Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего образования
«Пермский национальный исследовательский
политехнический университет»

Ю.Н. Липин, С.А. Сторожев

КРИПТОГРАФИЧЕСКАЯ ЗАЩИТА «ПОСЛЕДНЕЙ МИЛИ» СРЕДСТВАМИ WIN API

Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия

Издательство
Пермского национального исследовательского
политехнического университета
2021

УДК 004.056
Л612

Рецензенты:
канд. физ.-мат. наук, доцент кафедры информационной безопасности и систем связи *А.П. Шкарапута*
(Пермский государственный национальный исследовательский университет);
д-р техн. наук, профессор кафедры «Автоматика и телемеханика» *В.И. Фрейман*
(Пермский национальный исследовательский политехнический университет)

Липин, Ю.Н.

Л612 Криптографическая защита «последней мили» средствами WIN API : учеб. пособие / Ю.Н. Липин, С.А. Сторожев. – Пермь : Изд-во Перм. нац. исслед. политехн. ун-та, 2021. – 116 с.

ISBN 978-5-398-02566-8

Передача данных по туннелю через Интернет осуществляется через ранее зашифрованную информацию, таким образом через OpenVPN пропускается дважды зашифрованная информация. Исходя из этого, файлы в папках уже зашифрованы ранее, т.е. до момента их подачи на вход туннеля OpenVPN. Данная организация передачи данных относится к проблеме защиты «последней мили».

Предлагаемый материал удаленного защищенного доступа будет полезен учащимся средних общеобразовательных школ в качестве пособия для факультативных занятий, учащимся специализированных колледжей и лицеев, а также студентам вузов по соответствующим специальностям.

УДК 004.056

ISBN 978-5-398-02566-8

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Информационная безопасность означает выполнение требований к информации в виде конфиденциальности, целостности, авторства и правил допуска. Для соблюдения этих свойств используются соответствующие законы, положения, инструкции и криптографическое программное обеспечение. Для защиты информации в каналах передачи в системе Интернета используется, например, проект с открытым кодом (типа OpenVpn и не только), который функционирует в режиме «клиент-сервер», но за сохранность информации вне клиент-сервера последний не отвечает. В системах передачи данных давно широко используется в области информационной безопасности термин «последняя миля». На наш взгляд, наиболее приемлемым вариантом защиты информации являлся бы симбиоз закодированной парольной защиты папок, шифрование находящейся в них информации на базе WinApi и использование защищенных каналов передачи данных на основе OpenVpn. Таким образом, предлагается двойное шифрованинвсе: все, что хранится в папках, шифруется с помощью проекта WINApi, в OpenVpn используется свое шифрование, после получения пакета OpenVpn происходит его внутреннее расшифрование, и в папки получателя размещается шифрованный текст WinNApi. Сам проект WinNApi создан в Delphi, авторы имеют значительный опыт программирования в Delphi, и потому сам проект создан в среде программирования Delphi. Основные алгоритмы можно найти в Windows/System 32/, они имеют несколько названий: cryptngc.dll, CryptoWinRT.dll, cryptsp.dll и т.д. Доступ к ним возможен только через WinNApi., что и выполнено в данном проекте.

1. ПРОБЛЕМЫ ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ «ПОСЛЕДНЕЙ МИЛИ»

«Последняя миля» – канал, соединяющий конечное (клиентское) оборудование с узлом доступа провайдера (оператора связи). Например, при предоставлении услуги подключения к Интернету последний километр – это участок от порта коммутатора провайдера на его узле связи до порта маршрутизатора клиента в его офисе. Для услуг коммутируемого (dial-up, дайлапного) подключения последний километр – это участок между модемом пользователя и модемом (модемным пулом) провайдера. В последнюю милю обычно не включается разводка проводов внутри здания. Термин используется в основном специалистами из отрасли связи. К технологиям последней мили обычно относят xDSL, FTTx, Wi-Fi, WiMax, DOCSIS, связь по ЛЭП. К оборудованию последней мили можно отнести xDSL-модемы, мультиплексоры доступа, оптические модемы и преобразователи, радиомультиплексоры. Применительно к информационной безопасности под последней милей предлагается понимать участок от папки с файлами OPenVpn до Remote <IP адрес сервера> клиента и от туннельного шлюза сервера <172.16.0.1или 10.8.0.1> до папок с расшифрованным пакетом полученных данных.

Проблема последнего километра всегда была актуальной задачей для связистов. К настоящему времени появилось множество технологий последней мили, и перед любым оператором связи стоит задача выбора технологии, оптимально решающей задачу предоставления связи для своих абонентов. Универсального решения этой задачи не существует, у каждой технологии есть свои решения проблемы последней мили.

Статья о повсеместных вычислениях, в которой впервые был поднят вопрос обеспечения конфиденциальности сложных систем, была опубликована более десяти лет тому назад [1], но ее основные идеи вполне применимы и к Интернету вещей сегодня. При работе со своим компьютером у нас создается иллюзия контроля, поскольку

в принципе мы можем определить, какие приложения на нем работают и какие данные они собирают. А в Интернете вещей традиционные методы контроля практически бесполезны. Можно даже привести примеры ситуаций, когда люди являются уже не пользователями сервиса Интернета вещей, а исследуемыми им объектами – скажем, если речь идет о системе мониторинга уровня шума в «умном» городе. Другой пример – публичные сети Wi-Fi, пользователи которых в большинстве своем даже понятия не имеют о реальных параметрах их конфиденциальности.

Одна из главных проблем приватности Интернета вещей состоит в том, что пользователи не всегда знают, когда то или иное устройство собирает их персональные данные, а если такие устройства повсеместны, то факт наличия или отсутствия датчиков почти не поддается проверке. Вместе с тем, согласно основополагающим принципам приватности, сбор персональных данных должен происходить только при условии соответствующего уведомления. Каким же должен быть «стек приватности» – система мер для решения задач, связанных с извещением пользователей относительно намерений окружающих его устройств Интернета вещей.

Стек приватности. Многие аспекты сложных систем, например, обнаружение, удобство использования и приватность, непосредственно касаются конечных пользователей. Как именно люди должны узнавать о присутствии сервисов в публичных местах, на работе, дома? Как они могут выяснять характеристики приватности этих сервисов без перегрузки техническими терминами? Покажем на примерах, как можно было бы предоставлять уведомления о приватности Интернета вещей. Допустим, на парковке торгового центра установили камеры системы безопасности. Но вместо табличек с описанием назначения камер применяются радиомаячки, взаимодействующие со смартфонами прохожих. Маячок передает идентификатор, по которому можно выполнить поиск в облаке с помощью мобильного приложения. Таким образом пользователи получают уведомление о присутствии камер, причем это не только необходимое предупреждение, но и способ создать у людей ощущение безо-

пасности пребывания на территории. Концепция стека приватности, призванного повысить степень комфорта пользователей при их взаимодействии с системами Интернета вещей.

Стек приватности можно реализовать с помощью систем «умного» дома – например, с помощью колонки наподобие Amazon Echo, обеспечивающей возможность голосового управления музыкой, освещением и т.п. Гостям дома может быть не комфортно из-за наличия в доме системы, непрерывно записывающей их высказывания, и если гости укажут, что хотели бы получать предупреждения о записи звука в месте их пребывания, то хозяева могут попросить разрешения использовать такой сервис. В зависимости от предпочтений гостей они могут либо получить предупреждение при входе в дом, либо система может временно или частично отключить сервис. Все это может происходить автоматически или в интерактивном режиме.В этих примерах пользователи не знают, какие именно сервисы из их окружения активны, но можно описать сценарии применения стека приватности, задуманного как механизм уведомления, основанный на существующих сегодня протоколах Интернета вещей.

Осведомленность. Часть стека под названием «Осведомленность» отвечает за то, чтобы пользователи (или соответствующие агентские средства) могли выяснить характеристики приватности сервисов. Сложность заключается в том, что факт сбора данных в Интернете вещей незаметен, пользователи обычно не знают, что сведения о них собираются, поэтому при проектировании систем нужно предусмотреть возможность определить, в какой степени приватно то или иное пространство, чтобы знать, насколько откровенно можно в нем себя вести [2]. В связи с этим надо сделать так, чтобы, входя в помещение, человек мог в течение 30 секунд достоверно определять наличие всех датчиков и потоков данных. Осведомленность в основном касается вопроса о том, по каким каналам сервисы Интернета вещей могут выходить на связь с пользователями и объектами анализа. Например, можно пользоваться визуальными сигналами о сборе данных или сетевыми протоколами для пе-

редачи сведений от сервиса к пользовательским устройствам. Сегодня в рамках ряда отраслевых инициатив, например Open Connectivity Foundation, идет работа над стандартами интероперабельности Интернета вещей, обеспечивающими возможности обнаружения устройств и налаживания связи между ними.

Apple iBeacon – один из примеров технологии обнаружения устройств. Маячок передает по Bluetooth идентификатор, который совместимое устройство может использовать для получения ассоциированной с идентификатором информации (например, для позиционно-зависимых услуг). Маячки также позволяют определить местонахождение человека вблизи устройств на какой-либо территории. На сегодня разработка протоколов Интернета вещей еще не дошла до этапа стандартизации метаданных о приватности, которые бы позволили сервисам универсальным образом объявлять соответствующие политики. Преимуществом подобного подхода в мире повсеместных сервисов стало бы то, что решения, связанные с приватностью, можно было бы принимать при минимальной когнитивной нагрузке. Стандартов метаданных приватности по ряду причин пока нет даже для WWW. В принципе можно было бы обойтись и без стандартизации, например, реализовав средства обработки естественного языка, способные «понимать» политику конфиденциальности, и такие проекты уже есть. Однако в мире Интернета вещей риски сбора информации без ведома субъектов сервисов выше, поэтому стандартный «язык» приватности необходим.

Умозаключения. С помощью протоколов осведомленности сервисы Интернета вещей могут объявлять о своих действиях, но что именно включать в такие объявления? Можно было бы просто сообщить, какие показания собираются и для чего они будут использоваться, но этого может быть недостаточно. Пользователи не всегда понимают, какие выводы можно сделать из таких данных, а способности систем к выводу умозаключений со временем будут становиться все совершеннее. Сегодня, например, «дактилоскопия» устройств и браузеров (идентификация по набору характеристик) стала объективной реальностью WWW, и ясно, что в мире Интерне-

та вещей подобный сбор идентифицирующих сведений ведется не менее активно (например, установление закономерностей в перемещении человека или запись его голоса с целью аутентификации). Умозаключения напрямую влияют на приватность, но непонятно, как именно уведомлять о них, в частности, о вероятностных и сделанных с помощью косвенных данных. Можно предложить сбалансированный подход: поскольку пользователи не всегда понимают цели использования собранных данных, сервисы обязаны открыто сообщать об основных умозаключениях, которые они делают. В то же время нужно иметь в виду, что возможности делать выводы постоянно растут – не только из-за роста объема доступных данных, но и благодаря развитию методов машинного обучения. Сервисы Интернета вещей могли бы уведомлять об умозаключениях в рамках политики приватности; к примеру, сервис, собирающий данные GPS, должен объявить, что по ним можно установить личность, а устройство, регистрирующее сведения о расходе электроэнергии, обязано указать, что оно способно делать выводы о привычках абонента.

Зная вероятные умозаключения, пользователи смогут не только прояснить, какие сведения система о них собирает, но и повысить приватность, установив соответствующие ограничения. Допустим, система выполняет видеосъемку, параллельно регистрируя отметки времени и сведения в сетях Wi-Fi, в этом случае пользователь мог бы дать согласие вести слежение на работе, но не дома. Можно было бы реализовать политики безопасности для исходных данных, но только после того, как система по ним поймет, какие из них относятся к работе, а какие – к дому. Этот пример хорошо иллюстрирует дистанцию между сырыми данными и языком предпочтений, выражаемых с помощью более высокоуровневых понятий вроде «дом». Умозаключения позволяют устранить этот разрыв, помогая увеличить полезность системы при соблюдении ее дружелюбности.

Предпочтения конфиденциальности. Предположим, у системы есть достаточное понимание того, какие данные собирает сервис Интернета вещей и какие выводы он может из них сделать. Что

конкретно при этом волнует объект сбора данных (человека)? Влияет ли тот или иной сценарий сбора данных на приватность, зависит ли от конкретного человека, но и контекст имеет значение. Например, против записи звука в ресторане может не быть возражений в зависимости от того, кто именно ее выполняет (ресторан или ваш друг), и от того, с кем вы находитесь в заведении (с друзьями или деловыми партнерами). Контекст также может определять, что воспринимается неожиданным, а что – обычным. Например, видеонаблюдение на футбольном стадионе может представляться нормой, а в ресторане – нет. Поскольку на решение о приватности могут повлиять очень многие факторы, главный вопрос состоит в том, какие из них действительно имеют значение для людей в контексте Интернета вещей. Понимание всех значимых факторов поможет разработчикам систем предоставлять сведения, необходимые для принятия решений о конфиденциальности. Знание факторов, которые могут быть важными для тех или иных пользователей, также может повлиять на архитектуру системы. К примеру, сервис Интернета вещей в баре для успокоения посетителей может сообщать им, что видеосъемка выполняется только в целях проверки возраста.

Уведомление. Последний уровень стека отвечает за уведомление конечного пользователя. Предоставление уведомлений зависит от осведомленности о наличии сервисов в конкретной среде, выводимых умозаключений и предпочтений пользователя. Уведомление – относительно редкое событие, так как пользователи имеют дело лишь с малой частью сервисов Интернета вещей; кроме того, после первичного предупреждения повторять его нужно не всегда. Например, для водителей будет полезным предупреждение о том, что автомобиль въезжает на территорию, где за его перемещением могут следить, но получать каждый день уведомление об одной и той же уличной камере смысла нет. По поводу уведомлений возникают следующие вопросы: как система должна предоставлять предупреждения, как пользователю указывать свои предпочтения относительно предупреждений и как выяснять эти предпочтения. В частности, важно отметить, что если предупреждение отправлено не

приватно, то это может позволить посторонним делать нежелательные выводы об истории действий или предпочтениях пользователя. Кроме того, технологии и алгоритмы умозаключений будут совершенствоваться, в связи с чем тех, кто уже получал сообщения о каком-либо сервисе, возможно, придется уведомлять повторно. Уведомление – последний уровень стека приватности, и именно здесь происходит реальное взаимодействие с пользователями, которое может принимать разные формы. С одной стороны, уведомление требует прерывания текущей деятельности пользователя, с другой – более активные пользователи могут быть заинтересованы в наглядном отображении параметров приватности ближайших сервисов Интернета вещей. Как именно можно было бы визуализировать окружающие сервисы, их характеристики приватности и надежности?

Реализация стека. Опишем, как можно было бы реализовать стек на уровне «посредника» или ассистента по обеспечению приватности, работающего на пользовательском устройстве. На уровне осведомленности работают протоколы, позволяющие получать метаданные приватности от ближайших сервисов Интернета вещей. Такие метаданные могут в числе прочего содержать базовые сведения об умозаключениях. Кроме того, посредник может сам по себе обладать аналитическими способностями, позволяющими ему делать выводы на основе метаданных. Посредник отвечает за управление личными настройками приватности, предоставляя пользователю возможность указывать соответствующие предпочтения или делая выводы о них с учетом прошлых решений и демографических характеристик. Сохраняя предпочтения и решения пользователя и применяя средства аналитики, посредник может выдавать предупреждения только тогда, когда это уместно. Сбор личных данных возможен только при условии предоставления соответствующего предупреждения и получения согласия – это основополагающий принцип обеспечения конфиденциальности. Но реализовать этот принцип сложно даже для традиционных ИТ, не говоря уже о мире Интернета вещей, учитывая отсутствие универсального канала связи с пользователем. Человек легко может оказаться в среде, где его

персональные данные обрабатываются некими сервисами, которые он не устанавливал и об активности которых не подозревает. Предложенный концептуальный стек приватности основан на текущих знаниях о взаимодействии разрабатываемых сегодня систем Интернета вещей с пользователями, и исследования, направленные на реализацию элементов такого стека, уже ведутся. К таким элементам относятся пользовательские интерфейсы для указания и выяснения предпочтений, уведомления и отображения параметров приватности. Также исследуется проблема преобразования сведений о собираемых датчиками сырых данных в информацию об умозаключениях. Еще одна область изучения – принципы описания параметров приватности и репрезентации всевозможных сервисов Интернета вещей и их политик конфиденциальности. То и другое важно для автоматической фильтрации данных. Обеспечение конфиденциальности на «последней миле» Интернета вещей – одно из самых актуальных сегодня направлений исследований. Поиски решения этой задачи ведутся на пересечении машинного обучения, человеко-компьютерного взаимодействия и систем безопасности. Данный материал заимствован из приведенного в конце издания списка литературы. Решение проблемы может заключаться в шифровании данных различных журналов событий, и это относится к функциональным возможностям администраторов систем.

2. DELPHI И WINDOWS API ДЛЯ ЗАЩИТЫ СЕКРЕТОВ

2.1. Общие положения

Приведенные ниже алгоритмы предназначены для ознакомления студентов с основными принципами, составляющими ядро методов защиты информации. Они же по своим функциям определяют профессиональные системы защиты информации, на основании которых реализованы ГОСТы, принятые в России. К такому пакету относится приложение GryptoApi, входящее в состав библиотек Windows.. В семействе Windows, начиная с Windows 95, обеспечивается реализация шифрования, генерации ключей, создания и проверки цифровых подписей и других криптографических задач. Эти функции необходимы для работы операционной системы, однако ими может воспользоваться и любая прикладная программа – для этого программисту достаточно обратиться к нужной подпрограмме так, как предписывает криптографический интерфейс прикладных программ (CryptoAPI).

Разумеется, по мере совершенствования Windows расширялся и состав ее криптографической подсистемы. Помимо базовых операций в настоящее время в CryptoAPI 2.0 поддерживается работа с сертификатами, шифрованными сообщениями в формате PKCS. Опишем круг задач, на решение которых ориентирован Crypto API: существующие системы электронного документооборота «Верба, Omega, Сигнатура» в качестве основного ядра защиты информации используют CryptoAPI. Основные функции этой подсистемы:

- надежное сокрытие данных;
- возможность передачи сокрытых данных третьим лицам;
- надежная система проверки достоверности пришедшей от третьих лиц информации;
- расшифровывание полученных конфиденциальных данных;
- работа с «идентификационными удостоверениями» третьих лиц;

- обеспечение работы с признанными криптографическими стандартами;
- возможность расширения и работы с пока еще неизвестными алгоритмами.

Реализация всех алгоритмов (шифрования, цифровой подписи и т.п.) полностью выведена из состава самого Crypto API и реализуется в отдельных, независимых динамических библиотеках – «криптопровайдерах» (Cryptographic Service Provider – CSP). Сам же Crypto API просто предъявляет определенные требования к набору функций (интерфейсу) криптопровайдера и предоставляет конечному пользователю унифицированный интерфейс работы с CSP. Конечному пользователю для полноценного использования всех функций криптопровайдера достаточно знать его строковое имя и номер типа.

В программных решениях рано или поздно встает вопрос стандартизации передаваемых между приложениями данных. В сфере криптографии для решения данного вопроса применяют набор стандартов «PKCS», предложенный компанией RSA Security. В данном комплекте стандартов учитываются все возможные случаи, возникающие в криптографических приложениях. Предусмотрены стандарты для обмена сертификатами, зашифрованными и подписанными данными и многое другое. Crypto API как основная библиотека для обеспечения работы с криптографическими данными в Windows также достаточно полно поддерживает данный комплект стандартов и позволяет формировать криптографические приложения, которые могут быть обработаны в дальнейшем любыми программными продуктами.

Таким образом, мы можем разделить весь интерфейс Crypto API на 5 функциональных групп:

1. Базовые криптографические функции:

- функции шифрования/расшифровывания данных;
- функции хеширования и получения цифровой подписи данных;
- функции инициализации криптопровайдера и работы с полученным контекстом;

- функции генерации ключей;
- функции обмена ключами.

2. Функции кодирования/декодирования. Под кодированием в данном случае подразумевается получение на выходе информации, кодированной в формате ASN.1 (Abstract Syntax Notation One).

3. Функции работы с сертификатами.

4. Высокоуровневые функции обработки криптографических сообщений.

5. Низкоуровневые функции обработки криптографических сообщений.

2.2. Взаимодействие с CryptoAPI. Криптопровайдеры

Функции CryptoAPI можно вызвать из программы, написанной на Delphi, на языке C++, MS Visual C++.

Код функций криптографической подсистемы содержится в нескольких динамически загружаемых библиотеках Windows (advapi32.dll, crypt32.dll). Для обращения к такой функции из прикладной программы следует объявить ее как внешнюю. Заголовок функции в интерфейсной части модуля будет выглядеть, например, так:

function CryptAcquireContext (phPROV: PHCRYPTPROV; pszContainer: LPCTSTR;pszProvider: LPCTSTR;dwProvType: DWORD; dwFlags: DWORD): BOOL;

а в исполняемой части вместо тела функции нужно вписать директиву extern с указанием библиотеки, в которой содержится функция и, возможно, ее имени в этой библиотеке (если оно отличается от имени функции в создаваемом модуле), например:

function CryptAcquireContext; external ‘advapi32.dll’
name 'CryptAcquireContextA';

Таким образом, имея описание функций CryptoAPI, можно собрать заголовки функций в отдельном модуле, который будет обеспечивать взаимодействие прикладной программы с криптографической подсистемой. Разумеется, такая работа была проделана программистами Microsoft, и соответствующий заголовочный файл (wincrypt.h) был включен в поставку MS Visual C++, в Delphi-

версию (wcrypt2.pas). Подключив модуль к проекту, вы сможете использовать не только функции CryptoAPI, но и мнемонические константы режимов, идентификаторы алгоритмов и прочих параметров, необходимых на практике.

Функции CryptoAPI обеспечивают прикладным программам доступ к криптографическим возможностям Windows. Однако они являются лишь «передаточным звеном» в сложной цепи обработки информации. Основную работу выполняют скрытые от глаз программиста функции, входящие в специализированные программные (или программно-аппаратные) модули – провайдеры (поставщики) криптографических услуг (CSP – Cryptographic Service Providers), или криптопровайдеры. Программная часть криптопровайдера представляет собой dll-файл, подписанный Microsoft; периодически Windows проверяет цифровую подпись, что исключает возможность подмены криптопровайдера. Криптопровайдером называют независимый модуль, обеспечивающий непосредственную работу с криптографическими алгоритмами. Каждый криптопровайдер должен обеспечивать:

- реализацию стандартного интерфейса криптопровайдера;
- работу с ключами шифрования, предназначенными для обеспечения работы алгоритмов, специфичных для данного криптопровайдера;
- невозможность вмешательства третьих лиц в схему работы алгоритмов.

Как уже излагалось выше, сами криптопровайдеры реализуются в виде динамически загружаемых библиотек (DLL). Таким образом, достаточно трудно повлиять на ход алгоритма, реализованного в криптопровайдере, поскольку компоненты криптосистемы Windows (все) должны иметь цифровую подпись (т.е. подписывается и DLL криптопровайдера). У криптопровайдеров должны отсутствовать возможности изменения алгоритма через установку его параметров. Таким образом решается задача обеспечения целостности алгоритмов криптопровайдера. Задача обеспечения целостности ключей шифрования решается с использованием контейнера ключей, о котором рассказывается ниже.

Функции работы с криптопровайдерами можно разделить на следующие группы:

- функции инициализации контекста и получения параметров криптопровайдера;
- функции генерации ключей и работы с ними;
- функции шифрования/расшифровывания данных;
- функции хеширования и получения цифровой подписи данных.

В группу функций инициализации контекста входят следующие функции:

- CryptAcquireContext. С помощью данной функции в первую очередь производится инициализация контекста криптопровайдера (получение ссылки на HANDLE, которую в дальнейшем можно использовать в других функциях). Также с помощью последнего параметра данной функции можно создать или удалить контейнер ключей;
- CryptContextAddRef. Данная функция служит для увеличения внутреннего счетчика ссылок криптопровайдера. Эту функцию рекомендуется использовать при передаче контекста криптопровайдера в качестве члена различных структур, передаваемых функциям;
- CryptReleaseContext. Данная функция предназначена для освобождения контекста криптопровайдера, полученного с помощью функции CryptAcquireContext. Фактически производится только уменьшение внутреннего счетчика ссылок криптопровайдера (нечто вроде механизма подсчета ссылок у COM-объекта). Когда внутренний счетчик ссылок становится равным нулю, данный контекст криптопровайдера полностью освобождается и не может быть более нигде использован;
- CryptGetProvParam. С помощью этой функции можно получить значения различных параметров криптопровайдера. Нужно заметить, что для всех криптопровайдеров стандартом определен лишь ограниченный набор параметров. Набор параметров криптопровайдера может сильно варьироваться в зависимости от реализации криптопровайдера.

