^{\chi^2(obs)/2}$$

Если P-value меньше, чем 0,01, то делается вывод о неслучайности исследуемой последовательности. Иначе последовательность считается случайной.

Статистика теста

$\chi^2(obs)$ — оценка отношения наблюдаемого числа рангов различных значений к ожидаемому для случайной последовательности. Большое значение $c^2(obs)$ сигнализирует о том, что присутствует значительная девиация распределения значений рангов от ожидаемого распределения для случайной последовательности.

Эталонное распределение статистики теста

χ^2-распределение.

Условие верности оценки

Должно быть протестировано не менее 38 матриц, т.е. минимальная длина исследуемой последовательности:

38912 бит = 4864 байт.

Тест на сжимаемость

В этом тесте рассматривается общее число накапливаемых различных образцов (слон) в последовательности. Цель теста — определить, насколько тестируемую последовательность можно сжать. Последовательность рассматривается как неслучайная, если может быть значительно сжата. Случайная последовательность должна обладать характерным числом различных образцов.

Описание алгоритма теста

Производится разбор последовательности на последовательные отдельные различные слова, образующие «словарь» слов последовательности. Подстрока последовательно формируется из следующих друг за другом бит последовательности до тех пор, пока получаемая подстрока не окажется уникальной для уже обработанного участка последовательности (т.е. в формируемом словаре не найдётся такого слова). Полученная подстрока и есть новое слово в словаре. На этом этапе определяется W_{obs} — объем полученного словаря (количество слов в словаре). Для примера разберём последовательность вида e = 010110010 (таблица 5-12).

Таблица 5-12. Пример анализа подпоследовательностей

Позиция бита	Бит	Новое слово?	Слова
1	0	Да	0 (бит 1)
2	1	Ли	1 (бит 2)
3	0	Нет	
4	1	Да	01 (биты 3-4)
5	1	Нет	
6	0	Да	10 (биты 5-6)
7	0	Нет	
8	1	Нет	
9	0	Да	010 (биты 7-9)

В словаре пять слов: О, 1, 01, 10, 010. Значит, $W_{obs} = 5$.

Рассчитывается $P\text{-}value = \frac{1}{2} erfc\left(\frac{\mu - W_{obs}}{\sqrt{2\sigma^2}}\right)$

где m = 69586.25 и $\sigma = \sqrt{70.448718}$ для $n = 10^6$. Для иных значений n значения т и s должны быть пересчитаны. Отметим, что, так как в настоящее время не известна теория точного определения этих значений, они рассчитываются с помощью SHA-1 (при предположении его случайности). Blum-Blum-Shub-генератор даёт другие значения для μ и ст.

Если P-value меньше, чем 0,01, то делается вывод о неслучайности исследуемой последовательности. Иначе последовательность считается случайной.

Статистика теста

W_{obs} — количество последовательных отдельных различных слов в последовательности.

Эталонное распределение для статистики теста

Нормальное распределение.

Условие верности оценки

Длина последовательности не менее 10^6 бит.

Тест на вхождение неперекрывающихся шаблонов

Объект исследования данного теста — число вхождений заданной заранее битовой строки (шаблона). Задача теста — определить, не производит ли генератор слишком много заданных апериодических шаблонов. В данном тесте для поиска m-битного шаблона используется m-битное «окно». Если шаблон не найден, то окно сдвигается на один бит. Если шаблон найден, окно устанавливается на бит, следующий за найденным вхождением шаблона, и поиск продолжается.

Описание алгоритма теста

Последовательность делится на N непересекающихся блоков по M бит.

Для каждого блока подсчитывается количество вхождений Б (шаблон) W. (j = 1,..., N). В процессе поиска по последовательности движется m-битное окно, которое на каждом шаге побитно сравнивается с шаблоном. Если они не совпадают, окно сдвигается на один бит, например если окно длиной 3 бита (т.е. га = 3) содержало биты с 3 но 5, то на следующем шаге оно будет содержать биты с 4 по 6. Если окна совпадают, окно сдвигается на m бит, например в рассматриваемом случае окно, содержащее биты с 3 по 5, на следующем шаге будет содержать биты с 6 по 8.

Рассмотрим пример. Дана последовательность 10100100101110010110, т.е. n = 20. Пусть M = 10, тогда N = 2. Пусть m = 3 и B = 001. Тогда процесс поиска выглядит следующим образом (таблица 5-13).

Таблица 5-13. Пример теста на шаблоны

Биты последовательности	Блок 1		Блок 2	
	Биты	ω_1	Биты	ω_2
1-3	101	0	111	0
2-4	010	0	110	0
3-5	100	0	100	0
4-6	001	0+1=1	001	0+1=1
5-7	Нет проверки		Нет проверки	
6-8	Нет проверки		Нет проверки	
7-9	001	1+1=2	011	1
8-10	010	2	110	1

То. $W_1 = 2, W_2 = 1$.

Под предположением случайности рассчитываются теоретические значения μ и σ^2:

$$\mu = (M - m + 1)/2^m, \quad \sigma^2 = M\left(\frac{1}{2^m} - \frac{2m-1}{2^{2m}}\right)$$

Рассчитывается $\chi^2(obs) = \sum_{j=1}^{N} \frac{(W_j - \mu)^2}{\sigma^2}$

Рассчитывается, $P\text{-}value = igamc\left(\frac{N}{2}, \frac{\chi^2(obs)}{2}\right)$. Отметим, что для одного шаблона рассчитывается одно значение P-value, но для длины шаблона m существует 2^m шаблонов, для каждого из

которых можно рассчитать P-value, тем самым проведя тестирование по всем шаблонам данной длины.

Если P-value меньше 0,01. то делается вывод о неслучайности исследуемой последовательности. Иначе последовательность считается случайной.

Статистика теста

$\chi^2(obs)$ — опенка того, как хорошо наблюдаемое число вхождений шаблона соответствует ожидаемому числу вхождений (под предположением случайности).

Эталонное распределение для статистики теста

χ^2-распределение.

Условие верности оценки

Код теста поддерживает длины m от 1 до 32. Для получения поддающихся интерпретации результатов рекомендуется m = 9 или m = 10. Длина последовательности не менее 10^6 бит. По умолчанию длина блока M = 131072, по ее можно изменить. При этом должны соблюдаться следующее условие: M>0,01*n, при этом $N = \left\lfloor \frac{n}{M} \right\rfloor$.

В заключение покажем текст программы, проводя щей несколько статистических тестов, в том числе тесты на детерминированные последовательности и статистический тест ПДСЧ текущего установленного **криптопровайдера.**

```
#include <windows.h>
#include <stdio.h>
BOOL
TestRandomBuffer(
    BYTE  *pRndBuffer,
    DWORD  dwBufLen,      // Параметр зарезервирован
    DWORD *pdwBorderHiSquare,
    DWORD *pdwBufferHiSquare
    );
// Определение имени и типа рабочего провайдера
#define PROV_NAME             MS_DEF_PROV
#define PROV_TYPE             PROV_RSA_FULL
#define RND_BUFFER_LEN        1024          //Длина буфера
                                            фиксирована
void main() {
    BOOL        Result=TRUE;
    HCRYPTPROV  hCryptProv=0;
    PBYTE       pbRndBuffer=NULL;
```

```
    DWORD      dwBufferHiSquare=0,  dwBorderHiSquare=0,
                  dwCount=0;
    pbRndBuffer=LocalAlloc(LMEM_ZEROINIT,RND_BUFFER_LEN);
    if(!pbRndBuffer) {
       printf("Error: Out of memory.\n");
       Result=FALSE;
       goto ReleaseResource;
    }

    // Тест №1. Полностью нулевой буфер.
    printf("Test #1. Zero buffer.\n");
    Result=TestRandomBuffer(pbRndBuffer,RND_BUFFER_LEN,
       &dwBorderHiSquare, &dwBufferHiSquare);
    printf("Board Hi square = %d\n", dwBorderHiSquare);
    printf("Hi square = %d\n", dwBufferHiSquare);
    if (Result) printf("Random buffer test complete
       successful!\n\n");
    else printf("Random buffer test complete
       unsuccessful.\n\n");
/* Тест №2. Первая половина буфера заполнена единицами.
       Вторая нулевая. */
    printfC'Test ft2. Half buffer fill one. Other zero.\n");
    FillMemory(pbRndBuffer, RND_BUFFER_LEN/2, 1);

    Result=TestRandomBuffer(pbRndBuffer,RND_BUFFER_LEN,
       &dwBorderHiSquare, &dwBufferHiSquare);
    printf("Board Hi square = %d\n", dwBorderHiSquare);
    printf("Hi square = %d\n", dwBufferHiSquare);
    if (Result) printfC'Random buffer test complete
       successful!\n\n");
    else printfC'Random buffer test complete
       unsuccessful.\n\n");

/* Тест №3. Буфер заполнен возрастающими от 0 до 255
    последовательностями */
    printfC'Test fl3. Ascending sequences buffer.\n");
    for (dwCount=0; dwCount < RND_BUFFER_LEN; dwCount++)
       pbRndBuffer[dwCount]=(BYTE)dwCount;
    Result=TestRandomBuffer(pbRndBuffer,RND_BUFFER_LEN,
       &dwBorderHiSquare, &dwBufferHiSquare);
    printf("Board Hi square = %d\n", dwBorderHiSquare);
    printf("Hi square = %d\n", dwBufferHiSquare);
    if (Result) printfC'Random buffer test complete
       successful!\n\n");
    else printfC'Random buffer test complete
       unsuccessful.\n\n");
```

```
// Тест №4. Буфер заполненный функцией CryptGenRandom.
    printf("Test #4. CryptGenRandom buffer.\n");
// Открываем временный ключевого контейнер
    if  (!CryptAcquireContext(&hCryptProv,NULL,
        PROV_NAME,PROV_TYPE,CRYPT_VERIFYCONTEXT)){
        printf("Error: CryptAcquireContext=0x%X.\n",
            GetLastError());
        Result=FALSE;
        goto ReleaseResource;
    }

    ZeroMemory(pbRndBuffer,RND_BUFFER_LEN);
    if  (!CryptGenRandom(hCryptProv,RND_BUFFER_LEN,
        pbRndBuffer)){
        printf("Error: CryptGenRandom=0x%X.\n",
            GetLastError());
        Result=FALSE;
        goto ReleaseResource;
    }
    Result=TestRandomBuffer(pbRndBuffer,RND_BUFFER_LEN,
        &dwBorderHiSquare, &dwBufferHiSquare);
    printf("Board Hi square = %d\n", dwBorderHiSquare);
    printf("Hi square = %d\n", dwBufferHiSquare);
    if (Result) printf("Random buffer test complete
        successful!\n");
    else printf("Random buffer test complete
        unsuccessful.\n");
ReleaseResource:
    if (pbRndBuffer) LocalFree(pbRndBuffer);
    return;
}

BOOL
TestRandomBuffer(
    BYTE   *pRndBuffer,
    DWORD  dwBufLen,      // Параметр зарезервирован
    DWORD  *pdwBorderHiSquare,
    DWORD  *pdwBufferHiSquare
    )
{
    DWORD Class[16], dwCount;
    DWORD dwHiSquare=0, dwBorderHiSquare=0;

    dwBorderHiSquare=(24+2048)*RND_BUFFER_LEN/8;
    ZeroMemory(Class, sizeof(Class));
    for (dwCount=0; dwCount < RND_BUFFER_LEN; dwCount++) {
```

```
        Class[((BYTE*)pRndBuffer)[dwCount] & 0xF]++;
        Class[((BYTE*)pRndBuffer)[dwCount] >> 4]++;
    }
    for (dwCount = 0; dwCount < 16; dwCount ++ )
        dwHiSquare += Class[dwCount] * Class[dwCount];
    if (pdwBorderHiSquare)
        *pdwBorderHiSquare=dwBorderHiSquare;
    if (pdwBufferHiSquare) *pdwBufferHiSquare=dwHiSquare;
    if (dwHiSquare > dwBorderHiSquare) return FALSE;
    else return TRUE;
}
```

Надежность реализации криптографических преобразований

При реализации криптографических алгоритмов в рамках криптопровайдера, а также при использовании криптопровайдера из прикладных приложений возможны следующие группы угроз, связанные с нарушением надежности:

1. ошибки при программировании криптографических примитивов;
2. ошибки при использовании криптографических примитивов в рамках криптопровайдера;
3. ошибки при передаче параметров в криптопровайдер и возврате результата обработки;
4. ошибки и сбои аппаратной платформы;
5. случайные и преднамеренные нарушения целостности программ и данных криптопровайдера.

Первая группа угроз принципиально не отличается от угроз общего плана, связанных с технологией надежного программирования. Мы не будем ее подробно рассматривать.

Вторая и третья группа угроз связана с появившейся недавно проблемой корректного взаимодействия программных модулей друг другом. Необходимо отметить, что с точки зрения корректной передачи и возврата параметров криптопровайдер спроектирован и выполнен оптимально, поскольку использует передачу указателей на соответствующие данные.

Четвертая и пятая группа угроз становятся причиной таких последствий:

* нарушения работы фрагментов кода (например, уменьшение числа итераций в блочном алгоритме);

* использования некорректных данных к качестве ключевых;
* несанкционированной пересылки ключевых данных в область действия других прикладных программ или на внешние носители.

Такого рода последствия в прикладной криптографии принято называть опасными. Опасность их в том, что они уменьшают криптографическую стойкость реализованного криптографического алгоритма. Например, уменьшение итераций в блочном алгоритме приводит к резкому снижению трудоемкости дешифрования; сбой, из-за того что значение указателя на ключ изменится и начнет указывать на константный (например, нулевой) массив, вообще вызывает полную потерю стойкости шифра. Б результате использования дважды одной и той же величины k при выполнении ЭЦП (см. главу 1), связанной со сбоем работы ПДСЧ, становятся возможными определение секретного ключа и подделка электронных документов.

Основной вопрос, который приходится решать разработчикам и экспертам при изучении опасных последствий ошибок и сбоев. — оценка вероятности их наступления.

Вполне понятно, что изощренный ум эксперта может вообразить сколь угодно опасную ситуацию, вероятность наступления которой составляет приблизительно 10^{-10} в год, что существенно меньше вероятности даже такого редкого и экзотического события, как падение метеорита точнехонько вам на голову.

Возможность наступления опасных последствий представляет собой пересечение (одновременное наступление) следующих событий — сбоя (ошибки) аппаратной части или нарушения целостности и вероятности конкретных опасных последствий от данной ошибки.

Поскольку прогнозировать последствия ошибок достаточно сложно, считают опасной *любую ошибку аппаратной платформы,* которая наступила во время использования криптографических средств, либо *любое искажение кода и данных криптографического модуля* (в нашем случае — криптопровайдера).

Если предприняты меры защиты от ошибок и искажений целостности, то опасным считается пересечение таких событий — наступления ошибки (сбоя) и несрабатывания защитного механизма.

Пусть вероятность наступления опасного для криптомодуля последствия $P_{опс}$,

Тогда $P_{опк} - P_{ош} * P_{неср}$, где

$P_{ош}$ — вероятность наступления ошибки (сбоя) или нарушения целостности кода (мы условились, что любая ошибка, касающаяся криптомодуля, опасна),

$P_{неср}$, — вероятность несрабатывания средства контроля, ориентированного на блокирование конкретной ошибки.

Сформулируем классы защитных мер применительно к криптопровайдерам.

1. Контроль целостности кода и данных криптопровадера до его исполнения, а также во время исполнения.
2. Тестовый прогон криптографических функций до их реального использования с использованием тестовых примеров.
3. Защита ключевых данных во время их хранения но внешней и оперативной памяти.
4. Контроль качества случайной последовательности, выработанной ДСЧ и используемой при формировании ключей.
5. Логический контроль реального выполнения криптографических процедур.

Поясним подробнее смысл указанных мер защиты и проанализируем их возможную эффективность.

Контроль целостности кода и данных криптопровайдера можно осуществлять до его загрузки (что и делается при инициализации криптопровайдера — вычисляется цифровая подпись под ним, что позволяет убедиться в его целостности) либо после его загрузки в процессе работы. Во втором случае вычисляются хеш-значения от критичных фрагментов кода и данных и фиксируются как эталонные. Затем с заданной периодичностью на эти фрагменты вычисляются актуальные хеш-значения и сравниваются с эталоном. В случае несовпадения фиксируется сбой, и выполнение криптографических функций приостанавливается.

Если $P_{сб}$ — вероятность наступления сбоя во время работы, а P_h — вероятность необнаружения искажения контролируемого массива хеш-функцией, то вероятность наступления опасного события в ходе работы криптопровайдера $P_{оп}$ вычисляется как

$$p_{оп} = p_{сб} * p_h$$

Современная теория надежности оценивает вероятность сбоя в элементной базе компьютерной техники величиной порядка

10^{-6} в час. А вероятность необнаружения ошибки хеш-функцией может быть оценена длиной n хеш-значения в битах, как $P_h = 1/2^n$.

Тестовый прогон криптографических функций позволяет убедиться в том, что в память загружен неискаженный код криптографических модулей, указатели для работы с ключами и открытым текстом указывают на массивы, содержащие правильные данные. Его также можно выполнять с заданной периодичностью. Однако надо заметить, что тестовый прогон как механизм обеспечения надежности считается существенно более «тяжелым», чем контроль целостности внутренних данных и кода (выполняется медленнее, требует переустановки ключевыхданных).

Необходимость защиты ключевых данных при их хранении во внешней памяти очевидна. При хранении ключей в оперативной памяти вполне возможна их выгрузка па диск в ходе выгрузки/подкачки страниц памяти либо при наступлении ошибок и сбоев в пользовательском приложении, вызывающем криптопровайдер. Для избежания компрометации хранимых ключей необходимо использовать их «закрытое» хранение. Самым простым способом является суммирование ключа с защитной последовательностью, хранимой в достаточно удаленном (по адресу) месте, например в другой куче.

Например, для алгоритма ГОСТ, рассмотренного в первом параграфе данной главы, защитную последовательность следует налагать па ключ словами по 4 байта путем суммирования по mod 2^{32}. Тогда для использования защищенного ключа в процессе шифрования необходимо выполнить первую операцию (сумма по mod 2^{32}) блока открытого текста с элементом ключа, затем вычесть по тому же модулю элемент защитной последовательности.

Контроль качества случайной последовательности был подробно рассмотрен в параграфе, посвященном реализации ПДСЧ.

Логический контроль выполнения криптографических функций призван обеспечить контроль правильности использования шифрования или ЭЦП в конкретном приложении. Например, корректное выполнение зашифрования буфера в памяти можно проверить сравнением с буфером открытого текста (шифрованный текст должен отличаться от исходного), контрольным расшифрованием и сравнением с исходным (в этом случае должно получиться то же самое), вычислением статистик по шиф-

рованному тексту (ожидается получение последовательности, близкой к равновероятной по статистическим свойствам).

Контроль правильности работы ЭЦП проверяют, выполняя контрольную проверку подписанного сообщения либо намеренно искажая подписанное сообщение (подпись неверна, но адресат, естественно, получит неискаженное подписанное сообщение).

Все указанные меры позволяют объективно повысить надежность работы криптографических приложений, использующих криптопровайдер, и в ходе отладки выявить большинство программных ошибок разработчика.

Глава 6

Обзор CryptoAPI 2.0

Заключительная глава нашей книги посвящена краткому обзору функций CryptoAPI, реализующих стандарты инфраструктуры открытых ключей. Как уже отмечалось, знания основ криптографии и умение обращаться с набором базовых функций CryptoAPI, о которых мы говорили R предыдущих главах, позволит вам спроектировать и реализовать криптографическую систему любой сложности. Но при необходимости обмена информацией с другими системами возникает вопрос стандартизации используемых структур данных, алгоритмов и протоколов, применяемых в криптографических подсистемах. В рамках инфраструктуры открытых ключей создано и опубликовано множество стандартов. Их поддержка позволяет приложениям, функционирующим на совершенно разных аппаратных и операционных платформах, обмениваться защищенной информацией.

Набор функций CryptoAPI 2.0 реализует большинство из существующих стандартов инфраструктуры открытых ключей. Для удобства читателя функции CryptoAPI 2.0 мы сгруппировали по назначению,

Функции поддержки кодирования и декодирования структур данных

Данная группа функций (таблица 6-1) содержит базовые функции кодирования и декодирования данных. Функции реализуют стандарт ITU-T X.208 и X.209, называемый Abstract Syntax Notation One (ASN.1), и предназначены для кодирования различных структур данных (целые числа, строки символов, сертификаты и т.д.) в последовательные бинарные блоки данных, передаваемые по каналам связи. Для определения содержимого данных, передаваемых в функции, используется уникальный идентификатор объекта (Object Identifier, OID). В большин-

стве случаев идентификатор объекта — это просто строка, содержащая десятичные цифры разделенные точками. Например, строка «1.2.840.113549» является базовым идентификатором компании RSA (см. Приложение 1), а строка «1.2.840.113549.1.1.1» идентифицирует алгоритм RSA той же компании. Именно благодаря уникальности идентификаторов объектов структуры данных, которым присвоены идентификаторы, будут правильно интерпретироваться и обрабатываться на любых программных платформах.

Таблица 6-1. Функции кодирования и декодирования структур данных

Функция	Краткое описание
CryptDecodeObject	Декодирует структуру, тип которой определяется указателем *lpszStructType*.
CryptDecodeObjectEx	Аналогична функции ***CryptDecodeObject.*** Поддерживает дополнительные флаги.
CryptEncodeObject	Кодирует структуру, тип которой определяется указателем *lpszStructType*.
CryptEncodeObjectEx	Аналогична функции ***CryptEncodeObject.*** Поддерживает дополнительные флаги.

Функции управления хранилищами сертификатов

В некоторых случаях приложение должно работать со многими сертификатами одновременно. Хранилища сертификатов и функции их поддержки обеспечивают возможность хранения, получения, перечисления, проверки и использования информации, хранимой в сертификатах. Следующие функции поддерживают работу непосредственно с хранилищами сертификатов (таблица 6-2).

Таблица 6-2. Функции управления хранилищами сертификатов

Функция	Краткое описание
CertAddStoreToCollection	Добавляет определенное хранилище к коллекции хранилищ сертификатов.
CertCloseStore	Освобождает дескриптор хранилища сертификатов.
CertControlStore	Позволяет приложению получать сообщения в определенных системой случаях, относящихся к работе подсистемы обслуживания хранилищ сертификатов.

(см. след. стр.)

Функция	Краткое описание
CertDuplicateStore	Дублирует дескриптор хранилища сертификатов, увеличивая счетчик ссылок данного дескриптора.
CertEnumPhysicalStore	Используется для перечисления доступных физических хранилищ сертификатов для определенного системного хранилища.
CertEnumSystemStore	Используется для перечисления доступных системных хранилищ сертификатов.
CertEnumSystemStoreLocation	Используется для перечисления возможных вариантов хранения системного хранилища сертификатов.
CertGetStoreProperty	Используется для получения свойств указанного хранилища сертификатов.
CertOpenStore	Открывает хранилище сертификатов для определенного типа провайдера хранения сертификатов.
CertOpenSystemStore	Открывает системное хранилище сертификатов для определенного типа протокола.
CertRegisterPhysicalStore	Добавляет физическое хранилище к реестровому системному хранилищу сертификатов.
CertRegisterSystemStore	Регистрирует системное хранилище сертификатов.
CertRemoveStoreFromCollection	Удаляет определенное хранилище из коллекции хранилищ сертификатов.
CertSaveStore	Сохраняет определенное дескриптором хранилище сертификатов.
CertSetStoreProperty	Используется для установки свойств хранилища сертификатов.
CertUnregisterPhysicalStore	Удаляет физическое хранилище из определенного системного хранилища сертификатов.
CertUnregisterSystemStore	Отменяет регистрацию определенного системного хранилища сертификатов.

Общие функции поддержки

CryptoAPI предоставляет несколько процедур, реализующих общие функции поддержки сертификатов, списков отозванных сертификатов (Certificate Revocation List, CRL) и списков доверенных сертификатов (Certificate Trust List, CTL) (таблица 6-3).

Таблица 6-3. Общие функции поддержки

Функция	Краткое описание
CertAddSerializedElement-ToStore	Добавляет блок данных последовательной формы представления сертификата или CRL элемент в хранилище.
CertCreateContext	Создает контекст сертификата из кодированной структуры данных. Созданный контекст не помещается в хранилище.
CertEnumSubjectInSortedCTL	Используется для перечисления доверенных субъектов в сортированном CTL-контексте.
CertFindSubjectInCTL	Находит определенный субъект в CTL.
CertFindSubjectInSortedCTL	Находит определенный субъект в сортированном CTL.

Функции работы с сертификатами

Данная группа функции работает непосредственно с сертификатами (таблица 6-4). Большинство функций имеют аналогичные для работы со списками отозванных сертификатов и списками доверенных сертификатов.

Таблица 6-4. Функции работы с сертификатами

Функция	Краткое описание
CertAddCertificateContext-ToStore	Добавляет контекст сертификата в определенное хранилище.
CertAddCertificateLink-ToStore	Добавляет в хранилище сертификатов ссылку на сертификат, расположенный в другом хранилище
CertAddEncodedCertificate-ToStore	Конвертирует кодированную структуру данных в контекст сертификата и добавляет ею в хранилище сертификатов.
CertCreateCertificateContext	Конвертирует кодированную структуру данных в контекст сертификата.
CertCreateSelfSignCertificate	Создает корневой сертификат.
CertDeleteCertificateFrom-Store	Удаляет сертификат из хранилища сертификатов.
CertDuplicateCertificate-Context	Дублирует контекст сертификата, увеличивая счетчик ссылок данного контекста.
CertEnumCertificatesInStore	Используется для перечисления всех контекстов сертификатов, которые хранятся в определенном хранилище.
CertFindCertificateInStore	Используется для получения первого или последующего контекста сертификата, удовлетворяющего определенным критериям поиска.

(см. след. стр.)

Функция	Краткое описание
CertFree Certificate Context	Освобождает контекст сертификата.
CertGetIssuerCertificate-FromStore	Получает из хранилища контекст сертификата для первого или последующего поставщика для определенного сертификата субъекта.
CertGetSubjectCertificate-FromStore	Получает из хранилища контекст сертификата субъекта, идентифицированного поставщиком и серийным номером.
CertGetValidUsages	Получает массив сертификатов и возвращает массив идентификаторов типов использования сертификатов, общих для всех сертификатов.
CertSerializeCertificate-StoreElement	Создает последовательную форму представления сертификата и его свойств.
CertVerifySubject-Certificate Context	Выполняет проверку контекста сертификата субъекта, используя сертификат поставщика.

Функции работы со списками отозванных сертификатов

Следующая группа функций (таблица 6-5) используется для хранения, получения и обработки списков отозванных сертификатов.

Таблица 6-5. **Функции** работы со **списками** отозванных сертификатов

Функция	Краткое описание
CertAddCRLContextToStore	Добавляет CRL-контекст в хранилище сертификатов.
CertAddCRLLink ToStore	**Добавляет** в хранилище связь с контекстом CRL, находящимся в другом хранилище.
CertAddEncodedCRLToStore	Конвертирует кодированный CRL в CRL-контекст и добавляет контекст в хранилище сертификатов.
CertCreate CRLContext	Конвертирует кодированный CRL в CRL-контекст. Созданный контекст не добавляется в хранилище сертификатов.
CertDelete CRLFromStore	Удаляет CRL из хранилища сертификатов.
CertDuplicate CRLContext	Дублируют CRL-контекст, увеличивая счетчик ссылок.
CertEnumCRLsInStore	Используется для перечисления CRL-контекстов в хранилище.

