**Использование Crypto API**

**Автор:**[***Юрий Строжевский***](http://rsdn.org/account/info/34857) ***свободный художник*  
Источник:**[***RSDN Magazine #5-2004***](http://www.rsdn.ru/mag/main.htm)

Опубликовано: *23.04.2005*  
Исправлено: *10.12.2016*  
Версия текста: *2.0*

|  |  |
| --- | --- |
| [**Введение**](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#E5) [**Общие основы**](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EHB)  [Строение и возможности Crypto API](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#ELB) [Криптопровайдеры](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#ECD) [Контейнеры ключей](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#E3EAC) [Алгоритмы](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#ENGAC) [Сертификаты](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#E6HAC)  [**Базовые функции**](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EXIAC)  [Шифрование](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#E2IAC) [Экспорт сессионных ключей](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EJLAC) [Импорт сессионных ключей](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#E1NAC) [Расшифровывание](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#E6OAC) [Хеширование](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#ELAAE) [Цифровая подпись](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#ERFAE) [Проверка цифровой подписи](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EQHAE)  [**Высокоуровневые функции обработки сообщений**](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#ECJAE)  [Введение](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EGJAE) [Шифрование](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EEKAE) [Расшифровывание](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EINAE) [Цифровая подпись](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EWPAE) [Проверка цифровой подписи](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#ELTAE) [Совмещение цифровой подписи и шифрованных данных](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EXWAE) [Расшифровывание и проверка совмещенной цифровой подписи и шифрованных данных](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EKZAE) [Работа с форматом Base64 (PEM)](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EF2AE)  [**CAPICOM**](https://www.rsdn.org/article/crypto/usingcryptoapi.xml#EK4AE) | https://www.rsdn.org/mag/cover/mag0504.jpg |

Введение

Эта статья основана на моем личном опыте в деле написания программных продуктов криптографической направленности. Несмотря на данный, прежде всего коммерческий, опыт, здесь я постараюсь абстрагироваться от какого-то конкретного направления в Crypto API, и рассмотреть общий случай его применения.

Предполагается, что читатель знаком с некоторыми общими для криптографии понятиями: ключ шифрования, симметричные/ассиметричные алгоритмы шифрования, понятия протоколов шифрования и т.п. Также предполагается, что читатель имеет доступ к MSDN как можно более новой редакции.

Общие основы

**Строение и возможности Crypto API**

Прежде всего, постараемся описать круг задач, на решение которых ориентирован Crypto API:

* 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.

Сперва я постараюсь рассмотреть решение задачи расширения. В Crypto API эту задачу решили довольно элегантно. 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.

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.

В программных решениях рано или поздно встает вопрос стандартизации передаваемых между приложениями данных. В сфере криптографии для решения данного вопроса уже давно и успешно применяет 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.

**Криптопровайдеры**

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 говорилось 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 (то есть 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 c 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 значение различных параметров криптопровайдера. Нужно сказать, 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. То есть 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, скажем, 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, скажем, 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 сигнализирует 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**:

|  |
| --- |
| DWORD dwIndex=0;  DWORD dwType;  DWORD cbName;  LPTSTR pszName;  while (CryptEnumProviders(dwIndex, NULL, 0, &dwType, NULL, &cbName))  {  if (!cbName)  break;    if (!(pszName = (LPTSTR)LocalAlloc(LMEM\_ZEROINIT, cbName)))  return;    if (!CryptEnumProviders(dwIndex++, NULL, 0, &dwType, pszName, &cbName))  {  Error("CryptEnumProviders");  return;  }    std::cout<<"--------------------------------"<<std::endl;  std::cout<<"Provider name: "<<pszName<<std::endl;  std::cout<<"Provider type: "<<dwType<<std::endl;    LocalFree(pszName);  } |

Для перечисления типов криптопровайдеров, установленных в системе, можно использовать функцию **CryptEnumProviderTypes**:

|  |
| --- |
| DWORD dwIndex=0;  DWORD dwType;  DWORD cbName;  LPTSTR pszName;    while(CryptEnumProviderTypes(dwIndex,NULL,0,&dwType,NULL,&cbName))  {  if(!cbName)  break;    if(!(pszName=(LPTSTR)LocalAlloc(LMEM\_ZEROINIT, cbName)))  return;    if(!CryptEnumProviderTypes(dwIndex++,NULL,NULL,&dwType,pszName,&cbName))  {  Error("CryptEnumProvidersTypes");  return;  }    std::cout<<"--------------------------------"<<std::endl;  std::cout<<"Type name: "<<pszName<<std::endl;  std::cout<<"Type ID: "<<dwType<<std::endl;    LocalFree(pszName);  } |

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 типов.

Пример перечисления криптопровайдеров с прямым обращением к реестру:

|  |
| --- |
| // Все криптопровайдеры объявлены в этом ключе реестра  #define PROV\_REG\_PATH "SOFTWARE\\Microsoft\\Cryptography\\Defaults"char KeyName[1024];  strcpy(KeyName, PROV\_REG\_PATH);  strcat(KeyName, "\\Provider");  CRegKey RegKey;  if(RegKey.Open(HKEY\_LOCAL\_MACHINE, KeyName, KEY\_READ) != ERROR\_SUCCESS)  return;  DWORD iIndex = 0;  char ProvName[255];  DWORD iProvNameLen=255;  DWORD ProvType;  CRegKey SubKey;  while (RegKey.EnumKey(iIndex++, ProvName, &iProvNameLen) == ERROR\_SUCCESS)  {  if (SubKey.Open(RegKey, ProvName, KEY\_READ) != ERROR\_SUCCESS)  return;  SubKey.QueryDWORDValue("Type", ProvType);  std::cout << "--------------------------------" << std::endl;  std::cout << "Provider name: " << ProvName << std::endl;  std::cout << "Provider type: " << ProvType << std::endl;  iProvNameLen = 255;  } |