В группу генерации и работы с ключами входят следующие функции:

- CryptGenKey. Данная функция предназначена для генерации сессионного ключа (ключ, используемый только в течение текущей сессии работы), а также для генерации пар ключей для обмена (публичный и закрытый ключ) и цифровой подписи;
- CryptDuplicateKey. Функция предназначена для копирования ключа;
- CryptGetUserKey. Функция предназначена для получения значения публичного ключа для указанного контейнера ключей. Используется для получения значений публичных ключей, предназначенных для обмена ключами и цифровой подписи;
- CryptDestroyKey. Функция предназначена для освобождения ранее полученного хэндла ключа. Функцию следует вызывать всегда для предотвращения утечек памяти в приложении;
- CryptGetKeyParam. Функция предназначена для получения различных параметров ключа. В качестве параметров используются алгоритм ключа (его цифровой обозначение в системе), флаги разрешения использования ключа (например, если отсутствует установленный флаг ключа CRYPY_ENCYPT, данным ключом невозможно будет зашифровать данные), данные о длине блока ключа и многое другое. В практических реализациях данной функции следует уделять достаточно большое внимание, так как зачастую именно от параметров ключа зависит работа используемого алгоритма криптопровайдера;
- CryptSetKeyParam. Функция, обратная предыдущей. Используется для установки параметров ключа;
- CryptDeriveKey. Функция предназначена для генерации сессионного ключа на основе хеша данных, т.е. данная функция генерирует один и тот же сессионный ключ, если ей передаются одинаковые значения хеша данных. Функция полезна в случае генерации сессионного ключа на основе пароля;
- CryptGenRandom. Функция используется для заполнения переданного ей буфера случайными данными. Используется, например, для генерации нового имени контейнера ключей;

- CryptExportKey. Функция экспорта ключа для его передачи по каналам информации. Возможны различные варианты передачи ключа, включая передачу публичного ключа пары ключей, а также передачу секретного или сессионного ключа;
- CryptImportKey. Функция, обратная предыдущей. Предназначена для получения из каналов информации значения ключа.

В группу функций шифрования/расшифровывания данных входят:

- CryptEncrypt. Основная базовая функция шифрования данных. В качестве параметров использует ранее полученные контексты криптопровайдера и сессионного ключа. Данные, генерируемые на выходе этой функции, не являются форматированными и не содержат никакой другой информации, помимо шифрованного контента (в отличие от, например, стандарта PKCS #7, использующего совместную передачу кодированных данных и экспортированного сессионного ключа);
- CryptDecrypt. Основная базовая функция расшифровывания данных. В качестве параметров используются ранее полученные контекст криптопровайдера и хэндл сессионного ключа.

В группу функций хеширования и получения цифровой подписи входят:

- CryptCreateHash. Функция, создающая хеш-объект, предназначенный для генерации хеш-значения данных. В качестве основных параметров принимает ранее полученные контекст криптопровайдера и алгоритм формирования хеша данных;
- CryptHashData. Основная функция хеширования данных. Самым важным параметром этой функции является ссылка на хешируемые данные;
- CryptGetHashParam. Функция используется в основном для получения значения сформированного хеша данных. Функция CryptGetHashParam завершает процедуру создания хеш-значения, и дальнейшие вызовы функции CryptHashData будут возвращать ошибку;

- CryptSetHashParam. Функция используется для установки параметров хеша. Может быть использована, например, для изменения алгоритма формирования хеша;
- CryptDestroyHash. Функция используется для освобождения хеш-объекта;
- CryptDuplicateHash. Функция получения копии хеша. Используется при передаче хеша между функциями;
- CryptSignHash. Основная базовая функция получения цифровой подписи данных. Нужно отметить, что в Crypto API для уменьшения длины цифровой подписи (и ускорения работы ассиметричных алгоритмов, используемых для формирования цифровой подписи) в качестве входных данных используют хеш. Функций, использующих для получения цифровой подписи сами данные, в Crypto API нет;
- CryptVerifySignature. Основная базовая функция проверки цифровой подписи. В качестве входных данных опять-таки используется значение хеша.

В Crypto API криптопровайдеры принято группировать по их названиям (строковые величины), а также по номерам их типов. Тип криптопровайдера в общем случае ничего не сообщает обычному пользователю и служит лишь для вспомогательной группировки провайдеров. Исключение составляет тип PROV_RSA_FULL (его номер – 1), который присваивают себе только те криптопровайдеры, которые полностью поддерживают работу со стандартом RSA. Сначала получим полный перечень строковых имен криптопровайдеров. Наиболее простым способом для этого является использование функции CryptEnumProviders. Для перечисления типов криптопровайдеров, установленных в системе, можно использовать функцию CryptEnumProviderTypes. Но, к сожалению, использование данных функций иногда вызывает трудности, так как в Windows 98 эти функции не работают. В качестве выхода из ситуации используют прямое обращение к реестру для перечисления как всех криптопровайдеров, так и их.

2.3. Шифрование с использованием паролей

Для шифрования данных в CryptoAPI применяются симметричные алгоритмы. Симметричность означает, что для шифрования и расшифровки данных используется один и тот же ключ, известный как шифрующей, так и расшифровывающей стороне. При этом плохо выбранный ключ шифрования может дать противнику возможность взломать шифр. Поэтому одной из функций криптографической подсистемы должна быть генерация «хороших» ключей либо случайным образом, либо на основании некоторой информации, предоставляемой пользователем, например пароля.

В случае создания ключа на основании пароля должно выполняться следующее обязательное условие: при многократном повторении процедуры генерации ключа на одном и том же пароле должны получаться идентичные ключи. Ключ шифрования имеет, как правило, строго определенную длину, определяемую используемым алгоритмом, а длина пароля может быть произвольной. Даже интуитивно понятно, что для однозначной генерации ключей нужно привести разнообразные пароли к некоторой единой форме. Это достигается с помощью хеширования.

При соблюдении приведенных условий хеш-значение служит компактным цифровым отпечатком (дайджестом) сообщения. Существует множество алгоритмов хеширования. CryptoAPI поддерживает, например, алгоритмы MD5 (MD – Message Digest) и SHA (Secure Hash Algorithm).

Итак, чтобы создать ключ шифрования на основании пароля, нам нужно вначале получить хеш этого пароля. Для этого следует создать с помощью CryptoAPI хеш-объект, воспользовавшись функцией CryptCreateHash (провайдер, ID_алгоритма, ключ, флаги, хеш), которой нужно передать дескриптору криптопровайдера (полученный с помощью CryptAcquireContext), и идентификатор алгоритма хеширования (остальные параметры могут быть нулями). В результате мы получим дескриптор хеш-объекта. Этот объект можно представить себе как черный ящик, который принимает любые данные и «перемалывает» их, сохраняя внутри себя лишь хеш-значение. По-

дать данные на вход хеш-объекта позволяет функция CryptHashData (дескриптор, данные, размер_данных, флаги).

Непосредственно создание ключа выполняет функция CryptDeriveKey (провайдер, ID_алгоритма, хеш-объект, флаги, ключ), которая принимает хеш-объект в качестве исходных данных и строит подходящий ключ для алгоритма шифрования, заданного своим ID. Результатом будет дескриптор ключа, который можно использовать для шифрования.

Следует обратить внимание, что при работе с CryptoAPI мы все время имеем дело не с самими объектами или их адресами, а с дескрипторами – целыми числами, характеризующими положение объекта во внутренних таблицах криптопровайдера. Сами таблицы располагаются в защищенной области памяти, так что программы-«шпионы» не могут получить к ним доступ.

Алгоритмы шифрования, поддерживаемые CryptoAPI, можно разделить на блочные и поточные: первые обрабатывают данные относительно большими по размеру блоками (например, 64, 128 битов или более), а вторые – побитно (теоретически, на практике же – побайтно). Если размер данных, подлежащих шифрованию, не кратен размеру блока, то последний, неполный блок данных, будет дополнен необходимым количеством случайных битов, в результате чего размер зашифрованной информации может несколько увеличиться. Разумеется, при использовании поточных шифров размер данных при шифровании остается неизменным.

Шифрование выполняется функцией CryptEncrypt (ключ, хеш, финал, флаги, данные, рамер_данных, размер_буфера):

- через параметр ключ передается дескриптор ключа шифрования;
- параметр хеш используется, если одновременно с шифрованием нужно вычислить хеш-значение шифруемого текста;
- параметр «финал» равен true, если шифруемый блок текста – последний или единственный (шифрование можно осуществлять частями, вызывая функцию CryptEncrypt несколько раз);
- значение флага должно быть нулевым;

- параметр «данные» представляет собой адрес буфера, в котором при вызове функции находится исходный текст, а по завершении работы функции – зашифрованный;
- следующий параметр, соответственно, описывает размер входных/выходных данных,
- последний параметр задает размер буфера – если в результате шифрования зашифрованный текст не уместится в буфере, возникнет ошибка.

Для расшифровки данных используется функция CryptDecrypt (ключ, хеш, финал, флаги, данные, рамер_данных), отличающаяся от шифрующей функции только тем, что размер буфера указывать не следует: поскольку размер данных при расшифровке может только уменьшиться, отведенного под них буфера наверняка будет достаточно.

Конечно, шифрование всех файлов одним и тем же паролем облегчает «противнику» задачу их расшифровки, запоминание огромного числа паролей сильно усложняет жизнь, а их записывание в незашифрованном виде создает опасность раскрытия всей системы. CryptoAPI может предложить на этот случай ряд решений.

2.4. Проблема распределения ключей

В арсенале защиты должны быть не только методы, обеспечивающие секретность передачи информации (о них мы писали в первой части статьи). Не менее важными инструментами безопасности являются процедуры, позволяющие убедиться в целости и аутентичности данных. Кроме того, необходимо решать проблемы безопасного хранения и распределения ключей. Контейнером ключей называют часть базы данных ключей, которая содержит пару ключей для обмена ключами и формирования цифровой подписи.

В качестве контейнеров ключей (хранилищ пар ключей) используют, например, область временной памяти, участок реестра, файл на диске, смарт-карты. Контейнеры ключей в системе могут быть двух типов: пользовательские и уровня системы. Пользовательские контейнеры существуют в контексте работы текущего

пользователя. По умолчанию доступа к ним больше никто не имеет (правда, существуют возможности выдачи доступа к контейнерам для других пользователей). Контейнеры ключей уровня системы используются в основном не в пользовательских программах, а, например, в сервисах (доступ к контейнерам ключей уровня системы возможен без интерактивной идентификации пользователя в системе). Контейнеры ключей не существуют сами по себе, а существуют только в контексте криптопровайдера.

Для каждого криптопровайдера существует свой собственный набор контейнеров ключей. Это объясняется тем, что разные криптопровайдеры могут по-разному реализовывать даже один и тот же математический алгоритм. Следовательно, и способы хранения ключей могут также сильно варьироваться от криптопровайдера к криптопровайдеру.

Основными операциями с контейнерами ключей можно считать:

- создание нового контейнера ключей. Выполняется посредством функции CryptAcquireContext, описанной ранее в разделе о криптопровайдерах. Для создания нового контейнера ключей в качестве последнего параметра данной функции передается значение CRYPT_NEWKEYSET. Необходимо также заметить, что исходно в новом контейнере ключей никаких данных не содержится, и пары ключей необходимо генерировать самостоятельно посредством вызова функции CryptGenKey или импортировать;
- удаление существующего контейнера ключей. Выполняется посредством вызова функции CryptAcquireContext с последним параметром, установленным в CRYPT_DELETEKEYSET;
- генерацию новой пары ключей. Выполняется посредством вызова функции CryptGenKey. При генерации пары ключей для обмена ключами третий параметр данной функции устанавливают в AT_KEYEXCHANGE, а при генерации пары ключей для формирования цифровой подписи – в AT_SIGNATURE. Нужно заметить, что многие криптопровайдеры позволяют использовать пару ключей, созданную для обмена, также и для формирования цифровой подписи;

- получение значения пары ключей. Выполняется посредством вызова функции CryptGetUserKey. Флаги, передаваемые в данную функцию, соответствуют флагам для генерации пары ключей;
- установку параметров. Выполняется посредством вызова функции CryptSetProvParam. В основном устанавливают только один параметр – дескриптор безопасности контейнера ключей. Данный параметр используется, например, для выдачи доступа к контейнеру не только пользователю, создавшему контейнер. Остальные параметры контейнера ключей могут сильно варьироваться от криптопровайдера к криптопровайдеру и обычно хорошо расписаны в документации, поставляемой непосредственно с криптопровайдером;
- получение параметров. Выполняется посредством вызова функции CryptGetProvParam. Набор получаемых параметров также может сильно варьироваться в зависимости от криптопровайдера. В качестве интересного стандартного параметра можно назвать параметр PP_UNIQUE_CONTAINER. В качестве возвращаемого параметра передается уникальная (в масштабах всей системы) строка, идентифицирующая контейнер ключей. В частности, по этой строке часто можно узнать физическое расположение контейнера ключей (например, для контейнера ключей, расположенного в регистре системы, в начале возвращаемой строки будет «REGISTRY»);
- перечисление контейнеров ключей. Выполняется отдельно для каждого криптопровайдера в системе с помощью вызова функции CryptGetProvParam с параметром dwParam, установленным в PP_ENUMCONTAINERS.

Предположим, что отправитель и получатель при личной встрече договорились использовать для конфиденциальной переписки определенный пароль. Но если они будут шифровать все свои сообщения одним и тем же ключом, то возможный противник, перехватив корреспонденцию, будеть иметь хорошие шансы взломать шифр: при современных методах криптоанализа наличие нескольких шифртекстов, полученных путем использования одного и того же ключа, почти гарантирует успешный результат. Поэтому при использовании симметричных алгоритмов шифрования настоятельно рекомендуется не применять один и тот же ключ дважды!

Однако помнить отдельный пароль для каждого зашифрованного сообщения – задача достаточно трудоемкая. А для корреспондентов, не имеющих возможности встретиться лично для согласования ключей шифрования, конфиденциальный обмен сообщениями вообще становится недоступным. Такая практическая трудность называется проблемой распределения ключей.

Спасительный способ, позволяющий шифровать сообщения, обмениваясь ключами по открытым каналам связи, был придуман в середине 70-х годов прошлого столетия, а в начале 80-х появился первый реализующий его алгоритм – RSA. Теперь пользователь может сгенерировать два связанных между собой ключа – ключевую пару. Один из этих ключей по несекретным каналам рассылается всем, с кем пользователь хотел бы обмениваться конфиденциальными сообщениями. Этот ключ называют открытым (англ. public key). Зная открытый ключ пользователя, можно зашифровать адресованное ему сообщение, но вот расшифровать его позволяет лишь вторая часть ключевой пары – закрытый ключ (англ. private key). При этом открытый ключ не дает «практической» возможности вычислить закрытый: такая задача хотя и разрешима в принципе, но при достаточно большом размере ключа требует многих лет машинного времени. Для сохранения конфиденциальности получателю необходимо лишь хранить в строгом секрете свой закрытый ключ, а отправителю – убедиться, что имеющийся у него открытый ключ действительно принадлежит адресату.

Поскольку для шифрования и расшифровки используются различные ключи, алгоритмы такого рода назвали асимметричными. Наиболее существенным их недостатком является низкая производительность – они примерно в 100 раз медленнее симметричных алгоритмов. Поэтому были созданы криптографические схемы, использующие преимущества как симметричных, так и асимметричных алгоритмов:

- для шифрования файла или сообщения используется быстрый симметричный алгоритм, причем ключ шифрования генерируется случайным образом с обеспечением «хороших» статистических свойств;

- небольшой по размерам симметричный ключ шифрования шифруется при помощи асимметричного алгоритма с использованием открытого ключа адресата и в зашифрованном виде пересылается вместе с сообщением;
- получив сообщение, адресат своим закрытым ключом расшифровывает симметричный ключ, а с его помощью – и само сообщение.

2.5. Целостность и аутентичность информации

Как удостовериться в том, что пришедшее сообщение действительно отправлено тем, чье имя стоит в графе «отправитель»? Асимметричные схемы шифрования дают нам элегантный способ аутентификации. Если отправитель зашифрует сообщение своим закрытым ключом, то успешное расшифровывание убедит получателя в том, что послать корреспонденцию мог только хозяин ключевой пары и никто иной. При этом расшифровку может выполнить любой, кто имеет открытый ключ отправителя. Ведь наша цель – не конфиденциальность, а аутентификация.

Чтобы избежать шифрования всего сообщения при помощи асимметричных алгоритмов, используют хеширование: вычисляется хеш-значение исходного сообщения, и только эта короткая последовательность байтов шифруется закрытым ключом отправителя. Результат представляет собой электронную цифровую подпись. Добавление такой подписи к сообщению позволяет установить:

- аутентичность сообщения – создать подпись на основе закрытого ключа мог только его хозяин;
- целостность данных – легко вычислить хеш-значение полученного сообщения и сравнить его с тем, которое хранится в подписи: если значения совпадают, значит, сообщение не было изменено злоумышленником после того, как отправитель его подписал.

Таким образом, асимметричные алгоритмы позволяют решить две непростые задачи: обмена ключами шифрования по открытым каналам связи и подписи сообщения. Чтобы воспользоваться этими возможностями, нужно сгенерировать и сохранить две ключевые пары – для обмена ключами и для подписей. В этом нам поможет CryptoAPI.

2.6. Создание ключевых пар

После создания контейнера ключей необходимо сгенерировать ключевые пары обмена ключами и подписи. Эту работу в CryptoAPI выполняет функция CryptGenKey (провайдер, алгоритм, флаги, ключ):

- провайдер – дескриптор криптопровайдера, полученный в результате обращения к функции CryptAcquireContext;
- алгоритм – указывает, какому алгоритму шифрования будет соответствовать создаваемый ключ. Информация об алгоритме, таким образом, является частью описания ключа. Каждый криптопровайдер использует для обмена ключами и подписи строго определенные алгоритмы. Так, провайдеры типа PROV_RSA_FULL, к которым относится и Microsoft Base Cryptographic Provider, реализуют алгоритм RSA. Но при генерации ключей знать это не обязательно: достаточно указать, какой ключ мы собираемся создать – обмена ключами или подписи. Для этого используются мнемонические константы AT_KEYEXCHANGE и AT_SIGNATURE;
- флаги – при создании асимметричных ключей эта функция управляет их размером. Используемый нами криптопровайдер позволяет генерировать ключ обмена ключами длиной от 384 до 512 бит**, а ключ подписи – от 512 до 16 384 бит. Чем больше длина ключа, тем выше его надежность, поэтому трудно найти причины для использования ключа обмена ключами длиной менее 512 бит, а длину ключа подписи не рекомендуется делать меньше 1024 бит**. По умолчанию криптопровайдер создает оба ключа длиной 512 бит. Необходимую длину ключа можно передать в старшем слове параметра флаги;
- ключ – в случае успешного завершения функции в этот параметр заносится дескриптор созданного ключа.

2.7. Обмен ключами

Основными операциями с контейнерами ключей можно считать:

- создание нового контейнера ключей. Выполняется посредством функции CryptAcquireContext, описанной ранее в разделе о криптопровайдерах. Для создания нового контейнера ключей

в качестве последнего параметра данной функции передается значение CRYPT_NEWKEYSET. Необходимо также заметить, что исходно в новом контейнере ключей никаких данных не содержится, и пары ключей необходимо генерировать самостоятельно посредством вызова функции CryptGenKey, или импортировать;

- удаление существующего контейнера ключей. Выполняется посредством вызова функции CryptAcquireContext с последним параметром, установленным в CRYPT_DELETEKEYSET;
- генерация новой пары ключей. Выполняется посредством вызова функции CryptGenKey. При генерации пары ключей для обмена ключами третий параметр данной функции устанавливают в AT_KEYEXCHANGE, а при генерации пары ключей для формирования цифровой подписи – в AT_SIGNATURE. Нужно заметить, что многие криптопровайдеры позволяют использовать пару ключей, созданную для обмена, также и для формирования цифровой подписи;
- получение значения пары ключей. Выполняется посредством вызова функции CryptGetUserKey. Флаги, передаваемые в данную функцию, соответствуют флагам для генерации пары ключей;
- установка параметров. Выполняется посредством вызова функции CryptSetProvParam. В основном устанавливают только один параметр – дескриптор безопасности контейнера ключей. Данный параметр используется, например, для выдачи доступа к контейнеру не только пользователю, создавшему контейнер. Остальные параметры контейнера ключей могут сильно варьироваться от криптопровайдера к криптопровайдеру и обычно хорошо расписаны в документации, поставляемой непосредственно с криптопровайдером;
- получение параметров. Выполняется посредством вызова функции CryptGetProvParam. Набор получаемых параметров также может сильно варьироваться в зависимости от криптопровайдера. В качестве интересного стандартного параметра можно назвать параметр PP_UNIQUE_CONTAINER. В качестве возвращаемого параметра передается уникальная (в масштабах всей системы) строка, идентифицирующая контейнер ключей. В частности, по этой строке часто можно узнать физическое расположение контейнера ключей

(например, для контейнера ключей, расположенного в регистре системы, в начале возвращаемой строки будет «REGISTRY»);

- перечисление контейнеров ключей. Выполняется отдельно для каждого криптопровайдера в системе с помощью вызова функции CryptGetProvParam с параметром dwParam, установленным в PP_ENUMCONTAINERS.

Теперь мы располагаем набором ключей, однако все они останутся мертвым грузом до тех пор, пока мы не получим возможности обмена с другими пользователями открытыми ключами. Для этого необходимо извлечь их из базы данных ключей и записать в файл, который можно будет передать своим корреспондентам. При экспорте данные ключа сохраняются в одном из трех возможных форматов:

- PUBLICKEYBLOB – используется для сохранения открытых ключей. Поскольку открытые ключи не являются секретными, они сохраняются в незашифрованном виде;
- PRIVATEKEYBLOB – используется для сохранения ключевой пары целиком (открытого и закрытого ключей). Эти данные являются в высшей степени секретными, поэтому сохраняются в зашифрованном виде, причем для шифрования используется сеансовый ключ (и, соответственно, симметричный алгоритм);
- SIMPLEBLOB – используется для сохранения сеансовых ключей. Для обеспечения секретности данные ключа шифруются с использованием открытого ключа получателя сообщения.

Экспорт ключей в CryptoAPI выполняется функцией CryptExportKey (экспортируемый ключ, ключ адресата, формат, флаги, буфер, размер буфера):

- экспортируемый ключ – дескриптор нужного ключа;
- ключ адресата – в случае сохранения открытого ключа должен быть равен нулю (данные не шифруются);
- формат – указывается один из возможных форматов экспорта (PUBLICKEYBLOB, PRIVATEKEYBLOB, SIMPLEBLOB);
- флаги – зарезервирован на будущее (должен быть равен нулю);
- буфер – содержит адрес буфера, в который будет записан ключевой BLOB (Binary Large OBject – большой двоичный объект);

- размер буфера – при вызове функции в этой переменной должен находиться доступный размер буфера, а по окончании работы в нее записывается количество экспортируемых данных. Если размер буфера заранее не известен, то функцию нужно вызвать с параметром «буфер», равным пустому указателю, тогда размер буфера будет вычислен и занесен в переменную «размер буфера».

Экспорт ключевой пары целиком, включая и закрытый ключ, может понадобиться для того, чтобы иметь возможность подписывать документы на различных компьютерах (например, дома и на работе), или для сохранения страховочной копии. В этом случае нужно создать ключ шифрования на основании пароля и передать дескриптор этого ключа в качестве второго параметра функции CryptExportKey. Запросить у криптопровайдера дескриптор самого' экспортируемого ключа позволяет функция CryptGetUserKey (провайдер, описание ключа, дескриптор ключа). Описание ключа – это либо AT_KEYEXCHANGE, либо AT_SIGNATURE. Экспортированные таким образом открытые части ключей понадобятся нам для проверки подписи и расшифровки сеансового ключа.