(см. след. стр.)

Функция	Краткое описание
CertFindCertificateIn CRL	Используется для поиска списка CRL с вхождением определенного сертификата.
CertFindCRLInStore	Используется для перечисления CRL-контекстов из хранилища, удовлетворяющих определенным критериям.
CertFreeCRLContext	**Освобождает** CRL-контекст.
CertGetCRLFromStore	Используется для перечисления CRL-контекстов из хранилища. для определенного поставщика сертификатов.
CertSerializeCRLStoreElement	Конвертирует в последовательную форму представления CRL-контекст и его свойства.

Функции работы со списками доверенных сертификатов

Данная группа функций (таблица 6-6) используется для хранения, получения и обработки списков доверенных сертификатов.

Таблица 6-6. Функции работы со списками доверенных сертификатов

Функция	Краткое описание
CertAddCTLContextToStore	**Добавляет** CTL-контекст в хранилище сертификатов.
CertAddCTLLinkToStore	Добавляет в хранилище связь с контекстом CTL, находящимся в другом хранилище.
CertAddEncodedCTLToStore	Конвертирует кодированный CTL в CTL-контекст и добавляет контекст в хранилище сертификатов.
CertCreateCTLContext	Конвертирует кодированный CTL в CTL-контекст. Созданный контекст не *добавляется* в хранилище **сертификатов.**
CertDelete CTLFromStore	Удаляет CTL из хранилища сертификатов.
CertDuplicateCTLContext	Дублирует CTL-контекст, увеличивая счетчик ссылок.
CertEnum CTLsInStore	Используется для перечисления CTL-контекстов в хранилище,
Cert FindCTLInStore	Используется для перечисления CTL-контекстов из хранилища, удовлетворяющих определенным критериям.
CertFreeCTLContext	Освобождает CTL-контекст.
CertSerialize CTLStoreElement	Конвертирует в последовательную форму представления CTL-контекст и его свойства.

Функции управления свойствами сертификатов и списков сертификатов

Данная группа функций (таблица 6-7) используется для управления свойствами сертификатов, списков отозванных сертификатов и списков доверенных сертификатов.

Таблица 6-7. Функции управления свойствами сертификатов и списков сертификатов

Функция	Краткое описание
CertEnumCertificateContext-Properties	Используется для перечисления свойств указанного контекста сертификата.
CertEnumCRLContextProperties	Используется для перечисления свойств указанного контекста CRL.
CertEnumCTLContextProperties	Используется для перечисления свойств указанного контекста CTL.
CertGetCertificateContextProperty	Возвращает информацию об определенном свойстве сертификата.
CertGetCRLContextProperty	Возвращает информацию об определенном свойстве CRL.
CertGetCTLContextProperty	Возвращает информацию об определенном свойстве CTL.
CertSetCertificateContextProperty	Устанавливает свойства определенного контекста сертификата.
CertSetCRLContextProperty	Устанавливает свойства определенного контекста CRL.
CertSetCTLContextProperty	Устанавливает свойства определенного контекста CTL.

Функции проверки сертификатов и цепочек сертификатов

Процесс проверки сертификатов может основываться па списке доверенных сертификатов или цепочках сертификатов. CryptoAPI 2.0 включает в себя обе эти возможности.

В следующую группу вошли несколько функций (таблица 6-8), позволяющих построить процесс проверки сертификатов на основе списка доверенных сертификатов.

Таблица 6-8. Функции проверки сертификатов на основе списка доверенных сертификатов

Функция	Краткое описание
CertVerifyCTL Usage	Выполняет на основе CTL проверку субъекта для определенного использования сертификата.
CryptMsgEncodeAndSignCTL	Кодирует и подписывает CTL как сообщение.
CryptMsgGetAndVerifySigner	Получает CTL из сообщения и проводит его проверку.
CryptMsgSignCTL	Подписывает сообщение, содержащее CTL

Для использования цепочек сертификатов в процессе проверки CryptoAPI предоставляет функции создания, обработки и проверки цепочек сертификатов (таблица 6-9).

Таблица 6-9. Функции работы с цепочками сертификатов

Функция	Краткое описание
CertCreateCertificateChainEngine	Создает новую подсистему обработки цепочек сертификатов для приложения.
CertCreateCTLEntryFromCertificateContextProperties	Создает точку входа в CTL с атрибутами из свойств контекста сертификата.
CertDuplicateCertificateChain	Дублирует цепочку сертификатов, увеличивая счетчик ссылок, и возвращает указатель на цепочку.
CertFindChainInStore	Используется для перечисления цепочек сертификатов в хранилище.
CertFreeCertificateChain	Освобождает цепочку сертификатов, декрементируя счетчик ссылок.
CertFreeCertificateChainEngine	Уничтожает созданную в приложении новую подсистему обработки цепочек сертификатов.
CertGetCertificateChain	Создает цепочку сертификатов, начиная с конечного сертификата и, если возможно, заканчивая корневым сертификатом.
CertSetCertificateContextPropertiesFromCTLEntry	Устанавливает атрибуты контекста сертификата, используя атрибуты точки входа CTL.
CertIsValidCRLForCertificate	Проверяет CRL на наличие указанного сертификата.
CertVerifyCertificateChainPolicy	Проверяет цепочку сертификатов на соответствие критериям указанной политики,

Функции поддержки криптографических сообщений

Функции работы с криптографическими сообщениями в CryptoAPI делятся на две группы: *базовые функции*, иди *функции низкого уровня* (Low-level message functions), и *упрощенные функции* (Simplified message functions).

Базовые функции создают и обрабатывают непосредственно сообщения PKCS #7. Они кодируют данные для передачи и декодируют при приеме сообщения, а также используются для дешифрования и проверки подписи.

Упрощенные функции находятся на уровень выше, им требуется для работы базовые функции и некоторые функции обработки сертификатов, скрывая часть работы от пользователя.

В таблице 6-10 перечислены базовые функции обработки сообщений. Необходимость их использования возникает крайне редко, так как в большинстве приложений задачи удается решить с помощью упрощенных функций.

Таблица 6-10. Базовые функции поддержки криптографических сообщений

Функция	Краткое описание
CryptMsgCalculate-EncodedLength	Используется для вычисления длины кодированного криптографического сообщения.
CryptMsgClose	Освобождает дескриптор криптографического сообщения.
CryptMsgControl	Используется для проведения определенных операций над криптографическим сообщениям после завершающего вызова функции CryptMsgUpdate.
CryptMsgCountersign	**Подписывает** уже существующую подпись в сообщении.
CryptMsgCountersignEncoded	Подписывает уже существующую подпись в сообщении (кодирует структуру SignerInfo, как определено в PKCS #7).
CryptMsgDuplicate	**Дублирует** дескриптор криптографического сообщения, увеличивая счетчик ссылок данного дескриптора.
CryptMsgGetParam	Используется для получения параметров криптографического сообщения после его кодирования или декодирования.
CryptMsgOpenToDecode	Используется для получения дескриптора криптографического сообщения, предназначенного для операции декодирования.

(см. след. стр.)

Функция	Краткое описание
CryptMsgOpenToEncode	Используется для получения дескриптора криптографического сообщения, предназначенного для операции колирования.
CryptMsgUpdate	Изменяет содержание криптографического сообщения.
CryptMsgVerifyCounterSignature Encoded	Проверяет подпись в соответствии со структурой SignerInfo (как определено в PKCS # 7).
CryptMsgVerifyCounterSignatureEncodedEx	Проверяет подпись в соответствии со структурой SignerInfo. Подписывающий объект может быть определен структурой CERT_PUBLIC_KEY_INFO, контекстом сертификата или контекстом цепочки сертификатов.

Группа упрощенных функций (таблица 6-11) используется в большинстве приложений для работы с криптографическими сообщениями.

Таблица 6-11. Упрощенные функции поддержки криптографических сообщений

Функция	Краткое описание
CryptDecodeMessage	Декодирует криптографические сообщения.
CryptDecryptAndVerifyMessageSignature	Расшифровывает указанное криптографическое сообщение и проверяет подпись отправителя.
CryptDecryptMessage	Расшифровывает указанное криптографическое сообщение.
CryptEncryptMessage	Зашифровывает криптографическое сообщение для определенного получателя или получателей.
CryptGetMessageCertificates	Возвращает дескриптор хранилища сертификатов, которое содержит сертификат сообщения и CRL.
CryptGetMessageSignerCount	Используется для получения количества отправителей подписавших криптографическое сообщение.
CryptHashMessage	Хеширует содержание криптографического сообщения.
CryptSignAndEncryptMessage	Подписывает и зашифровывает криптографическое сообщение для определенного получателя или получателей.
CryptSignMessage	Подписывает криптографическое сообщение.
CryptVerifyDetachedMessageHash	Проверяет хешированное сообщение, содержащее отсоединенный хеш.

(см. след. ***стр.)***

Функция	Краткое описание
CryptVerifyDetached-MessageSignature	Проверяет подписанное сообщение, содержащее отсоединенное значение подписи или подписей.
CryptVerifyMessageHash	Используется для проверки хеша хешированного сообщения.
CryptVerifyMessageSignature	Используется для проверки подписи подписанного сообщения.

Вспомогательные функции

Группа многочисленных вспомогательных **функций** (таблица 6-12) используется для сравнения различных типов данных, поиска, получения необходимой информации о данных и работы с различными структурами криптографических данных и сертификатами.

Таблица 6-12. Функции управления структурами данных и сертификатами

Функция	Краткое описание
CertCompareCertificate	Используется для сраинения содержимого двух сертификатов на предмет их идентичности.
CertCompareCertificateName	Используется для сравнения имен двух сертификатов на предмет их идентичности.
CertCompareIntegerBlob	Используется для сравнения двух блоков данных, содержащихся в структурах типа DATA_BLOB, па предмет их идентичности.
CertComparePublicKeyInfo	Используется для сравнения двух открытых ключей на предмет их идентичности.
CertFindAttribute	Используется для нахождения первого атрибута, определенного его идентификатором.
CertFindExtension	**Используется** для нахождения первого расширения сертификата, определенного его идентификатором.
CertFindRDNAttr	Используется для нахождения первого RDN атрибута, определенного его **идентификатором,** в списке RDN-имен.
CertGetIntendedKeyUsage	Получает буфер с типами использования сертификата.
CertGetPublicKey Length	Получает длину открытого ключа.
CertIsRDNAttrsIn-CertificateName	Сравнивает атрибуты имени сертификата с определенным массивом RDN-имен.

(см. след. стр.)

Функция	Краткое описание
CertVerifyCRLRevocation	Проверяет наличие сертификата субъекта в CRL.
CertVerifyCRLTimeValidity	Проверяет верность CRL в соответствии с определенным временным параметром.
CertVerifyRevocation	Проверяет наличие сертификатов субъектов в CRL.
CertVerifyTimeValidity	Проверяет временную верность сертификата.
CertVerifyValidityNesting	Проверяет верность временных параметров сертификата в соответствии с временными параметрами поставщика.
CryptExportPublicKeyInfo	Экспортирует информацию открытого ключа, ассоциированного с определенным секретным ключом.
CryptExportPublicKeyInfoEx	Экспортирует информацию открытого ключа, ассоциированного с определенным секретным ключом. Приложение может определить алгоритм открытого ключа, отличающийся от используемого по умолчанию криптопровайдером.
CryptFindCertificateKeyProvInfo	Используется для перечисления криптопровайдеров и ключевых контейнеров для нахождения секретного ключа, соответствующего определенному открытому ключу.
CryptFindLocalizedName	Осуществляет поиск локального имени для специфицированного имени, такого, как имя системного хранилища «Root».
CryptHashCertificate	Хеширует кодированный сертификат.
CryptHashPublicKeyInfo	Вычисляет хеш кодированного открытого ключа.
CryptHashToBeSigned	Вычисляет хеш «to be signed» информации в структуре CERT_SIGNED_CONTENT_INFO.
CryptImportPublicKeyInfo	Конвертирует и импортирует в криптопровайдер открытый ключ, представленный структурой CERT_PUBLIC_KEY_INFO. Возвращает дескриптор открытого ключа.
CryptImportPublicKeyInfoEx	**Аналогична** функции *CryptImportPublicKeyInfo*. Поддерживает дополнительные флаги, передаваемые параметрами *dwFlags* и *pvAuxInfo*.

(см. след. стр.)

Функция	Краткое описание
CryptMemAlloc	Выделяет буфер памяти указанного размера. Данную функцию используют все процедуры библиотеки Crypt32.dll.
CryptMemFree	Освобождает память, выделенную функциями *CryptMemAlloc* или *CryptMemRealloc*.
CryptMemRealloc	Освобождает выделенную память и выделяет новый буфер.
CryptQueryObject	Возвращает информацию о содержании блоба или файла.
CryptSignAndEncode-Certificate	Кодирует «to be signed» информацию, подписывает кодированную информацию и кодирует результат операции
CryptSignCertificate	Подписывает «to be signed» информацию.
CryptVerify Certificate-Signature	Проверяет подпись субъекта сертификата или CRL, используя информацию об открытом ключе.
CryptVerifyCertificate-SignatureEx	Расширенная версия функции *CertificateSignature*.

Следующая группа (таблица 6-13) используется для конвертирования различных типов данных.

Таблица 6-13. Функции конвертирования структур данных

Функция	Краткое описание
CertAlgIdToOID	Конвертирует идентификатор алгоритма криптопровайдера в идентификатор объекта (OID).
CertGetNameString	Получает строку с именем субъекта или поставщика сертификата.
CertNameToStr	Конвертирует блоб с именем сертификата в строку символов.
CertOIDToAlgId	Конвертирует идентификатор объекта (OID) в идентификатор алгоритма криптопровайдера.
CertRDNValueToStr	Конвертирует RDN имя в строку символов.
CertStrToName	Конвертирует строку символов в формате X.500 в кодированное имя сертификата.
CryptBinaryToString	Конвертирует бинарную последовательность в форматированную строку.
CryptFormatObject	Форматирует кодированные данные и возвращает Unicode-строку
CryptStringToBinary	Конвертирует форматированную строку в бинарную последовательность.

Группа из четырех функций (таблица 6-14) помогает приложению работать с информацией о расширенных возможностях использования ключа (Enhanced Key Usage, EKU). Функции работают с EKU-*расширением* сертификата и EKU-*свойством* сертификата. Расширение и свойство EKU устанавливает ограничения на использования сертификата. Расширение является частью сертификата. Оно устанавливается поставщиком и доступно только для чтения. Свойство сертификата — это значение, ассоциированное с сертификатом, оно может быть установлено приложением.

Таблица 6-14. **Функции управления информацией о расширенных возможностях** использования ключа

Функция	Краткое описание
CertAddEnhancedKeyUsageIdentifier	Добавляет идентификатор использования ключа к EKU-свойству сертификата.
CertGetEnhancedKeyUsage	Получает информацию о EKU-расширениях или EKU-свойствах сертификата.
CertRemoveEnhancedKeyUsageIdentifier	Удаляет идентификатор использования ключа из EKU-свойства сертификата
CertSetEnhancedKeyUsage	Устанавливает EKU-расширение сертификата.

Следующая группа функций (таблица 6-15) обслуживает работу с *идентификатором ключа* (Key Identifier). Они позволяют создавать, устанавливать и искать идентификатор ключа или его свойства.

Идентификатор ключа — это уникальный идентификатор ключевой пары. Им можно назначить любое уникальное значение, по обычно используется 20 байт хеша по алгоритму SIIA-1 структуры CERT_PUBLIC_KEY_INFO. Как правило, его получают через свойство CERT_KEY_IDENTIFIER_PROP_ID. Идентификатор ключа позволяет использовать ключевую пару для зашифрования и расшифрования сообщения без использования сертификата. Идентификатор ключа не ассоциируется с CRL или CTL.

Таблица 6-15. Функции управления идентификатором ключа

Функция	Краткое описание
CryptCreateKeyIdentifierFromCSP	Создает идентификатор ключа из блоба открытого ключа CSP,
CryptEnumKeyIdentifierProperties	Используется для перечисления идентификаторов ключа и его свойств.

(см. след. стр.)

Функция	Краткое описание
CryptGetKeyIdentifierProperty	Получает определенное свойство из определенного ключевого идентификатора.
CryptSetKeyIdentifierProperty	Устанавливает свойство определенного ключевого идентификатора.

В группе функции провайдера хранилища сертификатов (таблица 6-16) функции обратного вызова используются для регистрации, установки и работы провайдера хранилища сертификатов, определенного приложением.

Таблица 6-16. Функции управления хранилищами сертификатов

Функция	Краткое описание
CertDllOpenStoreProv	Определяет функцию открытия провайдера хранилища сертификатов.
CertStoreProvClose Callback	Определяет действия, производимые при обнулении счетчика ссылок.
CertStoreProvDeleteCertCallback	Определяет действия, производимые перед удалением сертификата из хранилища.
CertStoreProvDeleteCRLCallback	Определяет действия, производимые перед удалением CRL из хранилища сертификатов.
CertStoreProvReadCertCallback	Зарезервирована.
CertStoreProvReadCRLCallback	Зарезервирована.
CertStoreProvSetCertProperty-Callback	Определяет действия, производимые перед вызовом функций *CertSetCertificateContextProperty* и *CertGetCertificateContextProperty*.
CertStoreProvSetCRLProperty-Callback	Определяет действия, производимые перед вызовом функций *CertSetCRLContextProperty* и *CertGetCRLContextProperty*.
CertStoreProv WriteCertCallback	Определяет действия, производимые перед добавлением сертификата в хранилище.
CertStoreProv Write CRLCallback	Определяет действия, производимые перед добавлением CRL в хранилище.
CertStoreProvReadCTL	Считывает копию CTL-контекста. и если она существует, то создает новый CTL-контекст.
CertStoreProv WriteCTL	Определяет возможность добавления CTL в хранилище,
CertStoreProvDelete CTL	Определяет возможность удаления CTL из хранилища.

(см. след. стр.)

Функция	Краткое описание
CertStoreProvSetCTLProperty	Определяет возможность назначения свойств CTL.
CertStoreProvControl	Предоставляет возможность приложению получать извещение о несовпадении содержимого кэша хранилища и содержимого хранила.
CertStoreProvFindCert	Используется для перечисления сертификатов удовлетворяющих определенным критериям.
CertStoreProvFreeFindCert	Освобождает контекст сертификата.
CertStoreProvGetCertProperty	Получает определенное свойство сертификата.
CertStoreProvFindCRL	Используется для перечисления CRL, удовлетворяющих определенным критериям.
CertStoreProvFreeFindCRL	Освобождает CRL-контекст.
CertStoreProvGetCRLProperty	Получаст определенное свойство CRL.
CertStoreProvFindCTL	Используется для перечисления CRL, удовлетворяющих определенным критериям.
CertStoreProvFreeFindCTL	ОсвобождаетCTL-контекст.
CertStoreProvGetCTLProperty	Получает определенное свойство CTL.

Следующая группа функций (таблица 6-17) поддерживает работу с идентификаторами объектов. Эти функции устанавливают, регистрируют и обрабатывают функции обработчики для определенных идентификаторов.

Функции этой группы вызываются такими функциями, как *CryptEncodeObject*, *CryptEncodeObjectEx*, *CryptDecodeObject*, *CryptDecodeObjectEx*, *CertVerifyRevocation*, *CertOpenStore*

Таблица 6-17. Функции управления идентификаторами объектов

Функция	Краткое описание
CryptEnumOIDFunction	Используется для перечисления зарегистрированных OID-функций, определенных типом кодирования, именем функции и идентификатором.
CryptEnumOIDInfo	Используется для перечисления зарегистрированных идентификаторов объектов. Функция возвращает информацию об идентификаторе объекта, который определен идентификатором группы.

(см. след. стр.)

Функция	Краткое описание
CryptFindOIDInfo	Используется для поиска информации об идентификаторе объекта, который определяется ключом и группой.
CryptFreeOIDFunction-Address	Освобождает дескриптор функции, возвращаемый процедурами *CryptGetOIDFunctionAddress* или *CryptGetDefaultOIDFunctionAddress*.
CryptGetDefaultOIDDllList	Возвращает список имен зарегистрированных DLL для определенного набора функций и типа кодирования.
CryptGetDefaultOID-FunctionAddress	Получает первую или последующую устанавливаемую функцию, используемую по умолчанию. Или загружает DLL, содержащую соответствующую функцию.
CryptGetOIDFunction-Address	Используется для поиска инсталлированных функций, определенных типом кодирования и идентификатором.
CryptGetOIDFunction Value	Получает значение переменной, ассоциированной с идентификатором, в соответствии с типом кодирования, именем функции и именем переменной.
CryptInitOIDFunctionSet	Инициализирует и возвращает дескриптор набора OID-функций в соответствии с именем функции.
CryptInstallOIDFunction-Address	Устанавливает набор OID-функций.
CryptRegisterDefaultOID-Function	Регистрирует DLL, содержащую функцию, которая используется умолчанию для определенного типа кодирования и имени функции.
CryptRegisterOIDFunction	Регистрирует DLL. содержащую функцию для определенного типа кодирования, имени функции и идентификатора объекта.
CryptRegisterOIDInfo	Регистрирует OID-информацию, определенную структурой CRYPT_OID_INFO и сохраняемую в реестре.
CryptSetOIDFunction Value	Устанавливает значение переменной для определенного типа кодирования, имени функции и имени неременной.
CryptUnregisterDefault-OIDFunction	Отменяет регистрацию DLL, содержащую функцию, используемую по умолчанию для определенного типа кодирования и имени функции.

(см. след. стр.)

Функция	Краткое описание
CryptUnregisterOIDFunction	Отменяет регистрацию DLL, содержащую функцию для определенного типа кодирования и имени функции.
CryptUnregisterOIDInfo	Отменяет регистрацию OID-информации.

Данная группа функций (таблица 6-18) позволяет пользователю получать объект инфраструктуры открытых ключей по URL сертификатов, CTL или CRL. А также дает возможность получить URL из объекта,

Таблица 6-18. Функции работы с URL объектов инфраструктуры открытых ключей

Функция	Краткое описание
CryptGetObjectUrl	Используется для получения URL **удаленного** объекта из сертификата, CTL или CRL.
CryptRetrieveObjectByUrl	Используется для получения PKI-объекта по указанному URL.

Последняя группа функций (таблица 6-19) позволяет организовать защиту данных на основе пароля. Данные функции поддерживают формат обмена личными секретными данными (Persona! Information Exchange, PFX), описанный в стандарте PKCS #12. К сожалению, данный набор процедур позволяет осуществлять экспорт и импорт на пароле только сертификатов и ассоциированных секретных ключей, не предоставляя возможности работать с произвольной пользовательской информацией.

Таблица 6-19. Функции поддержки обмена личными секретными данными

Функция	Краткое описание
PFXExportCertStore	Экспортирует из указанного хранилища сертификат и, если возможно, ассоциированный секретный ключ.
PFXExportCertStoreEx	Экспортирует из указанного хранилища **сертификат** и, если возможно, ассоциированный секретный ключ.
PFXImportCertStore	Импортирует PFX BLOB и возвращает дескриптор хранилища сертификатов.
PFXIsPFXBlob	Проверяет корректность PFX блоба.
PFXVerifyPassword	Проверяет целостность PFX блоба на указанном пароле.

Приложение 1

Словарь терминов

Приложение содержит наиболее часто используемые криптографические и программные термины, кроме того, для каждого приведен его английский эквивалент. Последнее очень важно, так как в русскоязычной криптографической литературе применяются не дословные переводы терминов, а определения, используемые и государственных стандартах Российской Федерации (ГОСТ 28147-89, ГОСТ Р 34.10-94, ГОСТ Р 34.11-94). Ряд применяемых в англоязычной литературе терминов вообще не имеет соответствующих аналогов в российских документах, и авторы взяли на себя смелость перевести эти термины, исходя из собственного опыта.

Для удобства термины перечислены не в алфавитном порядке, а по тематическим группам. Там, где это необходимо, указаны ссылки на тс параграфы приложения 2, где подробно рассматривается обсуждаемый вопрос или на соответствующее издание.

Общие определения

Шифр ~ Cipher — совокупность алгоритмов или отображений открытой (общедоступной) информации, представленной в формализованном виде, в недоступный для восприятия шифрованный текст, также представленный в формализованном виде.

Криптографический ключ - Cryptographic key, key — параметрический компонент криптографического преобразования, весь или частично неизвестный противнику (злоумышленнику), выполняющему криптоанализ (дешифрование).

Данным термином обозначаются как симметричные ключи шифрования, так и несимметричные ключи или ключевые пары.

При работе а рамках криптографического интерфейса под термином «ключ» понимается вся совокупность ключевой инфор-

мации, участвующей в процессе криптографического преобразования. Точный набор параметров ключевой информации зависит от конкретного алгоритма криптографического преобразования — шифрования, электронной цифровой подписи (ЭЦП) или ключевого обмена. Например, для ключа блочного шифрования совокупность ключевой информации составляют сам ключ, модификатор ключа (если он имеется), синхропосылка, содержимое внутренних регистров шифра. В рамках CryptoAPI для работы с ключевой информацией средствами криптографического интерфейса определяется дескриптор ключа.

Сессионный ключ - Session key — как правило, симметричный ключ блочного (например, DES, RC2, ГОСТ 28147-89) или поточного (например, RC4) шифрования. «Сессионный» означает временный, используемый только для одного сеанса защищенной (закрытой) связи. В рамках концепции PKI долговременными являются ключевые пары цифровой подписи и ключевого обмена

Ключевая пара открытый/секретный ключ ~ Public/private key pair, key pair — определяет ключевую пару несимметричного алгоритма ЭЦП (например RSA, DSS, ГОСТ Р 34.10-94) или ключевого обмена (например, DH).

Открытый ключ ключевой пары - Public key — точный набор параметров, которые составляют открытый ключ, зависит от конкретного алгоритма. Полагается известным *криптоаналитику* (противнику).

Секретный ключ ключевой пары - Private key — точный набор параметров, которые составляют секретный ключ, зависит от конкретного алгоритма. Для одного и того же алгоритма представление секретного ключа могут отличаться. Например, для стандарта RSA секретный ключ имеет два варианта представления.

Зашифрование - Encryption — процедура (детерминированный алгоритм) преобразования открытой информации (открытого текста) и шифрованную (шифртекст) с использованием ключа (ключа зашифрования).

Расшифрование ~ Decryption — процедура (детерминированный алгоритм) преобразования шифрованной информации (шифртекста) в открытую (открытый текст) с использованием ключа (ключа расшифрования).

Криптоанализ, дешифрование - Cryptanalysis — вся совокупность математических, программно-технических, оперативных

и других возможных методов и алгоритмов, направленных на получение ключа (открытого текста) при неизвестном ключе и заданном наборе исходных параметров (например, множества шифртекстов, зашифрованных на одном и том же ключе, шифртекстов, зашифрованных на неисправном шифраторе и т.д.). Классический криптоанализ описывается правилом Кирхгофа: противнику известна вся информация об алгоритме криптографического преобразования, кроме самого ключа.

Открытый текст - Plaintext — исходным материал для шифрования (получения шифртекста) либо результат расшифрования шифртекста.