Пример перечисления типов криптопровайдеров при прямом обращении к реестру:

|  |
| --- |
| // Все криптопровайдеры объявлены в этом ключе реестра  #define PROV\_REG\_PATH "SOFTWARE\\Microsoft\\Cryptography\\Defaults"char KeyName[1024];  strcpy(KeyName, PROV\_REG\_PATH);  strcat(KeyName, "\\Provider Types");  CRegKey RegKey;  if(RegKey.Open(HKEY\_LOCAL\_MACHINE, KeyName, KEY\_READ) != ERROR\_SUCCESS)  return;  DWORD iIndex = 0;  char TypeID[255];  DWORD iTypeIDLen = 255;  char TypeName[255];  DWORD TypeNameLen = 255;  DWORD dwType;  CRegKey SubKey;  while(RegKey.EnumKey(iIndex++, TypeID, &iTypeIDLen) == ERROR\_SUCCESS)  {  if(SubKey.Open(RegKey, TypeID, KEY\_READ) != ERROR\_SUCCESS)  return;  SubKey.QueryStringValue("TypeName", TypeName, &TypeNameLen);  TypeNameLen = 255;  std::string type = TypeID;  std::string::size\_type pos = type.find(" ");  if(pos != std::string::npos)  {  type.erase(0, pos + 1);  dwType = atol(type.c\_str());  }  std::cout<<"--------------------------------"<<std::endl;  std::cout<<"Type name: "<<TypeName<<std::endl;  std::cout<<"Type ID: "<<dwType<<std::endl;  iTypeIDLen = 255;  } |

**Контейнеры 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.
* 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 – 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** c параметром dwParam установленным в PP\_ENUMCONTAINERS.

**Алгоритмы**

В Crypto API все криптографические алгоритмы реализуются внутри криптопровайдеров. Таким образом, криптопровайдер позволяет устанавливать или получать различные параметры алгоритма, получать выходную информацию алгоритма и многое другое. Внутри Crypto API алгоритмы принято делить на следующие группы:

* алгоритмы шифрования/расшифровывания;
* алгоритмы получения значения хеша;
* алгоритмы получения цифровой подписи;
* алгоритмы обмена криптографическими ключами.

Для дальнейшего понимания сущности работы с алгоритмами необходимо также ввести понятия симметричных и ассиметричных алгоритмов шифрования. Симметричным алгоритмом называют алгоритм, в котором для шифрования и расшифровывания используется один и тот же ключ. Ассиметричными называют алгоритмы, в которых используется пара ключей (в общем случае может быть и более двух ключей), один из которых используется для шифрования (и только для шифрования) информации, а другой – для расшифровывания. В качестве ассиметричных алгоритмов принято применять так называемые алгоритмы с открытым ключом. В таких алгоритмах существует пара ключей – открытый (публичный) и закрытый (секретный) ключи. В большинстве случаев для шифрования информации используется публичный ключ, а для расшифровывания – секретный. Публичный ключ не является предметом секретности и предназначен для передачи по открытым каналам. Лицо, получившее публичный ключ, может зашифровать им данные и отослать их по тем же открытым каналам лицу, владеющему секретным ключом. В свою очередь на стороне получателя расшифровывание возможно только при наличии секретного ключа из той пары ключей, к которой принадлежит публичный ключ, применявшийся для шифрования. Таким образом, достигается возможность обмениваться шифрованными сообщениями, не заботясь о том, что ключ может быть перехвачен в процессе пересылки.

Ассиметричные алгоритмы в обычной жизни мало применимы. Причина тому – высокая трудоемкость шифрования, обусловленная повышенной сложностью ассиметричных алгоритмов. Обычно используют следующую схему: шифрование данных выполняется симметричным криптографическим алгоритмом некоторым сгенерированным для этого сессионным ключом. Затем для передачи получателю этот ключ шифруется с помощью асимметричного алгоритма (длина ключа несущественно мала по сравнению с возможным размером данных). На стороне получателя с помощью асимметричного алгоритма расшифровывается сессионный ключ, который используется для расшифровывания полученных данных с помощью симметричного алгоритма. Такой способ передачи шифрованной информации позволяет использовать преимущества обоих подходов.

Именно такой подход использован во всех ключевых алгоритмах работы Crypto API. Контейнеры ключей хранят пары ключей, необходимые для обмена сессионными ключами, а для непосредственно шифрования данных необходимо каждый раз генерировать новые сессионные ключи.

Рассмотрим основные операции, выполняемые над алгоритмами в Crypto API:

* Перечисление всех криптографических алгоритмов. Выполняется посредством вызова функции **CryptGetProvParam**с параметром *dwParam,*установленным в PP\_ENUMALGS.
* Получение параметров алгоритмов. Обычно выполняется прямо во время перечисления алгоритмов посредством вызова функции **CryptGetProvParam** с параметром *dwParam*установленным в PP\_ENUMALGS\_EX. В этом случае получаемая структура может содержать практически всю информацию об алгоритме.