Импорт ключевых пар во вновь созданный контейнер – это самостоятельная процедура. Необходимо запросить у пользователя название контейнера и пароль, подключиться к провайдеру, создать на основании пароля ключ, считать из файла импортируемые данные в буфер, после чего воспользоваться функцией CryptImportKey (провайдер, буфер, длина буфера, ключ для расшифровки, флаги, импортируемый ключ). Если нужно обеспечить возможность экспорта импортируемой ключевой пары впоследствии, то в параметре «флаги» необходимо передать значение CRYPT_EXPORTABLE; в противном случае вызов для данной ключевой пары функции CryptExportKey приведет к ошибке.

При работе с асимметричными алгоритмами важно убедиться, что открытый ключ действительно принадлежит тому, кого вы считаете его хозяином, и не был подменен злоумышленником. Простейшим способом обеспечить аутентичность ключа является побайтная сверка с оригиналом, хранящимся у его хозяина. Для этого

можно просто позволить пользователю просмотреть экспортированные данные в шестнадцатеричном виде – например, открыть файл, в который был записан открытый ключ, и вывести его.

2.8. Электронная цифровая подпись

Для создания электронной цифровой подписи необходимо вычислить хеш заданного файла и зашифровать этот «цифровой отпечаток сообщения» своим закрытым ключом – «подписать». Чтобы подпись впоследствии можно было проверить, необходимо указать, какой алгоритм хеширования использовался при ее создании. Поэтому подписанное сообщение должно иметь определенную структуру. Подписать вычисленный хеш в CryptoAPI позволяет функция CryptSignHash (хеш, описание ключа, комментарий, флаги, подпись, длина подписи). Вторым параметром может быть либо AT_KEYEXCHANGE, либо AT_SIGNATURE (в нашем случае логичнее использовать ключ подписи). Третий параметр в целях безопасности настоятельно рекомендуется оставлять пустым (nil). Флаги в настоящее время также не используются – на месте этого аргумента должен быть нуль. Готовую электронную подпись функция запишет в буфер, адрес которого содержится в предпоследнем параметре, последний же параметр будет содержать длину подписи в байтах.

Чтобы проверить правильность подписи, получатель подписанного сообщения должен иметь файл с открытым ключом подписи отправителя. В процессе проверки подписи этот ключ импортируется внутрь криптопровайдера. Проверка выполняется функцией CryptVerifySignature (хеш, подпись, длина подписи, открытый ключ, комментарий, флаги). О последних двух аргументах можно сказать то же, что и о параметрах «комментарий» и «флаги» функции CryptSignHash, назначение же остальных должно быть понятно. Если подпись верна, функция возвращает true. Значение false в качестве результата может свидетельствовать либо о возникновении ошибки в процессе проверки, либо о том, что подпись оказалась неверной. В последнем случае функция GetLastError вернет ошибку NTE_BAD_SIGNATURE. /

2.9. Создание сеансовых ключей

CryptoAPI позволяет генерировать сеансовые ключи случайным образом – эту работу выполняет функция CryptGenKey, о которой шла речь ранее. Однако при использовании этой возможности за пределами США и Канады приходится учитывать американские ограничения на экспорт средств «сильной криптографии». В частности, до января 2000 года был запрещен экспорт программного обеспечения для шифрования с использованием ключей длиной более 40 бит. Этим объясняется разработка Microsoft двух версий своего криптопровайдера – базовой и расширенной. Базовая версия предназначалась на экспорт и поддерживала симметричные ключи длиной 40 бит; расширенная же версия (Microsoft Enhanced Cryptographic Provider) работала с «полной» длиной ключа (128 бит). Поскольку алгоритм шифрования, как правило, требует использования ключа строго определенной длины, недостающее количество бит в урезанном «экспортном» ключе могло быть заполнено либо нулями, либо случайными данными, которые предлагалось передавать открыто.

В криптографической практике внесение в состав ключа определенной части несекретных данных, которые сменяются несколько раз в ходе обработки исходного или шифр-текста, используется для того, чтобы воспрепятствовать взлому шифра атакой «по словарю». В английской терминологии такие вставки называются salt values: их назначение – «подсолить» ключ (с учетом нашей ментальности можно перевести как «насолить» противнику). Поскольку этот термин используется и в CryptoAPI, будем употреблять его в транслитерированном виде – солт-значения.

Итак, CryptoAPI в экспортном исполнении практически вынуждает нас использовать солт-значения, составляющие бОльшую часть ключа – 88 бит из 128 для симметричных алгоритмов в RC2; и RC4. Конечно, при такой эффективной длине ключа криптозащита не может считаться достаточно надежной. В реальной ситуации выход один – воспользоваться криптопровайдером, не ограничивающим длину ключа. Обладатели Windows XP могут прибегнуть к услугам расширенных версий провайдера Microsoft (Enhanced или

Strong). Пользователям более старых версий Windows, по-видимому, придется воспользоваться продуктами сторонних разработчиков. Например, свои версии криптопровайдеров предлагают российская компания «Крипто-Про» и шведская «StreamSec». В Украине, насколько известно авторам, разработкой национального провайдера криптографических услуг занимается коллектив харьковских ученых под руководством профессора Горбенко, однако до широкого внедрения дело пока не дошло. Тем не менее благодаря архитектуре CryptoAPI прикладные программы могут разрабатываться и отлаживаться и с базовым провайдером Microsoft, так как интерфейс взаимодействия остается неизменным. Поэтому вернемся к обсуждению вопроса создания случайных сеансовых ключей.

Солт может быть сгенерирован вместе с ключом: для этого нужно в качестве флага передать функции CryptGenKey (или CryptDeriveKey) константу CRYPT_CREATE_SALT. Правда, при сохранении ключа (с помощью функции CryptExportKey) система уже не заботится о солт-значении, перекладывая ответственность на прикладную программу. Таким образом, корректная процедура создания и сохранения симметричного ключа предполагает:

1) при создании ключа функции CryptGenKey передается значение флага CRYPT_EXPORTABLE or CRYPT_CREATE_SALT;

2) с помощью функции CryptGetKeyParam с параметром KP_SALT сгенерированное солт-значение сохраняется в буфере;

3) ключ в зашифрованном виде сохраняется в буфере при помощи функции CryptExportKey, которой передается открытый ключ обмена ключами адресата;

4) зашифрованные ключевые данные сохраняются или передаются адресату вместе с экспортированным на втором шаге солт-значением.

С другой стороны, солт-значение может быть сгенерировано и отдельно от ключа. Для этого используется функция CryptGenRandom (провайдер, длина, буфер). Здесь параметр «длина» задает размер генерируемой случайной последовательности в байтах, а последний аргумент задает адрес буфера, в который будет за-

писан результат. Полученное таким образом солт-значение может быть внесено в ключ с помощью функции CryptSetKeyParam (ключ, параметр, данные, флаги). Ей вторым аргументом нужно передать KP_SALT, а третьим – адрес буфера, содержащего сгенерированную последовательность. (Последний аргумент функции зарезервирован на будущее и должен быть равен нулю.)

2.10. Блочные шифры

Блочные шифры считаются более надежными, нежели поточные, поскольку каждый блок текста подвергается сложным преобразованиям. Тем не менее одних только этих преобразований оказывается недостаточно для обеспечения должного уровня безопасности – важно, каким образом они применяются к исходному тексту в процессе шифрования.

Наиболее простой и интуитивно понятный способ состоит в том, чтобы разбить исходный текст на блоки соответствующего размера, а затем отдельно каждый блок подвергнуть шифрующему преобразованию. Такой режим использования блочных шифров называют электронной кодовой книгой (ECB – electronic codebook). Его главный недостаток состоит в том, что одинаковые блоки исходного текста при шифровании дадут одинаковые же блоки шифр-текста, а это может существенно облегчить противнику задачу взлома. Поэтому режим ECB не рекомендуется использовать при шифровании текстов, по длине превышающих один блок, в таких случаях лучше воспользоваться одним из режимов, связывающих различные блоки между собой. По умолчанию в CryptoAPI блочные шифры используются в режиме сцепления блоков шифр-текста (CBC – cipher block chaining). В этом режиме при шифровании очередной блок исходного текста вначале комбинируется с предыдущим блоком шифр-текста (при помощи побитового исключающего ИЛИ), а затем полученная последовательность битов поступает на вход блочного шифра. Образующийся на выходе блок шифр-текста используется для шифрования следующего блока. Самый первый блок исходного текста также должен быть скомбинирован с некото-

рой последовательностью битов, но «предыдущего блока шифр-текста» еще нет; поэтому режимы шифрования с обратной связью требуют использования еще одного параметра – он называется инициализирующим вектором (IV – initialization vector).

Инициализирующий вектор должен генерироваться отдельно с помощью уже известной нам функции CryptGenRandom и, как и солт-значение, передаваться вместе с ключом в открытом виде. Размер IV равен длине блока шифра. Например, для алгоритма RC2, поддерживаемого базовым криптопровайдером Microsoft, размер блока составляет 64 бита (8 байтов).

2.11. Базовые функции

2.11.1. Шифрование

Базовая функция шифрования данных имеет следующее объявление:

```
BOOL CryptEncrypt(HCRYPTKEY hKey,
                  HCRYPTHAS hHash,
                  BOOL Final,
                  DWORD dwFlags,
                  BYTE* pbData,
                  DWORD* pdwDataLen,
                  DWORD dwBufLen);
```

Первым параметром данной функции передается хендл сессионного ключа, применяемого для шифрования. Второй параметр достаточно редко используется и предназначен для получения хеша данных одновременно с их шифрованием. Такая возможность достаточно полезна при формировании одновременно как шифрованных данных, так и цифровой подписи этих же данных.

Эта функция может обрабатывать данные блоками, т.е. нет необходимости сразу загружать в память целиком весь массив данных, а лишь потом передавать ссылку на него криптографической функции. Достаточно передавать массив данных поблочно, специальным образом отметив лишь последний блок данных (это обычно нужно,

чтобы криптопровайдер провел некоторые действия после использования сессионного ключа). Для указания того, что это последний блок данных, в функции CryptEncrypt используется третий параметр Final. Четвертый параметр служит указателем на массив входных/выходных данных. Здесь нужно сразу отметить некоторую общую схему работы с данными в Crypto API. Если возвращаемые данные могут быть любого размера (а это возможно, ведь, например, в алгоритме может происходить простая замена, когда одна буква кодируется четырьмя цифрами), то работа с функцией состоит из двух этапов. На первом этапе в функцию передается общий размер входных данных и NULL в качестве ссылки на сам массив выходных данных. Функция возвращает длину выходного массива данных, пользователь инициализирует память необходимого размера и лишь затем заново передает функции ссылку на этот массив. Такая же схема используется и в работе с функцией CryptEncrypt. Параметр pdwDataLen служит для возврата размера данных, возвращаемых функцией. Параметр dwBufLen служит для указания длины входного буфера данных. Параметр dwFlags обычно не используется и устанавливается в 0.

2.11.2. Экспорт сессионных ключей

После выполнения операции шифрования встает проблема передачи шифрованных данных. Сами по себе данные, конечно, передавать можно вследствие их защищенности. Но напомним еще раз, что в Crypto API используются симметричные алгоритмы шифрования, и если на принимающей стороне не будет использован тот же самый сессионный ключ, который был использован для шифрования, то расшифровать данные на принимающей стороне не удастся. В самих шифрованных данных Crypto API самостоятельно сессионные ключи также не передает. Вместо этого Crypto API предоставляет развитые механизмы экспорта значения сессионного ключа во внешний массив данных.

Базовая функция экспорта ключей имеет следующее описание:

```
BOOL CryptExportKey(HCRYPTKEY hKey,
                    HCRYPTKEY hExpKey,
                    DWORD dwBlobType,
                    DWORD dwFlags,
                    BYTE* pdData,
                    DWORD* pdwDataLen);
```

Первым параметром данной функции передается хендл ключа, который будет экспортирован. Фактически экспорт ключа можно представить как отдельную операцию шифрования ключа. Следовательно, для такой операции необходим еще один ключ шифрования. Обычно в Crypto API сессионный ключ шифруют с помощью асимметричного алгоритма. Параметр hExpKey в большинстве случаев инициализируют контекстом публичного ключа получателя. Параметр dwBlobType определяет формат получаемого блока экспорта. Возможно, например, указать, что экспорту будет подлежать только лишь публичный ключ. В этом случае параметр hExpKey должен быть равен 0 (шифрование публичного ключа не нужно), и на выходе функции получается простое значение публичного ключа. Для такого случая параметр dwBlobType должен быть равен PUBLICKEYBLOB. Обычно же при экспорте сессионного ключа используется значение SIMPLEBLOB. Остальные значения данного параметра достаточно специфичны и применяются редко. Параметры pbData и pdwDataLen указывают на массив, выделенный для получения экспортируемого ключа, и на его размер.

2.11.3. Импорт сессионных ключей

Базовая функция импорта ключей имеет следующее описание:

```
BOOL CryptImportKey(HCRYPTPROV hProv,
                    BYTE* pbData,
                    DWORD dwDataLen,
                    HCRYPTKEY hPubKey,
                    DWORD dwFlags,
                    HCRYPTKEY* phKey);
```

В качестве первого параметра в данную функцию передается инициализированный контекст криптопровайдера. Следует отметить, для успешного завершения работы функции CryptImportKey необходимо, чтобы при инициализации криптопровайдера был указан контейнер ключей. В частности, это необходимо для успешного импорта секретных ключей в контейнер ключей. Параметр pbData представляет собой ссылку на импортируемые данные, параметр dwDataLen – длину этих данных. В параметре hPubKey указывают хендл ключа, применяемого при импорте (для расшифровывания сессионного ключа). Параметр dwFlags обычно не применяется и может быть установлен в 0. В параметре phKey возвращается импортированный ключ.

2.11.4. Расшифровывание

Базовая функция расшифровывания имеет следующее описание:

```
BOOL CryptDecrypt(HCRYPTKEY hKey,
                  HCRYPTHASH hHash,
                  BOOL Final,
                  DWORD dwFlags,
                  BYTE* pbData,
                  DWORD* pdwDataLen);
```

Первым параметром данной функции передается инициализированный контекст сессионного ключа, применяемого для расшифровывания данных. Второй параметр, как и в предыдущем примере, связан по большей части с функцией получения и проверки цифровой подписи. Обычно он не используется и устанавливается в 0. Параметр dwFlags чаще всего не используется и также устанавливается в 0. Параметры pbData и pdwDataLen используются точно так же, как и у CryptEncrypt, и представляют собой ссылку на входной/выходной массив данных и длину этого массива данных.

2.11.5. Хеширование

Под хешированием понимают применение некоторой математической функции (называемой хеш-функцией) к некоторым данным. При применении хеш-функции к произвольному объему дан-

ных всегда получается массив данных фиксированного размера. К хеш-значению предъявляется требование «устойчивости к коллизиям». Это значит, что хеш-функция тем лучше, чем труднее найти два таких случайных входных массива данных, для которых совпадали бы генерируемые хеш-значения. При обработке одних и тех же данных хеш-функция обязана возвращать одно и то же хеш-значение. Это свойство хеш-функций используется прежде всего для контроля над целостностью данных. Ведь если мы изменим хоть один бит во входном массиве информации, то результат работы хеш-функции (с высокой вероятностью) будет другим.

В Crypto API для манипуляции с хэшем используется специальный хэш-объект. Взаимодействие с этим объектом осуществляется с помощью следующих трех функций:

- CryptCreateHash;
- CryptHashData;
- CryptGetHashParam.

Для первичной инициализации хэш-объекта применяют функцию CryptCreateHash. Данная функция имеет следующее описание:

```
BOOL CryptCreateHash(HCRYPTPROV hProv,
                     ALG_ID AlgId,
                     HCRYPTKEY hKey,
                     DWORD dwFlags,
                     HCRYPTHASH* phHash);
```

В качестве первого параметра данной функции передается инициализированный контекст криптопровайдера. Вторым параметром указывается алгоритм получения значения хеша. Параметр hKey необходим лишь в случае применения специализированных алгоритмов типа MAC и HMAC.

Параметр dwFlags зарезервирован под возможное будущее использование и должен быть всегда равен 0. Через параметр phHash функция возвращает хендл созданного ею хеш-объекта. После того как хеш-объект станет ненужным, нужно освободить хеш-объект с помощью вызова функции CryptDestroyHash.

После инициализации хеш-объекта можно начать передачу данных хеш-функции с помощью вызова CryptHashData. Данная функция имеет следующее описание:

```
BOOL CryptHashData(HCRYPTHASH hHash,
                   BYTE* pbData,
                   DWORD dwDataLen,
                   DWORD dwFlags);
```

В качестве первого параметра данной функции передается ранее инициализированный хендл хеш-объекта. Вторым параметром передается порция данных для хеш-функции. Параметр dwDataLen представляет собой длину передаваемых данных. Параметр dwFlags обычно равен нулю.

После полной передачи всего массива входных данных функции CryptHashData возникает необходимость в получении значения хеш-функции. Данная задача решается с применением функции CryptGetHashParam. Данная функция имеет следующее описание:

```
BOOL CryptGetHashParam(HCRYPTHASH hHash,
                       DWORD dwParam,
                       BYTE* pbData,
                       DWORD* pdwDataLen,
                       DWORD dwFlags);
```

В качестве первого параметра данной функции передается ранее инициализированный хендл хеш-объекта. Второй параметр, dwParam, функции определяет тип запрашиваемого значения. Для получения хеш-значения необходимо передать вторым аргументом значение HP_HASHVAL. Параметры pdData и pdwDataLen отвечают за блок памяти, используемый под возвращаемое значение. Параметр dwFlags зарезервирован для будущего использования и должен быть равен нулю. Для проверки правильности хеш-значения нужно получить хэш-значение данных и сверить его с проверяемым хэш-значением.

2.11.6. Цифровая подпись

Под цифровой подписью понимают некую производную данных, по которой однозначно можно определить целостность и отправителя присланных данных. В простейшем случае под цифровой

подписью можно понимать даже собственно шифрованный контент в случае, когда ключ шифрования не скомпрометирован и однозначно принадлежит известному отправителю информации. На практике же используют гораздо более интересный метод: первично используют получение хеша данных, а затем шифруют полученное хеш-значение с помощью алгоритма с открытым ключом. Таким образом, по полученной цифровой подписи можно судить как о целостности данных (расшифровав цифровую подпись, мы можем затем проверить полученное значение хеша), так и об отправителе данных (для получения цифровой подписи используется не публичный ключ отправителя, а секретный, который может быть использован только самим отправителем).

Именно такой подход к формированию цифровой подписи используется в базовых функциях Crypto API для работы с подписью. Базовая функция получения подписи хеша данных имеет следующее описание:

```
BOOL CryptSignHash(HCRYPTHASH hHash,
                   DWORD dwKeySpec,
                   LPCTSTR sDescription,
                   DWORD dwFlags,
                   BYTE* pbSignature,
                   DWORD* pdwSigLen);
```

В качестве первого параметра используется значение хендла хеш-объекта, уже инициализированного данными (с помощью функции CryptHashData). Параметр dwKeySpec определяет, какая именно пара ключей будет использована для формирования подписи (AT_KEYEXCHANGE (пара для обмена ключами) или AT_SIGNATURE (пара для формирования цифровой подписи)). Еще раз хочется обратить внимание читателя, что во многих криптопровайдерах пара ключей, предназначенная для обмена ключами, может также использоваться и для формирования цифровой подписи (но не во всех криптопровайдерах). Параметр sDescription более не используется в данной функции, и его значение должно всегда быть установлено в NULL. Параметр dwFlags обычно устанавливают также в 0. Параметры pbSignature и pdwSigLen используют для корректного указания ссылки на массив выходных данных и его размера.

2.11.7. Проверка цифровой подписи

Для проверки цифровой подписи хеш-значения используется базовая функция, имеющая следующее описание:

```
BOOL CryptVerifySignature(HCRYPTHASH hHash,
                          BYTE* pbSignature,
                          DWORD dwSigLen,
                          HCRYPTKEY hPubKey,
                          LPCTSTR sDescription,
                          DWORD dwFlags);
```

В качестве первого параметра в функцию передается хендл хеш-объекта, предварительно инициализированный данными посредством функции CryptHashData. Второй и третий параметры отвечают за передачу значения проверяемой подписи. Параметр hPubKey используется для указания хендла публичного ключа отправителя подписи (того, кто собственно сформировал цифровую подпись). Параметр sDescription в настоящее время более не используется, и его значение должно быть установлено в NULL. Параметр dwFlags также обычно не несет полезной нагрузки и устанавливается в 0.

3. ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТОВ КРИПТОГРАФИИ В DELPHI-ПРИЛОЖЕНИЯХ

Защита информации от несанкционированного доступа и распространения играет все более важную роль в современной жизни. Задача криптографии – обеспечить эту защиту. О том, чем могут быть полезны достижения этой науки для программиста и как их использовать, будет изложено далее.

3.1. Основные понятия

Прежде всего введем несколько понятий и терминов, чтобы в дальнейшем их смысл не вызывал у вас вопросов или недопонимания. Открытый текст – собственно, это и есть та информация, которую мы будем пытаться защитить от несанкционированного доступа. «Открытый текст» – это вовсе не обязательно именно текст, это также могут быть двоичные данные, программный код и т.д.

Шифрованный текст – результат преобразования открытого текста, с использованием криптографических алгоритмов и дополнительного параметра (ключа), недоступный для восприятия.

Шифрование – процесс создания шифрованного текста при наличии открытого текста и ключа.

Дешифрование – процесс восстановления открытого текста из шифрованного при помощи ключа.

Ключ – параметр шифра, необходимый для шифрования и/или дешифрования.

Шифры подразделяются на две группы: симметричные и асимметричные.

Для шифрования и дешифрования используется один и тот же ключ. Очевидно, что «секретность» шифрованного текста зависит от «секретности» ключа, поэтому такие ключи так и называются «секретными». Тут есть одна проблема: при передаче сообщения собеседнику необходимо, чтобы у него был тот же ключ, что и у вас. А где гарантия, что при передаче ключа собеседнику его никто не

перехватит? Эта проблема решается с помощью асимметричных алгоритмов шифрования. Симметричные алгоритмы могут быть блочными (сообщение разбивается на блоки фиксированной длины, каждый из которых шифруется отдельно) и потоковыми (сообщение шифруется посимвольно). При использовании блочных шифров размер сообщения должен быть кратен размеру блока, в противном случае последний блок дополняется до необходимой длины. Блочные шифры считаются более надежными.

3.2. Асимметричные шифры

Для шифрования и дешифрования используются разные ключи. Один из ключей держится в строжайшем секрете (он называется «закрытый»), другой – публикуется («открытый»). Теперь представьте, что вы хотите передать какую-либо секретную информацию вашему другу или коллеге. Вы возьмете его открытый ключ (как уже было отмечено, он не является секретным, и узнать его может кто угодно) и зашифруете с его помощью свое сообщение. Получив шифрованный текст, он попытается расшифровать его с помощью своего закрытого ключа. Так как закрытый ключ кроме него не известен никому, то полученное сообщение не сможет восстановить никто посторонний.

Говоря об асимметричных шифрах, необходимо упомянуть еще одну замечательную возможность. Представьте, что вы зашифруете сообщение не открытым ключом получателя, а своим закрытым ключом. Расшифровать такое сообщение можно с помощью вашего открытого ключа, то есть все наоборот. Теперь получается, что расшифровать может кто угодно, а зашифровать – только вы. Кроме того, внести осмысленное изменение в сообщение (например, приписать нолик к денежной сумме) никто посторонний не сможет. Таким образом, получатель сможет аутентифицировать отправителя, т.е. он будет уверен, что отправитель сообщения именно вы и никто другой. Это называется электронно-цифровой подписью (ЭЦП), или просто цифровой подписью.

Какими бы замечательными асимметричные алгоритмы ни были, у них есть один существенный недостаток: по быстродействию они уступают симметричным раз в сто и используются в основном только для шифрования небольших сообщений. Поэтому на практике чаще всего применяют схемы, сочетающие в себе все достоинства и симметричных, и асимметричных алгоритмов, например:

1. Секретный ключ, с помощью которого будет зашифровано сообщение в данном сеансе связи (этот ключ не рекомендуют использовать повторно и называют «сеансовым»), шифруется асимметричным алгоритмом с помощью открытого ключа получателя.