Шифртекст ~ Ciphertext, encrypted text — результат шифрования открытого текста.

Шифрование

Блочный шифр ~ Block cipher — алгоритм криптографического преобразования, использующий возможность декомпозиции открытого текста на непересекающиеся блоки одинаковой длины. При этом преобразование (шифрование или расшифрование) каждого блока происходит автономно (независимо от предыдущих) либо зависит от результата преобразования предыдущего (нескольких предыдущих блоков).

Поточный шифр ~ Stream cipher — алгоритм криптографического преобразования, предназначенный для шифрования открытого текста произвольной длины.

Блок шифрования - Block — блок открытого текста (шифртекста), обрабатываемый за один цикл в блочном алгоритме.

Модификатор ключа ~ Salt value — блок случайных данных который иногда включается как часть сессионного ключа. Когда модификатор добавляется к сессионному ключу, то он передается вместе с шифртекстом в открытом виде.

Синхропосылка - Initialization vector — открытая информация, используемая в алгоритме блочного или поточного шифрования для шифрования различных открытых текстов.

Эффективная длина ключа - Effective key length — термин, описывающий использование в алгоритме шифрования элементов ключа для регулирования уровня криптографической стойкости преобразования. Эффективная длина ключа определяет число бит ключа, от которых существенно зависит шифрованный текст.

Дополнение - Padding — Процедура дополнения открытого **текста** для обеспечения возможности блочного шифрования данного открытого текста. Процедура дополнения **может** оказать влияние на криптографическую стойкость.

Режим шифрования - Cipher mode

Обратная связь - Feed back

Режим простой замены - Electronic codebook (ECB)

Режим гаммирования ~ Output feedback (OFB)

Режим гаммирования с обратной связью - Cipher block chaining (CFB)

Режим простой замены с зацеплением ~ Cipher block chaining (CBC)

Цифровая подпись и хеширование данных

Хеш-функция, функция хеширования, хеш ~ Hash, digest — преобразование, отображающее строки бит произвольной длины (исходные данные) в строки бит фиксированной длины и удовлетворяющее следующим двум свойствам: по результату преобразования **весьма** трудно (невозможно) вычислить исходные данные **отображенные** в этот результат; для **фиксированных** (заданных) исходных данных **трудно** найти другие исходные данные, отображаемые с тем же результатом.

Хеш-код, хеш-значение ~ Hash-code — строка бит, представляющая **выходной** результат хеш-функции. Определения хеш-функции и **хеш-кода** приведены по ИСО/МЭК 148881-1[3].

Значение хеш-функции - Hash value — *см. хеш-код.*

Хеш с ключем, секретный хеш, секретная хеш-функция - MAC (Message Authentication Code), Keyed-hash, secret hash — алгоритмы хеширования данных, для **вычисления** которых **необходим** секретный ключ симметричного шифрования.

Ключевой обмен - Key exchange

Электронная цифровая подпись (ЭЦП), цифровая подпись, *подпись* ~ Digital signature — строка бит, полученная в результате выполнения процесса формирования подписи.

Процесс формирования подписи - Signature process — процесс, в качестве исходных данных которого выступают сообщение (открытый текст), ключ подписи и **параметры схемы ЭЦП,** в результате чего формируется цифровая подпись.

Процесс проверки **подписи** - Verification process — процесс, в качестве исходных данных которого выступают подписанное сообщение (открытый текст и подпись), ключ проверки и параметры схемы

Подписывание значения хеш-функции ~ Signing a hash

Проверка подписи значения хеш-функции ~ Verifying the hash signature

Генератор (датчик) случайных чисел ~ Random number (sequence) generator — алгоритм получения (выработки) случайных векторов (значений, чисел) с заданными вероятностными и алгебраическими характеристиками. Случайные числа используются в различных криптографических алгоритмах для формирования ключей, синхропосылок, случайных компонентов схем ЭЦП и т.д.

Генератор (датчик) **псевдо-случайных** чисел ~ Pseudo-random number (sequence) **generator** — алгоритм получения псевдослучайных (случайных в статистическом понимании) векторов (значений, чисел) с заданными статистическими и алгебраическими характеристиками.

Начальное заполнение датчика случайных чисел - Seed — исходное значение для реализации генератора (датчика) псевдослучайных чисел.

Стандарты

PKCS (Public Key Cryptographic Standards) ~ Криптографические стандарты открытых ключей — криптографические стандарты RSADSI, описывающие различные аспекты применения инфраструктуры открытых ключей.

DES (Data Encryption Standard) — американский стандарт шифрования данных. DES - первый опубликованный в открытых источниках блочный алгоритм с длиной блока 64 бита и длиной ключа 56 бит. DES принят еще в 1976 году, в настоящее время в США принят новый стандарт шифрования данных (AES), прототипом которого послужил алгоритм шифрования Rijndail.

RC2, RC4 (Rivest Cipher) - Алгоритм шифрования Райвеста — группа алгоритмов шифрования данных, разработанных Роном Райвестом для RSADSI. RC2 — блочный алгоритм с длиной блока 64 бита и эффективной длиной ключа от] до 1024 бит. RC4 — поточный алгоритм с изменяющейся длиной ключа.

ГОСТ 28147-89 — российский стандарт шифрования данных. Это блочный алгоритм с длиной блока 64 бита и длиной ключа 256 бит. Разработан группой советских криптографов во главе с И. А. Заботиным.

MD2, MD4, MD5 (Message Digest) — группа алгоритмов хеширования данных, разработанных Роном Райвестом. Во всех трех алгоритмах вырабатывается хеш длиной 128 бит.

SHA (Secure Hash Algorithm) — алгоритм хеширования данных, разработанный NIST совместно с NSA для использования в паре с алгоритмом цифровой подписи DSA. Алгоритм вырабатывает хеш длиной 160 бит.

ГОСТ Р 34.11-94 — российский стандарт хеширования данных. Используется совместно с алгоритмом цифровой подписи ГОСТ Р 34.10-94. Алгоритм вырабатывает хеш длиной 256 бит,

CBC-MAC — алгоритм секретного хеша, основанный на использовании алгоритмов блочного шифрования, где в качестве значения хеш-функции используется последний зашифрованный блок. Шифрование осуществляется в режиме простой замены с зацеплением (CBC) или гаммирования с обратной связью (CFB).

HMAC (Hugo's Message Authentication Code) — Алгоритм секретного хеша, основанный на использовании итерационных алгоритмон хеширования данных с замешиванием секретного ключа. В качестве базовых алгоритмов хеширования обычно используются SHA (HMAC-SHA) или MD5 (HMAC-MD5).

Имитовставка — российский алгоритм секретного хетиа. Алгоритм является частью стандарта блочного шифрования ГОСТ 28147-89.

RSA (Rivest, Shamir, Adleman) — алгоритм шифрования на несимметричных ключах, разработанный Роном Райвестом, Ади Шамиром и Леонардом Адлеманом.

DSS (Digital Signature Standard) ~ Стандарт цифровой подписи — стандарт цифровой подписи в США принят 1994 году. В качестве алгоритма цифровой подписи в нем используется DSA, а в качестве хеш-функции алгоритм SHA.

DSA (Digital Signature Algorithm) — американский алгоритм цифровой подписи, разработанный NIST совместно с NSA. Используется в парс с алгоритмом хеширования данных SHA. Алгоритм использует секретный ключ длиной 160 бит и открытый ключ длиной от 512 до 1024 бит, длина цифрой подписи составляет 320 бит.

ГОСТ Р 34.10-94 — российский стандарт цифровой подписи. Используется совместно с **алгоритмом** хеширования данных **ГОСТ Р 34.11-94.** Алгоритм использует секретный ключ длиной **256** бит и открытый ключ длиной от **509** до 512 **бит** или **от 1020** до 1024 бит, длина цифрой подписи составляет 512 бит. В настоящее время рекомендуется использовать **открытый ключ** длиной 1024 бита.

ГОСТ Р 34.10-2001 — новый **Российский** стандарт **цифровой** подписи на основе эллиптических **кривых.** Используется совместно с алгоритмом хеширования данных **ГОСТ Р** 34.11-94.

DH (Diffie– Hellman) — алгоритм ключевого обмена, предложенный Уитфилдом Диффи и Мартином Хеллманом **в** 1976 году. Алгоритм **является** первой практической реализацией концепции несимметричных ключевых схем.

RSAES-OAEP (RSA Encryption Scheme — Optimal Assimetric Encryption Padding) - Схема шифрования по алгоритму RSA с использованием метода дополнения OAEP

RSAES-PKCSl-vl_5 (RSA Encryption Scheme - PKCS #1 version 1.5) ~ Схема шифрования по алгоритму RSA с использованием метода дополнения PKCS #1 версии 1.5

OAEP (Optimal Asymmetric Encryption Padding) ~ Алгоритм оптимального дополнения для несимметричного шифрования

Организации

RSADSI (RSA Data Security, Inc.) — основной разработчик и распространитель криптографических стандартов открытых ключей (Public-key cryptography standards, **PKCS).** Компания названа по именам трех ее основателей и владельцев Рона Райвеста **(Rivest),** Ади Шамира (Shamir) и Леонарда Адлемана (Adleman).

NIST (National Institute of Standards and Technology) - Национальный Институт Технологии и Стандартов - подразделение Департамента Торговли США (US Department of Commerce) которое публикует официальные стандарты для правительственных и частных секторов компьютерных систем. Стандарты публикуются под названием «Федеральные стандарты обработки информации» (Federal Information Processing Standards, FIPS).

ITU (International Telecommunication Union) - Международный Союз Телекоммуникаций — подразделение этой органи-

зации ITU-T публикует различные стандарты в области обмена информацией.

NSA (National Security Agency) - Агентство Национальной Безопасности — специальная служба США, одним из направлений деятельности которой является криптография.

Приложение 2

Приложение содержит описание функций и другие справочные данные, относящиеся к стандарту Microsoft Cryptographic Application Programming Interface (CryptoAPI). Процедуры, описанные в приложении, реализуют базовые криптографические преобразования, такие, как функции зашифрования/расшифрования, хеширования, создания/проверки ЭЦП, ключевого обмена, генерации случайной (или псевдослучайной) последовательности. Набор этих функций относится к интерфейсу Microsoft CryptoAPI версии 1.0.

Это приложение создано на основе справочника функций CryptoAPI, опубликованного в Microsoft MSDN Library (January 2002 Release). Мы попытались привести описание функций в соответствии с терминологией, принятой в российских криптографических стандартах, исправить некоторые неточности, которые имеются в оригинальном описании, а также дополнить те темы, которые в справочнике, по нашему мнению, освещены недостаточно.

Краткое описание криптографических алгоритмов и соответствие английских и российских криптографических терминов приведено в Приложении 1.

Функции CryptoAPI 1.0

CryptAcquireContext

Функция ***CryptAcquireContext*** используется для создания дескриптора определенного ключевого контейнера в рамках определенного криптопровайдера. Дескриптор, который возвращает функция, можно применять для вызовов выбранного криптопровайдера.

Функция выполняет две операции. Сначала она пытается найти криптопровайдер с характеристиками, описанными в пара-

метрах *dwProvType* и *pszProvider*. Если криптопровайдер найден, то функция пытается найти ключевой контейнер в рамках криптопровайдера в соответствии с именем, указанным параметром *pszContainer*.

Эту функцию также используют для создания и уничтожения ключевых контейнеров в зависимости от значения параметра *dwFlags*.

```
BOOL WINAPI CryptAcquireContext(
HCRYPTPROV *phProv,
LPCSTR pszContainer,
LPCSTR pszProvider,
DWORD dwProvType,
DWORD dwFlags
);
```

Параметры

phProv

[out] Указатель на дескриптор криптопровайдера.

pszContainer

[in] Имя ключевого контейнера. Это указатель на строку, идентифицирующую ключевой контейнер. Оно не зависит от метода, *используемого* для хранения ключей. Некоторые криптопровайдеры хранят ключевые контейнеры на внешних носителях (смарт-картах, Touche memory), другие — в системном реестре или файлах. Если в *dwFlags* установлен флаг CRYPT_VER1FYCONTEXT, *то pszContainer* должен быть равен NULL.

Если *pszContainer* ранен NULL, то будет использоваться имя ключевого контейнера по умолчанию. Криптопровайдеры Microsoft в этом случае задают имя пользователя, вошедшего в систему, в качестве имени контейнера.

Приложения не должны применять такие ключевые контейнеры для хранения секретных ключей. Когда несколько приложений используют один и тот же контейнер, одно из приложений может изменить или уничтожить ключи, необходимые для работы других.

Приложения могут получить имя контейнера, с которым они работают, прочитав значение PP_CONTATNER с помощью вызова функции ***CryptGetProvParam***.

pszProvider

[in] Указатель на строку, содержащую имя криптопровайдера. Если этот параметр равен NULL, то используется криптопровайдер по умолчанию.

Список доступных в настоящее время криптопровайдеров приведен в конце справочника.

Приложения могут получить имя криптопровайдера, прочитав значение PP_NAME с помощью вызова функции ***CryptGetProvParam***.

dwProvType

[in] Значение типа запрашиваемого криптопровайдера. Тины криптопровайдеров, определенных в настоящее время, и их характеристики приведены в конце справочника.

dwFlags

[in] Значение флагов. Параметр имеет нулевое или одно из значений, перечисленных далее.

* CRYPT_VERIFYCONTEXT

  Если флаг установлен, то приложение не имеет доступа к секретным ключам ключевого контейнера. При этом параметр *pszContainer* должен быть установлен в NULL.

  Флаг предназначен для работы с приложениями, цель которых — проверка цифровой подписи. Операции, обычно необходимые в этом случае, — получение дескрипторов открытых ключей, хеширование и проверка подписи.

  Когда вызывается функция ***CryptAcquireContext,*** многие криптопровайдеры требуют от пользователя ввода дополнительной информации. Например, секретный ключ в ключевом контейнере может быть зашифрован, поэтому необходимо, чтобы пользователь ввел пароль для доступа к ключевой информации. Однако, если определен флаг CRYPT_VERIFYCONTEXT, то доступа к секретному ключу не требуется, и нет необходимости вводить пароль пользователя.

* CRYPT_NEWKEYSET

  Если флаг установлен, то будет создан новый ключевой контейнер с именем, соответствующим *pszContainer*. Если *pszContainer* равен NULL, то создается ключевой контейнер с именем, используемым по умолчанию.

При создании ключевого контейнера большинство криптопровайдеров не создают автоматически ключевых пар цифровой подписи (и/или ключевого обмена). Для генерации этих ключей необходимо вызвать функцию ***CryptGenKey***.

* CRYPT_MACHINE_KEYSET

По умолчанию ключи и ключевые контейнеры сохраняются как пользовательские. Для Base Providers это значит, что контейнеры сохраняются в пользовательском профиле (например, в ключе реестра HKEY_CURRENT_USER). Флаг CRYPT_MACHINE_KEYSET в комбинации с другими флагами означает, что ключевой контейнер является машинным. Для Base Providers, это означает, что контейнеры сохраняются в профиле All Users (или в ключе реестра HKEY_LOCAL_MACHINE).

Если ключевой контейнер является машинным, то при любом обращении к функции ***CryptAcquireContext*** следует использовать флаг CRYPT_MACHINE_KEYSET.

Он применяется для ключевых контейнеров, к которым обращаются сервисы или пользователи, не имеющие интерактивного доступа к системе.

* CRYPT_DELETEKEYSET

Если флаг установлен, то ключевой контейнер, соответствующий *pszContainer*, удаляется. Если *pszContainer* — NULL, то удаляется ключевой контейнер с именем, заданным по умолчанию. Все ключевые пары в ключевом контейнере также уничтожаются.

Когда флаг CRYPT_DELETEKEYSET установлен, значение, возвращенное в *phProv*, не определено и функция ***CryptReleaseContext*** не должна вызываться.

* CRYPT_SILENT

Для данного запроса криптопровайдер не буден активизировать пользовательский диалог. Если криптопровайдер должен активизировать пользовательский диалог, то функция возвращает FALSE с кодом ошибки NTE_SILENT_CONTEXT. Также, если вызов функции ***CryptGenKey*** с флагом CRYPT_USER PROTECTED происходит в ключевом контейнере, созданном с флагом CRYPT_SILENT, функция вернет FALSE с кодом ошибки NTE_SILENT_CONTEXT.

Флаг CRYPT_SILENT предназначен приложениям, для которых криптопровайдер не может активизировать пользовательский диалог.

Данный флаг поддерживается, начиная с Microsoft Windows 2000. Его не поддерживает Windows 98 или Microsoft Internet Explorer 5.0.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки может быть получен через функцию *GetLastError*.

* ERROR_BUSY (некоторые криптопровайдеры могут возвращать данную ошибку, если установлен флаг CRYPT_DELETEKEYSET, а при этом другой поток или процесс использует данный ключевой контейнер)
* ERROR_INVALID_PARAMETER
* ERROR_NOT_ENOUGH_MEMORY
* NTE_NO_MEMORY
* NTE_BAD_FLAGS
* NTE_BAD_KEYSET
* NTE_BAD_KEYSET_PARAM
* NTE_BAD_PROV_TYPE
* NTE_BAD_SIGNATURE
* NTE_EXISTS
* NTE_KEYSET_ENTRY_BAD
* NTE_KEYSET_NOT_DEF
* NTE_PROV_DLL_NOT_FOUND
* NTE_PROV_TYPE_ENTRY_BAD
* NTE_PROV_TYPE_NO_MATCH
* NTE_PROV_TYPE_NOT_DEF
* NTE_PROVIDER_DLL_FAIL
* NTE_SIGNATURE_FILE_BAD

Дополнительная информация

Windows NT/2000: требуется Windows NT 4.0 или более поздняя версия.

Windows 95/98: требуется Windows 95 OSR2 или более поздняя версия (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

Unicode: на всех платформах реализованы Unicode- и ANSI-версии.

См. также

CryptGenKey, CryptGetProvParam, CryptReleaseContext

CryptContextAddRef

Функция *CryptContextAddRef* увеличивает на единицу *счетчик ссылок* (reference count) на дескриптор криптопровайдера *hProv*. Эта функция используется, когда дескриптор включается как член какой-либо структуры, передаваемой в другую функцию. Функцию *CryptReleaseContext* следует вызывать, когда в дескрипторе криптопровайдера больше нет необходимости.

```
BOOL WINAPI CryptContextAddRef(
HCRYPTPROV hProv,
DWORD pdwReserved,
DWORD dwFlags
);
```

Параметры

hProv

[in] Дескриптор криптопровайдера, для которого инкрементируется счетчик ссылок. Данный дескриптор следует предварительно получить через вызов функции *CryptAcquireContext*.

pdwReserved

[in] Параметр зарезервирован и должен быть равен NULL.

dwFlags

[in] Параметр зарезервирован и должен быть нулевым.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки может быть получен через функцию *GetLastError*:

* ERROR_INVALID_PARAMETER

Комментарии

Для освобождения дескриптора криптопровайдера, счетчик ссылок которого был увеличен функцией *CryptContextAddRef*, необходимо соответствующее количество вызовов функции *CryptReleaseContext*.

Дополнительная информация

Windows NT/2000: необходима Windows 2000,

Windows 95/98: необходима Windows 98.

Заголовок: прототип в файле wincrypc.h.

Библиотека: advapi32.dll

См. также

CrypiAcquire ***Context, CryptRelease Context***

CryptEnumProviders

Функция *CryptEnumProviders*используется для получения первого и следующего доступного криптопровайдера. При применении в цикле функция позволяет получить все криптопровайдеры, установленные на нашем компьютере.

```
BOOL WINAPI CryptEnumProviders(
DWORD dwIndex,
DWORD *pdwReserved,
DWORD dwFlags,
DWORD *pdwProvType,
LPTSTR pszProvName,
DWORD *pcbProvName
);
```

Параметры

dwlndex

[in] Индекс следующего криптопровайдера в перечислении.

pdwReserved

[in] Параметр зарезервирован и должен быть равен NULL.

dwFlags

[in] Параметр зарезервирован и должен быть нулевым.

pdwProvType

[out] Адрес значения типа DWORD, предназначенного для получения типа криптопровайдера.

pszProvName

[out] Указатель на буфер для получения имени криптопровайдера. Имя передается в виде строки, которая закапчивается нулевым символом. Параметр может содержать NULL-указатель. Тогда параметр *pcbProvName* используется для получения необходимого размера буфера. См. раздел «Получение данных неизвестного размера» Приложения 2.

pcbProvName

[in/out] Указатель на значение типа DWORD, определяющее размер в байтах буфера *pszProvName*. После завершения функции DWORD показывает количество байт, переданных в буфер. Если параметр *pszProvName* содержит NULL, то *pcbProvName* используется для получения необходимого размера буфера.

Примечание

Когда обрабатываются возвращаемые в буфере данные, приложение должно использовать фактический размер данных. Он может оказаться немного меньше, чем размер буфера, указанный на входе. На входе размер буфера обычно указывают таким, чтобы буфера хватало для получения максимально возможного объема данных. На выходе переменная, на которую указывает данный параметр, содержит фактический размер данных, скопированных в буфер.

Возвращаемое *значение*

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_MORE_DATA
* ERROR_NO_MORE_ITEMS
* ERROR_NOT_ENOUGH_MEMORY
* NTE_BAD_FLAGS
* NTE_FAIL

Комментарии

Функция ***CryptEnumProviders*** используется для перечисления криптопровайдеров, установленных на вашем компьютере. Для перечисления типов криптопровайдеров предназначена функция ***CryptEnumProviderTypes***.

Дополнительная информация

Windows NT/2000: необходима Windows 2000.

Windows 95/98: необходима Windows 98.

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

Unicode: на всех платформах реализованы Unicode- и ANSI-версии.

См. *также*

CryptEnumProviderTypes

CryptEnumProviderTypes

Функция ***CryptEnumProviderTypes*** применяется для получения первого и следующего типов доступных криптопровайдеров. При использовании в цикле функция позволяет получить все типы криптопровайдеров, установленные на вашем компьютере.

```
BOOL WINAPI CryptEnumProviders(
DWORD dwIndex,
DWORD *pdwReserved,
DWORD  rfwFlags,
DWORD *pdwProvType,
LPTSTR pszTypeName,
DWORD *pcbTypeName
);
```

Параметры

dwIndex

[in] Индекс следующего типа криптопровайдера в перечислении.

pdwReserved

[in] Параметр зарезервирован и должен быть равен NULL.

dwFlags

[in] Параметр зарезервирован и должен быть нулевым,

pdwProvType

[out] Адрес значения типа DWORD, предназначенного для получения типа криптопровайдера.

pszTypeName

[out] Указатель на буфер для получения имени типа криптопровайдера. Имя передается в виде строки, которая заканчива-

ется пулевым символом. Некоторые криптопровайдеры не содержат имен своих типов. В этом случае имя не возвращается, а значение, на которое указывает *pcbTypeName*, будет нулевым.

Параметр может содержать NULL-указатель. Тогда параметр *pcbTypeName* используется для получения необходимого размера буфера. См. раздел «Получение данных неизвестного размера» в Приложении 2.

pcbTypeName

[in/out] Указатель на значение типа DWORD, определяющее размер в байтах буфера *pszTypeName*. После завершения функции DWORD показывает количество байт, переданных в буфер. Если параметр *pszTypeName* содержит NULL, то *pcbTypeName* используется для получения необходимого размера буфера. Некоторые криптопровайдеры не содержат имен своих типов. В этом случае значение, на которое указывает *pcbTypeName,* будет нулевым.

Примечание

Когда обрабатываются возвращаемые в буфере данные, приложение должно использовать фактический размер данных. Он может быть немного меньше, чем размер буфера, указанный па входе. На входе размер буфера обычно задают таким, чтобы буфера хватало для получения максимально возможного объема данных. На выходе переменная, на которую указывает данный параметр, содержит фактический размер данных, скопированных в буфер.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию *GetLastError*:

* ERROR_NO_MORE_ITEMS
* ERROR_NOT_ENOUGH_MEMORY
* NTE_BAD_FLAGS
* NTE_FAIL

Комментарии

Функция *CryptEnumProviderTypes* используется для перечисления типов криптопровайдеров, установленных на вашем компьютере. Для перечисления криптопровайдеров, соответству-

ющих указанному типу, предназначена функция ***CryptEnum-Providers***.

Дополнительная информация

Windows NT/2000: необходима Windows 2000.

Windows 95/98: необходима Windows 98.

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

Unicode: на всех платформах реализованы Unicode- и ANSI-версии.

См, также

CryptEnumProviders

CryptGetDefaultProvider

Функция ***CryptGetDefaultProvider*** находит криптопровайдер, используемый по умолчанию, для указанного типа криптопровайдера. Функция осуществляет поиск криптопровайдера, используемого по умолчанию, или для текущего пользователя, или для компьютера (LocalMachine).

```
BOOL WINAPI CryptGetDefaultProviders(
DWORD dwProvType,
DWORD *pdwReserved,
DWORD dwFlags,
LPTSTR pszProvName,
DWORD *pcbProvName
);
```

Параметры

dwProvType

[in] Тип криптопровайдера, для которого выполняется поиск имени криптопровайдера. используемого по умолчанию. Типы криптопровайдеров, определенных в настоящее время, и их характеристики приведены в конце справочника.

pdwReserved

[in] Параметр зарезервирован и должен быть равен NULL.

dwFlags

[in] В настоящее время определены следующие значения флагов:

* CRYPT_MACHINE_DEFAULT

  Используется для поиска криптопровайдера, используемого по умолчанию для компьютера;

* CRYPT_USER_DEFAULT

  Используется для поиска криптопровайдера, используемого по умолчанию для текущего пользователя.

pszProvName

[out] Указатель па буфер для получения имени типа криптопровайдера. Имя передастся в виде строки, заканчивающейся пулевым символом. Параметр может содержать NULL-указатель. Тогда параметр *pcbProvName* применяется для получения необходимого размера буфера.

См. раздел «Получение данных неизвестного размера» в Приложении 2.

pcbProvName

[in/out] Указатель на значение типа DWORD, определяющее размер в байтах буфера *pszTypeName*.После завершения функции DWORD показывает количество байт, переданных в буфер. Если параметр *pszProvName* содержит NULL, то *pcbTypeName* используется для задания необходимого размера буфера.

Примечание

Когда обрабатываются возвращаемые в буфере данные, приложение должно использовать фактический размер данных. Он может быть немного меньше, чем размер буфера, указанный на входе. На входе размер буфера обычно задают таким, чтобы буфера хватало для получения максимально возможного объема данных. На выходе переменная, на которую указывает данный параметр, показывает фактический размер данных скопированных в буфер.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_INVALID_PARAMETER
* ERROR_MORE_DATA
* ERROR_NOT_ENOUGH_MEMORY
* NTE_BAD_FLAGS

Дополнительная информация

Windows NT/2000: необходима Windows 2000.

Windows 95/98: необходима Windows 98.

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

Unicode: на всех платформах реализованы Unicode- и ANSI-версии.