Зачастую также требуется найти криптопровайдер, реализующий некий известный нам алгоритм. Для идентификации алгоритмов в пределах системы все алгоритмы криптопровайдеров имеют уникальный цифровой номер. Данный номер в системе имеет тип ALG\_ID и представляет собой простое число типа DWORD. Наряду с нумерацией алгоритмов на основе ALG\_ID существует и другой идентификатор – OID. Данный идентификатор пришел из спецификаций криптографических стандартов фирмы RSA Security и представляет собой строку, состоящую из групп цифр, разделенных точками. В стандартах жестко закреплено значение каждой группы цифр в OID, и по данному номеру можно получить некоторую дополнительную информацию (например, фирму-разработчика алгоритма). Преобразование между типами номеров алгоритмов осуществляется функциями **CertAlgIdToOID** и **CertOIDToAlgId**.

**Сертификаты**

Полное пояснение по данному понятию будет дано в отдельном разделе. Здесь же о сертификатах упомянуто, поскольку сертификаты все-таки используются в высокоуровневых функциях обработки криптографических сообщений типа **CryptSignMessage**. Пока важно понимать, что сертификаты в системе используются для:

* хранения дополнительной информации о владельце сертификата;
* хранения значения публичного ключа владельца;
* получения ссылки на контейнер ключей. С сертификатом, установленным в системе, может быть физически связан контейнер ключей. То есть, имея только ссылку на контекст сертификата можно установить однозначную связь с секретным ключом и использовать это, например, при проверке цифровой подписи или расшифровывании данных.

В упрощенном виде сертификаты можно считать некоторыми удостоверениями личности. Правами выдавать сертификаты (как и паспорта), в идеале должны обладать только доверенные центры (например, ФСБ или милиция). Но, в принципе, можно генерировать сертификаты и самому. В этом случае тот пользователь, который будет использовать данный сертификат, вправе либо доверять, либо не доверять сертификату («...паспорт можно и самому нарисовать...»). Для целей дальнейшего изучения высокоуровневых функций обработки криптографической информации нам важны лишь два последних случая использования сертификатов из перечисленных: сертификаты хранят публичный ключ и могут быть связаны с контейнером ключей, в котором хранится секретный ключ. Этой информации на данном этапе будет достаточно.

Если вы не имели опыта работы с сертификатами, то для ознакомления с ними, можно вызвать Internet Explorer, выбрать в его меню пункт «Сервис / Свойства обозревателя» («Tools / Internet Options»), а затем на закладке «Содержание» («Content») нажать на кнопку «Сертификаты» («Certificates»). Вашему вниманию будет представлен список сертификатов, установленных в системе.

Базовые функции

**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. То есть 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, скажем, 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.

Пример использования функции **CryptEncrypt** приведен ниже:

|  |
| --- |
| HCRYPTPROV hProv;  HCRYPTKEY hSessionKey;  // Получение контекста криптопровайдераif (!CryptAcquireContext(&hProv, NULL, NULL,  PROV\_RSA\_FULL, CRYPT\_VERIFYCONTEXT))  {  Error("CryptAcquireContext");  return;  }  std::cout << "Cryptographic provider initialized" << std::endl;  // Генерация сессионного ключаif (!CryptGenKey(hProv, CALG\_RC4,  CRYPT\_ENCRYPT | CRYPT\_DECRYPT, &hSessionKey))  {  Error("CryptGenKey");  return;  }  std::cout << "Session key generated" << std::endl;  // Данные для шифрованияchar string[]="Test";  DWORD count=strlen(string);  // Шифрование данныхif (!CryptEncrypt(hSessionKey, 0, true, 0, (BYTE\*)string,  &count, strlen(string)))  {  Error("CryptEncrypt");  return;  }  std::cout << "Encryption completed" << std::endl;  // Тестовый вывод на экран  std::cout << "Encrypted string: " << string << std::endl; |

**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, скажем, 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.

|  |
| --- |
| **СОВЕТ**  Хотелось бы также обратить внимание читателя на достаточно важный момент: иногда для обмена ключами используются несколько более сложные схемы, чем просто шифрование данных сессионного ключа публичным ключем. Примером подобной усложненной схемы обмена может служить алгоритм обмена ключами по методу Диффи-Хеллмана. В данном алгоритме используются оба публичных ключа – как отправителя, так и получателя. Более подробную информацию о данном алгоритме читатель может найти в специализированной литературе. |

Пример использования этой функции приведен ниже:

|  |
| --- |
| HCRYPTPROV hProv;  HCRYPTKEY hKey, hPublicKey, hNewKey;  // Инициализация контекста криптопровайдера (с указанием имени  // контейнера ключей)if (!CryptAcquireContext(&hProv, "{EB57ED8A-CCCC-4bf5-8659-9DF2F05F24AD}",  NULL, PROV\_RSA\_FULL, 0))  return;  std::cout << "Cryptographic provider initialized" << std::endl;  // Генерация ключа для тестированияif (!CryptGenKey(hProv, CALG\_RC4,  CRYPT\_EXPORTABLE | CRYPT\_ENCRYPT | CRYPT\_DECRYPT, &hKey))  return;  std::cout << "Session key generated" << std::endl;  // Получение ключа для экспорта ключа шифрованияif (!CryptGetUserKey(hProv, AT\_KEYEXCHANGE, &hPublicKey))  return;  std::cout << "Public key is received" << std::endl;  count = 0;  // Получение размера массива, используемого для экспорта ключаif (!CryptExportKey(hKey, hPublicKey, SIMPLEBLOB, 0, NULL, &count))  return;  // Инициализация массива, используемого для экспорта ключа  BYTE\* data = static\_cast<BYTE\*>(malloc(count));  ZeroMemory(data, count);  // Экспорт ключа шифрованияif (!CryptExportKey(hKey, hPublicKey, SIMPLEBLOB, 0, data, &count))  return;  std::cout << "Key's export completed" << std::endl; |

**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. Должен отметить, что 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.