2. Сообщение шифруется симметричным алгоритмом с помощью сеансового ключа.

3. Зашифрованный сеансовый ключ и сообщение отправляются получателю.

4. Получатель сначала расшифровывает сеансовый ключ с помощью своего закрытого ключа, а потом и само сообщение.

А как же быть с цифровой подписью, ведь, как уже было изложено ранее, длинные сообщения шифровать с помощью асимметричных алгоритмов неразумно? Прежде чем ответить на этот вопрос, познакомимся с еще одним понятием – хэш-функция.

3.3. Хэш-функция

Хэш-функция – это такая функция, значение которой является необратимым преобразованием исходного значения. Другими словами, пусть у нас есть число A. Вычислим $Y=H(A)$. Функция H будет необратимой, если, зная значение Y, восстановить A будет невозможно. Такому условию удовлетворяет, например, простейшая контрольная сумма, однако к хэш-функциям есть еще одно серьезное требование: очень сложной задачей должно являться нахождение такого числа B, не равного A, что $H(B)$ также будет равняться Y (такие случаи называются коллизиями). Число Y называют дайджестом или отпечатком сообщения.

Где это может пригодиться? Например, хорошим решением будет хранить в базе данных паролей не сами пароли, а их отпечатки, при вводе пользователем пароля, высчитывать его отпечаток

и сравнивать со значением в базе данных. Если злоумышленник получит доступ к этой базе, то пароли он узнать не сможет, так как хэш-функция необратима. Также он вряд ли сможет подобрать другой пароль с аналогичным отпечатком.

А теперь вернемся к цифровым подписям. Как уже было замечено, подписывать целое сообщение неразумно. В цифровой подписи главное – не секретность самого сообщения, а гарантия того, что отправитель тот, за кого себя выдает, и текст сообщения не был изменен после подписания. Обычно поступают так: высчитывается отпечаток сообщения (обычно он составляет 16–64 байт), шифруется закрытым ключом отправителя и передается вместе с самим сообщением. Получатель вычисляет отпечаток сообщения, расшифровывает подпись открытым ключом отправителя и сравнивает полученные значения. Эта процедура называется верификацией.

3.4. Популярные алгоритмы

До этого мы рассматривали какие-то абстрактные алгоритмы, а теперь настало время назвать их по именам. Среди симметричных алгоритмов можно выделить алгоритм DES (разработанный фирмой IBM и утвержденный в 1977 году правительством США как официальный стандарт. Блочный алгоритм. Несмотря на популярность, алгоритм уязвим, в истории известны случаи взлома), 3-DES, который на самом деле представляет собой не что иное, как тройное шифрование DES тремя ключами, RC2 (блочный), RC4 (потоковый), IDEA (блочный). У каждого из них свои достоинства и недостатки.

Среди асимметричных алгоритмов следует выделить RSA, названный в честь Рона Ривеста, Ади Шамира и Лена Адельмана, разработавших алгоритм в 1977 году. Идея алгоритма заключается в следующем: перемножить два числа намного проще, чем разложить произведение на множители. Этот алгоритм будет описан ниже по тексту.

1. Для начала нужно сгенерировать два больших простых числа p и q.

2. Найти $n = pq$.

3. Выбрать число e (обычно порядка 10 000) взаимно простое с $phi=(p-1)(q-1)$, т.е. числа e и phi не имеют никаких общих делителей, кроме 1.

4. Генерируется число d такое, что $ed = 1(\mathrm{mod}\ phi)$ – запись означает, что $(ed-1)$ делится на phi.

5. Числа n и e публикуются как открытый ключ, а число d держится в строжайшей тайне – это закрытый ключ. Числа p и q желательно либо уничтожить, либо также хранить в тайне.

Сообщение зашифровывается по формуле $y = xe(\mathrm{mod}\ n)$, где x – исходное сообщение, а y – зашифрованное. Расшифровывается с помощью закрытого ключа d следующим образом: $x = yd(\mathrm{mod}\ n)$. Надежность алгоритма заключается в том, что для восстановления закрытого ключа d необходимо знать числа p и q. Их можно получить, разложив на множители число n, но если числа p и q достаточно большие, то эта задача становится практически неразрешимой. В настоящий момент рекомендуют выбирать p и q такие, чтобы произведение n было не короче 1024 бит.

Ну и среди алгоритмов хеширования можно назвать следующие: MD4 (128-разрядный отпечаток), MD5 (разработан в 1991 году, 128-разрядный отпечаток, пришел на смену MD4, в 2004 году в алгоритме обнаружена уязвимость, позволяющая довольно быстро находить коллизии), SHA-1 (разработан в 1995 году, 160-разрядный отпечаток, долгое время был наиболее популярным, однако в начале 2005 года с ним произошло то же самое, что и с MD5. Брюс Шнайер заявил: «SHA-1 has been broken»), SHA-224, SHA-256, SHA-384, SHA-512.

3.5. CryptoAPI

Криптографические функции являются частью операционной системы Windows, и обратиться к ним можно посредством интерфейса CryptoAPI. Основные возможности доступны еще с Windows 95, но со временем они расширялись. Описание функций CryptoAPI можно найти в MSDN, в работе [2] или в справочном файле к Delphi. Функции содержатся в библиотеках advapi32.dll и crypt32.dll. Их можно импортировать самостоятельно, а можно воспользоваться файлом Wcrypt2.pas, который прилагается к данной статье.

Подключение к криптопровайдеру. Контейнеры ключей.

Первая функция, которую мы рассмотрим, будет следующий:

```
function CryptAcquireContext(phProv      :PHCRYPTPROV;
                             pszContainer :LPAWSTR;
                             pszProvider :LPAWSTR;
                             dwProvType    :DWORD;
                             dwFlags       :DWORD) :BOOL; stdcall;
```

В большинстве случаев работа с криптографическими возможностями Windows начинается с вызова именно этой функции, которая выполняет подключение к криптопровайдеру и возвращает его дескриптор в параметре phProv. Криптопровайдер представляет собой dll, независимый программный модуль, который фактически исполняет криптографические алгоритмы. Криптопровайдеры бывают различные и отличаются составом функций (например, некоторые криптопровайдеры ограничиваются лишь цифровыми подписями), используемыми алгоритмами (некоторые шифруют алгоритмом RC2, другие – DES) и другими возможностями. В каждой операционной системе свой состав криптопровайдеров, однако в каждой присутствует Microsoft Base Cryptographic Provider v1.0. При вызове функции CryptAcquireContext, необходимо указать имя провайдера и его тип (соответственно в параметрах pszProvider и dwProvType). Тип провайдера определяет состав функций и поддерживаемые криптоалгоритмы, например:

Тип PROV_RSA_FULL

- Обмен ключами – алгоритм RSA
- Цифровая подпись – алгоритм RSA
- Шифрование – алгоритм RC2 и RC4
- Хэширование – алгоритмы MD5 и SHA

Тип PROV_RSA_SIG

- Обмен ключами – не поддерживается;
- цифровая подпись – алгоритм RSA,
- шифрование – не поддерживается,
- хэширование – алгоритмы MD5 и SHA.

Microsoft Base Cryptographic Provider v1.0 относится к типу PROV_RSA_FULL и для этого типа используется по умолчанию (если в параметре pszProvider указать nil). В параметре pszContainer необходимо указать имя контейнера ключей, который мы собираемся использовать. Дело в том, что каждый криптопровайдер содержит базу данных, в которой хранятся ключи пользователей. Эти ключи группируются в контейнерах. Сохраняются только ключевые пары для асимметричных алгоритмов, сеансовые ключи не сохраняются, так как их не рекомендуют использовать повторно. Таким образом, каждый контейнер имеет имя и содержит по одному ключу (точнее, по паре – открытый-закрытый ключ) для цифровой подписи и обмена ключами (помните, уже отмечалось, что из-за низкого быстродействия асимметричные алгоритмы используются в основном только для шифрования сеансовых ключей и подписи хэша). В зависимости от криптопровайдера база данных может храниться в файлах, реестре или в каких-либо аппаратных средствах, но это не влияет на работу программиста с контейнерами ключей. Если в качестве параметра pszContainer указать nil, то будет использоваться контейнер ключей, название которого совпадает с именем пользователя, под которым был осуществлен вход в систему. Но так делать не рекомендуется: дело в том, что если два приложения используют один и тот же контейнер, одно из них может изменить или уничтожить ключи, необходимые для корректной работы другого приложения. Поэтому рекомендуют использовать контейнеры, имена которых совпадают с именем приложения.

Параметр dwFlags может быть нулевым или принимать одно из следующих значений:

CRYPT_VERIFYCONTEXT – этот флаг предназначен для приложений, которые не должны иметь доступ к закрытым ключам контейнера. Такие приложения могут обращаться только к функциям хеширования, проверки цифровой подписи или симметричного шифрования. В этом случае параметр pszContainer должен быть равен nil;

CRYPT_NEWKEYSET – создает новый контейнер ключей, но сами ключи не создаются;

CRYPT_DELETEKEYSET – удаляет контейнер вместе с хранящимися там ключами. Если задан этот флаг, то подключение к криптопровайдеру не происходит и параметр phProv неопределен;

CRYPT_MACHINE_KEYSET – по умолчанию контейнеры ключей сохраняются как пользовательские. Для основных криптопровайдеров это означает, что контейнеры ключей сохраняются в пользовательских профилях. Этот флаг можно устанавливать в комбинации с другими, чтобы указать, что контейнер является машинным, то есть хранится в профиле All Users. В случае успеха функция возвращает true, в противном случае – false. GetLastError вернет код ошибки;

```
function CryptReleaseContext(hProv     :HCRYPTPROV;
                             dwFlags :DWORD) :BOOL; stdcall;
```

Она освобождает контекст криптопровайдера и контейнера ключей; hProv – дескриптор криптопровайдера, полученный при вызове CryptAcquireContext; dwFlags – зарезервирован и должен равняться нулю. В случае успеха функция возвращает true, в противном случае – false. GetLastError вернет код ошибки.

Приведем пример работы с этими функциями:

```
uses Wcrypt2;
...

procedure CryptProc;
var
  Prov: HCRYPTPROV;
begin

CryptAcquireContext(@Prov,nil,nil,PROV_RSA_FULL,CRYPT_VERIFY
CONTEXT);
// Работаем с функциями CryptoAPI
 CryptReleaseContext(Prov,0);
end;
```

Прежде чем перейти непосредственно к криптографическим функциям, упомяну еще о таких функциях, как CryptSetProvider, CryptGetDefaultProvider, CryptGetProvParam, CryptSetProvParam, CryptEmunProviders, CryptEnumProviderTypes, описание которых вы найдете сами.

3.6. Хэширование и электронно-цифровая подпись

```
function CryptCreateHash(hProv     :HCRYPTPROV;
                         Algid     :ALG_ID;
                         hKey      :HCRYPTKEY;
                         dwFlags :DWORD;
                         phHash    :PHCRYPTHASH) :BOOL; stdcall;
```

Эта функция создает в системе хэш-объект и возвращает в параметре phHash его дескриптор. Данные, поступающие на вход хэш-объекта, там преобразуются, и их отпечаток сохраняется внутри хэш-объекта. В параметре hProv нужно указать дескриптор провайдера, полученный с помощью CryptAcquireContext. Параметр Algid указывает на то, какой алгоритм хэширования будет использоваться. Для Microsoft Base Cryptographic Provider может принимать следующие значения: CALG_MAC, CALG_MD2, CALG_MD5, CALG_SHA. Смысл этих значений, наверное, понятен. Параметр hKey подробно рассматривать не будем, вы можете почитать о нем сами. Отметим лишь, что обычно (если не используется алгоритм с секретным ключом, такой как MAC) его указывают равным нулю. Параметр dwFlags зарезервирован на будущее и должен быть равен нулю. В случае успеха функция возвращает true, в противном случае – false. GetLastError вернет код ошибки.

```
function CryptDestroyHash(hHash :HCRYPTHASH) :BOOL; stdcall;
```

Эта функция уничтожает хэш-объект, созданный с помощью CryptCreateHash. В параметре hHash указывается дескриптор хэш-объекта. В случае успеха функция возвращает true, в противном случае – false. GetLastError вернет код ошибки.

```
function CryptHashData(hHash        :HCRYPTHASH;
                const pbData     :PBYTE;
                    dwDataLen  :DWORD;
                    dwFlags    :DWORD) :BOOL; stdcall;
```

Указанная функция позволяет добавлять данные к объекту хэш-функции. Функция может вызываться несколько раз, и данные, от которых мы вычисляем хэш, разбиты на порции. В параметре hHash указывается дескриптор хэш-объекта, созданный с помощью CryptCreateHash; pbData содержит указатель на данные, а dwDataLen содержит размер этих данных в байтах. Для Microsoft Base Cryptographic Provider параметр dwFlags должен быть равен нулю. В случае успеха функция возвращает true, в противном случае – false. GetLastError вернет код ошибки.

```
function CryptSignHash(hHash :HCRYPTHASH;
                dwKeySpec     :DWORD;
                sDescription :LPAWSTR;
                dwFlags      :DWORD;
                pbSignature   :PBYTE;
                pdwSigLen     :PDWORD) :BOOL; stdcall;
```

Данная функция вычисляет значение электронно-цифровой подписи от значения хэша. В параметре hHash указывается дескриптор хэш-объекта, созданный с помощью CryptCreateHash; dwKeySpec указывает, какой ключ будет использован для создания подписи. Как уже указывалось, в хранилище ключей содержатся две ключевые пары: для подписи и для обмена ключами. Соответственно этот параметр может принимать значения AT_SIGNATURE или AT_KEYEXCHANGE (логичнее использовать AT_SIGNATURE). Ключи должны существовать в контейнере. Параметр sDescription может содержать произвольную строку описания. Эта строка будет добавлена к хэшу и должна быть известна приемной стороне. Использовать этот параметр не рекомендуется, так как это снижает безопасность системы. Параметр dwFlags не поддерживается в Microsoft Base Cryptographic Provider, и на его месте следует указать ноль. Параметр pbSignature указывает на буфер, куда будет помещена цифровая под-

пись, а pdwSigLen – размер этого буфера. Если размер заранее не известен, то можно указать pbSignature равным nil, и тогда в параметре pdwSigLen мы получим необходимый размер буфера. В случае успеха функция возвращает true, в противном случае – false. GetLastError вернет код ошибки.

```
function CryptVerifySignature(hHash          :HCRYPTHASH;
                    const pbSignature   :PBYTE;
                        dwSigLen      :DWORD;
                        hPubKey       :HCRYPTKEY;
                        sDescription   :LPAWSTR;
                        dwFlags       :DWORD) :BOOL; stdcall;
```

Эта функция осуществляет проверку цифровой подписи. hHash – дескриптор хэш-объекта, значение которого является отпечатком сообщения, подпись которого мы проверяем; pbSignature – указатель на буфер, содержащий подпись; dwSigLen – размер этого буфера; hPubKey – дескриптор открытого ключа, с помощью которого мы будем проверять подпись. Открытый ключ должен соответствовать закрытому, которым осуществлялась подпись. О том, как получить этот ключ, узнаете позже. Параметры sDescription и dwFlags должны соответствовать параметрам функции CryptSignHash при осуществлении подписи. В случае успеха функция возвращает true, в противном случае – false. GetLastError вернет код ошибки. Приведем пример работы с этими функциями, упрощенно, без контроля возможных ошибок (в реальных приложениях на это не стоит закрывать глаза):

1. Создание подписи.

```
uses Wcrypt2;
...
function SignMessage(Message: String): String;

var
  Prov: HCRYPTPROV;
  Hash: HCRYPTHASH;
  BufLen: DWORD;
```

```
begin
  Result:='';
  CryptAcquireContext(@Prov,nil,nil,PROV_RSA_FULL,0);
  CryptCreateHash(Prov,CALG_MD5,0,0,@Hash);
  CryptHashData(Hash,PByte(Message),Length(Message),0);
  BufLen:=0;
  CryptSignHash(Hash,AT_SIGNATURE,nil,0,nil,@BufLen);
  if BufLen>0 then begin

    SetLength(Result,BufLen);
    CryptSignHash(Hash,AT_SIGNATURE,nil,0,PByte(Result),
@BufLen);
  end;
  CryptDestroyHash(Hash);
  CryptReleaseContext(Prov,0);
end;
```

2. Проверка подписи. В коде будут упущены некоторые фрагменты, о которых сообщим позже.

```
function VerifySign(Message, Sign: String): Boolean;
var
  Prov: HCRYPTPROV;
  Hash: HCRYPTHASH;
  PublicKey: HCRYPTKEY;
begin
  CryptAcquireContext(@Prov,nil,nil,PROV_RSA_FULL,0);
  CryptCreateHash(Prov,CALG_MD5,0,0,@Hash);
  CryptHashData(Hash,PByte(Message),Length(Message),0);
 // Здесь должен быть импорт открытого ключа для проверки
подписи

...
  Result:=CryptVerifySignature(Hash,PByte(Sign),Length(Sign),
    PublicKey,nil,0);
  // Здесь должно быть уничтожение открытого ключа
...
```

```
  CryptDestroyHash(Hash);
  CryptReleaseContext(Prov,0);
end;
```

Рекомендуется ознакомиться самостоятельно с функциями CryptHashSessionKey, CryptGetHashParam и CryptSetHashParam.

3.7. Шифрование на основе пользовательских данных или пароля

Для шифрования в CryptoAPI используются симметричные алгоритмы, ключ для которых может быть получен двумя путями: случайным образом или на основе каких-либо пользовательских данных, например пароля. Причем к последнему варианту генерации ключа есть одно важное требование: при использовании одних и тех же паролей должны получаться идентичные ключи. Такая возможность предусмотрена в CryptoAPI.

```
function CryptDeriveKey(hProv      :HCRYPTPROV;
                        Algid    :ALG_ID;
                        hBaseData :HCRYPTHASH;
                        dwFlags   :DWORD;
                        phKey     :PHCRYPTKEY) :BOOL; stdcall;
```

В параметре hProv нужно указать дескриптор провайдера, полученный с помощью CryptAcquireContext. Algid – идентификатор алгоритма, для которого генерируется ключ. Для Microsoft Base Cryptographic Provider может принимать следующие значения: CALG_RC2 и CALG_RC4. Пользовательские данные (пароль) предварительно хэшируются, и дескриптор хэш-объекта передается в функцию в качестве параметра hBaseData. Старшие 16 бит параметра dwFlags могут содержать размер ключа в битах или быть нулевыми (в этом случае будет создан ключ с размером по умолчанию). Младшие 16 бит могут быть нулевыми или принимать следующие значения или их комбинации:

CRYPT_EXPORTABLE, CRYPT_CREATE_SALT, CRYPT_USER_PROTECTED, CRYPT_UPDATE_KEY.

К первым двум мы еще вернемся, а со смыслом остальных вы можете ознакомиться самостоятельно. В параметре phKey возвращается дескриптор созданного ключа.

В случае успеха функция возвращает true, в противном случае – false. GetLastError вернет код ошибки. Когда ключ есть, можно приступать непосредственно к шифрованию. Для этого нам понадобятся функции CryptEncrypt и CryptDecrypt.

```
function CryptEncrypt(hKey        :HCRYPTKEY;
                      hHash    :HCRYPTHASH;
                      Final    :BOOL;
                      dwFlags  :DWORD;
                      pbData   :PBYTE;
                      pdwDataLen :PDWORD;
                      dwBufLen   :DWORD) :BOOL; stdcall;
```

В параметре hKey передается дескриптор ключа, необходимый для шифрования. Этот ключ также определяет алгоритм шифрования. Параметр hHash используется, если данные одновременно шифруются и хэшируются (шифроваться и хэшироваться будут исходные данные). В этом случае в параметре hHash передается дескриптор заранее созданного хэш-объекта. Эту возможность удобно использовать, если необходимо одновременно зашифровать и подписать сообщение. Иначе этот параметр следует установить в ноль. Параметр Final следует установить в true, если переданный в функцию блок данных является единственным или последним. В этом случае он будет дополнен до необходимого размера. Параметр dwFlags не используется в Microsoft Base Cryptographic Provider, и на его месте следует указать ноль. Параметр pbData – указатель на буфер, в котором содержатся данные для зашифрования. Зашифрованные данные помещаются в тот же буфер. Параметр pdwDataLen – размер данных, которые будут зашифрованы; dwBufLen – размер выходного буфера, для блочных шифров может быть больше, чем pdwDataLen. Узнать необходимый размер можно, передав в параметрах pbData nil, pdwDataLen размер данных, которые необходимо зашифровать, а в параметре dwBufLen передать что угодно, например ноль. После такого вызова необходимый размер буфера будет содержаться в параметре pdwDataLen (именно pdwDataLen, а не dwBufLen, немного нелогично, ну да ладно). Чтобы не было путаницы, приведем простой пример:

```
var
  Message: String;
  BufLen, DataLen: DWORD;
...

begin
...
  Message:='Hello World!';
  BufLen:=Length(Message);
  DataLen:=Length(Message);
  // Вычисляем необходимый размер выходного буфера
  CryptEncrypt(Key,0,true,0,nil,@BufLen,0);
  // Выделяем память для буфера и шифруем

  SetLength(Message,BufLen);
  CryptEncrypt(Key,0,true,0,PByte(Message),@DataLen,BufLen);
```

Теперь, рассмотрим функцию, которая позволяет расшифровать сообщение.

```
function CryptDecrypt(hKey      :HCRYPTKEY;
                      hHash     :HCRYPTHASH;
                      Final     :BOOL;
                      dwFlags   :DWORD;
                      pbData    :PBYTE;
                      pdwDataLen :PDWORD) :BOOL; stdcall;
```

Даже не надо подробно описывать все параметры – тут все очевидно. Уточним лишь, что в параметр pdwDataLen нужно передать число байт шифротекста, а после вызова в него будет помещена длина открытого сообщения. Если используется параметр hHash, то данные после расшифровки хэшируются. Это удобно использовать, если нужно одновременно расшифровать сообщение и проверить подпись. После того как работа с ключом закончена, необходимо освободить дескриптор:

```
function CryptDestroyKey(hKey :HCRYPTKEY) :BOOL; stdcall;
```

Если hKey относится к сеансовому ключу или импортированному открытому ключу (об этом ниже), то дескриптор освобождается, а ключ уничтожается. Если hKey относится к паре открытый/закрытый ключ, то дескриптор освобождается, а ключевая пара сохраняется в контейнере ключей. Ниже дан демонстрационный пример работы с этими функциями.

3.8. Генерация случайных ключей. Импорт/экспорт ключей

Только что мы рассмотрели случай, когда для зашифровки и расшифровки сообщения отправитель и получатель использовали пароль, известный только им. Сейчас рассмотрим другой: отправитель генерирует ключ случайно и передает его получателю в зашифрованном виде вместе с сообщением. При этом для шифрования сеансового ключа используется открытый ключ получателя. А где отправитель его возьмет? Как уже было изложено, при создании ключевого контейнера с помощью функции CryptAcquireContext ключи в контейнере не создаются, их нужно сгенерировать отдельно. Рассмотрим функцию:

```
function CryptGenKey(hProv   :HCRYPTPROV;
                     Algid   :ALG_ID;
                     dwFlags :DWORD;
                     phKey   :PHCRYPTKEY) :BOOL; stdcall;
```

Функция предназначена для генерации случайных сеансовых ключей и ключевых пар. Параметры этой функции аналогичны одноименным параметрам функции CryptDeriveKey, за исключением того, что Algid может также принимать значения AT_KEYEXCHANGE и AT_SIGNATURE. В этом случае будут сгенерированы ключевые пары соответственно для обмена ключами и цифровой подписи. Создание нового ключевого контейнера должно выглядеть примерно так:

```
uses Wcrypt2;
...
var
  Prov: HCRYPTPROV;
```

```
        ExchangeKey, SignKey: HCRYPTKEY;
      begin

CryptAcquireContext(@Prov,'My_Container',nil,PROV_RSA_FULL,CR
YPT_NEWKEYSET);
        // Создаем ключевые пары

        CryptGenKey(Prov,AT_KEYEXCHANGE,0,@ExchangeKey);
        CryptGenKey(Prov,AT_SIGNATURE,0,@SignKey);
        // Работаем с функциями CryptoAPI
      ...
        // Освобождаем дескрипторы ключевых пар. Сами ключи со-
храняются в контейнере
        CryptDestroyKey(SignKey);
        CryptDestroyKey(ExchangeKey);
        CryptReleaseContext(Prov,0);
      end;
```

Созданные таким образом ключевые пары впоследствии можно извлечь из контейнера, воспользовавшись функцией:

```
function CryptGetUserKey(hProv      :HCRYPTPROV;
                         dwKeySpec :DWORD;
                         phUserKey :PHCRYPTKEY) :BOOL; stdcall;
```

Параметр dwKeySpec может принимать два значения: AT_KEYEXCHANGE и AT_SIGNATURE, которые очевидны. Дескриптор ключа возвращается в параметре phUserKey.