См. *также*

CryptSetProvider, CryptSetProviderEx

CryptGetProvParam

Функция ***CryptGetProvParam*** возвращает параметры криптопровайдера,

```
BOOL WINAPI CryptGetProvParam(
HCRYPTPROV hProv,
DWORD dwParam,
BYTE *pbData,
DWORD *pdwDataLen,
DWORD dwFlags
);
```

Параметры

hProv

[in] Дескриптор криптопровайдера. Данный дескриптор должен быть предварительно получен через вызов функции ***CryptAcquireContext.***

dwParam

[in] Значение определяющее запрашиваемый параметр. В настоящее время определены следующие значения параметра *dwParam.*

* PP_CONTAINER

  Имя ключевого контейнера. Когда запрашивается этот параметр, функция заполняет *pbData* буфер именем текущего ключевого контейнера. Это имя представляется в форме строки, заканчивающейся нулевым символом.

  Это та же самая строка, которая передастся в параметре *pszContainer* функции ***CryptAcquireContext*** при задании ключевого контейнера.

  Параметр используется для определения имени ключевого контейнера по умолчанию.

* PP_ENUMALGS

  Информация об алгоритме, возвращаемая в виде структуры PROV_ENUMALGS. Когда этот параметр установлен, функция заполняет *pbData* буфер информацией об одном из алгоритмов, поддерживаемых криптопровайдером. Параметр PP_ENUMALGS следует передавать в цикле, чтобы перечислить все поддерживаемые алгоритмы. Порядок перечисления алгоритмов не определяется.

  Первый раз, когда передаётся параметр PP_ENUMALGS, следует установить флаг CRYPT_FIRST. Таким образом гарантируется, что будет возвращена информация относительно «первого» алгоритма в списке перечисления. Затем параметр PP_ENUMALGS можно неоднократно передавать для определения информации об остальных алгоритмах. Если функция ***CryptGetProvParam*** возвращает FALSE с кодом ошибки ERROR_NO_MORE_ITEMS, то достигнут конец списка перечисления.

  Функция не сохраняет информацию о потоках, поэтому в случае вызова в многопоточном контексте вы можете не подучить всей необходимой информации.

* PP_ENUMALGS_EX

  Информация об алгоритме, возвращаемая в виде структуры PROV_ENUMALGS_EX. Когда этот параметр установлен, функция заполняет *pbData* буфер информацией об одном из алгоритмов, поддерживаемых криптопровайдером. Параметр PP_ENUMALGS следует передавать в цикле, чтобы перечислить все поддерживаемые алгоритмы. Порядок перечисления алгоритмов не определяется.

  Первый раз, когда передаётся параметр PP_ENUMALGS_EX, необходимо установить флаг CRYPT_FIRST. Таким образом гарантируется, что будет возвращена информация относительно «первого» алгоритма в списке перечисления. Затем параметр PP_ENUMALGS можно неоднократно передавать для определения информации об остальных алгоритмах. Если функция ***CryptGetProvParam*** возвращает FALSE с кодом ошибки ERROR_NO_MORE_ITEMS, то достигнут конец списка перечисления.

  Функция не сохраняет информацию о потоках, поэтому в случае вызова в многопоточном контексте вы можете не получить всей необходимой информации.

Флаг PP_ENUMALGS_EX поддерживается Internet Explorer 4.0 или выше, Windows NT 4.0 Service Pack 4, Windows 98 или выше и Windows 2000.

* PP_ENUMCONTAINERS

  Имя одного из ключевых контейнеров в рамках указанного криптопровайдера. Имя копируется в *pbData* буфер в виде строки, заканчивающейся нулевым символом.

  Параметр PP_ENUMCONTAINERS следует передавать в цикле для получения имен всех ключевых контейнеров аналогично параметрам PP_ENUMALGS и PP_ENUMALGS_EX.

  Функция не сохраняет информацию о потоках, поэтому в случае вызова в многопоточном контексте вы можете не получить всей необходимой информации.

* PP_IMPTYPE

  Тип реализации криптопровайдера. Величина DWORD, содержащая тип реализации криптопровайдера.

  Возможные типы реализации перечислены в разделе «Комментарии».

* PP_NAME

  Имя криптопровайдера. Имя копируется в *pbData* буфер в виде строки, заканчивающейся нулевым символом.

  Это та же самая строка, которая передаётся в параметре *pszProvider* функции ***CrypAcquireContext*** при задании используемого криптопровайдера.

* PP_VERSION

  Версия криптопровайдера. Возвращается значение DWORD, содержащее номер версии криптопровайдера. Младший байт содержит младший номер версии, а следующий байт — старший номер версии. Например, версия 1.0 представлена как 0x00000100.

* PP_SIG_KEYSIZE_INC

  Значение DWORD, определяющее шаг инкремента длины ключа ЭЦП в битах. Эта информация используется в совокупности с информацией, полученной через параметр PP_ENUMALGS_EX, о допустимых длинах ключей ЭЦП.

  Например, если криптопровайдер определяет в структуре PROV_ENUMALGS_EX значение *dwMinLen* равным 512 битам, *dwMaxLen* — 1024 битам, а шаг инкремента — 64

битам, то допустимые длины ключа — 512, 567, 640, ..., 1024. Их можно использовать при вызове функции ***CryptGenKey***.

Этот параметр поддерживается, начиная с Windows 2000, и не поддерживается в Windows 98 или Internet Explorer 5.0.

* PP_KEYX_KEYSIZE_INC

  Значение DWORD, определяющее шаг инкремента длины ключа алгоритма ключевого обмена в битах. Эта информация используется в совокупности с информацией, полученной через параметр PP_ENUMALGS_EX, о допустимых длинах ключей алгоритма ключевого обмена.

  Например, если криптопровайдер определяет в структуре PROV_ENUMALGS_EX значение *dwMinLen* в 512 бит, *dwMaxLen* в 1024 бита, а шаг инкремента — в 64 бита, то допустимые длины ключа — 512, 567, 640, ..., 1024. Их можно использовать при вызове функции ***CryptGenKey***.

  Этот параметр поддерживается, начиная с Windows 2000, и не поддерживается в Windows 98 или Internet Explorer 5.0.

* PP_KEYSET_SEC_DESCR

  Дескриптор безопасности для ключевого контейнера. Не поддерживается операционной системой Microsoft Windows 95.

* PP_UNIQUE_CONTAINER

  Уникальное имя ключевого контейнера. Когда запрашивается этот параметр, функция заполняет буфер *pbData* уникальным именем текущего ключевого контейнера. Это имя представляется в форме строки, заканчивающейся нулевым символом. Для многих криптопровайдеров ато имя совпадает с именем ключевого контейнера.

  Функция ***CryptAcquireContext*** должна работать с именем ключевого контейнера.

* PP_PROVTYPE

  Значение DWORD, содержащее тип криптопровайдера. Типы криптопровайдеров, определенные в настоящее время, и их характеристики приведены в конце справочника.

* PP_USE_HARDWARE_RNG

  Значение BOOL, показывающее возможность поддержки криптопровайдером аппаратного датчика случайных чисел (ДСЧ). Если установлен параметр PP_USE_HARDWARE_RNG и функция возвращает TRUE, то аппаратный ДСЧ поддерживается, если FALSE — то нет. Если аппаратный

ДСЧ поддерживается криптопровайдером, то параметр PP_USE_HARDWARE_RNG можно использовать при вызове функции *CryptSetProvParam* для указания криптопровайдеру, что в контексте, определяемом текущим дескриптором, должен использоваться только аппаратный ДСЧ.

Если установлен параметр PP_USE_HARDWARE RNG, то *pbData* устанавливается в NULL и *dwFlags* должен быть нулевым.

* PP_KEYSPEC

Значение DWORD, определяющее типы ключей (ЭЦП - AT_SIGNATURE, ключевого обмена - AT_ KEYEXCHANGE), поддерживаемых криптопровайдером. Возвращаемое значение является результатом операции логического OR.

Например, Microsoft Base Cryptographic Provider v1.0 возвращает значение AT_SIGNATURE| AT_KEYEXCHANGE.

Данный параметр поддерживается только на Windows 2000.

pbData

[outj Буфер данных параметра. Функция копирует соответствующие параметру данные в буфер. Формат этих данных зависит от значения *dwParam*.

Если параметр равен NULL, то данные не копируются. Требуемый размер буфера (в байтах) возвращается в *pdwDataLen*. См. раздел «Получение данных неизвестного размера» в Приложении 2.

pdwDataLen

[in/out] Адрес длины данных параметра. При вызове функции этот параметр показывает число байт в буфере *pbData*. После её исполнения параметр показывает размер данных параметра в байтах, скопированных в буфер *pbData*.

Примечание

Когда обрабатываются возвращаемые в буфере данные, приложение должно использовать фактический размер данных. Он может быть немного меньше, чем размер буфера, указанный на входе. На входе размер буфера обычно задают таким, чтобы буфера хватало для получения максимально возможного объема данных. На выходе переменная, на которую указывает данный параметр, показывает фактический размер данных, скопированных в буфер.

Если функция завершается с кодом ошибки, отличным от ERROR_MORE_DATA, в этом параметре возвращается ноль.

Если функции передаётся один из параметров перечисления (PP_ENUMALGS, PP_ENUMALGS_EX или PP_ENUMCONTAINERS), параметр *pdwDataLen* работает несколько по-иному. Когда *pbData* – NULL или величина, указанная *pdwDataLen*, недостаточно велика, значение, возвращаемое в этом параметре, равно размеру самого большого элемента в списке перечисления вместо размера читаемого в настоящее время элемента.

Если функции передается параметр PP_ENUMCONTAINERS, то при первом вызове функция вернет размер максимально возможной длины имени ключевого контейнера. При следующих вызовах функции величина, указанная *pdwDataLen*, не изменяется. Фактическую длину имени контейнера, если необходимо, приложение определяет через вызовы соответствующих функций (например, ***lstrlen***).

Когда функции передаётся один из параметров перечисления *и pbData* параметр – NULL, флаг CRYPT_FIRST необходимо установить, чтобы информация о размере была правильно восстановлена.

dwFlags

[in) Значения флагов. В настоящее время следующие значения флагов определены для этой функции.

- CRYPT_FIRST

  Когда читается параметр перечисления (например, PP_ENUMALGS, PP_ENUMALGS_EX или PP_ENUMCONTAINERS) и установлен этот флаг, возвращается первый элемент в списке перечисления. Иначе — следующий элемент в списке.

  Если флажок установлен, когда читается параметр не перечисления, возвращается код ошибки NTE_BAD_FLAGS.

  Если параметр *dwParam* установлен в PP_KEYSET_SEC_DESCR, то *dwFlags* используется для передачи битовых флагов SECURITY_INFORMATION, которые определяют запрашиваемую информацию, как определено в *Win32 Programmer's Reference*. Значения битовых флагов SECURITY_INFORMATION могут быть скомбинированы операцией поразрядного OR.

  Следующие значения флагов определены для использования с параметром PP_KEYSET_SEC_DESCR.

* OWNER_SECURITY_INFORMATION

  Идентификатор владельца (Owner) объекта.

* GROUP_SECURITY_INFORMATION

  Идентификатор первичной группы (Primary group) объекта.

* DACL_SECURITY_INFORMATION

  Дискреционный список доступа (Discretionary ACL) объекта.

* SACL_SECURITY_INFORMATION

  Системный список доступа (System ACL) объекта.

Возвращаемое значение

При успешном завершении функция возвращает TRUE. в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* ERROR_MORE_DATA
* ERROR_NO_MORE_ITEMS
* NTE_BAD_FLAGS
* NTE_BAD_TYPE
* NTE_BAD_UID

Комментарии

Функция не сохраняет информацию о потоках, поэтому в случае вызова в многопоточном контексте вы можете не получить всей необходимой информации.

Если читается параметр перечисления PP_ENUMCONTAINERS, то перечисление ключевых контейнеров зависит от того, с каким флагом вызывалась функция ***CryptAcquireContext*** при получении дескриптора криптопровайдера *hProv*. Если при вызове функции ***CryptAcquireContext*** использовался флаг CRYPT_MACHINE_KEYSET, то перечисление контейнеров происходит в рамках машинного хранилища, если этот флаг не был определен, то запрашивается пользовательское хранилище ключевых контейнеров.

Ниже перечислены значения, которые возвращаются при запросе параметра PP_IMPTYPE:

* CRYPT_IMPL_HARDWARE - аппаратная реализация;
* CRYPT_IMPL_SOFTWARE - программная реализация;

* CRYPT_IMPL_MIXED — программно-аппаратная реализация;
* CRYPT_IMPL_UNKNOWN — неизвестный тип реализации.

Если параметр *dwParam* установлен в PP_KEYSET_SEC_DESCR, то *dwFlags* используется для передачи битовых флагов SECURITY_INFORMATION, которые определяют запрашиваемую информацию. Указатель на SECURITY_DESCRIPTOR возвращается *ftpbData,* а размер — в *pdwDataLen.* Подробнее материал изложен в описаниях функций ***RegGetKeySecurity*** и ***RegSetKeySecurity*** в *Win32 Programmer's Reference.*

Криптографические алгоритмы, полученные через параметры PP_ENUMALGS или PP_ENUMALGS_EX, классифицируются по типам. Для получения класса алгоритма используется макрос ***GET_ALG_CLASS***, где в качестве параметра применяется идентификатор алгоритма. В настоящее время определены следующие классы алгоритмов:

* ALG_CLASS_DATA_ENCRYPT
* ALG_CLASS_HASH
* ALG_CLASS_KEY_EXCHANGE
* ALG_CLASS_SIGNATURE

В *таблице П-1 перечислены алгоритмы,* поддерживаемые *криптопровайдерами* Microsoft, и классы этих алгоритмов.

Таблица **П-1.** Алгоритмы, поддерживаемые **криптопровайдерами Microsoft**

Имя	Идентификатор	Класс
«MD2»	CAШ_MD2	ALG_CLASS_HASH
«MD5»	CALG_MD5	ALG_CLASS_HASH
«SHA»	CALG_SHA	ALG_CLASS_HASH
«MAC»	CAШ_MAC	ALG_CLASS_HASH
«RSA_SIGN»	CALG_RSA_SIGN	ALG_CLASS_SIGNATURE
«RSA_KEYX»	CALG_RSA_KEYX	ALG_CLASS_KEY_EXCHANGE
«RC2»	CALG_RC2	ALG_CLASS_DATA_ENCRYPT
«RC4»	CALG_RC4	ALG_CLASS_DATA_ENCRYPT

Дополнительная информация

Windows NT/2000: необходима Windows 2000.

Windows 95/98: необходима Windows 98.

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptAcquireContext, CryptCreateHash, CryptDeriveKey, CryptGenKey, CryptGetKeyParam, CryptSetProvParam

CryptReleaseContext

Функция *CryptReleaseContext* используется для освобождения дескриптора криптопровайдера, созданного ***CryptAcquireContext.*** Каждый вызов этой функции декрементирует счетчик ссылок дескриптора. Когда счетчик достиганет нулевого значения, контекст действительно освобождается и приложение не может больше использовать дескриптор *hProv* для вызова криптопровайдера.

Это действие должно быть выполнено, когда приложение, использующее данный ключевой контейнер криптопровайдера, завершается. После вызова этой функции дескриптор ключевого контейнера, указанный *hProv* параметром, больше не существует. Функция не уничтожает ключевых пар и контейнеров.

```
BOOL WINAPI CryptReleaseContext(
HCRYPTPROV hProv,
DWORD dwFlags
);
```

Параметры

kProv

[in] Дескриптор криптопровайдера, полученный приложением через вызов функции ***CryptAcquireContext.***

dwFlags

[in] Значения флагов. Этот параметр зарезервирован для будущего использования и должен быть равен нулю.

Если параметр не равен нулю, функция вернет FALSE, но криптопровайдер будет освобожден.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки (см. таблицу П-1) можно получить через функцию *GetLastError*:

* ERROR_BUSY
* ERROR_INVALID_HANDLE

* ERROR_INVALID_PARAMETER
* NTE_BAD_FLAGS
* NTE_BAD_UID

Комментарии

После вызова этой функции дескриптор *hProv* становится недействительным. Все сессионные ключи и объекты хеш-функций, которые создавались в контексте *hProv*, уничтожаются. Рекомендуется, сделать это заранее через вызовы функций ***CryptDestroyKey*** и ***CryptDestroyHash***.

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptAcquireContext, ***CryptDestroyKey***, ***CryptDestroyHash***

CryptSetProvider

Функция ***CryptSetProvider*** позволяет задать имя и тип криптопровайдера для последующего использования имени криптопровайдера по умолчанию для текущего пользователя.

После того как эта функция вызвана, все последующие вызовы данного пользователя функции ***CryptAcquireContext*** можно осуществлять с параметром *pszProvName*, равным NULL, и заданным типом *dwProvType* криптопровайдера. В результате в качестве имени криптопровайдера применяется имя, установленное функцией ***CryptSetProvider***.

Примечание

Обычные прикладные программы не должны использовать эту функцию. Она предназначена исключительно для административных программ.

```
BOOL WINAPI CryptSetProvider(
LPCTSTR pszProvName,
DWORD dwProvTipe
);
```

Параметры

pszProvName

[in] *Имя* нового криптопровайдера, используемого по умолчанию. Этот криптопровайдер должен быть уже установлен на компьютере.

dwProvTipe

[in] Тип провайдера, указанного параметром *pszProvName*.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки может быть получен через функцию ***GetLastError***:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* ERROR_NOT_ENOUGH_MEMORY

Коды ошибок могут также наследоваться от функций ***RegCreateKeyEx*** и ***RegSetValueEx***.

Комментарии

Обычно прикладные программы не определяют имя криптопровайдера при запросе функции ***CryptAcquireContext*** однако если есть необходимость, можно явно указать имя криптопровайдера, необходимого для работы приложения. Это даёт пользователям свободу в выборе криптопровайдера, обеспечивающего соответствующий уровень безопасности,

Запрос к ***CryptSetProvider*** определяет тип криптопровайдера, который будет использоваться всеми прикладными программами, запущенными после этого момента. По этой причине функция ***CryptSetProvider*** не должна вызываться без согласия пользователя.

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше,

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

Unicode: на всех платформах реализованы Unicode- и ANSI-версии.

См. также

CryptAcquireContext

CryptSetProviderEx

Функция ***CryptSetProviderEx*** позволяет задать имя и тип криптопровайдера для последующего использования имени криптопровайдера по умолчанию для текущего пользователя или компьютера.

Криптопровайдер, установленный по умолчанию для текущего пользователя, имеет приоритет перед криптопровайдером, установленным по умолчания для компьютера.

После того как эта функция вызвана, все последующие вызовы данного пользователя функции ***CryptAcquireContext*** можно осуществлять с параметром *pszProvName*, равным NULL, и заданным типом *dwProvType* криптопровайдера. В результате в качестве имени криптопровайдера применяется имя, установленное функцией ***CryptSetProviderEx***.

Если определены параметры для компьютера, пользовательский вызов ***CryptAcquireContext*** не имеет криптопровайдера по умолчанию и явно не определяет криптопровайдер, то используется криптопровайдер, определенный для компьютера.

Примечание

Обычные прикладные программы не должны использовать эту функцию. Она предназначена исключительно для административных программ.

```
BOOL WINAPI CryptSetProviderEx(
LPCTSTR pszProvName,
DWORD dwProvTipe,
DWORD *pdwReserved,
DWORD dwFlags
);
```

Параметры

pszProvName

[in] Имя нового криптопровайдера, используемого по умолчанию. Этот криптопровайдер должен быть уже установлен на компьютере.

dwProvTipe

[in] Тип провайдера, указанного параметром *pszProvName*.

pdwReserved

[in] Параметр зарезервирован и должен быть равен NULL.

dwFlags

[in] В настоящее время определены следующие значения флагов.

- CRYPT_MACHINE_DEFAULT

  Применяется для поиска криптопровайдера, используемого по умолчанию для компьютера.

- CRYPT_USER_DEFAULT

  Применяется для поиска криптопровайдера, используемого по умолчанию для текущего пользователя.

- CRYPT_DELETE_DEFAULT

  Применяется в комбинации с флагами CRYPT_MACHINE_DEFAULT или CRYPT_USER_DEFAULT для снятия текущего параметра.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE. соответствующий код ошибки можно получить через функцию ***GetLastError***:

- ERROR_INVALID_PARAMETER
- ERROR_NOT_ENOUGH_MEMORY

Комментарии

Обычно прикладные программы не определяют имя криптопровайдера при запросе функции ***CrypAcquire Context,*** однако, если есть необходимость, можно явно указать имя криптопровайдера, необходимого для работы приложения. Это даёт пользователям свободу в выборе криптопровайдера, обеспечивающего соответствующий уровень безопасности.

Запрос к ***CryptSetProviderE***определяет тип криптопровайдера, который будет использоваться всеми прикладными программами, запущенными с этого момента. По этой причине функцию ***CryptSetProviderEx*** не стоит вызывать без согласия пользователя.

Дополнительная информация

Windows NT/2000: необходима Windows 2000.

Windows 95/98: необходима Windows 98.

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

Unicode: на всех платформах реализованы Unicode- и ANSI-версии.

См. также

CryptAcquireContext, CryptSetProvider

CryptSetProvParam

Функция ***CryptSetProvParam*** задает параметры криптопровайдера. Ее обычно применяют для установки дескриптора безопасности по ключевой контейнер, что позволяет контролировать доступ к секретным ключам в контейнере.

```
BOOL WINAPI CryptSetProvParam(
HCRYPTPROV hProv,
DWORD dwParam,
BYTE *pbData,
DWORD dwFlags
);
```

Параметры

hProv

[in] Дескриптор криптопровайдера. Данный дескриптор должен быть предварительно получен через вызов функции ***CryptAcquire Context.***

dwParam

[in] Значение параметра определяет тип задаваемой величины. В настоящее время определены следующие значения параметра *dwParam.*

* PP_CLIENT_HWND

  Указывает, что в буфере *pbData* содержится дескриптор окна.

* PP_CONTEXT_INFO

  Указывает, что в буфере *pbData* находится структура DATA_BLOB, содержащая пользовательский контекст. Параметры этой структуры передаются при вызове криптопровайдера функции ***CPAcquireContext*** Подробнее параметры функции ***CPAcquireContext*** обсуждаются в главе 3.

* PP_KEYSET_SEC_DESCR

  Указывает, что в буфере *pbData* содержится дескриптор безопасности для ключевого контейнера. Не поддерживается операционной системой Windows 95.

* PP_USE_HARDWARE_RNG

  Указывает, что криптопровайдер должен использовать исключительно аппаратный ДСЧ.

  Некоторые криптопровайдеры могут применять аппаратный ДСЧ, а также аппаратный и программный ДСЧ совместно. Когда устанавливается параметр PP_USE_HARDWARE_RNG, криптопровайдер должен брать случайные значения только с аппаратного ДСЧ.

  Если аппаратный ДСЧ поддерживается и криптопровайдер может работать в режиме эксклюзивного использования аппаратного ДСЧ, то функция возвращает TRUE, в противном случае — FALSE.

  Параметр *pbData* должен быть равен NULL и *dwFlags* - нулю.

pbData

[in] Буфер данных параметра. При обращении к функции он должен содержать данные, которые соответствуют типу параметра, помещённому в *dwParam*. Формат данных зависит от тина параметра.

dwFlags

[in] Если *dwParam* установлен в PP_KEYSET_SEC_DESCR, то *dwFlags* содержит битовые флаги SECURITY_INFORMATION, как определено в *Win32 Programmer's Reference*. Для Windows 2000 и старше операции установки/считывания дескриптора безопасности на ключевой контейнер выполняют функции ***SetFileSecurity*** и ***GetFileSecurity***. Для предыдущих версий используются функции ***RegSetKeySecurity*** и RegGetKeySecurity. Дополнительную информацию вы найдете в описании функции ***CryptGetProvParam***.

Если *dwParam* установлен в PP_USE_HARDWARE_RNG, то *dwFlags* должен быть равен нулю.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки устанавливается через функцию ***GetLastError***:

* ERROR_BUSY
* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER

* NTE_BAD_FLAGS
* NTE_BAD_TYPE
* NTE_BAD_UID
* NTE_FAIL

Комментарии

Если *dwParam* установлен в PP_CLIENT_HWND, то *pbData* содержит значение DWORD, являющееся дескриптором окна, которое криптопровайдер использует для создания интерактивного диалога с пользователем. Функцию ***CryptSetProvParam*** следует вызывать до вызова функции ***CryptAcquireContext,*** потому что многие криптопровайдеры активизируют пользовательский интерфейс в функции ***CryptAcquireContext.*** При вызове ***CryptSetProvParam*** параметр *hProv* устанавливается в нуль, в этом случае параметр PP_CLIENT_HWND устанавливается для всех криптографических контекстов, которые будут создаваться впоследствии в текущем процессе.

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старте.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См, также

CryptAcquireContext, CryptGetProvParam, CryptSetKeyParam

CryptDeriveKey

Функция ***CryptDeriveKey*** создает сессионные криптографические ключи из ключевого материала. Эта функция гарантирует, что при использовании тех же криптопровайдеров и алгоритмов все ключи будут идентичны. В качестве ключевого материала можно использовать пароль или любые другие пользовательские данные.

Эта функция подобна ***CryptGenKey*** однако здесь произведенные сессионные ключи получают из ключевого материала, а не берутся из ДСЧ. Кроме того, функцию ***CryptDeriveKey*** нельзя использовать только для выработки сессионных ключей. С ее помощью невозможно выработать ключевую пару (открытый/секретный ключи) ЭЦП или ключевого обмена.

Дескриптор сессионного ключа возвращается в параметре *phKey*. Этот дескриптор можно использовать по необходимости с любой из функций CryptoAPI, требующих дескриптор ключа.

```
BOOL WINAPI CryptDeriveKey(
HCRYPTPROV hProv,
ALG_ID Algid,
HCRYPTHASH hBaseData,
DWORD dwFlags,
HCRYPTKEY *phKey
);
```

Параметры

hProv

[in] Дескриптор криптопровайдера. Данный дескриптор следует предварительно получить через вызов функции ***CryptAcquireContext***.

Algid

[in] Идентификатор алгоритма шифрования, для которого необходимо произвести ключ. Различные криптопровайдеры поддерживают различные алгоритмы шифрования. Поддерживаемые в настоящее время алгоритмы см. в разделе «Типы криптопровайдеров» в Приложении 2.

hBaseData

[in] Дескриптор объекта хеш-функции, содержащего ключевой материал. Чтобы получить этот дескриптор, приложение сначала создает объект хеш-функции с помощью функции ***CryptCreateHash***, а затем добавляет ключевой материал к объекту хеш-функции через процедуру ***CryptHashData***.

dwFlags

[in] Флаги определяют тип создаваемого сессионного ключа.

Размер сессионного ключа задают при создании. Размер ключа в битах определяются в старших 16 битах этого параметра *dwFlags*. Например, если 128-битный RC4 сессионный ключ создается, то значение 0x00800000 комбинируется с другими значениями флагов с помощью операции поразрядного ИДИ.

Младшие 16 бит могут быть нулевыми либо принимать одно или белес следующих значений, скомбинированных операцией поразрядного ИЛИ.

* CRYPT_CREATE_SALT

  Обычно, когда создается сессионный ключ из ключевого материала объекта хеш-функции, возникает избыточная

информация. Например, если размер хеша 128 бит, а размер сессионного ключа 40 бит, то избыток составляет 88 бит.

Если установлен флаг CRYPT_CREATE_SALT то модификатор ключа создается на основе избыточных битов. Пользователь может получить модификатор ключа, используя функцию ***CryptGetKeyParam*** с *dwParam*, установленным в KP_SALT.