Пример использования данной функции приведен ниже:

|  |
| --- |
| // Импорт ключа шифрования из полученного массива данных  if(!CryptImportKey(hProv, data, count, hPublicKey, 0, &hNewKey))  return;  std::cout << "Key's import completed" << std::endl; |

**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 CryptEncryptи представляют собой ссылку на входной/выходной массив данных и длину этого массива данных.

Пример использования функции CryptDecrypt приведен ниже:

|  |
| --- |
| // Расшифровывание данных  if(!CryptDecrypt(hSessionKey, 0, true, 0, (BYTE\*)string, &count))  {  Error("CryptDecrypt");  return;  }  std::cout << "Decryption completed" << std::endl;  // Тестовый вывод на экран  std::cout << "Decrypted string: " << string << std::endl;  // Освобождение контекста локальных переменных  CryptDestroyKey(hSessionKey);  CryptReleaseContext(hProv, 0); |

**Хеширование**

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

|  |
| --- |
| **СОВЕТ**  *MAC* (Message Authentication Code, русский термин – «имитовставка»), переводится как «код проверки подлинности сообщения». Фактически, *MAC* предназначен для проверки значения хеш-значения только людьми, имеющими необходимый ключ. Можно воспринимать это как дополнительное шифрование значения, полученного с помощью хеш-функции (хотя это и не совсем так). Также *MAC* можно использовать в качестве простейшей цифровой подписи. *HMAC* (Hash-based Message Authentication Code) является разновидностью *MAC*. |

Параметр *dwFlags* зарезервирован под возможное будущее использование и должен быть всегда равен 0. Через параметр *phHash*функция возвращает хендл созданного ей 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.

Пример получения хеш-значения приведен ниже:

|  |
| --- |
| HCRYPTPROV hProv;  HCRYPTHASH hHash;  // Инициализация контекста криптопровайдераif(!CryptAcquireContext(&hProv,  "{EB57ED8A-CCCC-4bf5-8659-9DF2F05F24AD}", NULL, PROV\_RSA\_FULL, 0))  {  Error("CryptAcquireContext");  return;  }  std::cout << "Cryptographic provider initialized" << std::endl;  // Cоздание хеш-объектаif(!CryptCreateHash(hProv, CALG\_MD5, 0, 0, &hHash))  {  Error("CryptCreateHash");  return;  }  std::cout << "Hash created" << std::endl;  // Тестовые данные для хешированияchar string[] = "Test";  DWORD count = strlen(string);  // Передача хешируемых данных хэш-объекту.if(!CryptHashData(hHash, (BYTE\*)string, count, 0))  {  Error("CryptHashData");  return;  }  std::cout << "Hash data loaded" << std::endl;  // Получение хеш-значения  count = 0;  if(!CryptGetHashParam(hHash, HP\_HASHVAL, NULL, &count, 0))  {  Error("CryptGetHashParam");  return;  }  char\* hash\_value = static\_cast<char\*>(malloc(count + 1));  ZeroMemory(hash\_value, count + 1);  if(!CryptGetHashParam(hHash, HP\_HASHVAL, (BYTE\*)hash\_value, &count, 0))  {  Error("CryptGetHashParam");  return;  }  std::cout << "Hash value is received" << std::endl;  // Вывод на экран полученного хеш-значения  std::cout << "Hash value: " << hash\_value << std::endl; |

**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.

Пример использования данной функции приведен ниже:

|  |
| --- |
| // Цифровая подпись хеш-значения  count = 0;  if(!CryptSignHash(hHash, AT\_SIGNATURE, NULL, 0, NULL, &count))  {  Error("CryptSignHash");  return;  }  char\* sign\_hash = static\_cast<char\*>(malloc(count + 1));  ZeroMemory(sign\_hash, count + 1);  if(!CryptSignHash(hHash, AT\_SIGNATURE, NULL, 0, (BYTE\*)sign\_hash, &count))  {  Error("CryptSignHash");  return;  }  std::cout << "Signature created" << std::endl;  // Вывод на экран значения цифровой подписи  std::cout << "Signature value: " << sign\_hash << std::endl; |

**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.

Пример использования данной функции приведен ниже:

|  |
| --- |
| // Получение публичного ключа (для проверки цифровой подписи)  HCRYPTKEY hPublicKey;  if(!CryptGetUserKey(hProv, AT\_SIGNATURE, &hPublicKey))  {  Error("CryptGetUserKey");  return;  }  std::cout << "Public key is received" << std::endl;  // Проверка цифровой подписи  BOOL result = CryptVerifySignature(hHash, (BYTE\*)sign\_hash, count,  hPublicKey, NULL, 0);  std::cout << "Check is completed" << std::endl;  // Вывод на экран результата проверки цифровой подписи  std::cout << "Check result: "  << ((result)? "Verified!" : "NOT verified!") << std::endl; |

Высокоуровневые функции обработки сообщений

**Введение**

В рассмотренных выше базовых функциях работы с криптографическими сообщениями все выглядит достаточно хорошо. За исключением работы на уровне общепринятых в криптографии форматов данных. То есть выходные данные, полученные с помощью базовых функций Crypto API, можно обработать только этими же базовыми функциями Crypto API. Этот недостаток устраняется путем применения высокоуровневых функций для работы с криптографическими приложениями.

В основе работы данных функций лежит стандарт *PKCS #7* (RFC 2315). Более подробно об этом стандарте будет рассказано несколько позднее, а здесь будет сказано лишь о его основных особенностях.