Теперь ответим на вопрос, как отправитель сможет передать получателю свою открытую часть ключа.

```
function CryptExportKey(hKey      :HCRYPTKEY;
                        hExpKey   :HCRYPTKEY;
                        dwBlobType :DWORD;
                        dwFlags    :DWORD;
                        pbData     :PBYTE;
                        pdwDataLen :PDWORD) :BOOL; stdcall;
```

Функция позволяет экспортировать ключ в двоичный буфер, который впоследствии можно будет сохранить в файл и передать кому-либо. В параметре hKey должен содержаться дескриптор экспортируемого ключа. Экспортировать можно не только открытые ключи, а также ключевые пары целиком и сеансовые ключи. В последних двух случаях ключи и ключевые пары должны быть созданы функциями CryptGenKey или CryptDeriveKey с параметрами dwFlags, равными CRYPT_EXPORTABLE. Открытые же ключи всегда экспортируемы. Сеансовые ключи и ключевые пары экспортируются только в зашифрованном виде. Параметр hExpKey определяет ключ, которым они будут зашифрованы. Если экспортируется открытая часть ключа, то этот параметр следует установить в ноль, если экспортируется ключевая пара целиком, то здесь обычно передают дескриптор сеансового ключа (обычно полученный с помощью CryptDeriveKey), которым пара будет зашифрована, если экспортируется сеансовый ключ, то обычно он шифруется открытым ключом получателя (обычно используется ключ обмена, но никто не запрещает использовать ключ подписи). Параметр dwBlobType определяет тип экспортируемого ключа и может принимать следующие значения: SIMPLEBLOB – сеансовый ключ, PUBLICKEYBLOB – открытый ключ, PRIVATEKEYBLOB – ключевая пара целиком. Существуют и другие значения, но они не поддерживаются стандартным криптопровайдером. Параметр dwFlags для Microsoft Base Cryptographic Provider должен быть равен нулю; pbData – буфер, куда будут скопированы данные; pdwDataLen – размер этого буфера. Если он заранее не известен, то можно указать в качестве параметра pbData nil, и в pdwDataLen будет получен необходимый размер.

Вот пример экспорта открытого ключа:

```
procedure ExportPublicKey(FileName: TFileName);
var
  Prov: HCRYPTPROV;
  SignKey: HCRYPTKEY;
  Stream: TMemoryStream;
  BufSize: DWORD;
```

```
begin
  CryptAcquireContext(@Prov,'My_Container',nil,PROV_RSA_FULL,0);
  CryptGetUserKey(Prov,AT_SIGNATURE,@SignKey);
  Stream:=TMemoryStream.Create;
  CryptExportKey(SignKey,0,PUBLICKEYBLOB,0,nil,@BufSize);
  Stream.SetSize(BufSize);
  CryptExportKey(SignKey,0,PUBLICKEYBLOB,0,PByte(Stream.
Memory),@BufSize);
  Stream.SaveToFile(FileName);
  Stream.Free;
  CryptDestroyKey(SignKey);
  CryptReleaseContext(Prov,0);

  end;
```

Импорт ключа осуществляется с помощью функции:

```
function CryptImportKey(hProv     :HCRYPTPROV;
                        pbData    :PBYTE;
                        dwDataLen :DWORD;
                        hPubKey   :HCRYPTKEY;
                        dwFlags   :DWORD;
                        phKey     :PHCRYPTKEY) :BOOL; stdcall;
```

Тут практически все понятно. Поясним лишь, что в параметре hPubKey необходимо передать дескриптор ключа, которым будет расшифрован импортированный ключ. Если импортируется ключевая пара целиком, то параметр dwFlags можно установить в CRYPT_EXPORTABLE, тогда импортированная пара может быть впоследствии также экспортирована. В параметре phKey вернется дескриптор полученного ключа. Если это ключевая пара, то она будет сохранена в контейнере.

Вот пример импорта открытого ключа:

```
function ImportPublicKey(FileName: TFileName): HCRYPTKEY;

var
  Prov: HCRYPTPROV;
```

```
    Stream: TMemoryStream;
  begin
    Stream:=TMemoryStream.Create;
    Stream.LoadFromFile(FileName);
    CryptImportKey(Prov,PByte(Stream.Memory),Stream.Size,0,0,@Result);
    Stream.Free;
   end;
```

Теперь, воспользовавшись этой информацией, вы без труда сможете восстановить пропущенные фрагменты в функции проверки цифровой подписи, описанной ранее. Итак, как же передать собеседнику зашифрованное сообщение? Это происходит следующим образом:

1. Получатель экспортирует свой открытый ключ обмена в файл и передает его отправителю сообщения.

2. Отправитель генерирует случайный сеансовый ключ и шифрует им сообщение.

3. Отправитель импортирует открытый ключ обмена получателя, экспортирует сеансовый ключ, шифруя его полученным ключом обмена (ключ обмена в параметре hExpKey).

4. Зашифрованное сообщение передается вместе с зашифрованным сеансовым ключом – так называемый цифровой конверт.

5. Получатель импортирует сеансовый ключ, расшифровывая его своим закрытым ключом обмена (его можно получить, вызвав CryptGetUserKey) и с помощью сеансового ключа расшифровывает сообщение.

Рассуждая о сеансовых ключах, используемых в Microsoft Base Cryptographic Provider, нужно упомянуть об одной неприятности: до начала 2000 года действовал запрет на экспорт программного обеспечения, использующего средства «сильной криптографии» за пределами США и Канады. По этой причине в базовом криптопровайдере не поддерживаются ключи для симметричных алгоритмов длиной более 40 бит. Ключи длиной 56 бит разрешалось использовать только заграничным отделениям американских компаний. Для алгоритмов RC2 и RC4 рекомендуемая длина ключа должна состав-

лять 128 бит, поэтому недостающее количество бит заполняется нулями либо случайными значениями, которые должны передаваться открыто. Надежность защиты из-за этого, разумеется, сильно страдает. В состав Windows XP входит Microsoft Enhanced Cryptographic Provider, в котором этой проблемы нет, но при использовании базового криптопровайдера необходимо дополнять ключ до нужной длины, используя так называемые солт-значения (salt-values). Сгенерировать salt-value и внести его в ключ можно несколькими способами, но самый простой и очевидный – при вызове CryptGenKey или CryptDeriveKey передать в параметре dwFlags значение CRYPT_CREATE_SALT, примерно так:

```
CryptGenKey(Prov,CALG_RC2,CRYPT_EXPORTABLE or CRYPT_CREATE_SALT,@Key);
```

При экспорте ключа солт-значение не сохраняется, о нем должен позаботиться сам программист.

```
var
  SaltLen: DWORD;
  Stream: TMemoryStream;
...
begin
...
  // Определяем размер буфера для солт-значения

  CryptGetKeyParam(Key,KP_SALT,nil,@SaltLen,0);
  // Сохраняем его в файл
  Stream:=TMemoryStream.Create;
  Stream.SetSize(SaltLen);
  CryptGetKeyParam(Key,KP_SALT,PByte(Stream.Memory),@SaltLen,0);
  Stream.SaveToFile('Salt.dat');
  Stream.Free;
...
```

Сохраненное таким образом солт-значение необходимо передать вместе с сеансовым ключом, а на приемной стороне «вживить» его туда снова.

```
var

  Stream: TMemoryStream;
...
begin
...
  Stream:=TMemoryStream.Create;
  Stream.LoadFromFile('Salt.dat');
  CryptSetKeyParam(Key,KP_SALT,PByte(Stream.Memory),Stream.Size);
  Stream.Free;
...
```

Для работы с солт-значениями мы воспользовались функциями CryptGetKeyParam и CryptSetKeyParam, однако их возможности на этом не заканчиваются. Рекомендуем ознакомиться с ними самостоятельно, а также с другими функциями, которые в данной работе не упоминались: CryptGenRandom, CryptDuplicateKey, CryptDublicateHash.

3.9. Другие функции

К данному материалу еще прилагаются некоторые бесплатные библиотеки и программы, которые не имеют отношения к CryptoAPI, но также работают со средствами криптографии и могут быть очень полезны:

1. HashLib! 1.03 (C) Alex Demchenko, 2002, Moldova, Chishinev – очень хорошая и удобная библиотека, в которой реализовано множество алгоритмов хэширования: MD4, MD5, CRC32, HAVAL-128, SHA-1, SHA-256, TIGER-128, GOST, RIPEMD-128 и др.

2. FGInt copyright 2000, Walied Othman – отличная библиотека для работы с гигантскими целыми числами, необходимыми для работы алгоритма RSA и с самим RSA.

3. DCPCrypt Copyright (c) 1999-2003 David Barton – огромная библиотека компонент для работы с криптографическими функциями.

4. RSATool2v17 – генератор чисел p, q, e, n, d для алгоритма RSA.

4. КРАТКОЕ ОПИСАНИЕ ПРОЕКТА

Скачайте файл «setup.exe», доступный по ссылке https://disk.yandex.ru/d/sXgZ3eqX4ksDCQ. Запустите его.

Необходимо произвести запуск проекта. В Programfile и на рабочем столе появится папка PSTU с необходимой документацией (GriptoApi_Папки).

Важным содержанием являются папки проекта (Отправитель – А, Получатель – В):

- Filetext.txt – файл для хранения незашифрованного текста l;
- FileHesh.txt – файл Hesha;
- FileЭцп.txt – файл подписи Hesha;
- Filezash.txt – файл шифрования текста и Hesha;
- Filerac.txtr – файл резуьтата расшифрования;
- Kcc – сессионный разовый ключ шифрования и расшифрования, создает оправитель по утвержденной инструкции;
- Klkccо – Получатель генерирует алгоритмом RSA два ключа: открытый Klo, посылаемый открытоым образом отправителю, и закрытый Klz, хранимый у себя, Отправитель полученным открытым ключом Klo шифрует Kcc, получает шифр его Klkccо и поместит его в передаваемый пакет получателю;
- Отправитель имеет от центра сертификации два ключа ЭЦП – открытый klzpo, посылаемый получателю открыто, и закрытый ключ klzpz, хранимый у себя;
- Отправитель алгоритмом хеширования (предположим MD5) получает Hesh из текста Filetext.txt и помещает его в FileHesh.txt, далее закрытым ключом klzpz подписывает, шифрует и помещает в klzpz;
- Отправитель ключом Kcc по согласованному алгоритму шифрует текст их Filetext.txt, помещает впереди ЭЦП Hesha в папку Filezash.txt;
- Отправитель формирует пакет-адрес отправителя, адрес получатнля, временной блок, информационный блок, шифр КСС,

зашифрованный и подписанный HEsh, сам шифр текста и отправляет получателю;

- Получатель, получив пакет, закрытым ключом Klz расшифровывает сессионный ключ, расшифровывает текст, получает по нему свой Hesh, проверяет полученную ЭЦП, т.е. Hesh1, и если они совпадают, то процесс достоверен и пакет принимается. Этим самым обеспечиваются конфидициальность, целостность, авторство и доступ к информации. Результат размещается в папке Filerac.txtr. Иначе пакет бракуется.

4.1. Взаимодействие с CryptoAPI

Функции CryptoAPI можно вызвать из программы, написанной на любимом многими (в том числе и авторами) языке C++. Тем не менее Pascal де-факто признан стандартом в области обучения программированию. (Не будем спорить о том, хорошо это или плохо, чтобы не ввязываться в драку, пусть даже и виртуальную.) Кроме того, в ряде отечественных компаний Delphi является базовым средством разработки. Поэтому все примеры были реализованы в среде Delphi, хотя в качестве инструмента можно было бы выбрать и MS Visual C++.

Код функций криптографической подсистемы содержится в нескольких динамически загружаемых библиотеках Windows (advapi32.dll, crypt32.dll). Для обращения к такой функции из прикладной программы на Object Pascal следует объявить ее как внешнюю. Заголовок функции в интерфейсной части модуля будет выглядеть, например, так:

```
function CryptAcquireContext (
phPROV: PHCRYPTPROV;
pszContainer: LPCTSTR;
pszProvider: LPCTSTR;
dwProvType: DWORD;
dwFlags: DWORD): BOOL; stdcall;
```

А в исполняемой части вместо тела функции нужно вписать директиву extern с указанием библиотеки, в которой содержится

функция, и, возможно, ее имени в этой библиотеке (если оно отличается от имени функции в создаваемом модуле), например:

```
function CryptAcquireContext; external ‘advapi32.dll’
name 'CryptAcquireContextA';
```

Таким образом, имея описание функций CryptoAPI, можно собрать заголовки функций в отдельном модуле, который будет обеспечивать взаимодействие прикладной программы с криптографической подсистемой. Разумеется, такая работа была проделана программистами Microsoft, и соответствующий заголовочный файл (wincrypt.h) был включен в поставку MS Visual C++. К счастью, появилась и Delphi-версия (wcrypt2.pas). Подключив модуль к проекту, вы сможете использовать не только функции CryptoAPI, но и мнемонические константы режимов, идентификаторы алгоритмов и прочих параметров, необходимых на практике. И последнее замечание перед тем, как опробовать CryptoAPI в деле. Ряд функций был реализован только в Windows 2000. Но и на старушку Windows 98 можно найти управу: при установке Internet Explorer 5 интересующие нас библиотеки обновляются, позволяя использовать новейшие криптографические возможности. Нужно лишь задать для Delphi-проекта параметр условной компиляции NT5, после чего вызовы функций, появившихся лишь в Windows 2000, будут нормально работать.

4.2. Знакомство с криптопровайдерами

Функции CryptoAPI обеспечивают прикладным программам доступ к криптографическим возможностям Windows. Однако они являются лишь «передаточным звеном» в сложной цепи обработки информации. Основную работу выполняют скрытые от глаз программиста функции, входящие в специализированные программные (или программно-аппаратные) модули – провайдеры (поставщики) криптографических услуг (CSP – Cryptographic Service Providers), или криптопровайдеры (рис. 4.1).

Программная часть криптопровайдера представляет собой dll-файл, подписанный Microsoft; периодически Windows проверяет

цифровую подпись, что исключает возможность подмены криптопровайдера. Криптопровайдеры отличаются друг от друга:

- составом функций (например, некоторые криптопровайдеры не выполняют шифрование данных, ограничиваясь созданием и проверкой цифровых подписей);
- требованиями к оборудованию (специализированные криптопровайдеры могут требовать устройства для работы со смарт-картами для выполнения аутентификации пользователя);
- алгоритмами, осуществляющими базовые действия (создание ключей, хеширование и пр.).

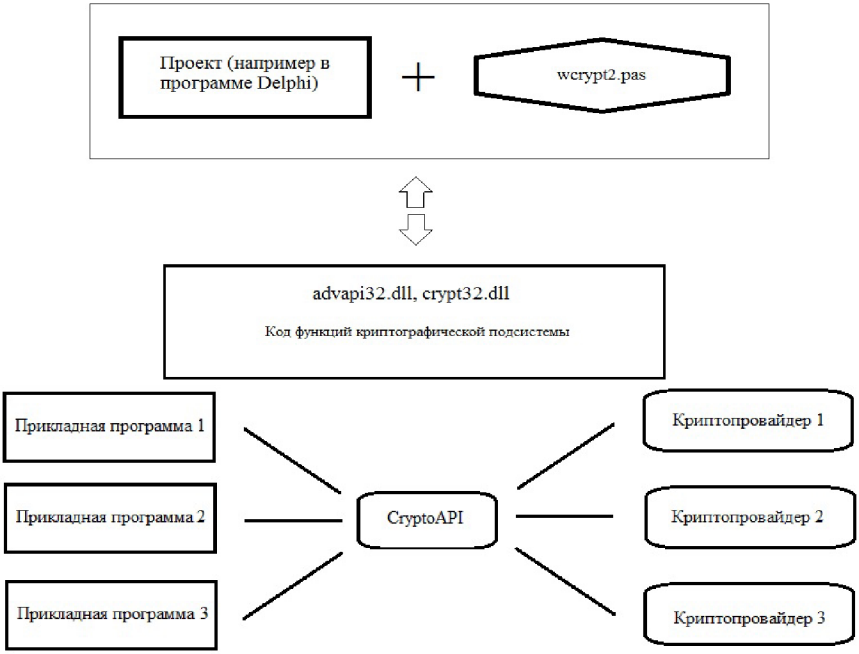

Рис. 4.1. Структура криптопровайдера

По составу функций и обеспечивающих их алгоритмов криптопровайдеры подразделяются на типы. Например, любой CSP типа PROV_RSA_FULL поддерживает как шифрование, так и цифровые

подписи, использует для обмена ключами и создания подписей алгоритм RSA, для шифрования – алгоритмы RC2 и RC4, а для хеширования – MD5 и SHA.

В зависимости от версии операционной системы состав установленных криптопровайдеров может существенно изменяться. Однако на любом компьютере с Windows можно найти Microsoft Base Cryptographic Provider, относящийся к уже известному нам типу PROV_RSA_FULL. Именно с этим провайдером по умолчанию будут взаимодействовать все программы.

Использование криптографических возможностей Windows напоминает работу программы с графическим устройством. Криптопровайдер подобен графическому драйверу: он может обеспечивать взаимодействие программного обеспечения с оборудованием (устройство чтения смарт-карт, аппаратные датчики случайных чисел и пр.). Для вывода информации на графическое устройство приложение не должно непосредственно обращаться к драйверу, вместо этого нужно получить у системы контекст устройства, посредством которого и осуществляются все операции. Это позволяет прикладному программисту использовать графическое устройство, ничего не зная о его аппаратной реализации. Точно так же для использования криптографических функций приложение обращается к криптопровайдеру не напрямую, а через CryptoAPI. При этом вначале необходимо запросить у системы контекст криптопровайдера. Выясним, какие же криптопровайдеры установлены в системе. Для этого нам понадобятся четыре функции CryptoAPI:

- CryptEnumProviders (*i*, резерв, флаги, тип, имя, длина_имени) – возвращает имя и тип *i*-го по порядку криптопровайдера в системе (нумерация начинается с нуля);
- CryptAcquireContext (провайдер, контейнер, имя, тип, флаги) – выполняет подключение к криптопровайдеру с заданным типом и именем и возвращает его дескриптор (контекст). При подключении мы будем передавать функции флаг CRYPT_VERIFYCONTEXT, служащий для получения контекста без подключения к контейнеру ключей;

- CryptGetProvParam (провайдер, параметр, данные, размер_данных, флаги) – возвращает значение указанного параметра провайдера, например, версии (второй параметр при вызове функции – PP_VERSION), типа реализации (программный, аппаратный, смешанный – PP_IMPTYPE), поддерживаемых алгоритмов (PP_ENUMALGS). Список поддерживаемых алгоритмов при помощи этой функции может быть получен следующим образом: при одном вызове функции возвращается информация об одном алгоритме; при первом вызове функции следует передать значение флага CRYPT_FIRST, а при последующих флаг должен быть равен 0;
- CryptReleaseContext (провайдер, флаги) – освобождает дескриптор криптопровайдера.

Каждая из этих функций, как и большинство других функций CryptoAPI, возвращает логическое значение, равное true, в случае успешного завершения, и false – если возникли ошибки. Код ошибки может быть получен при помощи функции GetLastError. Возможные значения кодов ошибки приведены в упоминавшейся выше документации. Например, при вызове функции CryptGetProvParam для получения версии провайдера следует учесть возможность возникновения ошибок следующим образом:

```
        if not CryptGetProvParam(hProv, PP_VERSION, (@vers),
@DataLen, 0)
        then
        begin
        case int64(GetLastError) of
        ERROR_INVALID_HANDLE: err :=
'ERROR_INVALID_HANDLE';
        ERROR_INVALID_PARAMETER: err :=
'ERROR_INVALID_PARAMETER';
        ERROR_MORE_DATA: err := 'ERROR_MORE_DATA';
        ERROR_NO_MORE_ITEMS: err :=
'ERROR_NO_MORE_ITEMS';
        NTE_BAD_FLAGS: err := 'NTE_BAD_FLAGS';
        NTE_BAD_TYPE: err := 'NTE_BAD_TYPE';
```

```
NTE_BAD_UID: err := 'NTE_BAD_UID';
else err := 'Unknown error';
end;
MessageDlg('Error of CryptGetProvParam: ' + err, mtError,
[mbOK], 0);
exit
end;
```

Ниже дан текст процедуры, выводящей в Memo-поле FileMemo формы информацию об установленных в системе криптопровайдерах. Предполагается, что процедура вызывается при выборе соответствующего элемента в главном меню формы. Для краткости в тексте программы опущены фрагменты, выполняющие обработку ошибок.

```
type algInfo = record
algID: ALG_ID;
dwBits: DWORD;
dwNameLen: DWORD;
szName: array[0..100] of char;
end;
{вспомогательная функция, преобразующая тип провайдера в
строку}
function ProvTypeToStr(provType: DWORD): string;
begin
case provType of
PROV_RSA_FULL: ProvTypeToStr := 'RSA full provider';
PROV_RSA_SIG: ProvTypeToStr := 'RSA signature provider';
PROV_DSS: ProvTypeToStr := 'DSS provider';
PROV_DSS_DH: ProvTypeToStr := 'DSS and Diffie-Hellman
provider';
PROV_FORTEZZA: ProvTypeToStr := 'Fortezza provider';
PROV_MS_EXCHANGE: ProvTypeToStr := 'MS Exchange
provider';
PROV_RSA_SCHANNEL: ProvTypeToStr := 'RSA secure chan-
nel provider';
```

```
PROV_SSL: ProvTypeToStr := 'SSL provider';
else ProvTypeToStr := 'Unknown provider';
end;
end;
{вспомогательная функция, преобразующая тип реализации в
строку}
function ImpTypeToStr(it: DWORD): string;
begin
case it of
CRYPT_IMPL_HARDWARE: ImpTypeToStr := 'аппаратный';
CRYPT_IMPL_SOFTWARE: ImpTypeToStr := 'программный';
CRYPT_IMPL_MIXED: ImpTypeToStr := 'смешанный';
CRYPT_IMPL_UNKNOWN: ImpTypeToStr := 'неизвестен';
else ImpTypeToStr := 'неверное значение';
end;
end;
{процедура вывода информации о криптопровайдерах}
procedure TMainForm.InfoItemClick(Sender: TObject);
var i: DWORD;
dwProvType, cbName, DataLen: DWORD;
provName: array[0..200] of char;
vers: array[0..3] of byte;
impType: DWORD;
ai: algInfo;
err: string;
begin
i:= 0;
FileMemo.Clear;
while (CryptEnumProviders(i, nil, 0, {проверяем наличие еще
одного}
@dwProvType, nil, @cbName)) do
begin
if CryptEnumProviders(i, nil, 0,   {получаем имя CSP}
@dwProvType, @provName, @cbName) then
```

```
begin
FileMemo.Lines.Add('Криптопровайдер: '+provName);
FileMemo.Lines.Add('Тип: '+IntToStr(dwProvType)+' – '+
ProvTypeToStr(dwProvType));
if not CryptAcquireContext(@hProv, nil, provName, dwProvType,
CRYPT_VERIFYCONTEXT)
then
begin
{обработка ошибок}
end;
DataLen := 4;
if not CryptGetProvParam(hProv, PP_VERSION, (@vers),
@DataLen, 0)
then
begin
{обработка ошибок}
end;
FileMemo.Lines.Add('Версия: ' + chr(vers[1]+) + '.' +
chr(vers[0]+));
if not CryptGetProvParam(hProv, PP_IMPTYPE, @impType,
@DataLen, 0)
then
begin
{обработка ошибок}
end;
FileMemo.Lines.Add('Тип реализации:
'+ImpTypeToStr(impType));
FileMemo.Lines.Add('Поддерживает алгоритмы:');
DataLen := sizeof(ai);
if not CryptGetProvParam(hProv, PP_ENUMALGS, @ai,
@DataLen, CRYPT_FIRST)
then
begin
{обработка ошибок}
end;
```

```
        with ai do
        FileMemo.Lines.Add(szName+#9+'длина ключа –
'+IntToStr(dwBits)+
        ' бит' +#9+ 'ID: '+IntToStr(AlgID));
        DataLen := sizeof(ai);
        while CryptGetProvParam(hProv, PP_ENUMALGS, @ai,
@DataLen, 0) do
        begin
        with ai do FileMemo.Lines.Add(szName+#9+'длина ключа – '
        +IntToStr(dwBits)+' бит'+#9+'ID: '+IntToStr(AlgID));
        DataLen := sizeof(ai);
        end;
        FileMemo.Lines.Add('');
        CryptReleaseContext(hProv, 0);
        end;
        inc(i);
        end;
        end;
```

4.3. Шифрование с использованием паролей

После того как мы узнали кое-что о структуре CryptoAPI, можно воспользоваться ею в практических целях. Пожалуй, самым ожидаемым действием криптографической подсистемы является шифрование файлов, но так, чтобы лишь пользователь, знающий определенный пароль, мог получить к ним доступ.