Если этот флаг не установлен, то модификатор ключа задают равным нулю.

Когда ключи с ненулевым модификатором экспортируются (функция ***CryptExportKey***) модификатор ключа необходимо получить и сохранить с ключевым блобом.

* CRYPT_EXPORTABLE

Если этот флаг установлен, то сессионный ключ можно передать из криптопровайдера в ключевой блоб с помощью функции ***CryptExportKey***. Поскольку сессионные ключи, как правило, должны быть экспортируемыми, то для них этот флаг обычно устанавливают.

Если этот флаг не установлен, то ключ будет доступен только в текущей сессии и только приложению, которое создало этот ключ.

* CRYPT_NO_SALT

Если флаг установлен, то для ключей размером 40 бит модификатор не создается.

* CRYPT_UPDATE_KEY

Некоторые криптопровайдеры используют сессионные ключи, полученные из нескольких объектов хеш-функций. В этом случае функция ***CryptDeriveKey*** должна вызываться несколько раз.

Если данный флаг установлен, новый сессионный ключ не создается. Вместо этого ключ, определенный указателем *phKey*, модифицируется. Точные действия, устанавливаемые флагом CRYPT_UPDATE_KEY, зависят от конкретного криптопровайдера.

Криптопровайдеры Microsoft игнорируют этот флаг.

phKey

[in/out] Адрес, по которому функция копирует дескриптор созданного ключа.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию *GetLastError*:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_ALGID
* NTE_BAD_FLAGS
* NTE_BAD_HASH
* NTE_BAD_HASH_STATE
* NTE_BAD_UID
* NTE_NO_MEMORY
* NTE_FAIL

Комментарии

По умолчанию функцией *CryptDeriveKey* вырабатывается ключ в режиме простой замены с зацеплением (CBC) с нулевым начальным вектором. Для смены этой установки используется функция *CryptSetKeyParam*.

После вызова функция *CryptDeriveKey* новые данные нельзя добавить к объекту хеш-функции. Вызовы функций *CryptHashData* и *CryptHashSessionKey* вернут FALSE. Функцию *CryptDestroyHash* необходимо вызвать для уничтожения объектахеш-функции.

Для определения длины ключа, рекомендуется следующая последовательность действий:

* перечислите алгоритмы, которые поддерживает криптопровайдер, и определите минимальную и максимальную длину ключа для каждого алгоритма. Если используется Internet Explorer 4.0 или старше либо Windows NT 4.0 (Service Pack 4) или старше, можно применить функцию *CryptGetProvParam* с параметром PP_ENUMALGS_EX;
* используйте полученные значения для определения рабочей длины ключа. Не всегда стоит задавать максимальную длину, поскольку это влияет на скорость работы;
* когда рабочая длина ключа выбрана, используйте старшие 16 бит параметра *dwFlags* для ее установки.

В таблице П-2 содержится список криптопровайдеров Microsoft и поддерживаемых ими алгоритмов шифрования.

Таблица П-2. Криптопровайдеры Microsoft и поддерживаемые ими алгоритмы шифрования

Криптопровайдер	Алгоритмы	Минимальная длина ключа, бит	Длина ключа по умолчанию, бит	Максимальная длина ключа, бит
MS Base	RC4 и RC2	40	40	56
MS Base	DES	56	56	56
MS Enhanced	RC4 и RC2	40	128	128
MS Enhanced	DES	56	56	56
MS Enhanced	3DES 112	112	112	112
MS Enhanced	3DES	168	168	168
MS Strong	RC4 и RC2	40	128	128
MS Strong	DES	56	56	56
MS Strong	3DES 112	112	112	112
MS Strong	3DES	168	168	168
DSS/DH Base	RC4 and RC2	40	40	56
DSS/DH Base	Cylink MEK	40	40	40
DSS/DH Base	DES	56	56	56
DSS/DH Enh	RC4 и RC2	40	128	128
DSS/DH Enh	Cylink MEK	40	40	40
DSS/DH Enh	DES	56	56	56
DSS/DH Enh	3DES 112	112	112	112
DSS/DH Enh	3DES	168	168	168

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h,

Библиотека: advapi32.dll

См. также

CryptAcquireContext, Crypt Create Hash, CryptDestroyHash, CryptDestroyKey, CryptExportKey, CryptGenKey, CryptGetKeyParam, CryptHashData, CryptSetKeyParam

CryptDestroyKey

Функция ***CryptDestroyKey*** освобождает дескриптор, на который указывает параметр *hKey*. После того как дескриптор ключа освобождён, его нельзя использовать вновь.

Если дескриптор относится к сессионному ключу или открытому ключу, который был импортирован в криптопровайдер через функцию ***CryptImportKey***то ключ уничтожается и память, которую он занимал, функция освобождает.

Если дескриптор относится к паре открытый/секретный ключ, то происходит только освобождение дескриптора, при этом ключевая пара не уничтожается.

```
BOOL WINAPI CryptDestroyKey(
HCRYPTKEY hKey
);
```

Параметры

hKey

[in] Дескриптор удаляемого ключа.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_BUSY (ключ, определенный параметром *hKey,* сейчас используется и не может быть уничтожен)
* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_KEY
* NTE_BAD_UID

Комментарии

Прикладные программы к целях экономии ресурсов при завершении работы с ключами должны уничтожать их с помощью функции ***CryptDestroyKey***.Это особенно важно при работе с криптопровайдерами с аппаратной реализацией, поскольку ключевая память в них иногда довольно ограничена.

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше),

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptDeriveKey, CryptGenKey, CryptGetUserKey, CryptImportKey

CryptDuplicateKey

Функция ***CryptDuplicateKe***делает точную копию ключа, его параметров и внутреннего состояния.

```
BOOL WINAPI CryptDuplicateKey(
HCRYPTKEY hKey,
DWORD *pdwReserved,
DWORD dwFlags,
HCRYPTKEY *phKey
);
```

Параметры

hKey

[in] Дескриптор дублируемого ключа.

pdwReserved

[in] Параметр зарезервирован для будущего использования и должен быть равен NULL.

dwFlags

[in] Значения флагов. Этот параметр зарезервирован для будущего использования и должен быть равен пулю.

phKey

[in/out] Указатель на дескриптор дублированного ключа.

возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE, Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_CALL_NOT_IMPLEMENTED
* ERROR_INVALID_PARAMETER
* NTE_BAD_KEY

Комментарии

Функция ***CryptDuplicateKey*** делает точную копию ключа, его параметров и внутреннего состояния. Эта функция используется приложениями, которые шифруют два различных блока данных па одном ключе, но с различными параметрами. Такие приложения генерируют ключ, дублируют его функцией *Crypt-*

DuplicateKey и выставляют соответствующие параметры функцией ***CryptSetKeyParam.***

Для уничтожения ключа, созданного функцией ***CryptDuplicateKey***, необходимо использовать функцию ***CryptDestroyKey***. Уничтожение исходного ключа не влечет за собой уничтожения дублированного. Созданный функцией ***CryptDuplicateKey*** ключ не содержит в себе параметров и внутренних состояний, разделяемых с исходным ключом.

Дополнительная информация

Windows NT/2000: необходима Windows 2000.

Windows 95/98: необходима Windows 98.

Заголовок: прототип и файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptDestroyKey, CryptSetKeyParam

CryptExportKey

Функция ***CryptExportKey*** используется для экспорта криптографических ключей и ключевых пар из ключевого контейнера криптопровайдера, сохраняя их в защищенном виде.

На вход функции передаётся дескриптор экспортируемого ключа, функция возвращает ключевой блоб, который передастся по открытым каналам связи или записывается на незащищённый носитель. Данную функцию применяют для экспорта Schannel сессионных ключей, обычных сессионных ключей, открытых ключей или ключевых пар. Использование ключа из ключевого блоба возможно только после того, как получатель ключа импортирует его в свой криптопровайдер, используя функцию ***CryptImportKey***.

```
BOOL WINAPI CryptExportKey(
HCRYPTKEY hKey,
HCRYPTKEY hExpKey,
DWORD dwBlobType,
DWORD dwFlags,
BYTE *pbData,
DWORD *pbDataLen
);
```

Параметры

hKey

[in] Дескриптор экспортируемого ключа.

hExpKey

[in] Дескриптор ключа получателя ключевого блоба. Данные внутри экспортируемого ключевого блоба шифруются с использованием этого ключа. Это гарантирует, что только получатель ключевого блоба сможет расшифровать и использовать экспортируемый ключ.

Чаще всего это открытый ключ алгоритма ключевого обмена, принадлежащий получателю ключевого блоба. Однако определенные протоколы в некоторых криптопровайдерах требуют, чтобы это был сессионный ключ, принадлежащий получателю ключевого блоба.

В Windows 2000 поддерживается экспорт на симметричном ключе (Symmetric Key Wrap), однако эта возможность не поддерживается на более ранних платформах ни криптопровайдером Microsoft Base Cryptographic Provider, ни Microsoft Enhanced Cryptographic Provider.

Если тип ключевого блоба, определяемый параметром *dwBlobType*, установлен в PUBLICKEYBLOB, то этот параметр не используется и должен быть равен нулю.

Если тип ключевого блоба, определяемый параметром *dwBlobType*, установлен в PRIVATEKEYBLOB, то обычно этот параметр является дескриптором сессионного ключа, используемого для шифрования ключевого блоба. Некоторые криптопровайдеры позволяют устанавливать этот параметр в нуль, в этом случае приложение должно самостоятельно зашифровать блоб секретного ключа.

dwBlobType

[in] Тип ключевого блоба, предназначенного для экспорта ключа. В настоящее время определены следующие типы ключевых блобов.

* OPAQUEKEYBLOB

  Используется для сохранения сессионных ключей в Schannel-криптопровайдерах. OPAQUEKEYBLOB не транспортируется и должен использоваться внутри криптопровайдера, создавшего ключевой блоб.

* PRIVATEKEYBLOB

  Используется для транспортировки пары открытый/секретный ключи.

* SIMPLEBLOB

  Используется для экспорта сессионных ключей.

* PUBLICKEYBLOB

  Используется для экспорта открытых ключей ключевых пар подписи или ключевого обмена.

* PUBLICKEYBLOBEX

  Используется для обмена значениями P, G, и (G^X) mod P при обмене ключами по алгоритму Диффи — Хеллмана. Данный тип ключевого блоба в Windows 2000 не применяется и не поддерживается.

* SYMMETRICWRAPKEYBLOB

  Используется для экспорта симметричного ключа на другом симметричном ключе. Формат экспорта описан в стандарте IETF SMIME X9.42.

dwFlags

[in] В настоящее время определены следующие значения флагов.

* CRYPT_DESTROYKEY

  При установке этого флага исходный ключ в OPAQUEKEYBLOB уничтожается. Флаг доступен только в Schannel-криптопровайдерах.

» CRYPT_SSL2_FALLBACK

  При установке этого флага первые 8 байт дополнения в блоке, зашифрованном по алгоритму RSA. заполнены значением 0x03, а не случайной последовательностью. Это предотвращает rollback-атаку. Подробнее об этом — в спецификации на SSL3. Флаг доступен только в Schannel-криптопровайдерах.

* CRYPT_OAEP

  При установке этого флага зашифрование по алгоритму RSA при экспорте SIMPLEBLOB происходит с применением схемы RSAES-OAEP, предусмотренной в стандарте PKCS # 1 версия 2.0. По умолчанию зашифрование по алгоритму RSA происходит с применением схемы RSAES-PKCSl-vl_5.

pbData

[out] Буфер данных, куда функция копирует ключевой блоб. Формат ключевого блоба зависит типа, определяемого параметром *dwBlobType*.

Если параметр равен NULL, то данные не копируются. Требуемый размер буфера (в байтах) должен возвращается в *pdwDataLen*. См. раздел «Получение данных неизвестного размера» в Приложении 2.

pbDataLen

[in/out] Адрес размера ключевого блоба. При вызове функции указанный параметр содержит число байтов в буфере *pbData*. После ее исполнения параметр задается числом байт данных параметра, скопированных в буфер *pbData*.

Примечание

Когда обрабатываются возвращаемые в буфере данные, приложение должно использовать фактический размер данных. Он может быть немного меньше, чем размер буфера, указанный на входе. На входе размер буфера обычно указывают таким, чтобы буфера хватало для получения максимально возможного объема данных. На выходе переменная, на которую указывает данный параметр, содержит фактический размер данных, скопированных в буфер.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию *GetLastError*:

* ERROR_INVALID_PARAMETER
* ERROR_MORE_DATA
* NTE_BAD_FLAGS
* NTE_BAD_KEY
* NTE_BAD_KEY_STATE
* NTE_BAD_PUBLIC_KEY
* NTE_NO_KEY (экспортируется сессионный ключ, а параметр *hExpKey* не определяет дескриптор открытого ключа)
* NTE_BAD_TYPE
* NTE_PERM
* NTE_BAD_UID
* NTE_NO_MEMORY

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptImportKey

CryptGenKey

Функция *CryptGenKey* генерирует случайные сессионные ключи и ключевые пары. Дескриптор созданного ключа возвращается через параметр *phKey*. Этот дескриптор можно затем использовать по необходимости с любой из функций CryptoAPI, требующих дескриптор ключа на входе.

При вызове этой функции приложение должно определить алгоритм криптографических преобразований. Криптопровайдер сохраняет алгоритм как один из параметров ключа, поэтому приложению нет необходимости обращаться к этому параметру позже, при выполнении криптографических функций.

```
BOOL WINAPI CryptGenKey(
HCRYPTPROV hProv,
ALG_ID Algid,
DWORD dwFlags,
HCRYPTKEY *phKey
);
```

Параметры

hProv

[in] Дескриптор криптопровайдера. Его необходимо предварительно получить через вызов функции *CryptAcquireContext*.

Algid

[in] Идентификатор алгоритма шифрования, ЭЦП или ключевого обмена, для которого следует создать ключ. Значение этого параметра зависит от используемого криптопровайдера.

Для Microsoft Base Cryptographic Provider определены следующие значения идентификаторов алгоритмов.

* CALG_RC2

  Алгоритм блочного шифрования RC2.

* CALG_RC4

  Алгоритм поточного шифрования RC4.

Для криптопровайдеров, поддерживающих ключевой обмен по алгоритму Диффи — Хеллмана, определены следующие значения идентификаторов алгоритмов.

* CALG_DH_EPHEM

  Определяет «Ephemeral» ключ Диффи — Хеллмана.

* CALG_DH_SF

  Определяет «Store and Forward» ключ Диффи — Хеллмана.

Вместо идентификаторов алгоритмов для генерации ключевых пар приложение может использовать следующие параметры:

* AT_KEYEXCHANGE

  Определяет ключевую пару обмена.

* AT_SIGNATURE

  Определяет ключевую пару подписи.

  При использовании параметров AT_KEYEXCHANGE и AT_SIGNATURE идентификаторы криптографических алгоритмов, которые будут применяться при создании ключа, зависят от криптопровайдера. Для определения идентификатора алгоритма используйте функцию ***CryptGetKeyParam*** с параметром KP_ALGID.

dwFlags

[in] Флаги определяют тин создаваемого сессионного ключа.

Размер криптографического ключа задают при создании. Размер ключа в битах определяется в старших 16 битах этого параметра *dw Flags*. Например, если 2048-битный RSA ключ ЭЦП создается, то значение 0x08000000 комбинируется с другими значениями флагов с помощью операции поразрядного ИЛИ.

Если старшие 16 бит нулевые, то используется длина ключа по умолчанию. Если запрашиваемая длина ключа меньше минимально поддерживаемой или больше максимально поддерживаемой криптопровайдером, то функция вернет FALSE с кодом ошибки ERROR_INVALID_PARAMETER.

Младшие 16 бит могут быть нулевыми либо принимать одно или более значений, скомбинированных операцией поразрядного ИЛИ (они перечислены далее).

» CRYPT_CREATE_SALT

  Если этот флаг установлен, то модификатор ключа берется из ДСЧ. Приложение может получить модификатор ключа, вызвав функцию ***CryptGetKeyParam*** с параметром KP_SALT.

  Если флаг не установлен, то задают нулевой модификатор ключа.

  Когда ключи с ненулевым модификатором экспортируются (функция ***CryptExportKey***)модификатор ключа необходимо получить и сохранить с ключевым блобом.

* CRYPT_EXPORTABLE

  Если этот флаг установлен, то сессионный ключ можно передать из криптопровайдера в ключевой блоб через функцию ***CryptExportKey***Поскольку ключи, как правило, должны быть экспортируемыми, этот флаг следует установить.

  Если этот флаг не установлен, то сессионный ключ нельзя экспортировать. Это означает, что он доступен только в пределах текущей сессии и только приложению, которое создало этот ключ.

  Для ключевых пар это означает, что секретный ключ нельзя экспортировать из криптопровайдера.

  Действие этого флага не распространяются на открытые ключи ключевых пар, они всегда экспортируемые.

* CRYPT_NO_SALT

  Если флаг установлен, то для ключей размером 40 бит модификатор не создается.

* CRYPT_PREGEN

  Этот флаг определяет инициализацию генерации ключевых пар Диффи — Хеллмана и DSS. Флаг используется только с криптопровайдерам и, поддерживающими алгоритмы Диффи — Хеллман/DSS.

  Параметры, используемые при дальнейших вызовах функции ***CryptSetKeyParam***должны соответствовать длине ключа, установленной при инициализации генерации.

* CRYPT_USER_PROTECTED

  Если этот флаг установлен, то пользователь извещается через диалог или другим способом, когда приложение пытается использовать данный ключ. Точные действия по этому флагу зависят от конкретного криптопровайдера.

  Данный флаг поддерживается, начиная с Internet Explorer 4.0, а также в Windows 98, Windows NT 4.0 (Service Pack 4) и Windows 2000.

  Если контекст криптопровайдера открыт с флагом CRYPT_SILENT, то в результате использования флага CRYPT_USER_PROTECTED функция вернет FALSE с кодом ошибки NTE_SILENT_CONTEXT.

phKey

[out] Адрес, по которому функция копирует дескриптор произведённого ключа.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию *GetLastError*:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_ALGID
* NTE_BAD_FLAGS
* NTE_BAD_UID
* NTE_NO_MEMORY
* NTE_FAIL
* NTE_SILENT_CONTEXT

Комментарии

Если порождаются ключи для симметричных блочных шифров, по умолчанию ключ устанавливается в шифратор в режиме простой замены с зацеплением (Cipher Block Chaining Mode CBC) (см. таблицу П-4) с нулевым начальным вектором. Данный параметр можно изменить, используя функцию *CryptSetKeyParam*.

Для определения длины ключа рекомендуется следующая последовательность действий:

* перечислите алгоритмы, которые поддерживает криптопровайдер, и определите минимальную и максимальную длину ключа для каждого алгоритма. Если используется Internet Explorer 4.0 или старше либо Windows NT 4.0 (Service Pack 4) или старше, можно применить функцию *CryptGetProvParam* с параметром PP_ENUMALGS_EX;
* используйте полученные значения для определения рабочей длины ключа. Но всегда имеет смысл задавать максимальную длину, поскольку это влияет на скорость работы;
* когда рабочая длина ключа выбрана, используйте старшие 16 бит параметра *dwFlags* для ее установки.

В таблице II-3 перечислены возможные алгоритмы и их параметры. Для сессионных ключей данные приведены в таблице П-2 в описании функции *CryptDeriveKey*.

Таблица П-3. Алгоритмы и их параметры

Тип ключа и криптопровайдер	Минимальная длина ключа	Длина ключа по умолчанию	Максимальная длина ключа
Ключи ЭЦП / RSA Base и Strong providers	384	512	16000
Ключи ЭЦП / RSA Enhanced provider	384	1024	16000
Ключи обмена / Base и Strong providers	384	512	1024
Ключи обмена / Enhanced provider	384	1024	16000

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. *также*

CryptAcquireContext, CryptDestroyKey, CryptExportKey, CryptGetKeyParam, CryptImportKey, CryptSetKeyParam

CryptGenRandom

Функция ***CryptGenRandom*** **вырабатывает** случайную последовательность и сохраняет ее в буфер *pbBuffer*.

```
BOOL WINAPI CryptGenRandom(
HCRYPTPROV hProv,
DWORD dwLen,
BYTE *pbBuffer
);
```

Параметры

hProv

[in] Дескриптор криптопровайдера. Данный дескриптор должен быть предварительно получен через вызов функции ***CryptAcquire Context.***

dwLen

[in] Длина случайной последовательности в байтах.

pbBuffer

[in/out] Буфер, куда копируются случайная последовательность. Длина этого буфера должна быть не меньше, чем определено в параметре *dwLen*.

Приложение может заранее поместить в буфер данные, которые *криптопровайдер использует* как начальное заполнение для ДСЧ.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_UID
* NTE_FAIL

Комментарии

Последовательность, вырабатываемая данной функцией, является криптографически случайной. Требования к случайным данным, используемым в криптографических преобразованиях, значительно строже, чем, например, к датчикам из библиотеки С-компилятора.

Эту функцию часто применяют для генерации синхропосылки и модификатора ключа.

Все программные ДСЧ по принципу работы аналогичны. ДСЧ инициализируется случайной последовательностью, называемой *начальным заполнением* (seed), и, используя определенный алгоритм генерации псевдо-случайных чисел, генерируется последовательность. Наиболее сложная часть этого процесса — получение начального заполнения, которое можно считать случайным. Обычно методы получения начального заполнения базируются на пользовательском вводе данных с обработкой случайных задержек при работе пользователя или на *дрожании* (jitter) каких-либо аппаратных данных.

Если приложение имеет доступ к надежному источнику случайных данных, оно может заполнить буфер *pbBuffer* до вызова функции ***CryptGenRandom***. Криптопровайдер использует эту последовательность для рандомизации внутреннего состояния ДСЧ,

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptAcquire Context, CryptGenKey, CryptSetKeyParam

CryptGetKeyParam

Функция *CryptGetKeyParam* возвращает параметры ключа, определяемого дескриптором *hKey*.

```
BOOL WINAPI CryptGetKeyParam(
HCRYPTKEY hKey,
DWORD dwParam,
BYTE *pbData,
DWORD *pdwDataLen,
DWORD dwFlags
);
```

Параметры

hKey

Дескриптор ключа, параметры которого запрашиваются.

dwParam

Значение параметра определяет тип запроса.

Для всех типов ключей величину *dwParam* можно установить в одно из следующих значений.

* KP_ALGID

  Алгоритм ключа. Идентификатор алгоритма (ALG_ID), соответствующий данному ключу, будет возвращён через буфер *pbData*. Значение 4 возвращается в *pdwDataLen*.

* KP_BLOCKLEN

  Если *hKey* указывает на сессионный ключ, величина DWORD. указывающая длину блока шифра в битах, возвращается через буфер *pbData*. Для поточных шифров эта величина всегда нулевая. Значение 4 возвращается в *pdwDataLen*.

  Если *hKey* ссылается па ключевую пару, через *pbData* возвращается блоб, содержащий гранулярность ключевой пары в битах. Например, Microsoft Base Cryptographic Provider

генерирует 512-битный RSA ключ, и функция вернет значение 512. Если алгоритм с открытым ключом не поддерживает шифрование, возвращается код ошибки NTE_BAD_TYPE.

* KP_KEYLEN

  Значение используется для получения длины ключа в битах. Величина DWORD, содержащая длину ключа, возвращается в буфере *pbData*.

* KP_SALT

  В *pbData* копируется модификатор ключа. Размер модификатора ключа зависит от криптопровайдера и используемого алгоритма шифрования. Ключевые пары модификатора ключа не содержат.

* KP_PERMISSIONS

  В *pbData* копируется значение DWORD, содержащее битовые флаги, которые определяют ограничения на использование ключа. Подробнее — в разделе «Комментарии».

  Если *hKey* является ключем DSS или Диффи — Хеллмана, то *dwParam* можно установить также в следующие значения.

* KP_P

  Буфер *pbData* содержит значение модуля P. Длина данных возвращается в *pdwDataLen*.

* KP_Q

  Буфер *pbData* содержит значение параметра Q. Длина данных возвращается в *pdwDataLen*.

* KP_G

  Буфер *pbData* содержит значение параметра G. Длина данных возвращается и *pdwDataLen*.

Если *hKey* параметр относится к сессионному ключу, величину *dwParam* можно установить в одно из следующих значений параметра.

* KP_EFFECTIVE_KEYLEN

  Запрашивает эффективную длину ключа RC2. *pbData* содержит значение DWORD, равное эффективной длине ключа.

* KP_IV

  Запрашивает значение синхропосылки. Последовательность байт, содержащая синхропосылку, возвращается через буфер *pbData*. Размер синхропосылки в байтах возвращается через *pdwDataLen*. Размер синхропосылки в байтах равен

размер блока в битах/8. Например, если размер блока 64 бита, то синхропосылка составляет 8 байт.

* KP_PADDING

  Запрашивает метод дополнения. Величина DWORD, содержащая метод дополнения, который используется при шифровании на данном ключа, возвращается через буфер *pbData*. Значение 4 возвращается через *pdwDataLen*.

  В настоящее время определен метод дополнения по алгоритму PKCS 5: PKCS5_PADDING — PKCS 5.

* KP_MODE

  Запрашивает режим шифрования. Величина DWORD, содержащая режим, возвращается через буфер *pbData*. Значение 4 возвращается через *pdwDataLen*. Список режимов шифрования приведен в разделе «Комментарии».

* KP_MODE_BITS

  Запрашивает число бит обратной связи. Величина DWORD, содержащая величину обратной связи в битах, возвращается через буфер *pbData*. Значение 4 возвращается через *pdwDataLen*. Этот параметр применяется, только когда используются режимы шифра режим гаммирования (Output Feedback Mode, OFB) или режим гаммирования с обратной связью (Cipher Feedback Mode, CFB) (таблица П-4).

pbData

Буфер данных параметра. Функция копирует соответствующие параметру данные в буфер. Формат этих данных зависит от значения *dwParam*.

Если параметр — NULL, то данные не копируются. Требуемый размер буфера (Б байтах) возвращается в *pdwDataLen*. См. раздел «Получение данных неизвестного размера» в конце Приложения 2.

pdwDataLen

Адрес размера данных параметра. При вызове функции он показывает число байт в буфере *pbData*. После её исполнения параметр задастся числом байт данных параметра, сконированных в буфер *pbData*.

Примечание

Когда обрабатываются возвращаемые в буфере данные, приложение должно использовать фактический размер данных. Он немного меньше, чем размер буфера, указанный на входе. На

входе размер буфера обычно задают таким, чтобы буфера хватало для получения максимально возможного объема данных. На выходе переменная, на которую указывает данный параметр, показывает фактический размер данных, скопированных в буфер.

dwFlags

Значения флагов. Этот параметр зарезервирован для будущего использования и должен быть нулевым.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию *GetLastError*:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* ERROR_MORE_DATA
* NTE_BAD_FLAGS
* NTE_BAD_KEY
* NTE_NO_KEY
* NTE_BAD_TYPE
* NTE_BAD._UID

Комментарии

В таблице II-4 перечислены режимы шифрования, определённые в настоящее время.

Таблица П-4. Режимы шифрования

Режим шифрования	Описание
CRYPT_MODE_CBC	Шифр простой замены с зацеплением (Cipher Block Chaining Mode, CBC).
CRYPT_MODE_CFB	Режим гаммирования с обратной связью (Cipher Feedback Mode, CFB).
CRYPT_MODE_ECB	Шифр простой замены (Electronic Codebook Mode, ECB).
CRYPT_MODE_OFB	Режим гаммирования (Output Feedback Mode, OFB).