Основу стандарта *PKCS #7* составляют способы кодирования и описания различных данных, используемых в криптографических приложениях. Так, предоставляются стандарты по кодированию шифрованных и подписанных сообщений, стандарт кодирования сессионного ключа, стандарт кодирования информации о сертификате и многое другое. Работа со стандартом *PKCS #7* реализована на многих платформах, отличных от *MS Windows,*что позволяет создавать кроссплатформенные приложения*.* Высокоуровневые функции скрывают многие не очень существенные моменты использования стандарта *PKCS #7,*благодаря чему упрощается работа с криптографическими данными.

Между тем, одним из ограничений в работе высокоуровневых функций следует признать недопустимость поблочной загрузки входных данных – эти функции работают только с одним, загруженным в память блоком. Для работы с большими объемами данных и более тонкого использования всех возможностей Crypto API в работе со стандартом *PKCS* предназначены низкоуровневые функции обработки сообщений, о которых будет рассказано позднее.

**Шифрование**

Для шифрования используется функция **CryptEncryptMessage**. Данная функция имеет следующее описание:

|  |
| --- |
| BOOL CryptEncryptMessage(PCRYPT\_ENCRYPT\_MESSAGE\_PARA pEncryptPara,  DWORD cRecipientCert,  PCERT\_CONTEXT rgpRecipientCert[],  const BYTE\* pbToBeEncrypted,  DWORD cbToBeEncrypted,  BYTE\* pbEncryptedBlob,  DWORD\* pcbEncyptedBlob); |

В качестве первого параметра данной функции передается указатель на структуру CRYPT\_ENCRYPT\_MESSAGE\_PARA. В полях данной структуры передается информация об используемом для шифрования криптопровайдере, используемом алгоритме и вспомогательная информация, нужная для правильной работы выбранного алгоритма. Параметр *cRecipientCert* передает количество используемых сертификатов получателей. Сам массив используемых сертификатов получателей передается в параметре *rgpRecipientCert*.

|  |
| --- |
| **СОВЕТ**  Собственно сертификаты в данных функциях используются, в основном, для экспорта сессионного ключа, сформированного при шифровании. Того, что сертификаты сами по себе хранят публичные ключи получателей, вполне должно хватить для экспорта сессионного ключа. В случае нескольких получателей данных для каждого получателя формируется свой массив экспортированного ключа. Кроме того, используется также другое свойство сертификатов: каждый из них имеет некий уникальный идентификационный номер. Вставляя этот номер в полученный массив экспортированных сессионных ключей, мы получаем возможность на стороне получателя найти необходимый для корректного импорта сессионного ключа элемент данного массива. |

Параметры *pbToBeEncrypted*и*cbToBeEncrypted*идентифицируют область обрабатываемых данных. Зашифрованный контент и его размер возвращается через параметры *pbEncryptedBlob*и*pcbEncryptedBlob*соответственно.

Пример использования данной функции приведен ниже:

|  |
| --- |
| // Получение сертификата(ов) получателей  HCERTSTORE hSystemStore;  HCRYPTPROV hProv;  if (!(hSystemStore = CertOpenSystemStore(NULL, "MY")))  {  Error("CertOpenSystemStore");  return;  }  std::cout << "System store is opened" << std::endl;  PCCERT\_CONTEXT pCert = NULL;  while (pCert = CertEnumCertificatesInStore(hSystemStore, pCert))  {  // Выбираем первый попавшийся сертификат, использующий алгоритм sha1RSA,  // и имеющий секретный ключ.if (!strcmp(pCert->pCertInfo->SubjectPublicKeyInfo.Algorithm.pszObjId,  "1.2.840.113549.1.1.1"))  {  DWORD dwKeySpec;  if(CryptAcquireCertificatePrivateKey(pCert, 0, NULL, &hProv,  &dwKeySpec, NULL))  break;  }  }  std::cout << "Certificate found" << std::endl;  // Инициализация структуры, необходимой для шифрования  CRYPT\_ENCRYPT\_MESSAGE\_PARA EncryptPara;  ZeroMemory(&EncryptPara, sizeof(EncryptPara));  EncryptPara.cbSize = sizeof(EncryptPara);  EncryptPara.dwMsgEncodingType = X509\_ASN\_ENCODING | PKCS\_7\_ASN\_ENCODING;  EncryptPara.ContentEncryptionAlgorithm.pszObjId =  (LPSTR)CertAlgIdToOID(CALG\_RC4);  EncryptPara.hCryptProv = hProv;  char string[] = "Test";  if(!CryptEncryptMessage(&EncryptPara, 1, &pCert, (BYTE\*)string,  strlen(string), NULL, count))  {  Error("CryptEncryptMessage");  return;  }  \*encrypted = static\_cast<BYTE\*>(malloc(\*count));  if(!CryptEncryptMessage(&EncryptPara, 1, &pCert, (BYTE\*)string,  strlen(string), \*encrypted, count))  {  Error("CryptEncryptMessage");  return;  }  std::cout << "Encryption completed" << std::endl;  CertCloseStore(hSystemStore, 0);  CryptReleaseContext(hProv, 0); |

**Расшифровывание**

Для расшифровывания зашифрованного контента применяется функция **CryptDecryptMessage**, имеющая следующее описание:

|  |
| --- |
| BOOL CryptDecryptMessage(PCRYPT\_DECRYPT\_MESSAGE\_PARA pDecryptPara,  const BYTE\* pbEncryptedBlob,  DWORD cbEncryptedBlob,  BYTE\* pbDecrypted,  DWORD\* pcbDecrypted,  PCCERT\_CONTEXT\* ppXchgCert); |