Для шифрования данных в CryptoAPI применяются симметричные алгоритмы. Симметричность означает, что для шифрования и расшифровки данных используется один и тот же ключ, известный как шифрующей, так и расшифровывающей стороне. При этом плохо выбранный ключ шифрования может дать противнику возможность взломать шифр. Поэтому одной из функций криптографической подсистемы должна быть генерация «хороших» ключей либо случайным образом, либо на основании некоторой информации, предоставляемой пользователем, например пароля.

В случае создания ключа на основании пароля должно выполняться следующее обязательное условие: при многократном повторении процедуры генерации ключа на одном и том же пароле должны получаться идентичные ключи. Ключ шифрования имеет, как правило, строго определенную длину, определяемую используемым алгоритмом, а длина пароля может быть произвольной. Даже интуитивно понятно, что для однозначной генерации ключей нужно привести разнообразные пароли к некоторой единой форме. Это достигается с помощью хеширования. Хешированием (от англ. hash – разрезать, крошить, перемешивать) называется преобразование строки произвольной длины в битовую последовательность фиксированной длины (хеш-значение, или просто хеш) с обеспечением следующих условий:

- по хеш-значению невозможно восстановить исходное сообщение;
- практически невозможно найти еще один текст, дающий такой же хеш, как и наперед заданное сообщение;
- практически невозможно найти два различных текста, дающих одинаковые хеш-значения (такие ситуации называют коллизиями).

При соблюдении приведенных условий хеш-значение служит компактным цифровым отпечатком (дайджестом) сообщения. Существует множество алгоритмов хеширования. CryptoAPI поддерживает, например, алгоритмы MD5 (MD – Message Digest) и SHA (Secure Hash Algorithm). Итак, чтобы создать ключ шифрования на основании пароля, нам нужно вначале получить хеш этого пароля. Для этого следует создать с помощью CryptoAPI хеш-объект, воспользовавшись функцией CryptCreateHash (провайдер, ID_алгоритма, ключ, флаги, хеш), которой нужно передать дескриптор криптопровайдера (полученный с помощью CryptAcquireContext) и идентификатор алгоритма хеширования (остальные параметры могут быть нулями). В результате мы получим дескриптор хеш-объекта. Этот объект можно представить себе как черный ящик, который принимает любые данные и «перемалывает» их, сохраняя внутри себя лишь хеш-

значение. Подать данные на вход хеш-объекта позволяет функция CryptHashData (дескриптор, данные, размер данных, флаги).

Непосредственно создание ключа выполняет функция CryptDeriveKey (провайдер, ID_алгоритма, хеш-объект, флаги, ключ), которая принимает хеш-объект в качестве исходных данных и строит подходящий ключ для алгоритма шифрования, заданного своим ID. Результатом будет дескриптор ключа, который можно использовать для шифрования (рис. 4.2)

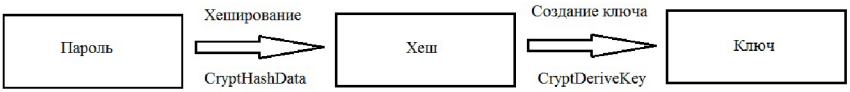

Рис. 4.2. Схема создания ключей

Следует обратить внимание, что при работе с CryptoAPI мы все время имеем дело не с самими объектами или их адресами, а с дескрипторами – целыми числами, характеризующими положение объекта во внутренних таблицах криптопровайдера. Сами таблицы располагаются в защищенной области памяти, так что программы-«шпионы» не могут получить к ним доступ.

Алгоритмы шифрования, поддерживаемые CryptoAPI, можно разделить на блочные и поточные: первые обрабатывают данные относительно большими по размеру блоками (например, 64, 128 битов или более), а вторые – побитно (теоретически, на практике же – побайтно). Если размер данных, подлежащих шифрованию, не кратен размеру блока, то последний, неполный блок данных, будет дополнен необходимым количеством случайных битов, в результате чего размер зашифрованной информации может несколько увеличиться. Разумеется, при использовании поточных шифров размер данных при шифровании остается неизменным. Шифрование выполняется функцией CryptEncrypt (ключ, хеш, финал, флаги, данные, рамер_данных, размер_буфера):

- через параметр ключ передается дескриптор ключа шифрования;

- параметр хеш используется, если одновременно с шифрованием нужно вычислить хеш-значение шифруемого текста;
- параметр финал равен true, если шифруемый блок текста – последний или единственный (шифрование можно осуществлять частями, вызывая функцию CryptEncrypt несколько раз);
- значение флага должно быть нулевым;
- параметр «данные» представляет собой адрес буфера, в котором при вызове функции находится исходный текст, а по завершении работы функции – зашифрованный;
- следующий параметр, соответственно, описывает размер входных/выходных данных,
- последний параметр задает размер буфера, если в результате шифрования зашифрованный текст не уместится в буфере, возникнет ошибка.

Для расшифровки данных используется функция CryptDecrypt (ключ, хеш, финал, флаги, данные, размер_данных), отличающаяся от шифрующей функции только тем, что размер буфера указывать не следует: поскольку размер данных при расшифровке может только уменьшиться, отведенного под них буфера наверняка будет достаточно. Приведем лишь фрагменты программы, реализующей шифрование файла с использованием заданного пароля, опустив громоздкие проверки успешности выполнения криптографических операций (что в реальной программе делать крайне нежелательно).

```
{«описание» используемых переменных}
hProv: HCRYPTPROV;
hash: HCRYPTHASH;
password: string;
key: HCRYPTKEY;
plaintext, ciphertext: string;
inFile, outFile: file;
data: PByte;
l: DWORD;

{получаем контекст криптопровайдера}
```

```
CryptAcquireContext(@hProv, nil, nil, PROV_RSA_FULL,
CRYPT_VERIFYCONTEXT);
{создаем хеш-объект}
CryptCreateHash(hProv, CALG_SHA, 0, 0, @hash);
{хешируем пароль}
CryptHashData(hash, @password[1], length(password), 0);
{создаем ключ на основании пароля для потокового шифра RC4}
CryptDeriveKey(hProv, CALG_RC4, hash, 0, @key);
{уничтожаем хеш-объект}
CryptDestroyHash(hash);
{открываем файлы}
AssignFile(inFile, plaintext);
AssignFile(outFile, ciphertext);
reset(inFile, 1);
rewrite(outFile, 1);
{выделяем место для буфера}
GetMem(data, 512);
{шифруем данные}
while not eof(inFile) do
begin
BlockRead(inFile, data^, 512, l);
CryptEncrypt(key, 0, eof(inFile), 0, data, @l, l);
BlockWrite(outFile, data^, l);
end;
{освобождаем место и закрываем файлы}
FreeMem(data, 512);
CloseFile(inFile);
CloseFile(outFile);
{освобождаем контекст криптопровайдера}
CryptReleaseContext(hProv, 0);
```

Конечно, шифрование вами всех файлов одним и тем же паролем облегчает «противнику» задачу их расшифровки, запоминание огромного числа паролей сильно усложняет жизнь, а их записывание в незашифрованном виде создает опасность раскрытия всей сис-

темы. CryptoAPI может предложить на этот случай ряд решений. В арсенале защиты должны быть не только методы, обеспечивающие секретность передачи информации (о них мы писали в первой части статьи). Не менее важными инструментами безопасности являются процедуры, позволяющие убедиться в целости и аутентичности данных. Кроме того, необходимо решать проблемы безопасного хранения и распределения ключей.

4.4. Проблема распределения ключей

В прошлый раз при помощи CryptoAPI мы решали такую «классическую» задачу, как шифрование на основе пароля. Напомним, что пароль использовался для создания ключа шифрования какого-либо симметричного алгоритма. В таком случае расшифровать файл может лишь тот, кто знает пароль. А значит, для обеспечения конфиденциальности нужно держать пароль в строжайшем секрете – желательно, чтобы его знали лишь отправитель и получатель информации. (А еще лучше, если отправитель и получатель – одно и то же лицо.)

Предположим, что отправитель и получатель при личной встрече договорились использовать для конфиденциальной переписки определенный пароль. Но если они будут шифровать все свои сообщения одним и тем же ключом, то возможный противник, перехватив корреспонденцию, будеть иметь хорошие шансы взломать шифр: при современных методах криптоанализа наличие нескольких шифр-текстов, полученных путем использования одного и того же ключа, почти гарантирует успешный результат. Поэтому при использовании симметричных алгоритмов шифрования настоятельно рекомендуется не применять один и тот же ключ дважды!

Однако помнить отдельный пароль для каждого зашифрованного сообщения – задача достаточно трудоемкая. А для корреспондентов, не имеющих возможности встретиться лично для согласования ключей шифрования, конфиденциальный обмен сообщениями вообще становится недоступным. Такая практическая трудность называется проблемой распределения ключей.

Спасительный способ, позволяющий шифровать сообщения, обмениваясь ключами по открытым каналам связи, был придуман в середине 70-х годов прошлого столетия, а в начале 80-х появился первый реализующий его алгоритм – RSA. Теперь пользователь может сгенерировать два связанных между собой ключа – ключевую пару. Один из этих ключей по несекретным каналам рассылается всем, с кем пользователь хотел бы обмениваться конфиденциальными сообщениями.

Этот ключ называют открытым (англ. public key). Зная открытый ключ пользователя, можно зашифровать адресованное ему сообщение, но вот расшифровать его позволяет лишь вторая часть ключевой пары – закрытый ключ (private key). При этом открытый ключ не дает «практической» возможности вычислить закрытый: такая задача, хотя и разрешима в принципе, но при достаточно большом размере ключа требует многих лет машинного времени. Для сохранения конфиденциальности получателю необходимо лишь хранить в строгом секрете свой закрытый ключ, а отправителю – убедиться, что имеющийся у него открытый ключ действительно принадлежит адресату.

Поскольку для шифрования и расшифровки используются различные ключи, алгоритмы такого рода назвали асимметричными. Наиболее существенным их недостатком является низкая производительность – они примерно в 100 раз медленнее симметричных алгоритмов. Поэтому были созданы криптографические схемы, использующие преимущества как симметричных, так и асимметричных алгоритмов:

- для шифрования файла или сообщения используется быстрый симметричный алгоритм, причем ключ шифрования генерируется случайным образом с обеспечением «хороших» статистических свойств;
- небольшой по размерам симметричный ключ шифрования шифруется при помощи асимметричного алгоритма с использованием открытого ключа адресата и в зашифрованном виде пересылается вместе с сообщением;

- получив сообщение, адресат своим закрытым ключом расшифровывает симметричный ключ, а с его помощью – и само сообщение.

4.5. Целостность и аутентичность информации

Как удостовериться в том, что пришедшее сообщение действительно отправлено тем, чье имя стоит в графе «отправитель»? Асимметричные схемы шифрования дают нам элегантный способ аутентификации. Если отправитель зашифрует сообщение своим закрытым ключом, то успешное расшифровывание убедит получателя в том, что послать корреспонденцию мог только хозяин ключевой пары. При этом расшифровку может выполнить любой, кто имеет открытый ключ отправителя. Ведь наша цель – не конфиденциальность, а аутентификация.

Чтобы избежать шифрования всего сообщения при помощи асимметричных алгоритмов, используют хеширование: вычисляется хеш-значение исходного сообщения, и только эта короткая последовательность байтов шифруется закрытым ключом отправителя. Результат представляет собой электронную цифровую подпись. Добавление такой подписи к сообщению позволяет установить:

- аутентичность сообщения – создать подпись на основе закрытого ключа мог только его хозяин;
- целостность данных – легко вычислить хеш-значение полученного сообщения и сравнить его с тем, которое хранится в подписи: если значения совпадают, значит, сообщение не было изменено злоумышленником после того, как отправитель его подписал.

Таким образом, асимметричные алгоритмы позволяют решить две непростые задачи: обмена ключами шифрования по открытым каналам связи и подписи сообщения. Чтобы воспользоваться этими возможностями, нужно сгенерировать и сохранить две ключевые пары – для обмена ключами и для подписей. В этом нам поможет CryptoAPI.

5. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ

Создадим программу при помощи функции CryptoAPI. На рис. 5.1 показан принцип работы программы.

5.1. Создание контейнера

Каждый криптопровайдер располагает базой данных, в которой хранятся долговременные ключи пользователей. База данных содержит один или более контейнеров ключей.

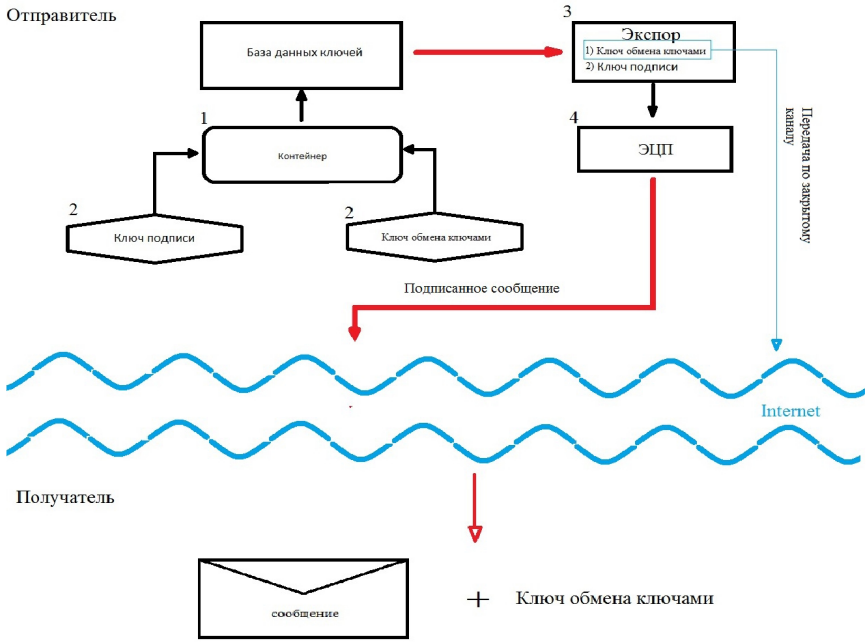

Рис. 5.1. Принцип работы

Пользователь может создать несколько контейнеров с различными именами (именем контейнера по умолчанию является имя пользователя в системе). Подключение к контейнеру производится одновременно с получением контекста криптопровайдера при вызо-

ве функции CryptAcquireContext – имя контейнера ключей передается функции вторым ее аргументом. Если второй аргумент содержит пустой указатель (nil), то используется имя по умолчанию, т. е. имя пользователя. В том случае, если доступ к контейнеру не нужен, можно передать в последнем аргументе функции флаг CRYPT_VERIFYCONTEXT; при необходимости создать новый контейнер используется флаг CRYPT_NEWKEYSET, а для удаления существующего контейнера вместе с хранящимися в нем ключами – CRYPT_DELETEKEYSET.

Каждый контейнер может содержать, как минимум, две ключевые пары – ключ обмена ключами и ключ подписи. Ключи, используемые для шифрования симметричными алгоритмами, не сохраняются. Как мы уже указывали, такие ключи не рекомендуется применять более одного раза, поэтому их называют сеансовыми (англ. session key).

Окно программы в которой создается контейнер (рис. 5.2).

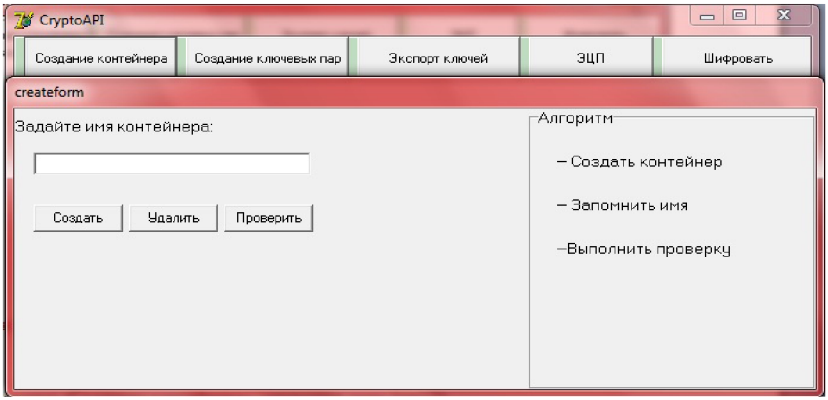

Рис. 5.2. Создание контейнера

5.2. Создание ключевых пар

После создания контейнера ключей необходимо сгенерировать ключевые пары обмена ключами и подписи. Эту работу в CryptoAPI выполняет функция CryptGenKey (провайдер, алгоритм, флаги, ключ):

- провайдер – дескриптор криптопровайдера, полученный в результате обращения к функции CryptAcquireContext;
- алгоритм – указывает, какому алгоритму шифрования будет соответствовать создаваемый ключ. Информация об алгоритме, таким образом, является частью описания ключа. Каждый криптопровайдер использует для обмена ключами и подписи строго определенные алгоритмы. Так, провайдеры типа PROV_RSA_FULL, к которым относится и Microsoft Base Cryptographic Provider, реализуют алгоритм RSA. Но при генерации ключей знать это не обязательно: достаточно указать, какой ключ мы собираемся создать – обмена ключами или подписи. Для этого используются мнемонические константы AT_KEYEXCHANGE и AT_SIGNATURE;
- флаги – при создании асимметричных ключей этот параметр управляет их размером. Используемый нами криптопровайдер позволяет генерировать ключ обмена ключами длиной от 384 до 512 бит**, а ключ подписи – от 512 до 16384 бит. Чем больше длина ключа, тем выше его надежность, поэтому трудно найти причины для использования ключа обмена ключами длиной менее 512 бит, а длину ключа подписи не рекомендуется делать меньше 1024 бит**. По умолчанию криптопровайдер создает оба ключа длиной 512 бит. Необходимую длину ключа можно передать в старшем слове параметра «флаги»;
- ключ – в случае успешного завершения функции в этот параметр заносится дескриптор созданного ключа. Рассмотрим пример создания ключевых пар при помощи формы нашей программы, показанной на (рис. 5.3).

В поле «Контейнер» можно указать имя контейнера ключей; если оставить это поле пустым, будет использован контейнер по умолчанию. Назначение остальных элементов управления должно быть интуитивно понятным. После генерации ключа в memo-поле выводится отчет о его параметрах. Для этого используется функция CryptGetKeyParam (ключ, параметр, буфер, размер, флаги). Чтобы получить информацию о требуемом параметре, нужно через второй аргумент функции передать соответствующую константу: KP_ALGID – идентификатор алгоритма, KP_KEYLEN – размер ключа, и т.д.

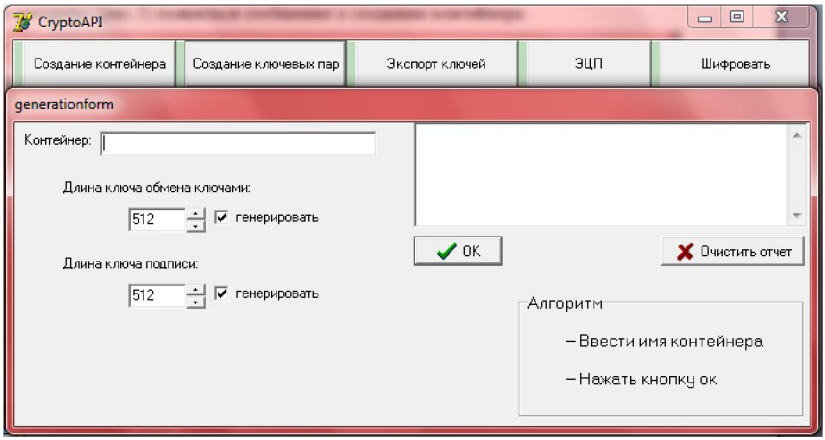

Рис. 5.3. Создание ключей

Вот так будет выглядеть наша программа:

```
procedure TGenerateForm.OKBtnClick(Sender: TObject);
var cont: PChar;
err: string;
hProv: HCRYPTPROV;
KeyExchKey, SignKey: HCRYPTKEY;
flag, keyLen: DWORD;
begin
{если ни один ключ не выбран – выход}
if not (KEKCheckBox.Checked or SKCheckBox.Checked) then exit;
{«считываем» имя контейнера}
if length(ContainerEdit.Text) = 0
then cont := nil
else
begin
err := ContainerEdit.Text;
cont := StrAlloc(length(err) + 1);
StrPCopy(cont, err);
end;
```

```
        CryptAcquireContext(@hProv, cont, nil, PROV_RSA_FULL, 0);
        {генерация ключа обмена ключами (Key Exchange Key)}
        if KEKCheckBox.Checked then
        begin
        {«считываем» длину ключа и помещаем ее в
        старшее слово параметра ФЛАГИ}
        keyLen := strtoint(KeyExchLenEdit.text);
        flag := keyLen shl 16;
        if not CryptGenKey(hProv, AT_KEYEXCHANGE, flag,
@KeyExchKey) then
        begin
        {«обработка ошибок»}
        end
        else
        begin
        ReportMemo.Lines.Add('');
        ReportMemo.Lines.Add('Создан ключ обмена ключами:');
        flag := 4;
        if not CryptGetKeyParam(KeyExchKey, KP_KEYLEN, @keyLen,
@flag, 0) then
        begin
        jобработка ошибокj
        end
        else ReportMemo.Lines.Add(' длина ключа – ' +
inttostr(keyLen));
        flag := 4;
        if not CryptGetKeyParam(KeyExchKey, KP_ALGID, @keyLen,
@flag, 0) then
        begin
        jобработка ошибокj
        end
        else ReportMemo.Lines.Add(' алгоритм – ' + algIDtostr(keyLen));
        {функция algIDtostr здесь не приводится. Она состоит из един-
ственного
```

```
оператора case, отображающего целый идентификатор алго-
ритма в строку}
end;
end;
{генерация ключа подписи (Signature Key)}
if SKCheckBox.Checked then
begin
{«выполняется аналогично генерации ключа обмена
ключами»}
end;
CryptReleaseContext(hProv, 0);
end;
```

5.3. Обмен ключами

Теперь мы располагаем набором ключей, однако все они останутся мертвым грузом до тех пор, пока мы не получим возможности обмена с другими пользователями открытыми ключами. Для этого необходимо извлечь их из базы данных ключей и записать в файл, который можно будет передать своим корреспондентам. При экспорте данные ключа сохраняются в одном из трех возможных форматов:

- PUBLICKEYBLOB – используется для сохранения открытых ключей. Поскольку открытые ключи не являются секретными, они сохраняются в незашифрованном виде;
- PRIVATEKEYBLOB – используется для сохранения ключевой пары целиком (открытого и закрытого ключей). Эти данные являются в высшей степени секретными, поэтому сохраняются в зашифрованном виде, причем для шифрования используется сеансовый ключ (и, соответственно, симметричный алгоритм);
- SIMPLEBLOB – используется для сохранения сеансовых ключей. Для обеспечения секретности данные ключа шифруются с использованием открытого ключа получателя сообщения.