В таблице П-5 перечислены битовые флаги, которые функция возвращает при запросе параметра KP_PERMISSIONS. Флаги могут быть скомбинированы операцией поразрядного ИЛИ. Эти флаги игнорируются в криптопровайдерах Microsoft в Windows 95/98 и Windows NT 4.0. Windows 2000 используют только флаг CRYPT_EXPORT. Однако другие криптопровай-

деры могут использовать данные флаги для установки ограничений на использование ключа.

Таблица П-5. Битовые флаги ограничений на использование ключа

Флаг	Значение
CRYPT_DECRYPT	Разрешено зашифрование.
CRYPT_ENCRYPT	Разрешено расшифрование.
CRYPT_EXPORT	Разрешен экспорт ключа.
CRYPT_MAC	Разрешено использовать ключ для создания объекта хеш-функции.
CRYPT_READ	Разрешено чтение значения ключа.
CRYPT_WRITE	Разрешено устанавливать значение ключа.

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptSetKeyParam

CryptGetUserKey

Функция ***CryptGetUserKey*** возвращает дескриптор одной из постоянных ключевых пар. Эта функция используется только владельцем ключевых пар и только когда получен дескриптор соответствующего ключевого контейнера.

```
BOOL WINAPI CryptGetUserKey(
HCRYPTPROV hProv,
DWORD dwKeySpec,
HCRYPTKEY *phUserKey
);
```

Параметры

hProv

[in] Дескриптор криптопровайдера. Данный дескриптор должен быть предварительно получен через вызов функции ***CryptAcquireContext***.

dwKeySpec

[in] Идентификатор запрашиваемого секретного ключа. Параметр может принимать значения AT_SIGNATURE или AT_KEYEXCHANGE.

Дополнительно некоторые криптопровайдеры реализуют через эту функцию доступ к другим пользовательским ключам. Подробнее об этом — и документации на соответствующий криптопровайдер.

ph UseKey

[out] Адрес, по которому функция копирует дескриптор ключа.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***.

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_UID
* NTE_BAD_KEY
* NTE_BAD_TYPE

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле *wincrypt.h*.

Библиотека: advapi32.dll

См. *также*

CryptAcquire Context, CryptDestroyKey, CryptGenKey

CryptImportKey

Функция ***CryptImportKey*** используется для импорта криптографического ключа из ключевого блоба в контейнер криптопровайдера, а также для импорта Schannel сессионных ключей, обычных сессионных ключей, открытых ключей и ключевых пар. Во всех случаях, кроме случая открытых ключей, информация в ключевом блобе зашифрована.

```
BOOL WINAPI CryptImportKey(
HCRYPTPROV hProv,
CONST BYTE *pbData,
DWORD dwDataLen,
HCRYPTKEY hImpKey,
```

```
DWORD dwFlags,
HCRYPTKEY *phKey
);
```

Параметры

hProv

[in] Дескриптор криптопровайдера. Его следует предварительно получить через вызов функции ***CryptAcquireContext.***

pbData

[in) Буфер, содержащий ключевой блоб. Он экспортирован функцией ***CryptExportKey*** из данного приложения или другого приложения, возможно запущенного на удаленном компьютере.

Ключевой блоб состоит из стандартного заголовка и зашифрованного ключа.

dwDataLen

[in] Длина ключевого блоба в байтах.

hImpKey

[in] Значение этого параметра зависит от используемого криптопровайдера и типа импортируемого ключевого блоба.

Если ключевой блоб зашифрован на открытом ключе обмена, например SIMPLEBLOB, этот параметр может быть дескриптором секретного ключа обмена.

Если ключевой блоб зашифрован на сессионном ключе, например PRIVATEKEYBLOB, этот параметр содержит дескриптор сессионного ключа.

Если ключевой блоб не зашифрован, например PUBLICKEYBLOB. этот параметр не используется и должен быть нулевым.

Если ключевой блоб зашифрован на сессионном ключе в Schannel криптопровайдере, например OPAQUEKEYBLOB, этот параметр не используется и должен быть нулевым.

dwFlags

[in] Значение флага. В настоящее время определены следующие значения флагов.

* CRYPT_EXPORTABLE

  Импортированный ключ можно снова экспортировать. Если этот флаг не используется, то вызов функции ***CryptExportKey*** вернет FALSE.

+ CRYPT_OAEP

  Флаг задает использование при импорте SIMPLEBLOB расшифрование по алгоритму RSA с применением схемы RSAES-OAEP, предусмотренной а стандарте PKCS #1 версии 2.0. По умолчанию расшифрование по алгориму RSA происходит с применением схемы RSAES-PKCS1-v1_5.

* CRYPT_NO_SALT

  Для сессионных ключей длиной 40 бит модификатор не создается.

phKey

[out] Адрес, по которому функция копирует дескриптор импортированного ключа.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию *GetLastError*:

* ERROR_BUSY
* ERROR_INVALID_HANDLE
+ ERROR_INVALID_PARAMETER
* NTE_BAD_DATA
* NTE_BAD_FLAGS
* NTE_BAD_TYPE
* NTE_BAD_UID
* NTE_NO_MEMORY

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше,

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип и файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptAcquireContext, CryptDestroyKey, CryptExportKey

CryptSetKeyParam

Функция *CryptSetKeyParam* устанавливает параметры ключа. Действие установленных параметров распространяются только на одну сессию.

Microsoft Base Cryptographic Provider не позволяет задавать параметры на ключи ЭЦП и ключевого обмена. Однако, другие криптопровайдеры, например Microsoft Base DSS and Diffie-Hellman Cryptographic Provider, используют эту функцию с целью определения параметров на несимметричные ключи,

```
BOOL WINAPI CryptSetKeyParam(
HCRYPTKEY hKey,
DWORD dwParam,
BYTE *pbData,
DWORD dwFlags
);
```

Параметры

hKey

[in] Дескриптор ключа, параметры которого устанавливаются.

dwParam

[in] Значение параметра.

Для всех типов ключей величину *dwParam* можно установить в одно из следующих значений.

- KP_SALT

Параметр *pbData* указывает на массив, содержащий новый модификатор ключа. Размер модификатора ключа зависит от используемого криптопровайдера. Перед тем как устанавливать этот параметр, приложение может определить необходимый размер, вызвав функцию ***CryptGetKeyParam.*** В криптопровайдере Microsoft Base Cryptographic Provider модификатор ключа по умолчанию является нулевым.

* KP_SALT_EX

  Параметр *pbData* указывает на структуру BLOB, в которой содержится новый модификатор ключа. Дополнительную информацию вы найдете в описании параметра KP_SALT.

* KP_PERMISSIONS

  Параметр *pbData* указывает на значение DWORD, содержащее битовые флаги, которые определяют ограничения на использования ключа. Подробнее об этом — в описании функции ***CryptGetKeyParam,***

* KP_ALGID

  Параметр *pbData* указывает па идентификатор алгоритма. Этот параметр используется при ключевом обмене в Microsoft Base DSS and Diffie-Hellman Cryptographic Provider или совместимых криптопровайдерах. Когда открытый ключ ключевой пары обмена импортируется в криптопровайдер функцией ***CryptImportKey*** то параметр KP_ALGID устанавливает алгоритм шифрования полученного сессионного ключа. Только после этого ключ готов к использованию.

Если параметр *hKey* относится к сессионному ключу, величину *dwParam* можно установить в одно из следующих значений.

* KP_EFFECTIVE_KEYLEN

  Этот параметр используется только с ключами RC2 и добавлен для совместимости с Microsoft Enhanced Cryptographic Provider Windows NT 4.0 Service Pack 2. В этой реализации длина ключей RC2 составляла 128 бит, но эффективная длина ключа — всего 40 бит. Для обратной совместимости это свойство сохранено. Однако эффективную длину ключа можно установить более 40 бит с помощью функции ***CryptSetKeyParam*** с параметром KP_EFFECTIVE_KEYLEN. Устанавливаемое значение передается в параметре *pbData* как указатель на значение DWORD. Минимальная эффективная длина ключа в криптопровайдере Microsoft Base Cryptographic Provider равна 1, а максимальная - 40 бит. В криптопровайдере Microsoft Enhanced Cryptographic Provider минимальная равна 1, а максимальная — 1024 бита. Эффективную длину следует установить до проведения операций зашифрования или расшифрования.

* KP_IV

  Параметр *pbData* указывает на синхропосылку. Размер синхропосылки в байтах равен [размер блока в битах/8]. Например, если размер блока 64 бита, то синхропосылка равна 8 байтам.

  По умолчанию в криптопровайдере Microsoft Base Cryptographic Provider синхропосылка равна нулю.

* KP_PADDING

  Параметр *pbData* указывает на значение DWORD, определяющее способ дополнения. В настоящее время определен способ дополнения PKCS5_PADDING.

* KP_MODE

  Параметр *pbData* указывает на значение DWORD, определяющее режим шифрования. Список возможных режимов — в описании функции ***CryptGetKeyParam***.

  По умолчанию в криптопровайдере Microsoft Base Cryptographic Provider режим шифрования устанавливается в CRYPT_MODE_CBC.

* KP_MODE_BITS

  Параметр *pbData* указывает на значение DWORD, опрсделяющее число бит обратной связи. Ом применяется, только когда используются режимы простой замены с зацеплением (OFB) или гаммирования с обратной связью (CFB) (таблица П-4).

  По умолчанию в криптопровайдере Microsoft Base Cryptographic Provider число бит обратной связи равно восьми.

Для ключей DSS и Диффи — Хеллмана определяются следующие параметры.

* KP_P

  Параметр *pbData* указывает на значение модуля P. Данные передаются в форме BLOB структуры, где член *pbData* указывает на буфер, а член *cbData* определяет размер. Данные передаются без заголовка, первым передается самый младший байт (формат «little-endian»).

* KP_Q

  Параметр *pbData* указывает на параметр Q. Данные передаются в форме BLOB структуры, где член *pbData* указывает на буфер, а член *cbData* определяет размер. Данные перелаются без заголовка, первым передается самый младший байт (формат «little-endian»)

* KP_G

  Параметр *pbData* указывает на значение генератора G. Данные передаются в форме BLOB структуры, где член *pbData* указывает на буфер, а член *cbData* определяет размер. Данные передаются без заголовка, первым передается самый младший байт (формат «little-endian»)

* KP_PUB_PARAMS

  Параметр *pbData* указывает на значение параметров P,Q,G. Данные передаются в форме BLOB структуры, где член *pbData* указывает на структуру DHPUBKEY_VER3 или

DSSPUBKEY_VER3 BLOB. Данные передаются без заголовка, первым передается самый младший байт (формат little endian). Дескриптор ключа, для которого устанавливаются эти параметры, следует получить с использованием флага CRYPT_PREGEN. После такого вызова приложение должно вызвать функцию ***CryptSetKeyParam*** с параметром KP_X для завершения генерации ключевой пары. Параметр KP_PUB_PARAMS позволяет установить все параметры с помощью одного вызова функции, а не трех с параметрами KP_P, KP_Q и KP_G.

pbData

[in] Буфер данных. При вызове функции он должен содержать данные, соответствующие значению параметра в *dwParam*. Формат данных зависит от типа параметра.

dwFlags

[in] Значения флагов. В настоящее время используется только с параметром KP_ALG1D. Параметр *dwFlags* можно использовать для установки значения длины ключа и других значений флагов, которые применяются при генерации того же типа ключа в функции ***CryptGenKey***. Описание возможных значений флагов вы найдете в описании функции ***CryptGenKey***.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

- ERROR_BUSY
- ERROR_INVALID_HANDLE
- ERROR_INVALID_PARAMETER
- NTE_BAD_FLAGS
- NTE_BAD_TYPE
- NTE_PERM
- NTE_FIXEDPARAMETER
- NT NTE_BAD_UID
- NTE_FAIL

Комментарии

Если параметры KP_Q, KP_P или KP_A устанавливают для ключа DSS или Диффи — Хеллмана, созданного с флагом

CRYPT_PREGEN, их длины должны быть совместимы с длиной ключа, указанной в функции ***CryptGenKey***.Если в старших 16 битах параметра *dwFlasg* функции ***CryptGenKey*** явно не указана длина ключа, то используется размер по умолчанию. В этом случае размеры параметров P, Q и A должны быть совместимы с длиной ключа по умолчанию.

Схема дополнения PKCS5_PADDING определена в PKCS #5. При ее использовании последовательность всегда дополняется, даже в случае, когда её длина кратна длине блока шифрования. Последовательность дополнения состоит из байт, каждый из которых ранен числу дополнительных байт. Если дополняется последовательность из 24 бит, последовательность дополнения равна «03 03 03». Если дополняется 64 бита, последовательность дополнения равна «08 08 08 08 08 08 08 08».

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98; необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См, *также*

CryptGenKey, CryptGetKeyParam

CryptDecrypt

Функция ***CryptDecrypt*** используется для расшифрования данных, предварительно зашифрованных функцией ***CryptEncrypt***. Одновременно возможно хеширование данных.

```
BOOL WINAPI CryptDecrypt(
HCRYPTKEY hKey,
HCRYPTHASH hHash,
BOOL Final,
DWORD dwFlags,
BYTE *pbData,
DWORD *pdwDataLen
);
```

Параметры

hKey

[in| Дескриптор сессионного ключа, применяемого для расшифрования данных. Приложение получает его, используя функцию ***CryptImportKey***или ***CryptGenKey,***

Этот ключ определяет используемый алгоритм шифрования.

hHash

[in] Дескриптор объекта хеш-функции. Если данные одновременно расшифровываются и хешируются, то дескриптор объекта хеширования передается в этом параметре. В процедуру хеширования данные передаются после расшифрования. Эту возможность удобно использовать, если данные одновременно расшифровываются и проверяется ЭЦП.

До того как вызвать функцию ***CryptDecrypt***приложение должно получить дескриптор объекта хеш-функции, вызвав функцию ***CryptCreateHash***После расшифрования значение хеша можно получить вызовом функции ***CryptGetHashParam***подписать функцией ***CryptSignHash***или передать для проверки ЭЦП в функцию ***CryptVerifySignature.***

Если значение хеша не вычисляется, этот параметр должен быть равен нулю.

Final

[in] Булева величина. Определяет, является ли переданный функции блок последним расшифрованным блоком данных. Она должна быть установлена TRUE, если это последний (или единственный) блок, и FALSE — в противном случае. См. раздел «Комментарии».

dwFlags

[in] Значения флагов. В настоящее время определено одно значение флагов.

* CRYPT_OAEP

  При установке этого флага расшифрование при использовании алгоритма RSA происходит с применением схемы RSAES-OAEP, предусмотренной в стандарте PKCS #1 версии 2.0. По умолчанию расшифрование по алгоритму RSA выполняется с применением схемы RSAES-PKCS1-v1_5.

  Флаг используется с криптопровайдером Microsoft Enhanced Cryptographic Provider, начиная с Windows 2000.

pbData

[in/out] Буфер, содержащий данные для расшифрования. После расшифрования открытый текст помещается в тот же самый буфер.

pdwDataLen

[in/out] Указатель на значение DWORD, которое содержит длину буфера данных. При вызове функции этот параметр определяет число байт шифртекста в буфере pbData. Если используется блочный шифр, а признак Final — FALSE, это значение должно быть кратным длине блока шифра.

Через этот параметр функция возвращает указатель на число байт открытого текста, помещенного в буфер *pbData.*

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_ALGID
* NTE_BAD_FLAGS
* NTE_BAD_HASH
* NTE_BAD_DATA
* NTE_BAD_UID
* NTE_BAD_LEN
* NTE_FAIL

Комментарии

Расшифрование большого объёма данных можно выполнять по блокам, многократно вызывая ***CryptDecrypt.*** Флаг *Final* следует установить в значение TRUE только при последнем обращении к ***CryptDecrypt,*** чтобы процесс расшифрования завершился должным образом. Когда параметр *Final* равен TRUE, выполняются следующие дополнительные действия:

* если применяется ключ блочного шифрования, данные дополняются до длины, кратной размеру шифрблока. Для определения длины шифрблока используйте ***CryptGetKeyParam*** с пармстром KP_BLOCKLEN;
* если используется шифр в режиме гаммирования с обратной связью, функция ***CryptDecrypt*** установит регистр обратной связи шифра, равный размеру синхропосылки ключа, что позволит её использовать в следующем блоке данных;

* если используется поточный шифр, функция *CryptDecrypt* установит шифр в его начальное состояние для следующего запроса.

В Windows 2000 криптопровайдер Microsoft Enhanced Cryptographic Provider v1.0 поддерживает непосредственное шифрование на открытом ключе RSA и расшифрование на секретном ключе RSA. При шифровании используется формат дополнения PKCS #1 версии 2, при расшифровании дополнение проверяется. Длина шифртекста при расшифровании должна совпадать с длинами модулей RSA ключа, используемого при расшифровании. Если шифртекст имеет нули в старших байтах, то их следует включить во входной буфер и учесть в длине входного буфера. Шифртекст надо передавать в формате, где первым передается самый младший байт (формат «little endian»).

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptCreateHash, CryptEncrypt, CryptGenKey, CryptGetKeyParam, CryptGetHashParam, CryptImportKey, CryptSignHash, CryptVerifySignature

CryptEncrypt

Функция ***CryptEncrypt*** используется для зашифрования данных. Алгоритм шифрования данных определяется ключом, передаваемым через параметр *hKey*.

```
BOOL WINAPI CryptEncrypt(
HCRYPTKEY hKey,
HCRYPTHASH hHash,
BOOL Final,
DWORD dwFlags,
BYTE  *pbData,
DWORD *pdwDataLen,
DWORD dwBufLen
);
```

Параметры

hKey

[in] Дескриптор сессионного ключа, используемого для зашифрования данных. Приложение получает этот дескриптор, используя функцию ***CryptGenKey*** или ***CryptImportKey***.

Этот ключ определяет применяемый алгоритм шифрования.

hHash

[in] Дескриптор объекта хеш-функции. Если данные одновременно зашифровываются и хешируются, то дескриптор объекта хеш-функции передается в этом параметре. В процедуру хеширования данные передаются до зашифрования. Эту возможность удобно использовать, если данные одновременно подписываются и зашифровываются.

До того как вызвать функцию ***CryptEncrypt*** приложение должно получить дескриптор объекта хеш-функции, вызвав функцию ***CryptCreateHash***. После зашифрования значение хеша можно получить вызовом функции ***CryptGetHashParam*** и подписать функцией ***CryptSignHash***.

Если значение хеша не вычисляется, этот параметр должен быть равен нулю.

Final

[in] Булева величина. Определяет, является ли переданный функции блок последним зашифрованным блоком данных. Ее следует установить в TRUE, если это последний (или единственный) блок, и в FALSE — в противном случае. См. раздел «Комментарии».

dwFlags

[in] Значения флагов. В настоящее время определено одно значение флагов.

* CRYPT_OAEP

  Когда флаг установлен, зашифрование при использовании алгоритма RSA происходит с применением схемы RSAES-OAEP, предусмотренной в стандарте PKCS #1 версии 2.0. По умолчанию зашифрование при использовании алгоритма RSA выполняется с применением схемы RSAES-PKCS1-v1_5.

  Флаг используется с криптопровайдером Microsoft Enhanced Cryptographic Provider, начиная с Windows 2000.

pbData

[in/out] Буфер, содержащий данные для зашифрования. После зашифрования шифртекст помещается в тот же самый буфер.

Размер этого буфера определен параметром *dwBufLen*. Количество байт, подлежащих зашифрованию, определяется в параметре*pdwDataLen*.

Этот параметр устанавливают в NULL, если необходимо только определить размер буфера, требуемый для размещения шифртекста.

pdwDataLen

[in/out] Указатель на значение DWORD, содержащее длину данных. Перед вызовом функции приложение должно установить этот параметр равным количеству байт открытого текста, которые подлежат зашифрованию. После возврата этот адрес показывает число байт шифртекста.

Если буфера, указанного *pbData,* не достаточно, то функция возвращает FALSE с кодом ошибки ERROR_MORE_DATA, при этом копирует требуемый размер буфера в байтах в переменную, указанную парметром *pdwDataLen*.

Если pbData — NULL, функция копирует размер данных в байтах в переменную, указанную параметром *pdwDataLen*. Это позволяет приложению определять требуемый размер буфера.

Если используется блочный шифр, длина шифруемых данных должна быть кратной размеру шифрблока, за исключением последнего блока данных, передаваемых на зашифрование, при этом флаг *Final* устанавливается в TRUE.

dwBufLen

[in] Размер буфера *pbData* в байтах.

В зависимости от используемого алгоритма, длина зашифрованного текста может несколько больше, чем исходного открытого текста. В этом случае буфер *pbData* должен иметь соответствующую длину.

Как правило, если используется поточный шифр, длина шифртекста такая же, как открытого текст. Если используется блочный шифр, шифртекст дополняется до длины, кратной размеру шифрблока, и становится длиннее открытого текста.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE,

соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_ALGID
* NTE_BAD_FLAGS
* NTE_BAD_DATA
* NTE_BAD_UID
* NTE_BAD_LEN
* ERROR_MORE_DATA
* NTE_FAIL

Комментарии

Когда шифруется большой объём данных, шифрование можно осуществлять по блокам многократным вызовом ***CryptEncrypt***. Параметр *Final* следует установить в TRUE только при последнем обращении к *CryptEncrypt,* чтобы завершить процесс шифрования должным образом. Когда параметр *Final* ранен TRUE, выполняются следующие дополнительные действия:

* если применяется ключ блочного шифрования, данные будут дополнены до длины, кратной размеру шифрблока. Для определения длины шифрблока используйте ***CryptGetKeyParam*** с параметром KP_BLOCKLEN;
* если используется шифр в режиме гаммирования с обратной связью, функция *CryptEncrypt* установит регистр обратной связи шифра равной вслич ине синхропосылки ключа для сё использования в следующем блоке данных;
* если используется поточный шифр, функция *CryptEncrypt* установит шифр и его начальное состояние для следующего запроса.

В Windows 2000 криптопровайдер Microsoft Enhanced Cryptographic Provider v1.0 поддерживает непосредственное шифрование на открытом ключе RSA и расшифрование на секретном ключе RSA. При шифровании используется формат дополнения PKCS #1 версии 2, при расшифронамии дополнение проверяется. Длина шифртекста при расшифровании должна совпадать с длинами модулей RSA ключа, используемого при расшифровании. Если шифртскст имеет пули в старших байтах, то их следует включить во входной буфер и учесть в длине

входного буфера. Шифртекст надо передавать в формате, где первым передается самый младший байт (формат «little endian»).

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptCreateHash, CryptDecrypt, CryptGenKey, CryptGetHashParam, CryptImportKey, CryptSignHash

CryptCreateHash

Функция ***CryptCreateHash*** используется для инициализации хеширования потока данных. Она создает объект хеш-функции и возвращает приложению его дескриптор. Этот дескриптор используется при вызовах функций ***CryptHashData*** и ***CryptHashSessionKey*** для хеширования потока данных и сессионных ключей.

```
BOOL WINAPI CryptCreateHash(
HCRYPTPROV hProv,
ALG_ID Algid,
HCRYPTKEY hKey,
DWORD dwFlags,
HCRYPTHASH *phHash
);
```

Параметры

hProv

[in] Дескриптор криптопровайдера. Его следует предварительно получить через вызов функции ***CryptAcquireContext***.

Algid

[in] Идентификатор используемого алгоритма хеширования. Значение параметра зависит от используемого криптопровайдера. Список алгоритмов, используемых криптопровайдерами Microsoft, см. в разделе «Комментарии».

hKey

[in] Если используется алгоритм хеширования с секретным ключом, например HMAC или MAC, то в этом параметре пере-

даётся сессионный ключ для объекта хеш-функции. Ключ должен определять алгоритм блочного шифрования, например RC2, в режиме CBC.

Для алгоритмов, не требующих ключа, параметр должен быть равен нулю,

dwFlags

[in] Значения флагов. Этот параметр зарезервирован для будущего использования и должен быть нулевым.

phHash

[in/out] Адрес, по которому функция копирует дескриптор нового объекта хеш-функции.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию *GetLastError*:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* ERROR_NOT_ENOUGH_MEMORY
* NTE_BAD_ALGID
* NTE_BAD_FLAGS
* NTE_BAD_KEY
* NTE_BAD_LEN
* NTE_NO_MEMORY
* NTE_FAIL

Комментарии

В криптопровайдерах Microsoft определены различные алгоритмы хеширования (таблица П-6).

Таблица **П-6.** Алгоритмы хеширования **криптопровайдеров Microsoft**

Идентификатор	Описание
CALG_HMAC	HMAC, алгоритм хеширования с ключом
CALG_MAC	Message Authentication Code
CALG_MD2	MD2
CALG_MD5	MD5
CALG_SHA	US DSA Secure Hash Algorithm
CALG_SHA1	Тоже самое, что CALG_SHA
CALG_SSL3_SHAMD5	Клиентская аутентификация в SSL3

Хеш-функции вычисляются средствами функций ***CryptHashData*** **и** ***CryptHashSessionKey.*** Эти функции получают на входе дескриптор объекта хеш-функции. После того как все необходимые данные переданы объекту хеш-функции, выполняется одно и только одно из следующих действий:

* для получения значения хеша используют функцию ***CryptGetHashParam*** с параметром HP_HASHVAL;
* для получения сессионного ключа используют функцию ***CryptDeriveKey***;
* для подписи значения хеша используют функцию ***CryptSignHash***;
* для проверки подписи используют функцию ***CryptVerifySignature***.

После вызова одной из этих функций единственная функция, где можно использовать дескриптор объекта хеш-функции, — ***Cryp tDestroyHash***.

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98; необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип л файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptAcquire Context, CryptDeriveKey, CryptDestroyHash, CryptGetHashParam, CryptHashData, CryptHashSessionKey, CryptSignHash, Crypt Verify Signature

CryptDestroyHash

Функция ***CryptDestroyHash*** уничтожает объект хеш-функции, определённый дескриптором *hHash*. После уничтожения объект хеш-функции более использовать нельзя.

По завершении работы приложения с объектами хеш-функций, все они уничтожаются функцией ***CryptDestroyHash.***

```
BOOL WINAPI CryptDestroyHash(
HCRYPTHASH hHash
);
```

Параметры

hHash

[in] Дескриптор уничтожаемого объекта хеш-функции.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию *GetLastError*:

* ERROR_BUSY (объект хеш-функции, определенный параметром *hKey*, сейчас используется и не может быть уничтожен)
* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_ALGID
* NTE_BAD_HASH
* NTE_BAD_UID

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptCreateHash, CryptHashData, CryptSignHash

CryptDuplicateHash

Функция *CryptDuplicateHash* делает точную копию объекта хеш-функции. Новый объект хеш-функции содержит значение хеша, параметры и внутреннее состояние дублируемого объекта на момент вызова функции.

```
BOOL WINAPI CryptDuplicateHash(
HCRYPTKEY hKey,
DWORD *pdwReserved,
DWORD dwFlags,
HCRYPTKEY *phHash
);
```

Параметры

hKey

[in] Дескриптор дублируемого объекта хеш-функции.

pdwReserved

[in] Параметр зарезервирован для будущего использования и должен быть равен NULL.

dwFlags

[in] Значения флагов. Этот параметр зарезервирован для будущего использования и должен быть равен нулю.

phHash

Указатель на дескриптор, созданного объекта хеш-функции.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

- ERROR_CALL_NOT_IMPLEMENTED
- ERROR_INVALID_PARAMETER
- NTE_BAD_HASH

Комментарии

Функция ***CryptDuplicateHash*** делает копию хеша, параметров и внутреннего состояния объекта хеш-функции. Эта функция может быть использована, когда приложению необходимо вычислить два значения хеша, у которых начальное состояние является одинаковым. Для этого создается объект хеш-функции, хешируются общие данные, функцией ***CryptDuplicateHash*** создается второй объект хеш-функции, и далее хешируются различные данные отдельно для каждого объекта.