В качестве первого параметра данной функции передается указатель на структуру типа CRYPT\_DECRYPT\_MESSAGE\_PARA. В полях данной структуры передается список хранилищ сертификатов, в которых будет производиться поиск сертификата для импорта сессионного ключа (сертификата обмена). В параметрах *pbEncryptedBlob*и *cbEncryptedBlob*передается информация о блоке входных данных, подлежащих расшифровке. В параметрах *pbDecrypted*и *pcbDecrypted* передаются принимающий буфер и его размер соответственно. В этот буфер будут помещены расшифровываемые данные. В параметре *ppXchgCert* возвращается ссылка на контекст сертификата, который был использован для обмена (если эта информация не нужна, то данный параметр должен быть установлен в *NULL*).

|  |
| --- |
| **ПРЕДУПРЕЖДЕНИЕ**  Необходимо отметить, что сертификат, используемый для импорта сессионного ключа (сертификат обмена) на стороне получателя должен быть связан с локальным контейнером ключей. Иначе не будет найден секретный ключ пары ключей, и функция будет выполнена с ошибкой (никаких расшифрованных данных получить не удастся). |

Пример использования этой функции приведен ниже:

|  |
| --- |
| HCERTSTORE hSystemStore;  if(!(hSystemStore = CertOpenSystemStore(NULL, "MY")))  {  Error("CertOpenSystemStore");  return;  }  std::cout << "System store is opened" << std::endl;  // Инициализация структуры, необходимой для расшифровывания  CRYPT\_DECRYPT\_MESSAGE\_PARA DecryptPara;  ZeroMemory(&DecryptPara, sizeof(DecryptPara));  DecryptPara.cbSize = sizeof(DecryptPara);  DecryptPara.dwMsgAndCertEncodingType =  X509\_ASN\_ENCODING | PKCS\_7\_ASN\_ENCODING;  HCERTSTORE StoreArray[] = { hSystemStore };  DecryptPara.cCertStore = 1;  DecryptPara.rghCertStore = StoreArray;  DWORD dwSize = 0;  if(!CryptDecryptMessage(&DecryptPara, encrypted, count, NULL,  &dwSize, NULL))  {  Error("CryptDecryptMessage");  return;  }  BYTE\* decrypted = static\_cast<BYTE\*>(malloc(dwSize + 1));  ZeroMemory(decrypted, dwSize + 1);  if(!CryptDecryptMessage(&DecryptPara, encrypted, count, decrypted,  &dwSize, NULL))  {  Error("CryptDecryptMessage");  return;  }  std::cout << "Decryption completed" << std::endl;  std::cout << "Result string: " << decrypted << std::endl;  free(decrypted);  free(encrypted);  CertCloseStore(hSystemStore, 0); |

**Цифровая подпись**

В отличие от простейшего случая формирования цифровой подписи базовыми функциями в стандарте PKCS предусмотрено существование двух видов цифровой подписи: подпись, совмещенная с подписываемыми данными (*attached signature*) и подпись, отдельная от данных (*detached signature*). Функция **CryptSignMessage**, формирующая оба эти вида подписей, имеет следующее описание:

|  |
| --- |
| BOOL CryptSignMessage(PCRYPT\_SIGN\_MESSAGE\_PARA pSignPara,  BOOL fDetachedSignature,  DWORD cToBeSigned,  const BYTE\* rgpbToBeSigned[],  DWORD rgcbToBeSigned[],  BYTE\* pbSignedBlob,  DWORD\* pcbSignedBlob); |

В качестве первого параметра данной функции передается указатель на структуру типа CRYPT\_SIGN\_MESSAGE\_PARA. В полях данной структуры передается информация о сертификате, с помощью которого будет производиться подпись (сертификат, заверяющий подписчика), алгоритм формирования хеш-значения переданных данных, массив дополнительных сертификатов, которые необходимо включить в состав цифровой подписи, и некоторые вспомогательные величины. Параметр *fDetachedSignature* задает тип получаемой цифровой подписи (отделенная или совмещенная с данными). Параметр *cToBeSigned* указывает на количество элементов массива подписываемых данных.

|  |
| --- |
| **ПРЕДУПРЕЖДЕНИЕ**  Если флаг *fDetachedSignature* установлен в *false*, параметр *cToBeSigned* всегда должен быть равен 1. Это связано с особенностями реализации низкоуровневых функций работы с криптографическими сообщениями, особенности которых будут рассмотрены позднее. |

Параметр *rgpbToBeSigned* представляет собой массив данных, передаваемых для формирования цифровой подписи. Параметр *rgcbToBeSigned* представляет собой массив размеров элементов переданного массива данных. Выходное значение функции (цифровая подпись) формируется в параметрах *pbSignedBlob* и *pcbSignedBlob*.