Экспорт ключей в CryptoAPI выполняется функцией CryptExportKey (экспортируемый ключ, ключ адресата, формат, флаги, буфер, размер буфера):

- экспортируемый ключ – дескриптор нужного ключа;
- ключ адресата – в случае сохранения открытого ключа должен быть равен нулю (данные не шифруются);
- формат – указывается один из возможных форматов экспорта (PUBLICKEYBLOB, PRIVATEKEYBLOB, SIMPLEBLOB);
- флаги – зарезервирован на будущее (должен быть равен нулю);
- буфер – содержит адрес буфера, в который будет записан ключевой BLOB (Binary Large OBject – большой двоичный объект);
- размер буфера – при вызове функции в этой переменной должен находиться доступный размер буфера, а по окончании работы в нее записывается количество экспортируемых данных. Если размер буфера заранее не известен, то функцию нужно вызвать с параметром «буфер», равным пустому указателю, тогда размер буфера будет вычислен и занесен в переменную «размер буфера».

Экспорт ключевой пары целиком, включая и закрытый ключ, может понадобиться для того, чтобы иметь возможность подписывать документы на различных компьютерах (например, дома и на работе), или для сохранения страховочной копии. В этом случае нужно создать ключ шифрования на основании пароля и передать дескриптор этого ключа в качестве второго параметра функции CryptExportKey. Запросить у криптопровайдера дескриптор самого' экспортируемого ключа позволяет функция CryptGetUserKey (провайдер, описание ключа, дескриптор ключа). Описание ключа – это либо AT_KEYEXCHANGE, либо AT_SIGNATURE. Экспорт асимметричных ключей во всем возможном многообразии можно осуществить при помощи формы, показанной на (рис. 5.4).

Код программы:

```
begin
{если ни один ключ не выбран – выход}
if not (KEKCheckBox.Checked or SKCheckBox.Checked) then exit;
{если нужен пароль, т.е. экспортируется ключевая пара целиком}
if PasswEdit.Enabled and (PasswEdit.Text <> Passw2Edit.Text) then
begin
```

```
MessageDlg('Ошибка при вводе пароля! Повторите ввод.', mtError, [mbOK], 0);
exit;
end;
…
«считываем» имя контейнера и подключаемся к криптопровайдеру
...
если нужен ключ шифрования – создаем его на основании пароля
…
{ключ обмена ключами}
if KEKCheckBox.Checked then
repeat
{получаем дескриптор ключа}
CryptGetUserKey(hProv, AT_KEYEXCHANGE, @key);
{пытаемся определить размер буфера для экспорта ключа}
if (WhatRadioGroup.ItemIndex = 0) then
CryptExportKey(key, 0, PUBLICKEYBLOB, 0, nil, @bufLen)
else CryptExportKey(key, expKey, PRIVATEKEYBLOB, 0, nil, @bufLen);
GetMem(pbuf, bufLen);
{экспортируем данные}
if (WhatRadioGroup.ItemIndex = 0) then
CryptExportKey(key, 0, PUBLICKEYBLOB, 0, pbuf, @bufLen)
else CryptExportKey(key, expKey, PRIVATEKEYBLOB, 0, pbuf, @bufLen);
{освобождаем дескриптор ключа обмена ключами
(сам ключ при этом не уничтожается)}
CryptDestroyKey(key);
SaveDialog1.Title := 'Укажите файл для сохранения ключа обмена ключами';
if SaveDialog1.Execute then
begin
AssignFile(f, SaveDialog1.FileName);
```

```
        rewrite(f, 1);
        BlockWrite(f, pbuf^, bufLen);
        CloseFile(f);
        MessageDlg('Ключ обмена ключами успешно сохранен',
mtInformation, [mbOK], 0);
        end;
        until true; {KeyExchange}
        {ключ подписи}
        if SKCheckBox.Checked then
        repeat
        {аналогично ключу обмена ключами}
        until true; {Signature}
        …
        если создавался ключ на основании пароля – уничтожаем его,
        после чего освобождаем контекст криптопровайдера
        …
        end;
```

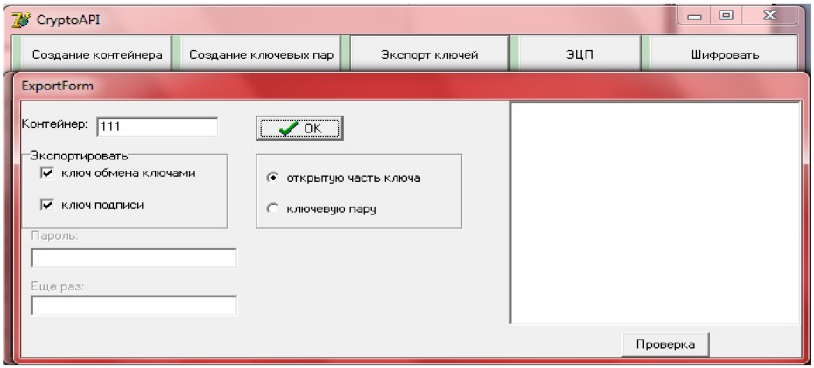

Рис. 5.4. Экспорт асимметричных ключей

5.4. Электронная цифровая подпись

Для создания электронной цифровой подписи необходимо вычислить хеш заданного файла и зашифровать этот «цифровой отпечаток сообщения» своим закрытым ключом – «подписать». Чтобы

подпись впоследствии можно было проверить, необходимо указать, какой алгоритм хеширования использовался при ее создании. Поэтому подписанное сообщение должно иметь структуру, показанную на рис. 5.5.

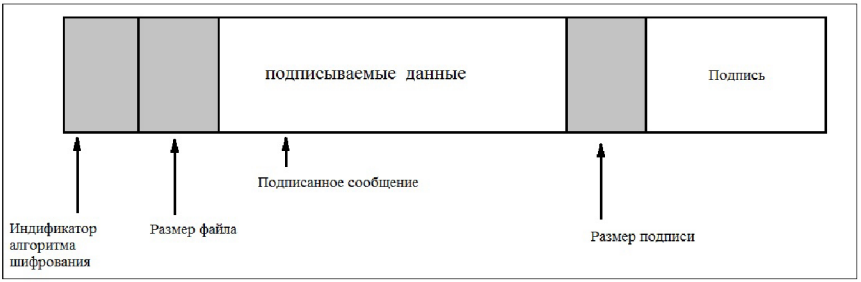

Рис. 5.5. Электронно-цифровая подпись

Подписать вычисленный хеш в CryptoAPI позволяет функция CryptSignHash (хеш, описание ключа, комментарий, флаги, подпись, длина подписи). Вторым параметром может быть либо AT_KEYEXCHANGE, либо AT_SIGNATURE (в нашем случае логичнее использовать ключ подписи). Третий параметр в целях безопасности настоятельно рекомендуется оставлять пустым (nil). Флаги в настоящее время также не используются – на месте этого аргумента должен быть нуль. Готовую электронную подпись функция запишет в буфер, адрес которого содержится в предпоследнем параметре, последний же параметр будет содержать длину подписи в байтах.

На рис. 5.6 показана форма, позволяющая подписать заданный файл.

Код программы:

```
procedure TSigningForm.SignBtnClick(Sender: TObject);
var cont: PChar;
err: string;
hProv: HCRYPTPROV;
key: HCRYPTKEY;
alg: ALG_ID;
hash: HCRYPTHASH;
```

```
infile, outfile: file;
size: DWORD;
buf: array [0..511] of byte;
signature: PBYTE;
begin
{проверка существования выбранного файла}
if not FileExists(DataNameEdit.Text) then
begin
MessageDlg('Неверное имя файла!', mtError, [mbOK], 0);
exit;
end;
AssignFile(infile, DataNameEdit.Text);
…
«считываем» имя контейнера и подключаемся к нему
…
case HashRadioGroup.ItemIndex of
0: alg := CALG_MD5;
1: alg := CALG_SHA;
end;
CryptCreateHash(hProv, alg, 0, 0, @hash);
SaveDialog1.Title := 'Задайте имя файла для хранения подпи-
санных данных';
if SaveDialog1.Execute then
begin
AssignFile(outfile, SaveDialog1.FileName);
rewrite(outfile, 1);
{записываем в файл идентификатор алгоритма хеширования}
BlockWrite(outfile, alg, 4);
reset(infile, 1);
size := FileSize(infile);
{записываем размер подписываемых данных}
BlockWrite(outfile, size, 4);
{пишем сами данные и вычисляем хеш:}
while not eof(infile) do
```

```
begin
BlockRead(infile, buf, 512, size);
BlockWrite(outFile, buf, size);
CryptHashData(hash, @buf, size, 0);
end;
CloseFile(infile);
{выясняем размер подписи}
CryptSignHash(hash, AT_SIGNATURE, nil, 0, nil, @size);
{создаем подпись}
GetMem(signature, size);
CryptSignHash(hash, AT_SIGNATURE, nil, 0, signature, @size);
BlockWrite(outfile, size, 4);
BlockWrite(outfile, signature^, size);
CloseFile(outfile);
end;
…
уничтожаем хеш-объект и освобождаем контекст
…
end;
```

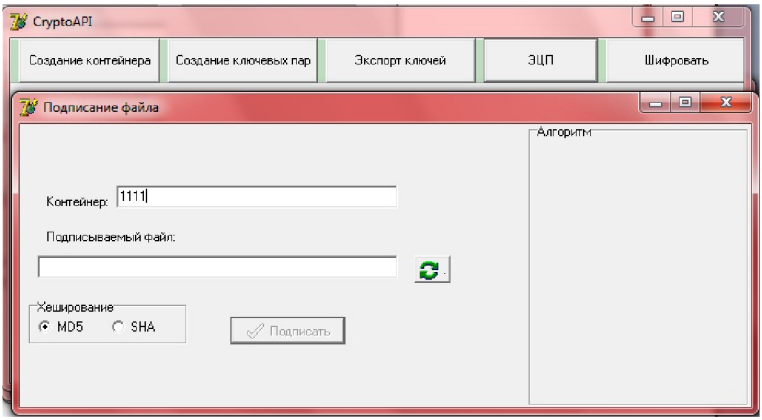

Рис. 5.6. Подписание файла

Чтобы проверить правильность подписи, получатель подписанного сообщения должен иметь файл с открытым ключом подписи отправителя. В процессе проверки подписи этот ключ импортируется внутрь криптопровайдера. Проверка выполняется функцией CryptVerifySignature (хеш, подпись, длина подписи, открытый ключ, комментарий, флаги). О последних двух аргументах можно сказать то же, что и о параметрах «комментарий» и «флаги» функции CryptSignHash, назначение же остальных должно быть понятно. Если подпись верна, функция возвращает true. Значение false в качестве результата может свидетельствовать либо о возникновении ошибки в процессе проверки, либо о том, что подпись оказалась неверной. В последнем случае функция GetLastError вернет ошибку NTE_BAD_SIGNATURE. Для примера приведем наиболее значимые фрагменты программы проверки подписи. Форма в программе представлена на рис. 5.7.

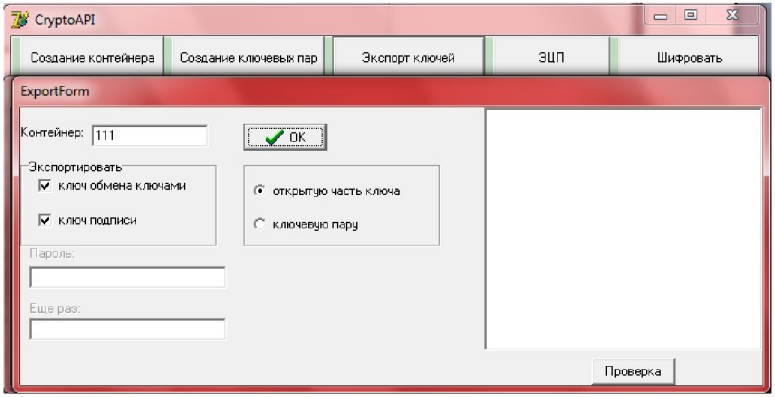

Рис.5.7. Проверка правильности подписи

Код программы:

```
procedure TMainForm.VerifyItemClick(Sender: TObject);
var err: string;
hProv: HCRYPTPROV;
key: HCRYPTKEY;
alg: ALG_ID;
```

```
hash: HCRYPTHASH;
infile: file;
size, test, textsize: DWORD;
buf: PBYTE;
signature, signkey: PBYTE;
begin
…
получаем контекст криптопровайдера
…
OpenDialog1.Title := 'Укажите файл с подписанными данными';
if OpenDialog1.Execute then
begin
AssignFile(infile, OpenDialog1.FileName);
reset(infile, 1);
{считываем идентификатор алгоритма хеширования}
BlockRead(infile, alg, 4);
{считываем размер подписанных данных и сами данные}
BlockRead(infile, textsize, 4);
GetMem(buf, textsize);
BlockRead(infile, buf^, textsize, test);
if test < textsize then
begin
MessageDlg('Неверный формат файла! Процесс прерван.', mtError, [mbOK], 0);
exit;
end;
{считываем размер подписи и саму подпись}
BlockRead(infile, test, 4);
GetMem(signature, test);
BlockRead(infile, signature^, test);
CloseFile(infile);
end
else exit;
```

```
…
создаем хеш-объект и хешируем данные
…
OpenDialog1.Title := 'Укажите файл с открытым ключом подписи';
if OpenDialog1.Execute then
begin
AssignFile(infile, OpenDialog1.FileName);
reset(infile, 1);
size := FileSize(infile);
GetMem(signkey, size);
BlockRead(infile, signkey^, size);
CloseFile(infile);
end
else exit;
{импортируем открытый ключ подписи отправителя}
CryptImportKey(hProv, signkey, size, 0, 0, @key);
FreeMem(signkey, size);
{проверяем подпись}
if CryptVerifySignature(hash, signature, test, key, nil, 0) then
begin
MessageDlg('Подпись верна.', mtInformation, [mbOK], 0);
{сохраняем подписанные данные}
SaveDialog1.Title := 'Укажите имя файла для сохранения данных';
if SaveDialog1.Execute then
begin
AssignFile(infile, SaveDialog1.FileName);
rewrite(infile, 1);
BlockWrite(infile, buf^, textsize);
CloseFile(infile);
end;
end
else
```

```
begin
case int64(GetLastError) of
NTE_BAD_SIGNATURE: err := 'Подпись неверна!';
{обработка других ошибок}
else err := 'Ошибка при проверке подписи: Unknown error';
end;
MessageDlg(err, mtError, [mbOK], 0);
end;
…
уничтожаем хеш-объект и импортированный ключ
и освобождаем контекст криптопровайдера
…
end;
```

5.5. Способы и методы безопасной передачи шифрованных данных

Для конфиденциального обмена информацией с корреспондентом в любой точке земного шара приходится использовать целый арсенал современных криптографических инструментов: симметричные и асимметричные алгоритмы шифрования, механизмы генерирования криптографических ключей и случайных последовательностей, специфические режимы работы шифров и пр. Теперь рассмотрим реализацию этих инструментов в CryptoAPI и воспользуемся ими для шифрования файла случайным ключом.

Асимметричные алгоритмы позволяют легко обменяться ключами шифрования по открытому каналу связи, но работают слишком медленно. Симметричные алгоритмы работают быстро, но для обмена ключами требуют наличия защищенного канала связи и к тому же нуждаются в частой смене ключей. Поэтому в современных криптосистемах используются сильные стороны обоих подходов. Так, для шифрования сообщения используется симметричный алгоритм со случайным ключом шифрования, действующим только в пределах одного сеанса, – сеансовым ключом. Чтобы впоследствии сообщение могло быть расшифровано, сеансовый ключ подвергает-

ся шифрованию асимметричным алгоритмом с использованием открытого ключа получателя сообщения. Зашифрованный таким образом сеансовый ключ сохраняется вместе с сообщением, образуя цифровой конверт. При необходимости цифровой конверт может содержать сеансовый ключ в нескольких экземплярах, зашифрованный открытыми ключами различных получателей (рис. 5.8).

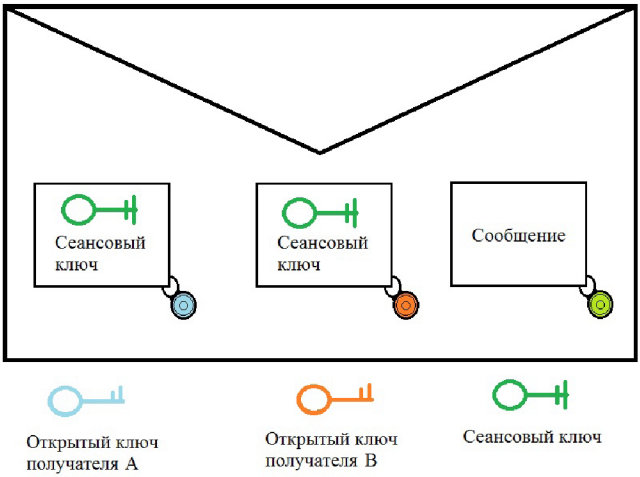

Рис. 5.8. Цифровой конверт

5.6. Создание сеансовых ключей

CryptoAPI позволяет генерировать сеансовые ключи случайным образом, эту работу выполняет функция CryptGenKey, о чем писалось ранее. Однако при использовании этой возможности за пределами США и Канады приходится учитывать американские ограничения на экспорт средств «сильной криптографии». В частности, до января 2000 года был запрещен экспорт программного обеспечения для шифрования с использованием ключей длиной более 40 бит. Этим объясняется разработка Microsoft двух версий своего криптопровайдера – базовой и расширенной. Базовая версия предназначалась на экспорт и поддерживала симметричные ключи длиной 40 бит; расширенная же версия (Microsoft Enhanced Cryptographic

Provider) работала с «полной» длиной ключа (128 бит). Поскольку алгоритм шифрования, как правило, требует использования ключа строго определенной длины, недостающее количество бит в урезанном «экспортном» ключе могло быть заполнено либо нулями, либо случайными данными, которые предлагалось передавать открыто.

В криптографической практике внесение в состав ключа определенной части несекретных данных, которые сменяются несколько раз в ходе обработки исходного или шифр-текста, используется для того, чтобы воспрепятствовать взлому шифра атакой «по словарю». В английской терминологии такие вставки называются salt values: их назначение – «подсолить» ключ (с учетом нашей ментальности можно перевести как «насолить» противнику). Поскольку этот термин используется и в CryptoAPI, будем употреблять его в транслитерированном виде – солт-значения.

Итак, CryptoAPI, в экспортном исполнении практически вынуждает нас использовать солт-значения, составляющие бОльшую часть ключа – 88 бит из 128 для симметричных алгоритмов в RC2, и RC4. Конечно, при такой эффективной длине ключа криптозащита не может считаться достаточно надежной. В реальной ситуации выход один – воспользоваться криптопровайдером, не ограничивающим длину ключа. Обладатели Windows XP могут прибегнуть к услугам расширенных версий провайдера Microsoft (Enhanced или Strong). Пользователям более старых версий Windows, по-видимому, придется воспользоваться продуктами сторонних разработчиков.

Например, свои версии криптопровайдеров предлагают российская компания «Крипто-Про» и шведская «StreamSec». В Украине, насколько известно авторам, разработкой национального провайдера криптографических услуг занимается коллектив харьковских ученых под руководством профессора Горбенко, однако до широкого внедрения дело пока не дошло. Тем не менее благодаря архитектуре CryptoAPI прикладные программы могут разрабатываться и отлаживаться и с базовым провайдером Microsoft, так как интерфейс взаимодействия остается неизменным. Поэтому вернемся к обсуждению создания случайных сеансовых ключей.

Солт может быть сгенерирован вместе с ключом: для этого нужно в качестве флага передать функции CryptGenKey (или CryptDeriveKey) константу CRYPT_CREATE_SALT. Правда, при сохранении ключа (с помощью функции CryptExportKey) система уже не заботится о солт-значении, перекладывая ответственность на прикладную программу. Таким образом, корректная процедура создания и сохранения симметричного ключа предполагает блочные цифры.

5.7. Блочные шифры

Блочные шифры считаются более надежными, нежели поточные, поскольку каждый блок текста подвергается сложным преобразованиям. Тем не менее одних только этих преобразований оказывается недостаточно для обеспечения должного уровня безопасности, а важно, каким образом они применяются к исходному тексту в процессе шифрования.

Наиболее простой и интуитивно понятный способ состоит в том, чтобы разбить исходный текст на блоки соответствующего размера, а затем отдельно каждый блок подвергнуть шифрующему преобразованию. Такой режим использования блочных шифров называют электронной кодовой книгой (ECB – electronic codebook). Его главный недостаток состоит в том, что одинаковые блоки исходного текста при шифровании дадут одинаковые же блоки шифр-текста, а это может существенно облегчить противнику задачу взлома. Поэтому режим ECB не рекомендуется использовать при шифровании текстов, по длине превышающих один блок. В таких случаях лучше воспользоваться одним из режимов, связывающих различные блоки между собой. По умолчанию в CryptoAPI блочные шифры используются в режиме сцепления блоков шифр-текста (CBC – cipher block chaining). В этом режиме при шифровании очередной блок исходного текста вначале комбинируется с предыдущим блоком шифр-текста (при помощи побитового исключающего ИЛИ), а затем полученная последовательность битов поступает на вход блочного шифра (рис. 5.9).

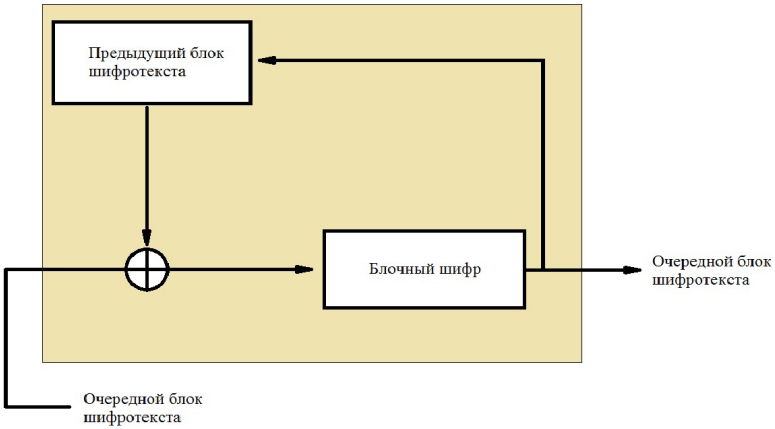

Рис. 5.9. Электронная кодовая книга

Образующийся на выходе блок шифр-текста используется для шифрования следующего блока. Самый первый блок исходного текста также должен быть скомбинирован с некоторой последовательностью битов, но «предыдущего блока шифр-текста» еще нет; поэтому режимы шифрования с обратной связью требуют использования еще одного параметра, он называется инициализирующим вектором (IV – initialization vector). Инициализирующий вектор должен генерироваться отдельно с помощью уже известной нам функции CryptGenRandom и, как и солт-значение, передаваться вместе с ключом в открытом виде. Размер IV равен длине блока шифра. Например, для алгоритма RC2, поддерживаемого базовым криптопровайдером Microsoft, размер блока составляет 64 бита (8 байтов).

```
CryptSignHash(hash, AT_SIGNATURE, nil, 0, nil, @size);
{создаем подпись}
GetMem(signature, size);
CryptSignHash(hash, AT_SIGNATURE, nil, 0, signature, @size);
BlockWrite(outfile, size, 4);
BlockWrite(outfile, signature^, size);
CloseFile(outfile);
end;
```

```
…
уничтожаем хеш-объект и освобождаем контекст
…
end;
```

Чтобы проверить правильность подписи, получатель подписанного сообщения должен иметь файл с открытым ключом подписи отправителя. В процессе проверки подписи этот ключ импортируется внутрь криптопровайдера. Проверка выполняется функцией CryptVerifySignature (хеш, подпись, длина подписи, открытый ключ, комментарий, флаги). О последних двух аргументах можно сказать то же, что и о параметрах «комментарий» и «флаги» функции CryptSignHash, назначение же остальных должно быть понятно. Если подпись верна, функция возвращает true. Значение false в качестве результата может свидетельствовать либо о возникновении ошибки в процессе проверки, либо о том, что подпись оказалась неверной. В последнем случае функция GetLastError вернет ошибку NTE_BAD_SIGNATURE.