Функция ***CryptDestroyHash*** должна быть вызвана для любого объекта хеш-функции, созданного функцией ***CryptDuplicateHash.***

Уничтожение исходного объекта хеш-функции не влечет за собой уничтожения дублированного. Созданный функцией ***CryptDuplicateHash*** объект хеширования не содержит в себе параметров и внутренних состояний, разделяемых с исходным объектом.

Дополнительная информация

Windows NT/2000: необходима Windows 2000.

Windows 95/98: необходима Windows 98.

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptDestroyHash

CryptGetHashParam

Функция ***CryptGetHashParam*** возвращает параметры объекта хеш-функции. Через эту функцию можно получить фактическое значение хеша.

```
BOOL WINAPI CryptGetHashParam(
HCRYPTKEY hHash,
DWORD dwParam,
BYTE «pbData,
DWORD *pdwDataLen,
DWORD dwFlags
);
```

Параметры

hHash

[in] Дескриптор объекта хеш-функции, параметры которого определяются.

dwParam

[in] Значение параметра. Величина *dwParam* может быть установлена в одно из следующих значений.

* HP_ALGID

  Алгоритм хеш-функции. Идентификатор алгоритма (ALG_ID), соответствующий данному объекту хеш-функции возвращается через буфер *pbData*. Значение 4 возвращается *в pdwDataLen*.

  Список возможных алгоритмов хеша см. в описании функции ***CryptCreateHash***.

* HP_HASHSIZE

  Размер значения хеша. Величина DWORD, определяющая количество байт в значении хеша, возвращается через буфер *pbData*. Значение 4 возвращается *в pdwDataLen*.

Приложение должно определить этот параметр заранее, чтобы выделить необходимый объем памяти для получения значения хеша.

* HP_HASHVAL

  Значение хеша для объекта хеш-функции, указанного дескриптором *hHash*. Это значение вычисляется от данных переданных ранее приложением в функции ***CryptHashData*** **и** ***CryptHashSessionKey***.

  Функция ***CryptGetHashParam*** закрывает объект хеш-функции. После этого вызова, невозможно передать новые данные объекту хеш-функции. Функции ***CryptHashData*** **и** ***CryptHashSessionKey*** вернут FALSE. После того как приложение завершает работу с объектом хеш-функции, оно уничтожает объект вызовом функции ***CryptDestroyHash***.

pbData

[out] Буфер данных параметра. Функция копирует соответствующие параметру данные к буфер. Формат этих данных зависит от значения *dwParam*.

Если параметр — NULL, то данные не копируются. В этом случае требуемый размер буфера (в байтах) возвращается в *pdwDataLen*.

См. раздел «Получение данных неизвестного размера» в Приложении 2.

pdwDataLen

[in/out] Указатель на значение типа DWORD, содержащее размер, в байтах, буфера ***pbData***. После завершения функции, DWORD содержит количество байт, переданных в буфер.

Примечание

Когда обрабатываются возвращаемые в буфере данные, приложение должно использовать фактический размер данных. Он немного меньше, чем размер буфера, указанный на входе. На входе размер буфера обычно задают таким, чтобы буфера хватало для получения максимально возможного объема данных. На выходе переменная, на которую указывает данный параметр, содержит фактический размер данных скопированных в буфер.

dwFlags

[in] Значения флагов. Этот параметр зарезервирован для будущего использования и должен быть равен нулю.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

- ERROR_INVALID_HANDLE
- ERROR_INVALID_PARAMETER
- ERROR_MORE_DATA
- NTE_BAD_FLAGS
- NTE_BAD_HASH
- NTE_BAD_TYPE
- NTE_BAD_UID

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См, также

CryptCreateHash, CryptGetKeyParam, CryptHashData, CryptHashSessionKey, CryptSetHashParam

CryptHashData

Функция ***CryptHashData*** используется для добавления данных к объекту хеш-функции. Ее и функцию ***CryptHashSessionKey*** можно вызывать многократно при вычислении хеша для данных разбитых па блоки.

Перед вызовом этой функции следует вызвать функцию ***CryptCreateHash*** для получения дескриптора объекта хеш-функции.

```
BOOL WINAPI CryptHashData(
HCRYPTHASH hHash,
CONST BYTE *pbData,
DWORD dwDataLen,
DWORD dwFlags
);
```

Параметры

hHash

[in] Дескриптор объекта хеш-функции.

pbData

[in] Буфер, содержащий хешируемые данных.

dwDataLen

[in] Размер хешируемых данных в байтах. Если установлен флаг CRYPT_USERDATA, то параметр должен быть равен нулю.

dwFlags

[in] Значения флагов. В настоящее время определено одно возможное значение.

* CRYPT_USERDATA

  Криптопровайдеры Microsoft не поддерживают этот флаг. Для тех криптопровайдеров, которые поддерживают флаг CRYPT_USERDATA, он означает, что пользователь будет вводить данные для хеширования в интерактивном режиме. Данные, введенные пользователем, и будут хешироваться функцией. Приложение, таким образом, не получает доступа к данным. Этот флаг используют для ввода пользователем своего персонального идентификатора (Personal Identification Number, PIN).

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_ALGID
* NTE_BAD_FLAGS
* NTE_BAD_HASH
* NTE_BAD_HASH_STATE
* NTE_BAD_KEY
* NTE_BAD_LEN
* NTE_BAD_UID
* NTE_FAIL
* NTE_NO_MEMORY

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок; прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptCreateHash, ***CryptHashSessionKey***, *CryptSignHash*

CryptHashSessionKey

Функция ***CryptHashSessionKey*** используется для добавления к объекту хеш-функции значения сессионного ключа. Ее можно вызывать много раз с одним и тем же дескриптором объекта хеш-функции для вычисления хеша от нескольких ключей. Вызовы функции ***CryptHashSessionKey*** могут пересекаться с вызовами функции ***CryptHashData***.

Перед вызовом этой функции следует вызвать функцию ***CryptCreateHash*** для получения дескриптора объекта хеш-функции.

```
BOOL WINAPI CryptHashSessionKey(
HCRYPTHASH hHash,
HCRYPTKEY hKey,
DWORD dwFlags
);
```

Параметры

hHash

[in] Дескриптор объекта хеш-функции. Приложение получает этот дескриптор, используя функцию ***CryptCreateHash.***

hKey

[in] Дескриптор хешируемого сессионного ключа.

dwFlags

[in] Значения флагов. В настоящее время определено одно значение флагов:

» CRYPT_LITTLE_ENDIAN

Когда установлен этот флаг, байты ключа хешируются, начиная с младшего (формат «little-endian»). По умолчанию (параметр *dwFlags* равен нулю) хеширование происходит, начиная со старшего байта (формат «big–endian»).

Флаг CRYPT_LITTLE_ENDIAN поддерживается в Internet Explorer 4.0 или старше, Windows 98 или в Windows 2000 или старше.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию *GetLastError*:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_ALGID
* NTE_BAD_FLAGS
* NTE_BAD_HASH
* NTE_BAD_HASH_STATE
* NTE_BAD_KEY
* NTE_BAD_UID
* NTE_FAIL

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См, также

CryptCreateHash, CryptGenKey, CryptHashData

CryptSetHashParam

Функция ***CryptSetHashParam*** устанавливает параметры объекта хеш-функции.

```
BOOL. WINAPI CryptSetHashParam(
HCRYPTHASH hHash,
DWORD dwParam,
BYTE *pbData,
DWORD dwFlags
);
```

Параметры

hHash

Дескриптор объекта хеш-функции, параметры которого устанавливаются.

dwParam

Значение параметра. Величину *dwParam* можно установить в одно из следующих значений параметра хеш-функции.

- HP_HMAC_INFO

  Указатель на структуру HMAC_INFO, определяющую алгоритм хеша, а также параметры inner string и outer string.

- HP_HASHVAL

  Массив байт содержит значение хеша, которое напрямую копируется в объект хеш-функции. Размер хеша можно определить вызовом функции ***CryptGetHashParam*** с параметром HP_HASHSIZE.

  Некоторые криптопровайдеры не поддерживают данную возможность.

pbData

Буфер данных параметра. При вызове функции буфер должен содержать данные, соответствующие значению параметра в *dwParam*. Формат данных зависит от типа параметра *dwParam*.

dwFlags

Значения флагов. В настоящее время для этой функции значения флагов не определены.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

- ERROR_INVALID_HANDLE
- ERROR_BUSY
- ERROR_INVALID_PARAMETER
- NTE_BAD_FLAGS
- NTE_BAD_HASH
- NTE_BAD_TYPE
- NTE_BAD_UID
- NTE_FAIL

Комментарии

Вызов функции ***CryptSetHashParam*** с параметром HP_HASHVAL позволяет копировать значение хеша непосредственно в объект хеш-функции. Иногда необходимо подписать значение хеша, выработанное в другом месте. Обычно это делается таким образом:

* приложение создает объект хеш-функции с помощью функции ***CryptCreateHash***;
* приложение устанавливает значение хеша, используя параметр HP_HASHVAL;
* приложение подписывает значение хеша, используя функцию ***CryptSignHash*** и получает ЭЦП;
* приложение уничтожает объект хеш-функции, используя функцию ***CryptDestroyHash***.

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

См. также

CryptCreateHash, CryptDestroyHash, CryptGetHashParam, CryptSetKeyParam, CryptSignHash

CryptSignHash

Функция ***CryptSignHash*** вычисляет значение ЭЦП от значения хеша, определенного дескриптором объекта хеш-функции *hHash*.

```
BOOL WINAPI CryptSignHash(
HCRYPTHASH hHash,
DWORD dwKeySpec,
LPCWSTR sDescription,
DWORD dwFlags,
BYTE *pbSignature,
DWORD *pdwSigLen
);
```

Параметры

hHash

[in] Дескриптор объекта хеш-функции, для которого вычисляется ЭЦП.

dwKeySpec

[in] Параметр идентифицирует секретный ключ, который будет использоваться для ЭЦП. Возможные значения параметра: AT_SIGNATURE или AT_KEYEXCHANGE.

sDescription

[in] Строка, описывающая подписываемые данные. Текст описания добавляется к хешу перед тем, как будет произведена подпись. Всякий раз, когда подпись проверяется, в функцию ***CryptVerifySignature*** должна передаваться та же самая строка. Это гарантирует, что и подписывающая сторона и проверяющая знают то, что подписывается или проверяется.

Если строка описания не включается в подпись, этот параметр может быть равен NULL.

Примечание

В настоящее время этот параметр поддерживается для совместимости с предыдущими версиями криптопровайдеров. Использование его не рекомендуется, так как это снижает безопасность системы.

dwFlags

[in] Значения флагов. В настоящее время определено одно значение флага.

* CRYPT_NOHASHOID

  Используется с криптопровайдерами RSA. В случае установки данного флага подписывается непосредственно значение хеш-функции. По умолчанию, если флаг не устанавливается, подписывается структура DigestInfo, определенная к стандарте PKCS #7.

  Флаг CRYPT_NOHASHOID поддерживается в Windows NT Service Pack 6 и старше. Флаг не поддерживается в Windows 98 или Internet Explorer 5.0.

pbSignature

[out] Адрес буфера, в который копируется значение подписи.

Этот параметр может быть равен NULL, если необходимо только определить размер буфера, требуемый для размещения подписи.

См. раздел «Получение данных неизвестного размера» в Приложении 2.

pdwSigLen

[in/out] Указатель на значение типа DWORD, содержащее размер в байтах буфера *pbSignature*. После завершения функции DWORD показыывает число байт, переданных в буфер.

Примечание

Когда обрабатываются возвращаемые в буфере данные, приложение должно использовать фактический размер данных. Он может быть немного меньше, чем размер буфера, указанный на входе. На входе обычно задают такой размер буфера, чтобы его хватило для получения максимально возможного объема данных. На выходе переменная, на которую указывает данный параметр, показывает фактический размер данных, скопированных в буфер.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

- ERROR_INVALID_HANDLE
- ERROR_INVALID_PARAMETER
- ERROR_MORE_DATA
- NTE_BAD_ALGID
- NTE_BAD_FLAGS
- NTE_BAD_HASH
- NTE_BAD_UID
- NTE_NO_KEY
- NTE_NO_MEMORY

Комментарии

Перед вызовом этой функции дескриптор объекта хеш-функции следует получить через вызов функции ***CryptCreateHash***. Далее для хеширования данных или сессионных ключей используется функция ***CryptHashData*** или ***CryptHashSessionKey***.

Функция ***CryptSignHash*** выполняет следующие операции:

- объект хеш-функции «закрывается» и значение хеша извлекается;

* значение хеша дополняется, как это требуется алгоритмом подписи;
* вычисляется значение подписи.

После того как значение хеша подписано, приложение не может добавлять новые данные к объекту хеш-функции. Приложение должно уничтожить объект хеш-функции, вызвав функцию *CryptDestroyHash*.

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

Unicode: на всех платформах реализованы Unicode- и ANSI-версии.

См. также

CryptCreateHash, CryptHashData, CryptHashSessionKey, CryptVerifySignature

CryptVerifySignature

Функция *CryptVerifySignature* осуществляет проверку подписи, соответствующей объекту хеш-функции *hHash*.

```
BOOL WINAPI CryptVerifySignature(
HCRYPTHASH hHash,
CONST BYTE *pbSignature,
DWORD dwSigLen,
HCRYPTKEY hPubKey,
LPCWSTR sDescription,
DWORD dwFlags);
```

Параметры

hHash

Дескриптор объекта хеш-функции, подпись которого проверяется.

pbSignature

Буфер, содержащий значение проверяемой подписи.

dwSigLen

Длина в байтах, значения подписи, содержащейся в буфере *pbSignature*.

hPubKey

Дескриптор открытого ключа, который используется для проверки подписи. Этот открытый ключ должен соответствовать ключевой паре, на которой вычислялось значение подписи.

sDescription

Строка описания подписанных данных. Она должна быть точно такой, что использовалась в функции ***CryptSignHash*** при вычислении подписи. Если она не соответствует использованной в функции ***CryptSignHash,*** проверка подписи вернет отрицательный результат.

Примечание

В настоящее время этот параметр поддерживается для совместимости с предыдущими версиями криптопровайдеров. Использование его не рекомендуется, так как это снижает безопасность системы,

dwFlags

Значения флагов. В настоящее время определено одно значение флага.

* CRYPT_NOHASHOID

  Используется с криптопровайдерами RSA. При проверке подписи проверяется только значение хеш-функции. По умолчанию, если флаг не устанавливается, проверяется подпись для структуры *DigestInfo,* определенной в стандарте PKCS #7.

  Флаг CRYPT_NOHASHOID поддерживается в Windows NT Service Pack 6 и старше. Флаг не поддерживается в Windows 98 или Internet Explorer 5.0.

Возвращаемое значение

При успешном завершении функция возвращает TRUE, в противном случае — FALSE. Если возвращается величина FALSE, соответствующий код ошибки можно получить через функцию ***GetLastError***:

* ERROR_INVALID_HANDLE
* ERROR_INVALID_PARAMETER
* NTE_BAD_FLAGS
* NTE_BAD_HASH
* NTE_BAD_KEY

* NTE_BAD_SIGNATURE
* NTE_BAD_UID
* NTE_NO_MEMORY

Комментарии

Функция *CryptVerifySignature* выполняет следующие операции:

* если параметр описания *sDescription* определен, он добавляется к хешу;
* объект хеш-функции «закрывается» и значение хеша извлекается;
* значение хеша дополняется, как это требуется алгоритмом подписи;
* осуществляется проверка подписи на открытом ключе *hPubKey*.

Если подпись, переданная через буфер *pbSignature*, неверна, то функция возвращает FALSE с кодом ошибки NTE_BAD_SIGNATURE.

После выполнения проверки подписи приложение не может добавлять новые данные к объекту хеш-функции. Приложение должно уничтожить объект хеш-функции через вызов функции *CryptDestroyHash*.

Дополнительная информация

Windows NT/2000: необходима Windows NT 4.0 или старше.

Windows 95/98: необходима Windows 95 OSR2 или старше (или Windows 95 и Internet Explorer 3.02 или старше).

Заголовок: прототип в файле wincrypt.h.

Библиотека: advapi32.dll

Unicode: на всех платформах реализованы Unicode- и ANSI-версии.

См. *также*

CryptCreateHash, CryptDestroyHash, CryptHashData, CryptHashSessionKey, CryptSignHash

Получение данных неизвестного размера

Многие функции потенциально могут возвращать большие массивы данных по адресу, который указывает один из параметров, переданный приложением. Во всех случаях операция выполняется по одному сценарию. Параметр, который указывает на буфер для копирования данных, имеет нотацию, где

первые две буквы в имени — «pb» или «pv». Первые три буквы имени другого параметра — «pcb». Он определяет размер в байтах массива данных, который размещается в буфере «pb» или «pv». Например, рассмотрим следующую спецификацию функции:

```
BOOL WINAPI SomeFunction(
PCCRL_CONTEXT pCrlContext,    // in
DWORD dwPropId,               // in
BYTE *pbData,                 // out
DWORD *pcbData                // in/out
);
```

Здесь *pbData* — указатель па буфер *для размещения* массива данных, а *pcbData* — размер в байтах возвращаемого массива данных.

Если буфера, определенного параметром *pbData,* не достаточно для размещения необходимого массива данных, функция возвращает FALSE с кодом ошибки ERROR MORE_DATA (код ошибки получается посредством вызова функции ***GetLastError***). Необходимый размер буфера в байтах возвращается через параметр *pcbData.*

Если приложение передает ц параметре *pbData* NULL, а параметр *pcbData* содержит ненулевое значение, то функция возвращает TRU E, а необходимый размер буфера в байтах возвращается через параметр *pcbData.* Этот способ рекомендуется для определения размера буфера, необходимого для размещения массива данных.

Следует отметить, что фактический размер данных иногда немного меньше, чем размер возвращенный через параметр *pcbData.* При вызове функции с параметром *pbData,* содержащим указатель на выделенный под данные буфер, фактический размер массива данных, переданных в буфер, содержится в *pcbData.* При дальнейшей обработке возвращенных данных необходимо использовать именно это значение размера массива.

Следующий пример демонстрирует возможный вариант реализации описанной процедуры:

```
ftinclude <stdio.h>
ftinclude  <windows.h>
ftinclude  <wincrypt. h>
void  HandleError(char *s);
void  maln() {
   //--------------------------------
   // Декларируем переменные функции SomeFunction.
```

```
PCCRL_CONTEXT pCrlContext;    // Устанавливается где-либо
                                 в программе.
DWORD dwPropId; // Устанавливается где-либо в программе.
DWORD cbData;
BYTE      *pbData;
//--------------------------------
// Вызываем функцию SomeFunction для установки cbData.
// После вызова функции cbData будет содержать
// необходимый размер буфера.
if (SomeFunction(
         pCrlContext,
         dwPropId,
         NULL,
         &cbData)) {
   printf("The function succeeded.\n");
}
else {
// При вызове функции SomeFunction произошла ошибка.
// Вызываем функцию обработки ошибок.
   HandleError("Function call failed.");
}
//--------------------------------
// Вызов функции SomeFunction завершился успешно.
// Необходимый размер буфера содержится в переменной
// cbData.
//
// Выделяем необходимый размер памяти.
if (pbData = (BYTE*)malloc(cbData)) {
   printf("Memory has been allocated.\n");
}
else {
   // При попытке выделения памяти произошла ошибка.
   // Вызываем функцию обработки ошибок.
   HandleError("Malloc operation failed.");
}
//
// Память под буфер успешно выделена.
// Вызываем функцию SomeFunction для получения массива
// данных.
if (SomeFunction(
         pCrlContext,
         dwPropId,
         pbData,
         &cbData)){
   printf("The function succeeded.\n");
}
```

```
    else {
        // При повторном вызове функции SomeFunction
        // произошла ошибка. Вызываем функцию обработки
        //  ошибок.
        HandleError("The second call to the function
            failed.");
    }
    //*********************************************************
    // Повторный вызов функции SomeFunction завершился
    // успешно. Массив данных находится в буфере pbData.
    // Отметим, что теперь переменная cbData содержит
    // фактический размер переданного буфера данных.
    // При обработке переданных данных, приложение должно
    // использовать именно это значение.
}
//---------------------------------
// В этом примере используется функция обработки ошибок
// HandleError. Простейшая функция обработки ошибок содержит
// печать сообщения об ошибке в стандартный error file
// (stderr) и завершение программы.
void
HandleError(
    char *s
    )
{
    fprintf(stderr, "An error occurred in running the
        program.\n");
    fprintf(stderr, "%s\n", s);
    fprintf(stderr, "Error number %x.\n", GetLastError());
    fprintf(stderr, "Program terminating.\n");
    exit(1);
}
```

Недостатком описанной процедуры является то, что при каждом получении массива данных необходим дополнительный вызов функции для получения размера буфера. Альтернативная процедура состоит в использовании заранее выделенного буфера. Если размера выделенного буфера хватает для массива данных, то вызов функции будет успешен и экономится время на дополнительный вызов. Если размера буфера не достаточно, то требуется освободить буфер и выделить новый в соответствии с размером, возвращенным в параметре *pcbData*. Такая технология имеет преимущество, если известен размер буфера, которого в большинстве случаев будет хватать для запрашиваемого массива данных.

Следующий фрагмент показывает использование технологии заранее выделенного буфера:

```
//-----------------------------------
// Принимаем, что функция SomeFunction (из предыдущего
// примера) многократно вызывается приложением,
// и в большинстве случаев буфер размером 50 байт будет
// достаточен для сохранения массива данных.
//-----------------------------------
// Декларируем переменные функции SomeFunction.
PCCRL_CONTEXT pCrlContext;  // Устанавливается где-либо
                            в программе.
DWORD dwPropId;      // Устанавливается где-либо в программе.
//-----------------------------------
// Выделяем буфер размером 50 байт и используем его для
// вызова функции SomeFunction.
// Однако, в случае если размер массива данных больше 50
// байт, используем переменную cbData для выделения
// необходимого буфера. После выделения нового буфера снова
// вызываем функцию SomeFunction для получения массива
// данных.
DWORD dwTries = 0; // the number of times the function has
been tried
BOOL  fMore=TRUE;  // flag indicating the function needs to
be tried
DWORD cbData = 50;
BYTE  *pbData;
//-----------------------------------
// Начало цикла. Цикл выполняется самое большое два раза.
while (fMore) {
   if (pbData = (BYTE *)malloc(cbData)) {
      printf("Memory has been allocated.\n");
   }
   else {
      HandleError("Memory allocation error.");
   }
   //-----------------------------------
   // Память для буфера выделена успешно. Пытаемся вызвать
   // функциюSomeFunction.
   if (SomeFunction(pCrlContext, dwPropId, pbData,
      &cbData)) {
      // Вызов функции завершился успешно. Завершаем цикл.
      printf("The function succeeded. Exit the loop.\n");
      fMore=FALSE;
   }
   else {
      // При вызове функции SomeFunction произошла ошибка.
```

```
            // Освобождаем выделенную память.
            free(pbData);
            // Если ошибка произошла не по причине недостаточного
            // размера буфера (код ошибке не равен
            // ERROR_MORE_DATA), то выходим из цикла с ошибкой.
            if(GetLastError() != ERROR_MORE_DATA ) {
                HandleError("General function failure.");
            }
            // Если ошибка произошла по причине недостаточного
            // размера буфера (код ошибке равен ERROR_MORE_DATA),
            // то увеличиваем счетчик количества ошибок.
            // Количество попыток не может быть больше двух.
            if( ++dwtries > 1 ) {
                HandleError("Function call failed twice");
            }
        }
    } // конец цикла
//--------------------------------------------------------
// Первый или второй вызов функции SameFunction завершился
// успешно. В этом случае переменная cbData содержит
// фактический размер массива данных. При обработке
// переданных данных, приложение должно использовать именно
// это значение.
//--------------------------------------------------------
// Если необходимо, обрабатываем буфер pbData.
//--------------------------------------------------------
// После обработки освобождаем буфер данных.
free(pbData);
```

Добиться увеличения производительности возможно также с помощью использования функции ***alloca*** для выделения памяти из стека, вместо использования функции ***malloc***, которая выделяет память из кучи. Поскольку намять выделяется из стека, то нет необходимости вызывать функцию ***free*** для освобождения памяти после использования буфера. Если применяется функция ***alloca,*** то использовать буфер можно только локально, поскольку при возвращении из функции стек освобождается. Также необходимо отметить, что стек имеет ограниченный размер, и если непродумано пользоваться функцией ***alloca,*** то возможно его переполнение.

Коды ошибок

ERROR_BUSY

Криптографический объект, который пытаются уничтожить, занят и не может быть уничтожен.

ERROR_INVALID_PARAMETER

Один или несколько параметров имеют неверное значение. Чаще всего это неверный указатель.

ERROR_NOT_ENOUGH_MEMORY

В течение операции операционная система исчерпала память.

NTE_NO_MEMORY

В течение операции криптопровайдер исчерпал память.

NTE_BAD_FLAGS

Параметр *dwFlags* имеет неверное значение.

NTE_BAD_KEYSET

Ключевой контейнер не может быть открыт. Обычно это означает, что он не существует. Для создания контейнера необходимо вызвать функцию ***CryptAcquire Context*** с флагом CRYPT_NEWKEYSET.

NTE_BAD_KEYSET_PARAM

Параметр *pszContainer* или *pszProvider* не установлен или имеет неверное значение.

NTE_BAD_PROV_TYPE

Параметр *dwProvType* имеет неверное значение. Все типы криптопровайдеров должны иметь значение от 1 до 999 включительно.

NTE_BAD_SIGNATURE

Не прошла проверка цифровой подписи DLL криптопровайдера. DLL или цифровая подпись искажена.

NTE_EXISTS

Параметр *dwFlags* установлен в CRYPT_NEWKEYSET, а ключевой контейнер уже существует.

NTE_KEYSET_ENTRY_BAD

Ключевой контейнер, соответствующий *pszContainer*, обнаружен, но оказался искаженным.

NTE_KEYSET_NOT_DEF

Ключевой контейнер, соответствующий *pszContainer*, не существует, или запрашиваемый криптопровайдер не определен.

NTE_PROV_DLL_NOT_FOUND

Библиотеки криптопровайдера не существует, или не описан путь до нее.

NTE_PROV_TYPE_ENTRY_BAD

Тип криптопровайдера, указанный в *dwProvType* неверный. Эта ошибка относится к списку криптопровайдеров, используемых ноумолчанию.