Пример использования данной функции приведен ниже:

|  |
| --- |
| // Получение сертификата(ов) получателей  HCERTSTORE hSystemStore;  if (!(hSystemStore = CertOpenSystemStore(NULL, "MY")))  {  Error("CertOpenSystemStore");  return;  }  std::cout << "System store is opened" << std::endl;  PCCERT\_CONTEXT pCert = NULL;  while (pCert = CertEnumCertificatesInStore(hSystemStore, pCert))  {  // Выбираем первый попавшийся сертификат, использующий алгоритм sha1RSA,  // и имеющий секретный ключ.if(!strcmp(pCert->pCertInfo->SubjectPublicKeyInfo.Algorithm.pszObjId,  "1.2.840.113549.1.1.1"))  {  DWORD dwKeySpec;  HCRYPTPROV hProv;  if(CryptAcquireCertificatePrivateKey(pCert, 0, NULL, &hProv,  &dwKeySpec, NULL))  {  CryptReleaseContext(hProv, 0);  break;  }  }  }  std::cout << "Certificate found" << std::endl;  //---  CRYPT\_SIGN\_MESSAGE\_PARA SignPara;  ZeroMemory(&SignPara, sizeof(SignPara));  SignPara.cbSize = sizeof(SignPara);  SignPara.dwMsgEncodingType = X509\_ASN\_ENCODING | PKCS\_7\_ASN\_ENCODING;  SignPara.HashAlgorithm.pszObjId = (LPSTR)CertAlgIdToOID(CALG\_MD5);  SignPara.pSigningCert = pCert;  SignPara.cMsgCert = 1;  SignPara.rgpMsgCert = &pCert;  char string[] = "Test";  const BYTE\* DataArray[] = { (BYTE\*)string };  DWORD SizeArray[] = { strlen(string) };  \*count = 0;  if(!CryptSignMessage(&SignPara, true, 1, DataArray, SizeArray,  NULL, count))  {  Error("CryptSignMessage");  return;  }  \*signature = static\_cast<BYTE\*>(malloc(\*count));  if(!CryptSignMessage(&SignPara, true, 1, DataArray, SizeArray,  \*signature, count))  {  Error("CryptSignMessage");  return;  }  std::cout << "Signature content is received" << std::endl; |

**Проверка цифровой подписи**

Для проверки цифровой подписи в состав высокоуровневых функций работы с сообщениями входят функции **CryptVerifyMessageSignature** и **CrypVerifyDetachedMessageSignature**. Первая функция предназначена для проверки цифровой подписи, совмещенной с данными. Результат проверки (правильна или неправильна цифровая подпись) можно узнать, проанализировав возвращаемое значение этой функции. Вторая функция предназначена для проверки цифровой подписи, не содержащей сами подписываемые данные. Первая из этих функций имеет следующее описание:

|  |
| --- |
| BOOL CryptVerifyMessageSignature(PCRYPT\_VERIFY\_MESSAGE\_PARA pVerifyPara,  DWORD dwSignerIndex,  const BYTE\* pbSignedBlob,  DWORD cbSignedBlob,  BYTE\* pbDecoded,  DWORD\* pcbDecoded,  PCCERT\_CONTEXT\* ppSignerCert); |

В качестве первого параметра данной функции передается указатель на структуру типа CRYPT\_VERIFY\_MESSAGE\_PARA. В полях данной структуры передается информация о контексте криптопровайдера, применяемом для проверки подписи, и ссылке на функцию, с помощью которой находится сертификат подписчика в локальных хранилищах данных. В случае, когда ссылка на данную функцию равна *NULL*, функция **CryptVerifyMessageSignature** ищет данный сертификат внутри самой цифровой подписи. Параметр *dwSignerIndex* задает номер подписчика, для которого необходимо проверить подпись (в одном файле подписи потенциально может быть несколько цифровых подписей от различных подписчиков). Для первого подписчика параметр *dwSignerIndex* должен быть равным 0. В параметрах *pbSignedBlob*и*cbSignedBlob* передается информация о входном блоке данных, подлежащем проверке. В параметрах *pbDecoded*и*pcbDecoded* может быть передана информация о блоке памяти, в который помещается раскодированное проверяемое сообщение (для которого, собственно, и была сформирована цифровая подпись). В случае, когда данная информация не нужна, параметры *pbDecoded* и *pcbDecoded* должны быть установлены в *NULL*. В параметре *ppSignerCert* возвращается двойной указатель на контекст сертификата подписчика.

Пример использования этой функции примерно соответствует примеру для функции **CryptVerifyDetachedMessageSignature**, приведенному ниже.

Функция, проверяющая отсоединенную от данных цифровую подпись, имеет следующее описание:

|  |
| --- |
| BOOL CryptVerifyDetachedMessageSignature(PCRYPT\_VERIFY\_MESSAGE\_PARA pVerifyPara,  DWORD dwSignerIndex,  const BYTE\* pbDetachedSignedBlob,  DWORD cbDetachedSignedBlob,  DWORD cToBeSigned,  const BYTE\* rgpbToBeSigned[],  DWORD\* rgcbToBeSigned[],  PCCERT\_CONTEXT\* ppSignerCert); |

Параметры *pVerifyPara* и *dwSignerIndex*имеют точно такой же смысл, что и соответствующие параметры функции **CryptVerifyMessageSignature**. Параметры *pbDetachedSignedBlob* и *cbDetachedSignedBlob* описывают область памяти, которая хранит собственно проверяемые данные. Параметр *cToBeSigned* описывает количество элементов в массивах, передаваемых в параметрах *rgpbToBeSigned* и *rgcbToBeSigned*. Параметр *rgpbToBeSigned* представляет собой массив областей памяти, содержащий в себе данные, для которых проверяется цифровая подпись. Параметр *rgcbToBeSigned* содержит массив размеров блоков памяти, указанных в параметре *rgpbToBeSigned*. В параметре *ppSignerCert* возвращается двойной указатель на контекст сертификата подписчика. Результат проверки (правильна или неправильна цифровая подпись) возвращается как результат выполнения функции **CryptVerifyDetachedMessageSignature**.