6. РАБОЧАЯ ИНСТРУКЦИЯ. ОПИСАНИЕ ИСПОЛЬЗОВАНИЯ ПРОГРАММЫ CRYPTO

Для примера приведем наиболее значимые фрагменты программы проверки подписи. Форма в программе (рис. 6.1–6.9). Описание программ представлено в виде двух частей:

I) Инструкция для отправителя

II) Инструкция для получателя

6.1. Инструкция для отправителя

Перед открытием программы необходимо подготовить файл **Filetxt.txt** для шифрования и поместить его в Папку Filetext:

“Кроме задачи расширения одной из основных задач Crypto API является возможность однозначной идентификации передающей/принимающей стороны в протоколе передачи данных. Общепризнанным решением в данном вопросе является использование механизма сертификатов. Сертификаты как бы стали «цифровыми паспортами», несущими информацию о своих владельцах. Полный рассказ о данном механизме последует несколько позже, а сейчас стоит упомянуть, что Crypto API также полно реализует весь спектр функций работы с ним. Большинство функций Crypto API, работающих с передаваемыми данными, так или иначе используют сертификаты в своей работе. “;

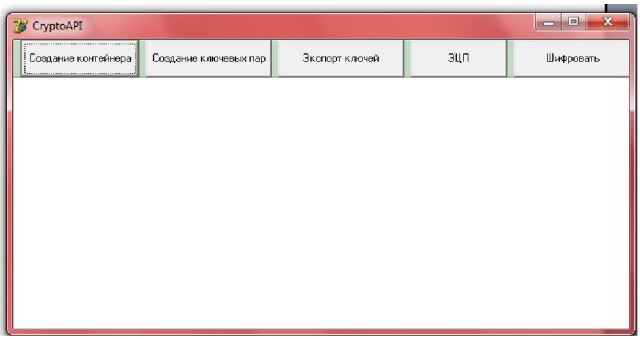

Рис. 6.1. Создание контейнера

1) Для начала работы необходимо открыть программу (меню ПУСК → psty → crypto.exe).

2) Откроется основное меню программы (рис. 6.1).

3) Щелкните по кнопке **«Создание контейнера»,** откроется окно (рис. 6.2).

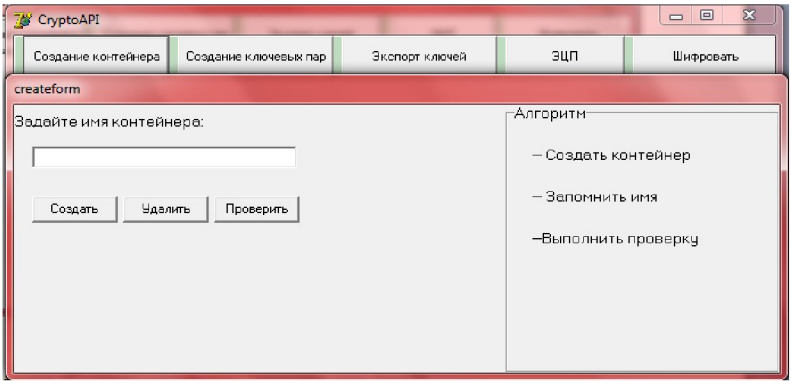

Рис. 6.2. Создание контейнера

4) В поле ввода текста напишите любое имя контейнера (цифры, буквы как латинского, так и русского языка), примечание: *ЗАПОМНИ ИМЯ КОНТЕЙНЕРА.*

5) Нажмите кнопку «Создать», появится сообщение о создании контейнера (рис. 6.3).

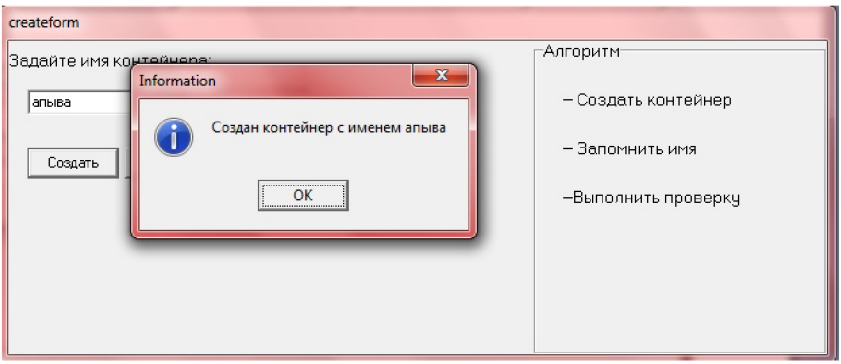

Рис. 6.3. Создание контейнера

6) Вы можете удалить ваш контейнер, также можно выполнить проверку для того, чтобы убедиться, что контейнер действительно был создан.

7) Перейдем на окно «Создание ключевых пар» (рис. 6.4).

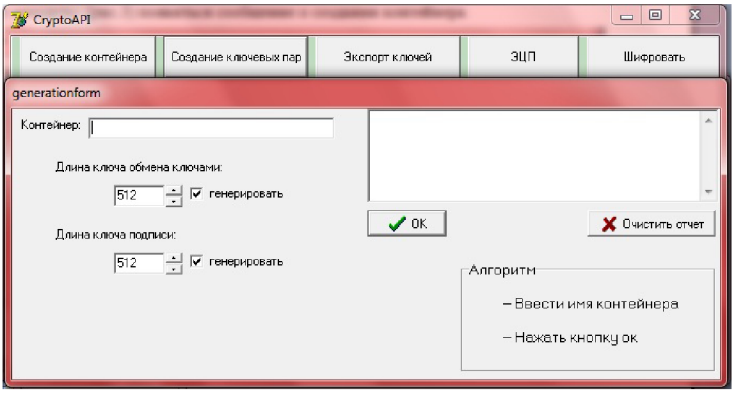

Рис. 6.4. Создание ключевых пар

8) В поле ввода текста напишем имя контейнера, созданного ранее, и нажмем кнопку «Ок», программа напишет отчет, если все выполнено правильно (рис. 6.5).

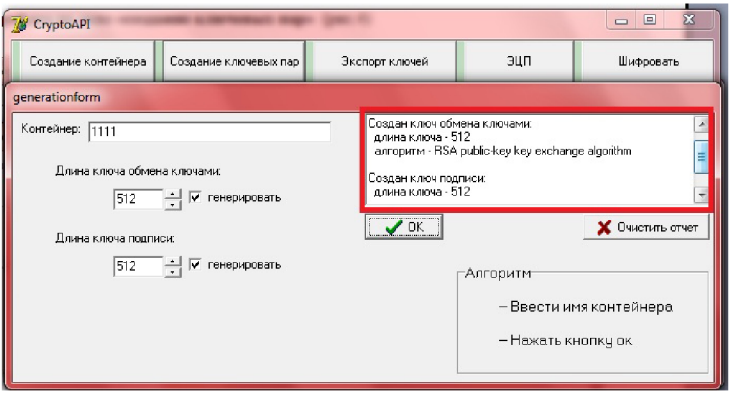

Рис. 6.5. Создание ключевых пар

9) Перейдите на вкладку «Экспорт ключей» (рис 6.6).

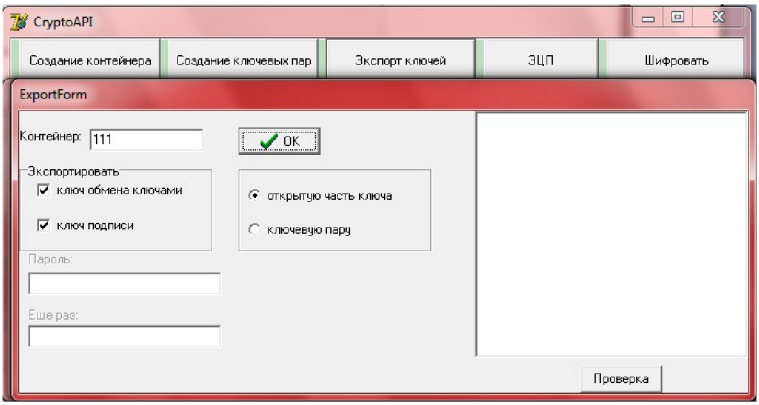

Рис. 6.6. Создание ключевых пар

10) Нажав кнопку «Ок», программа предложит вам сохранить файлы:

а) ключ обмена ключами **Klsobk**l:

¤ RSA1 г<ь, уп-юЩ°@П§А]ѓГщц|t”‡jЛіФД[Ач Ґ1П^ДjQиЌРеПсІ~ЧR6{#}ўҐ

б) ключ подписи **KlЭцп:**

$ RSA1 Б+ґ%›4?@¬mHC·#ħ'B‘2uґІ»rґк'эКЦ…д:Лj%~ZnЯЗ»¬яіо®gЁПҐ·_щ= ЯmaI

По умолчанию они находятся в директории **«C:\Program Files\pstu\».** Если вы меняли директорию при установке, то следует это учесть.

11) Перейдем на вкладку **«ЭЦП»** (рис. 6.7).

12) Не забудем вписать имя нашего контейнера, нажмем кнопку в виде двух стрелочек и укажем файл, который необходимо подписать. Файл может быть только в формате **.txt**.

Программа запросит место расположения вашего файла, он может находиться на любом доступном для вас носителе (HDD, flesh, CD\DVD и т.д).

13) Далее жмем на кнопку **«Подписать»** и укажем место для хранения подписанного файла. По умолчанию **«C:\Program Files\pstu\GryptoApi-Папки\FileHesh»:**

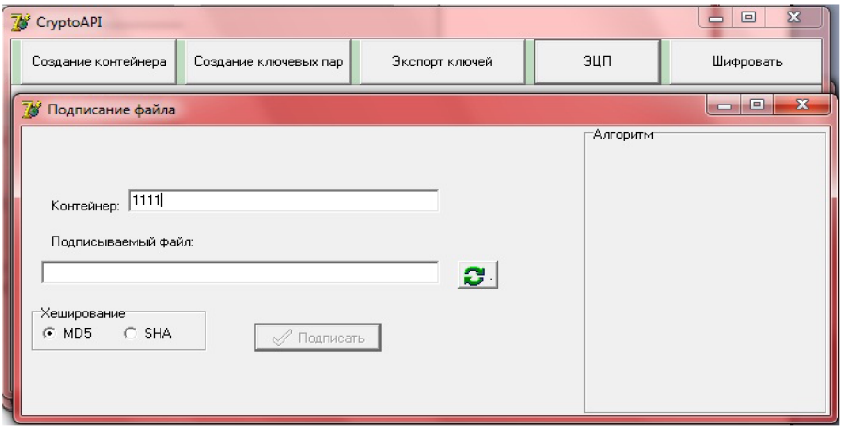

Рис. 6.7. Создание электронной цифровой подписи

“Кроме задачи расширения одной из основных задач Crypto API является возможность однозначной идентификации передающей/принимающей стороны в протоколе передачи данных. Общепризнанным решением в данном вопросе является использование механизма сертификатов. Сертификаты как бы стали «цифровыми паспортами», несущими информацию о своих владельцах. Полный рассказ о данном механизме последует несколько позже, а сейчас стоит упомянуть, что Crypto API также полно реализует весь спектр функций работы с ним. Большинство функций Crypto API, работающих с передаваемыми данными, так или иначе используют сертификаты в своей работе”

@ И?ЉVWЩ¬c@r:ЁЯЪФЊ ®я»Эiy1Ы™#+y»…лИjjXY,‡ Ћ–mtиљ§5KRBй LAъ§#~”

14) Теперь нам необходимо зашифровать подписанный файл перейдем на вкладку «шифровать» (рис. 6.8).

15) Напишем имя нашего контейнера, убедитесь, что в поле «**Действие»** выбрано «**шифровать»**, далее можно выбрать алгоритм шифрования RC2 или RC4. Нажмем кнопку «**Ок»**.

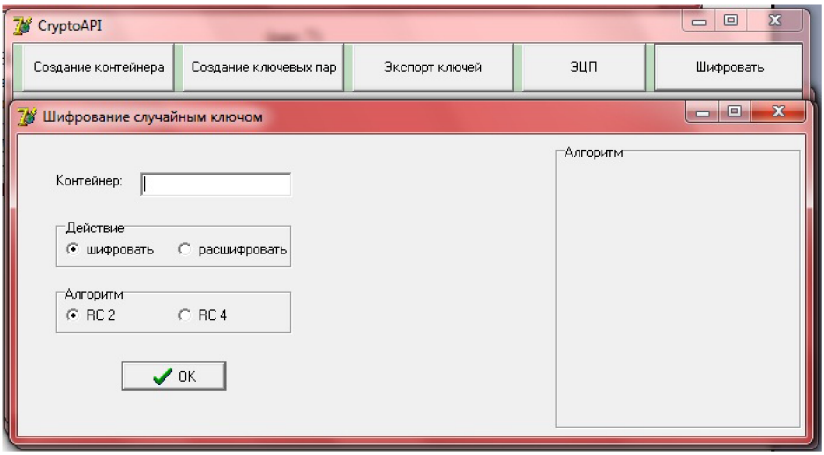

Рис. 6.8. Шифрование данных

Программа запросит место расположения файла для шифрования, **FileHesh** нам необходимо указать файл, **который мы ранее подписали** (также можно просто зашифровать, но тогда он будет без ЭЦП) + нужно указать файл **«ключ обмена ключами» Klsobkl.** Сохраним наш файл в файле **«FileZash»**:

“L f ¤ ‹[S<‹hIк¶ic I*,Л¬ЎлЄOVjH6”4BзTќт^“j%ehж©№ї†ЏqЌJ›)WЩ{pO)2qЏгЎеҐ$„?кJ©3ThIEєQ-Гw^®wQ3ЭБ·ЎBмHx†u©.EҐ¤ьср¦rHEqA’-·ЪmПЙ(RќЧЩ0єи=Ж4Лабюb($iµХзк†ґuъK)7ЧBЛЕ•¦3Щrl¬‹h-P»Г0(п67Фе¬ЦҐ‘(яМя`ы}I+мЩLЙ¦ЦBэAYЎгьЦNrю6XJзi3фA{bZ•BђHeA`кюш‘T06Є<мЮЛЂ'±l1‰†±…U7*!J]6Ъъгђ‰ЇЁд€rSYV(¦o8K$N№`UЮД9Лb
ЩЂf6гУ™™l7IK$ёD 8ќ1•‰KqbBUR/тO<5¶{i9Ш ДЖЛїпЪG@к6“
_T?¶E¬ЇРЎµ#ЛUHдцJЪсвёЙАчЗяЧРtHЬMФ№%,|]›rёa!КшЪ&9C єЂўi(-r~·j%r„m? «o v”яw–}‡ъ№ЭҐкHґ|l“R–M®Нд#MbE†зЮЮ2µMQЧ_fи&шы=&VcV8яLG–-ZЊU°»„1xUV8№эЉNµBЊj™o€EҐ:ИЂSЊGjЖ¬]bblJЄr¶»fgWf©ъ·D&пэ‘iк„]ЪCI~O ЌW•»Cj:.ЙчЏCчг* RД-Э¶Ъ‹Fб‡ЦдTCБ0L; o^5u7Sn®&G–‹X_-&|Є®jпЪ[ЛtjSr“Jf\й“ЊTvZX»ЛвягэЁг¬ќ”°1mHa§ЌёSд'кҐK”

6.2. Инструкция для получателя

1) В нашем случае отправитель будет и получателем. Для расшифровки нам понадобятся ключи, которые мы создали с вами ранее (напомню, это ключ обмена ключами и ключ подписи). Укажите в поле «Действие» –> «расшифровать» (рис. 6.9).

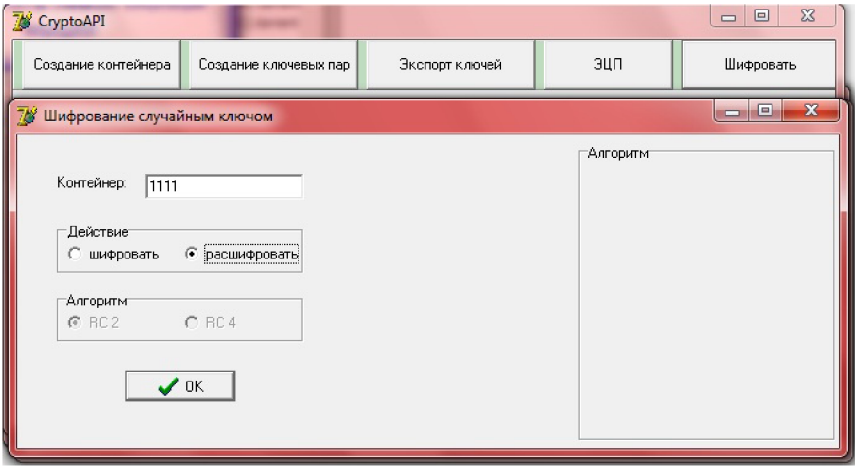

Рис. 6.9. Расшифрование файла

2) Нажмем кнопку **«Ок»** и укажем файл, который нужно расшифровать:

«C:\Program Files\pstu\ FileZash, и сохраним в файл **«C:\Program Files\pstu\FileRac»**. Получим расшифрованный файл, аналогичный исходному Gjkexbv.

“Кроме задачи расширения одной из основных задач Crypto API является возможность однозначной идентификации передающей/принимающей стороны в протоколе передачи данных. Общепризнанным решением в данном вопросе является использование механизма сертификатов. Сертификаты как бы стали «цифровыми паспортами», несущими информацию о своих владельцах. Полный рассказ о данном механизме последует несколько позже, а сейчас стоит упомянуть, что Crypto API также полно реализует весь спектр

функций работы с ним. Большинство функций Crypto API, работающих с передаваемыми данными, так или иначе, используют сертификаты в своей работе.”

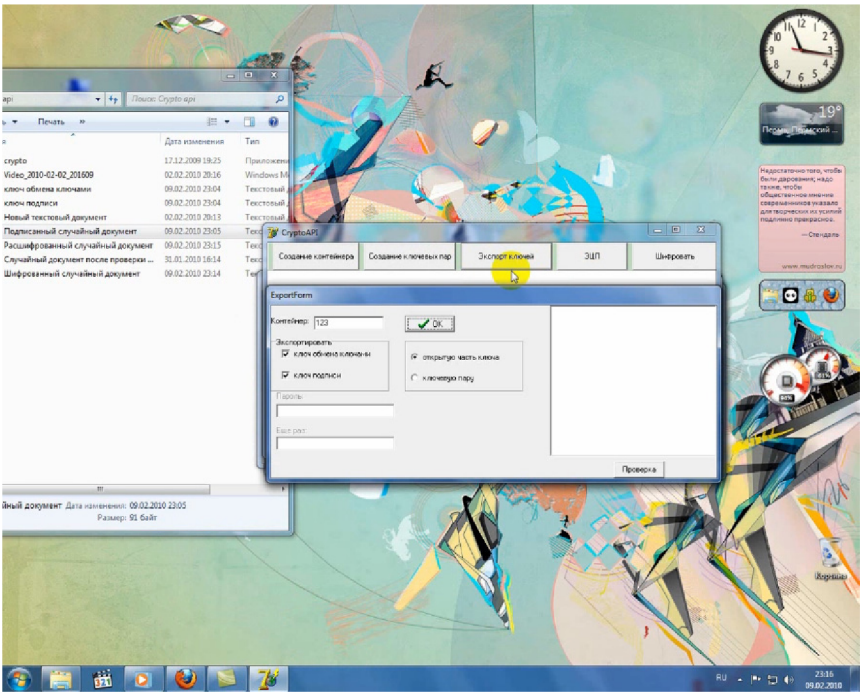

Рис. 6.10. Фрагмент видеоинструкции

3) Далее перейдем на вкладку «Экспорт ключей» и нажать кнопку «Проверка», в нашем случае нужно указать расшифрованный документ **Filerac** (напомним, он находится в **«C:\Program Files\pstu\Flerac»),** + укажем ключ подписи **FileЭцп** и *сохраним файл* – вот и все.

Скачать **видеоинструкцию** можно по ссылке https://disk.yandex.ru/i/2mvqmY2datDn9Q. Скриншот видеоинструкции представлен на рис. 6.10.

ЗАКЛЮЧЕНИЕ

Изложенный материал включает в себя теорию, алгоритмы, технологию, методы обучения, инструкции по организации криптографической защиты информации и, на наш взгляд, будет представлять интерес:

– для работников предприятий и организаций профиля защиты информации;

– слушателей курсов по защите данных;

– преподавателей высших и средних учебных заведений, ведущих курсы криптографических методов защиты данных;

– студентов специальности по защите информации в качестве методического пособия.

P.S. Будем рады вашим замечаниям, присылайте их по электронным адресам: Ю.Н. Липин – ur-lip193530@yandex.ru, С.А. Сторожев – cepra5@mail.ru

СПИСОК ЛИТЕРАТУРЫ

1. The Origins of Ubiquitous Computing Research at PARC in the Late 1980s / M. Wieser [et al.] // IBM Syst. J. – 1999. – Vol. 38, no. 4. – P. 693–696.

2. Five Minutes with Judith Donath. The MIT Press. – 30 July 2014. – URL: mitpress.mit.edu/blog/five-minutes-judith-donath (дата обращения: 10.03.2018).

3. Chow R. The Last Mile for IoT Privacy, IEEE Security & Privacy // IEEE Computer Society. All rights reserved. Reprinted with permission. – November/December 2017.

4. Спектор Ю. Королевство Delphi. – М., 2008.

5. Шнайер Б. Прикладная криптография. – Киев: Диалектика, 2016.

6. Щербаков А., Домашен А. Прикладная криптография. – М.: Русская редакция, 2003.

7. ALT Linux снаружи. – М.: ДМК Пресс, 2010. – 576 с.

8. Gookin D. DOS For Dummies®. – М.: Иностранная литература, 2014. – 384 с.

9. Войтов Н.М. Основы работы с Linux: учебный курс. – М.: ДМК Пресс, 2010. – 461 с.

10. Гецманн П., Хакфорт С., Новак П. Разработка приложений для Windows Phone. Архитектура, фреймворки, API. – М.: БХВ-Петербург, 2014. – 880 с.

11. Уолтер Г., Нортроп Т. Техническая поддержка и устранение неполадок в Microsoft Windows XP (+ CD-ROM). – М.: Русская редакция, 2009. – 926 с.

12. Джанус Фило. InfoPath 2007 для профессионалов. – М.: Вильямс, 2009. – 272 с.

13. Зубков С.В. Assembler для DOS, Windows и Unix. – М.: ДМК Пресс, 2017. – 868 с.

14. Клименко Р. Большая книга Windows Vista. – М.: Питер, 2009. – 688 с.

15. Книттель Б. Windows 7. Скрипты, автоматизация и командная строка. – М.: Питер, 2011. – 165 с.

16. Кокорева О. Реестр Windows XP. – М.: БХВ-Петербург, 2016. – 548 с.

17. Крелль Б.Е. Windows Mobile. Разработка приложений для КПК. – М.: Книга по требованию, 2009. – 352 с.

18. Матвеев М.Д., Альтер В.Е., Апанасевич Н.Р. Хакинг Windows 7: практич. руководство (+ CD). – М.: Наука и техника, 2012. – 320 с.

19. Петцольд Ч., Эспозито Д., Эспозито Ф. Программирование для Microsoft Windows 8. Разработка приложений для Windows 8 на HTML5 и JavaScript: в 2 кн. – М.: Питер, 2014. – 492 с.

20. Теджасви Редкар, Гвидичи Тони. Платформа Windows Azure. – М.: ДМК Пресс, 2014. – 656 с.

21. Румянцев П.В. Работа с файлами в Win 32 API. – М.: Горячая линия – Телеком, 2009. – 494 с.

22. Уилсон М. Расширение библиотеки STL для C++. Наборы и итераторы (+ CD-ROM). – М.: ДМК Пресс, БХВ-Петербург, 2015. – 608 с.

23. Эклер Ю. Прогрессивный самоучитель работы на компьютере. – М.: Книга по требованию, 2009. – 496 с.

Учебное издание

ЛИПИН Юрий Николаевич,

СТОРОЖЕВ Сергей Александрович

КРИПТОГРАФИЧЕСКАЯ ЗАЩИТА «ПОСЛЕДНЕЙ МИЛИ» СРЕДСТВАМИ WIN API

Учебное пособие

Редактор и корректор *И.Н. Жеганина*

Подписано в печать 10.06.2021. Формат 60×90/16.
Усл. печ. л. 7,25. Тираж 28 экз. Заказ № 130/2021.

Издательство
Пермского национального исследовательского
политехнического университета.
Адрес: 614990, г. Пермь, Комсомольский пр., 29, к. 113.
Тел. (342) 219-80-33.