NTE_PROV_TYPE_NO_MATCH

Тип криптопровайдера, указанный в *dwProvType*, не соответствует действительному тину криптопровайдера. Эта ошибка может иметь место, когда *pszProvider* указывает на реальный криптопровайдер.

NTE_PROV_TYPE_NOT_DEF

Тип криптопровайдера, указанный в параметре *dwProvType,* не описан.

NTE_PROVIDER_DLL_FAIL

Файл DLL криптопровайдера не может быть загружен, или произошла ошибка при инициализации.

NTE_SIGNA TURE_FILE_BAD

Ошибка произошла при загрузке образа библиотеки до проверки подписи.

ERROR_MORE_DATA

Буфера, переданного приложением, не хватает для сохранения необходимых данных

ERROR_NO_MORE_ITEMS

Перечисление закончено.

ERROR_INVALID_HANDLE

Один из параметров указывает на неверный дескриптор.

NTE_BAD_UID

Параметр *hProv* содержит неверный дескриптор криптопровайдера.

NTE_BAD_ALGID

Параметр *Algid* определяет алгоритм, который данный криптопровайдер не поддерживает.

NTE_BAD_HASH

Дескриптор объекта хеширования содержит неверное значение.

NTE_BAD_HASH_STATE

Попытка добавить данные к «закрытому» объекту хеширования.

NTE_SILENT_CONTEXT

Криптопровайдер не может выполнить запрашиваемую операцию, поскольку функция ***CryptAcquireContext*** вызвана с флагом CRYPT_SILENT.

ERROR_ CALL_NOT_IMPLEMENTED

Эта ошибка возвращается, если Криптопровайдер не поддерживает данную функцию.

NTE_BAD_KEY

Дескриптор криптографического ключа содержит неверное значение.

NTE_BAD_KEY_STATE

Криптопровадерне может выполнить экспорт ключа, поскольку при его создании не определен флаг CRYPT_EXPORTABLE.

NTE_BAD_PUBLIC_KEY

Тип ключевого блоба, определенный в параметре *dwBlobType*, установлен в PUBLICKEYBLOB, по *hExpKey* не содержит дескриптора открытого ключа.

NTE_NO_KEY

Дескриптор криптографического ключа содержит неверное значение.

NTE_BAD_VER

Данный Криптопровайдер не поддерживает версию импортируемого ключевого блоба. Обычно это означает, что необходимо обновить версию криптопровайдера.

NTE_FIXEDPARAMETERS

Некоторые криптопровайдеры имеют фиксированные параметры P, Q и G (A). В этом случае при попытке установить эти параметры через функцию ***CryptSetKeyParam*** возвращается данная ошибка.

NTE_DOUBLE_ENCRYPT

Приложение пытается зашифровать/расшифровать одни и те же данные второй раз.

NTE_BAD_DATA

Данные для процедуры зашифрования/расшифрования неверны. Например, когда используется блочное шифрование и флаг Final установлен в FALSE, параметр *pdwDataLen* должен содержать значение, кратное размеру блока шифрования.

При расшифровании эта ошибка возвращается при неверном содержании дополнения.

NTE_BAD_LEN

Буфера, переданного приложением, недостаточно для сохранения данных.

NTE_FAIL

При выполнении операции в функции произошла ошибка.

Криптопровайдеры Microsoft

В файле WinCrypt.h даны определения имен криптопровайдеров, доступных в настоящее время. Знаком * отмечены криптопровайдеры, *включенные* Microsoft в ОС, *начиная* с Windows 2000:

Microsoft Base Cryptographic Provider v1.0

Определение: MS_DEF_PROV

Библиотека: rsabase.dll

Microsoft Enhanced Cryptographic Provider*

Определение: MS_ENHANCED_PROV

Библиотека: rsaenh.dll

Microsoft Strong Cryptographic Provider*

Определение: MS_STRONG_PROV

Библиотека: rsaenh.dll

Microsoft RSA Signature Cryptographic Provider

Определение: MS_DEF_RSA_SIG_PROV

Библиотека: rsabase.dll

Microsoft RSA SChannel Cryptographic Provider*

Определение: MS_DEF_RSA_SCHANNEL_PROV

Библиотека:rsaenh.dll

Microsoft Base DSS Cryptographic Provider

Определение: MS_DEF_DSS_PROV

Библиотека: dssbase.dll

Microsoft Base DSS and Diffie-Hellman Cryptographic Provider

Определение: MS_DEF_DSS_DH_PROV

Библиотека: dssbase.dll

Microsoft Enhanced DSS and Diffie-Hellman Cryptographic Provider*

Определение: MS_ENH_DSS_DH_PROV

Библиотека: dssenh.dll

Microsoft DH SChannel Cryptographic Provider*

Определение: MS_DEF_DH_SCHANNEL_PROV

Библиотека: dssenh.dll

Типы криптопровайдеров

В файле WinCrypt.h даны определения типов криптопровайдеров, определенных в настоящее время. Знаком * отмечены типы криптопровайдеров, включенные Microsoft в ОС, начиная с Windows 2000:

PROV_RSA_FULL (1)

Криптопровайдеры данного типа поддерживают и цифровую подпись и шифрования данных. Для всех операций с открытым ключом используется алгоритм RSA. Поддерживаемые алгоритмы:

* ключевой обмен — RSA;
* цифровая подпись — RSA;
* шифрование — RC2, RC4;
* хеширование — MD5, SIIA.

PROV_RSA_SIG (2)

Криптопровайдеры данного типа поддерживают только функции и алгоритмы, необходимые для хеширования и цифровой подписи. Поддерживаемые алгоритмы:

* ключевой обмен — не поддерживается;
* цифровая подпись — RSA;
* шифрование — не поддерживается;
* хеширование — MD5, SHA.

PROV_RSA_ SCHANNEL (12)*

Криптопровайдеры данного типа поддерживают и алгоритмы RSA и протоколы Schannel. Поддерживаемые алгоритмы;

* ключевой обмен — RSA;
* цифровая подпись — RSA;
* шифрование — RC4. DES, Triple DES (не обязательно все);
* хеширование — MD5, SHA.

PROV_DSS (3)

Криптопровайдеры данного типа, как и PROV_RSA_SIG, поддерживают только функции и алгоритмы, необходимые для хеширования и цифровой подписи. В качестве алгоритма подписи используется Digital Signature Algorithm (DSA). Поддерживаемые алгоритмы:

* ключевой обмен — не поддерживается;
* цифровая подпись — DSS;
* шифрование — не поддерживается;
* хеширование — MD5, SHA.

PROV_DSS_DH (13)

Криптопровайдеры данного типа поддерживают в *дополнение* к PROV_DSS и шифрования данных. Поддерживаемые алгоритмы:

* ключевой обмен — DH;
* цифровая подпись — DSS;
* шифрование - CYLINK MEK;
* хеширование — MD5, SHA.

PROV_DH_ SCHANNEL (18)*

Криптопровайдеры данного типа поддерживают и алгоритм Диффи — Хеллмана, и Schannel протоколы. Поддерживаемые алгоритмы:

* ключевой обмен — DH (Ephemeral);
* цифровая подпись — DSS;
* шифрование — DES. Triple DES;
* хеширование — MD5, SHA.

PROV_FORTEZZA (4)

Криптопровайдеры данного типа поддерживают и протоколы и алгоритмы, разработанные National Institute of Standards and Technology (NIST). Поддерживаемые алгоритмы:

* ключевой обмен — KEA;
* цифровая подпись — DSS;
* шифрование — Skipjack;
* хеширование — SHA.

PROV_MS_EXCHANGE (5)

Криптопровайдеры данного типа поддерживают криптографические операции Microsoft Exchange mail и приложений совместимых *с* Microsoft mail. Поддерживаемые алгоритмы:

* ключевой обмен — RSA;
* цифровая подпись — RSA;
* шифрование — CAST%;
* хеширование — MD5.

PROV_SSL (6)*

Криптопровайдеры данного типа поддерживают Secure Sockets Layer (SSL) протокол. Поддерживаемые алгоритмы:

» ключевой обмен — RSA;
* цифровая подпись — RSA;
* шифрование — различные алгоритмы;
* хеширование — различные алгоритмы.

Приложение 3

Цель данного приложения — напомнить читателю смысл основных математических понятий, использованных в предыдущих параграфах и примерах, а также пояснить некоторые факты, на которые и тексте ссылаются авторы.

Основные комбинаторные задачи

Пусть имеется множество М. Мощность множества М — число элементов в данном множестве п. Дан вектор размерности k $X=(x_1, x_2, ..., x_k)$, где x принадлежит множеству М. Каково число Т таких векторов?

На первое место можно независимо поместить любой из элементов множества М — выбрать из n, аналогично на второе, третье, -.., k-е. Таким образом,

T = п x п x ... x n (k раз),

$T = n^k$

Пусть имеется п чисел 1, 2, ..., п. Найдем число Т всевозможных перестановок из них,

На первое место число можно выбрать из п, на второе — из n-1, на третье — из n-2 и т.д., на последнее — останется одно число.

Следовательно,

T = п x (п-1) x (п-2) x (n-3) x ... x 1 = п!

Трудоемкость алгоритма (пример)

Пусть имеется п неупорядоченных целых чисел от 1 до п. За сколько операций их можно упорядочить но возрастанию — расставить по порядку от 1 до п? Введем две операции: сравнение двух элементов и их перестановка между собой; назовем их в данном случае *элементарными операциями.*

Алгоритм упорядочения опишем следующим образом. Выбираем i-й элемент i = 1, ..., n-1 и последовательно сравниваем его

с n-i элементами. Если i-й элемент больше j-го, то переставляем их местами.

Алгоритм сделает n-1 шаг, на Км шаге будет сравниваться n-i элементов. Общее число операций сравнения Ts равно:

Ts = (n-1) + (n-2) + ... + 2 + 1 (сумма арифметической прогрессии)

Ts = 0.5 x п x (п-1) = 0.5 x п x n - 0.5 x п

Говорят, что трудоемкость данного алгоритма есть $O(n^2)$ (читается «о большое от п квадрат»), поскольку наибольшая степень полинома — вторая и, следовательно, при увеличении п в два раза время работы алгоритма увеличится в 4 раза.

Понятие группы

Пусть задано некоторое произвольное множество S, обозначим S x S декартово произведение этих множеств, т.е. множество вида

M = S x S = {(r, t)}, где r и t принадлежат множеству S.

Произвольное отображение M в S, т.е. соответствие между парами (r, t) и элементами множества S, при котором каждой паре соответствует элемент из S, назовем операцией па множестве S (или бинарной операцией).

Группой (G,*) назовем некоторое множество G *с* операцией * (* означает упомянутое выше отображение) на G, для которых соблюдаются следующие условия:

* операция * ассоциативна для любых а и b, принадлежащих G

  a*(b*c)=(a*b)*c

* во множестве G существует некоторый элемент с, для которого

  a*e=e*a=a

* для каждого элемента множества G существует обратный элемент о (также принадлежащий G)

  a*o=o*a=c

  Если дополнительно выполнено для любых а и b из G

  a*b=b*a

  то группа называется *коммутативной или абелевой.*

Примеры групп

Множество целых чисел с операцией сложения (Z,+) образует бесконечную абелевугруппу.

Подстановки

Рассмотрим объект «подстановка» — таблицу вида

1 2 3 ... n

$x_1\ x_2\ x_3\ \ x_n$

где x_i принадлежит множеству {1, 2, ..., n}

n называется *степенью подстановки*.

Введем операцию умножения подстановок p1 и p2

p1= 1 2 ... n p2= 1 2... n

x1 x2 ... xn y1 y2 ... yn

Обозначим xi=p1(i), yi=p2(i)

p1#p2 = 1 2... n

p2(p1(1)) p2(p1(2)) ... p2(p1(n))

Множество всех подстановок степени n с введенной операцией образует некоммутативную конечную группу.

Порядком конечной группы называется число ее элементов.

Цикличность группы

Группа (G,*) считается циклической, если ее каждый элемент b можно представить как:

$b = a^n$,

где степень n понимается как n-кратное выполнение операции * с элементом a.

Элемент a группы в этом случае называется *примитивным или образующим*.

Так, в аддитивной группе целых чисел с операцией сложения образующими будут элементы 1 и -1.

В конечной циклической группе порядка m элемент a в степени k порождает подгруппу порядка t, причем

c = m/НОД(k, т)

Если p — простое число, то НОД(p-1, p) = 1, и следовательно, t = m.

Кольца и поля

Рассмотрим множество R, на котором заданы две нетождественные операции; обозначим их + и *.

Множество R называется кольцом R(+,*), если

* операция + на множестве R образует коммутативную группу;
* для любых a, b, c выполнено a*(b+c)=a*b+a*c (дистрибутивность операции).

Кольцо коммутативно, если коммутативна операция *.

Нейтральный элемент по операции * называется единицей кольца.

Коммутативное кольцо с единицей, совокупность отличных от нуля элементов которого образуют абелеву группу относительно операции *, называется полем.

В поле можно выделить мультипликативную группу (по операции *) и аддитивную группу поля (по операции +). Наибольший интерес представляют мультипликативные группы полей.

Можно утверждать, что мультипликативная группа поля является циклической группой. Порядок данной циклической группы есть p-1, где p — число элементов поля (нулевой элемент исключен).

Для простого p:

$a^{P-1} = 1$

Кольца и поля вычетов

Целые числа a и b сравнимы по модулю p, если разность a-b делится на p.

Множество M целых чисел называется i-м классом вычетов по модулю p, если для любого числа t из $M_t = i \pmod p$.

Нетрудно видеть, что все множество целых чисел разбивается на непересекающиеся классы (их окажется ровно p) чисел, сравнимых по модулю p с 0, 1, ..., p-1. На множестве классов K_0, K_1, ..., K_{p-1} введем операции + и *.

$K_i + K_j = K_t$ если d, равное арифметической сумме чисел i и j, сравнимо с t по модулю p.

$K_i * K_j = K_t$ если d, равное целочисленному произведению чисел i и j, сравнимо с t по модулю p.

Легко видеть, что заданное множество классов с введенными операциями образует коммутативное кольцо, которое называется кольцом классов вычетов по модулю p. Данное кольцо является полем, если p — простое число. Обозначение GF(p) — поле Галуа.

Задача дискретного логарифмирования

Пусть задана конечная группа G и ее элемент a, порождающий циклическую подгруппу P (подмножество группы) порядка t.

Пусть задан b — произвольный элемент P.

Требуется найти такое число x (очевидно, меньшее т), чтобы

$a^x = b$

Метод полного перебора

Перебираем все числа i от 0 до т, вычисляем a в степени i и сравниваем с b.

Трудоемкость метода T = O(m).

Метод встречного движения (согласования)

Выберем два числа c и d такие, чтобы их произведение не было меньше т. Тогда любое число *x,* не превосходящее т, можно представить в виде

x = nc + d, n < c, d < c.

Для всех чисел i от 0 до c вычислим значения a в степени i и поместим их в некоторый массив результат:

в 0-й элемент — a

в 1-й элемент — a^2

. . .

в c-й элемент — a^{c-1}

Далее для всех чисел j от 0 до d - 1 вычислим

выражение $t = ba^{jc}$

и сверим его с j-м элементом накопленного массива.

Поскольку

$a^x = a^{nc-d}$

$b = a^{nc}a^d$

$a^d = b^{-nc}a$

то совпадение выражения t с содержимым ячейки, вычисленным в ходе первого этапа, всегда произойдет, и если t совпало со значением элемента r, то x = j c + r.

Логично выбирать c и d как корень квадратный из т.

Тогда трудоемкость метода $T = O(m^{0,5})$ — корневая оценка.

Группа точек эллиптической кривой

Пусть задано простое число $p > 3$. Символом $GF(p)$ будем обозначать конечное простое поле из p элементов, которое можно представлять, как множество целых чисел от 0 до *p-i c* операциями сложения и умножения mod *p*.

Эллиптической кривой E, определённой над полем $GF(p)$, называется множество пар чисел (x, y), x, y е $GF(p)$, удовлетворяющих тождеству

$$y^2 \equiv x^3 + ax + b \pmod{p}, \quad (1)$$

где a, b Е $GF(p)$ и число $4a^3 + 27b^2$ не делится на p.

Пара чисел (x, y), где x, y е $GF(p)$ удовлетворяют тождеству (1), называется точкой кривой E, а числа x и y — *ее* координатами x и y.

Двумерное евклидово пространство над $GF(p)$ можно расширить, добавив бесконечную точку O = (ос, со). Естественным образом можно считать, что координаты этой точки тоже удовлетворяют тождеству (1). Т.е. O тоже можно считать точкой кривой E.

На множестве точек эллиптической кривой £ следующим образом можно осуществить операцию сложения. Пусть координаты точек $Q_1(x_1,y_1)$ и $Q_2(x_2,y_2)$ удовлетворяют условию $x_1 \neq x_2$. В этом случае их суммой будем называть точку $Q_3(x_3,y_3)$, координаты которой определяются сравнениями

$$\begin{cases} x_3 \equiv \Lambda^2 - x_1 - x_2 \pmod{p}, \\ y_3 \equiv \Lambda(x_1 - x_3) - y_1 \pmod{p}, \end{cases} \text{ где } \Lambda \equiv \frac{y_2 - y_1}{x_2 - x_1} t \pmod{p}.$$

Если выполнены равенства $x_1 = x_2$, но $y_1 = y_2 \neq 0$, определим координаты точки Q_3 следующим образом

$$\begin{cases} x_3 \equiv \lambda^2 - 2x_1 \pmod{p}, \\ y_3 \equiv \lambda(x_1 - x_3) - y_1 \pmod{p}, \end{cases} \text{ где } \lambda \equiv \frac{3x_1^2 + a}{2 \cdot 2y_1} \pmod{p}.$$

В случае, если $x_1 = x_2$ и y_1 е $-y_2 \pmod{p}$ суммой точек Q_1 и Q_2 будет описанная выше точка О.

Примечание Часто возникает вопрос: неужели всегда в формулах для вычисления I числитель делится нацело на знаменатель $x_1 - x_2$ и $2y_1$. Нет, не всегда. Здесь приведены формулы вычислений в поле GF(p). И операцию деления тоже нужно выполнять в этом поле, а не в кольце целых чисел. А в поле можно делить на все ненулевые элементы.

Итак, для любых двух точек Q_1 и Q_2 кривой Е определена точка $Q_3 = Q_1 + Q_2$.

Точка О называется нулевой точкой. Для нее выполнены равенства

$$Q + O = O + Q = Q,$$

где Q — произвольная точка эллиптической кривой Е.

Относительно введенной операции сложения множество всех точек эллиптической кривой E, вместе с нулевой точкой, образуют конечную абелеву группу. Нейтральным элементом этой группы является точка O.

Для любой точки Q кривой E и любого натурального числа k можно определить кратную точку

$$kQ = \underbrace{Q + \cdots + Q}_{k}.$$

Стойкость алгоритмов ЭЦП

Стойкость схемы подписи ГОСТ Р 34.10-94 основана на сложности решения частной задачи дискретного логарифмирования в простом поле GF(p). Задачу эту можно сформулировать следующим образом:

- заданы простые числа p, q и натуральное число g < p порядка q, т.е. $g^q \equiv 1 \pmod p$;
- зная значение $y = g^x \pmod p$, необходимо найти подходящее целое x.

В настоящее время наиболее быстрым алгоритмом решения общей задачи дискретного логарифмирования (при произвольном выборе g) является алгоритм обобщенного решета числового поля, временная сложность которого оценивается как

$$o\left(\exp\left(c + o(1)\right)(\ln p)^{1/3}(\ln\ln p)^{2/3}\right)$$ операций в поле GF(p),

где $c \approx \left(\frac{64}{9}\right)^{1/3}$.

Кроме него действенными методами решения частной задачи дискретного логарифмирования считается r-метод и l-метод Полларда и некоторые аналогичные методы, требующие для ее решения выполнения порядка

$$\sqrt{\frac{\pi q}{4}}$$ операций умножения в поле GF(p). (2)

При 1024-битном p и 256-битном q мы получаем приблизительно $1.3*10^{26}$ для метода решета числового поля и $3.0*10^{38}$ для r- и l-методов Полларда.

Стойкость схемы цифровой подписи ГОСТ Р 34.10-2001 основана на сложности решения задачи дискретного логарифмиро-

вания в группе точек эллиптической кривой. Формулируется она следующим образом:

* задана эллиптическая кривая E над полем GF(p), где p — простое число;
* выбрана точка G, имеющая простой порядок q в группе точек кривой E;
* зная точку kG, необходимо восстановить натуральное число k.

В настоящее время наиболее быстрыми алгоритмами решения задачи дискретного логарифмирования в группе точек эллиптической кривой при правильном выборе параметров считаются r-метод и l-метод Полларда и некоторые аналогичные. Их сложность оценивается той же формулой (2) и, стало быть, тем же числом $3{,}0*10^{38}$ операций сложения точек кривой. Сравнив с $1{,}3*10^{26}$ можно сказать, что новый ГОСТ предлагает более стойкую схему цифровой подписи.

Предметный указатель

Список литературы для углубленного изучения

1. Аршинов М.Н. Садовский Л.Е. Коды и математика М.: Наука, 1983.
2. Диффи У, Хеллмен Э. Новое направление в криптографии //ТИИЭР. IT-22, 1976.
3. Диффи У., Хеллмен Э. Защищенность и имитостойкость: введение в криптографию //ТИИЭР, т. 67, 3, 1979.
4. Лидл Р., Нидеррайтер Г. Конечные ноля: Пер с англ. М.: Мир, 1988. Т. 1, 425 с., т. 2, 390 с.
5. Перельман Н.Я. Живая математика. Гл.6. М.:Наука,1978.
6. Сачков В.Н. Введение в комбинаторные методы дискретной математики. М.: Наука,1982.
7. Фролов Г. Тайны тайнописи. М.: Инфосервис Экспресс Лтд.,1992, 125 с.
8. Шеннон К.Э. Математическая теория связи. В кн.: Работы по теории информации и кибернетике. М.: ИЛ, 1963.
9. Шеннон К.Э. Теория связи в секретных системах, В кн.: Работы по теории информации и кибернетике. М.: ИЛ, 1963.
10. Система обработки информации. Защита криптографическая. Алгоритм криптографического преобразования. ГОСТ 28147-89. Госкомстат. М.: 1989.
11. А.Винокуров. ГОСТ не прост, а ... очень прост. М.: Монитор, №1, 1995.
12. Программно-аппаратные средства обеспечения информационной безопасности. Защита программ и данных: Учебное пособие для вузов/ П.Ю.Белкин, О.О. Михальский, А.С.Першаков и др. М.: Радио и связь, 1999.
13. ГОСТ Р 34.10–94. Государственный стандарт Российской Федерации. Криптографическая зашита информации. Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма.
14. ГОСТ Р 34.11 -94. Государственный стандарт Российской Федерации. Криптографическая защита информации. Функция хеширования.
15. Д. Кнут. Искусство программирования на ЭВМ. Том 2. Получисленные алгоритмы.

16. А. Ахо, Дж. Хопкрофт, Дж. Ульман. Построение и анализ вычислительных алгоритмов: Пер. с англ. М.: Мир, 1986.
17. А, Акритас. Основы компьютерной алгебры с приложениями: Пер. с англ. М.: Мир, 1994.
18. В.В.Шорин. Программная реализация системы электронной цифровой подписи и оптимизация скорости ее работы на платформе Digital Alpha. Магистерская диссертация. М.: МФТИ.
19. В.В.Липаев. Программно-технологическая безопасность информационных систем. Информационный бюллетень Jet Info № 6/7, 1997.
20. Э.Дейкстра. Дисциплина программирования: Пер. с англ. М.: Мир, 1978.
21. Бейбер Р.Л. Программное обеспечение без ошибок: Пер. *с* англ./ Под ред. Д.И.Правикова. М.: Джон Уайли энд Санз, Радио и связь, 1996.
22. Семьянов П.В. Почему криптосистемы ненадежны? Тезисы доклада на конф. «Методы и технические средства обеспечения безопасности информации», СПбГТУ, 1996.
23. Peter L. Montgomery. Modular multiplication without trial division, Mathematics of Computation, v.44, n.170, 1985, pp. 519-521.
24. B. Schneier. Applied Cryptography, Second Edition, 1996.
25. R.L. Rivest. A Description of the RC2(r) Encryption Algorithm, RFC 2268, March 1998.
26. B.S. Kaliski. The MD2 Message Digest Algorithm, RFC 1319, Apr 1992.
27. R.L. Rivest. The MD4 Message Digest Algorithm, RFC 1320, Apr 1992.
28. R.L. Rivest. The MD5 Message Digest Algorithm, RFC 1321, Apr 1992.
29. H. Krawczyk, M. Bellare, R. Canetti. HMAC: Keyed-Hashing for Message Authentication, RFC 2104, Feb 1997.
30. R. Housley. Cryptographic Message Syntax, RFC 2630, June 1999.
31. E. Rescorla. Diffie-Hellman Key Agreement Method, RFC 2631, June 1999.
32. R. Zuccherato. Methods for Avoiding the «Small-Subgroup» Attacks on the Diffie-Hellman Key Agreement Method for S/MIME, RFC 2785, March 2000.
33. RSA Laboratories. PKCS ffl: RSA Encryption Standard, version 2.0, Oct 1998.
34. RSA Laboratories. PKCS #3: Diffie-Hellman Key-Agreement Standard, version 1.4, Nov 1993.
35. RSA Laboratories. PKCS #5: Password-Based Cryptography Standard, version 2.0, March 1999.
36. RSA Laboratories. PKCS #7: Cryptographic Message Syntax Standard, version 1.5, Nov 1993.

Об авторах

Домашев Алексей Владимирович — один из ведущих российских специалистов-практиков в области использования криптографических интерфейсов и систем цифровой подписи, соавтор ряда основополагающих трудов по прикладной криптографии.

Щербаков Андрей Юрьевич — один из ведущих российских специалистов по компьютерной безопасности и прикладной криптографии, автор основополагающих научных и учебных трудов в данной области, академик РАЕН, доктор технических наук, профессор.

Рецензенты: Московское представительство Microsoft, Московский институт электроники и математики, кафедра №42 «Криптология и дискретная математика» МИФИ.

Благодарности

Авторы выражают глубокую признательность сотрудникам московского представительства Microsoft Алексею Чубарю и Дмитрию Вешнякову, а также представителям организаций-рецензентов Тамаре Петровой и Игорю Прокофьеву.

Щербаков Андрей Юрьевич,
Домашев Алексей Владимирович

Прикладная криптография

Использование и синтез криптографических интерфейсов

Редактор *Ю. П. Леонова*

Технический редактор *Н. Г. Тимченко*

Компьютерная верстка *В. Б. Хильченко*

Дизайнер обложки *Е. В. Козлова*

Оригинал-макет выполнен с использованием издательской системы Adobe PageMaker 6.0

Главный редактор *А. И. Козлов*

Подготовлено к печати издательством «Русская Редакция»
121087, Москва, ул. Заречная, д.9
тел.: (095) 142-0571, тел./факс: (095) 145-4519
e-mail: info@rusedit.ru, http://www.ruscdit.ru

РУССКАЯ РЕДАКЦИЯ

Подписано в печать 11.11.02 г. Тираж 2 000 экз.
Формат 84x108/32. Физ. п. л. 13
Отпечатано в ОАО «Типография «Новости»
107105, Москва, ул. Фр. Энгельса, 46