Пример использования данной функции приведен ниже:

|  |
| --- |
| CRYPT\_VERIFY\_MESSAGE\_PARA VerifyPara;  ZeroMemory(&VerifyPara, sizeof(VerifyPara));  VerifyPara.cbSize = sizeof(VerifyPara);  VerifyPara.dwMsgAndCertEncodingType = X509\_ASN\_ENCODING | KCS\_7\_ASN\_ENCODING;  VerifyPara.hCryptProv = NULL;  char string[] = "Test";  const BYTE\* DataArray[] = { (BYTE\*)string };  DWORD SizeArray[] = { strlen(string) };  BOOL result = CryptVerifyDetachedMessageSignature(&VerifyPara, 0,  signature, count, 1, DataArray, SizeArray, NULL);  std::cout << "Result: signature "  << ((result) ? "verified!" : "NOT verified!") << std::endl; |

**Совмещение цифровой подписи и шифрованных данных**

Кроме применения формирования отдельно шифрованного и отдельно подписанного контентов в стандарте PKCS #7 предусмотрено получение контента, который представляет собой цифровую подпись, совмещенную с зашифрованными данными (сначала формируется цифровая подпись открытых данных, затем данные шифруются). Таким образом, собственно проверка данных может быть осуществлена получателем шифрованных данных, для которого они предназначались. Такой способ передачи данных хорош, прежде всего, отсутствием передачи проверяемых данных в открытом виде.

Функция **CryptSignAndEncryptMessage**, осуществляющая подобную работу в Crypto API, имеет следующее описание:

|  |
| --- |
| BOOL CryptSignAndEncryptMessage(PCRYPT\_SIGN\_MESSAGE\_PARA pSignPara,  PCRYPT\_ENCRYPT\_MESSAGE\_PARA pEncryptPara,  DWORD cRecipientCert,  PCCERT\_CONTEXT rgpRecipientCert[],  const BYTE\* pbToBeSignedAndEncrypted,  DWORD cbToBeSignedAndEncrypted,  BYTE\* pbSignedAndEncryptedBlob,  DWORD\* pcbSignedAndEncryptedBlob); |

Так как, по сути, выполнение данной функции состоит из последовательного применения функций для получения цифровой подписи и функции шифрования, то параметры у функции **CryptSignAndEncryptMessage** точно совпадают с параметрами составляющих ее функций. Значение всех этих параметров было объяснено ранее в соответствующих разделах.

Пример использования данной функции приведен ниже:

|  |
| --- |
| HCRYPTPROV hProv;  HCERTSTORE hSystemStore;  // Получение сертификата(ов) получателейif (!(hSystemStore = CertOpenSystemStore(NULL, "MY")))  {  Error("CertOpenSystemStore");  return;  }  std::cout << "System store is opened" << std::endl;  PCCERT\_CONTEXT pCert = NULL;  while (pCert = CertEnumCertificatesInStore(hSystemStore, pCert))  {  // Выбираем первый попавшийся сертификат, использующий алгоритм sha1RSA,  // и имеющий секретный ключ.if(!strcmp(pCert->pCertInfo->SubjectPublicKeyInfo.Algorithm.pszObjId,  "1.2.840.113549.1.1.1"))  {  DWORD dwKeySpec;  if (CryptAcquireCertificatePrivateKey(pCert, 0, NULL, &hProv,  &dwKeySpec, NULL))  break;  }  }  std::cout << "Certificate found" << std::endl;  // Инициализация структуры, необходимой для цифровой подписи  CRYPT\_SIGN\_MESSAGE\_PARA SignPara;  ZeroMemory(&SignPara, sizeof(SignPara));  SignPara.cbSize = sizeof(SignPara);  SignPara.dwMsgEncodingType = X509\_ASN\_ENCODING | PKCS\_7\_ASN\_ENCODING;  SignPara.HashAlgorithm.pszObjId = (LPSTR)CertAlgIdToOID(CALG\_MD5);  SignPara.pSigningCert = pCert;  SignPara.cMsgCert = 1;  SignPara.rgpMsgCert = &pCert;  // Инициализация структуры, необходимой для шифрования  CRYPT\_ENCRYPT\_MESSAGE\_PARA EncryptPara;  ZeroMemory(&EncryptPara, sizeof(EncryptPara));  EncryptPara.cbSize = sizeof(EncryptPara);  EncryptPara.dwMsgEncodingType = X509\_ASN\_ENCODING | PKCS\_7\_ASN\_ENCODING;  EncryptPara.ContentEncryptionAlgorithm.pszObjId =  (LPSTR)CertAlgIdToOID(CALG\_RC4);  EncryptPara.hCryptProv = hProv;  char string[] = "Test";  if (!CryptSignAndEncryptMessage(&SignPara, &EncryptPara, 1, &pCert,  (BYTE\*)string, strlen(string), NULL, count))  {  Error("CryptSignAndEncryptMessage");  return;  }  \*encrypted = static\_cast<BYTE\*>(malloc(\*count));  if (!CryptSignAndEncryptMessage(&SignPara, &EncryptPara, 1, &pCert,  (BYTE\*)string, strlen(string), \*encrypted, count))  {  Error("CryptSignAndEncryptMessage");  return;  }  std::cout << "Encrypted and signed content is received" << std::endl;  CertFreeCertificateContext(pCert);  CertCloseStore(hSystemStore, 0);  CryptReleaseContext(hProv, 0); |

**Расшифровывание и проверка совмещенной цифровой подписи и шифрованных данных**

Для расшифровывания и проверки совмещенных цифровой подписи и шифрованных данных применяют специальную функцию **CryptDecryptAndVerifyMesageSignature**, имеющую следующее описание:

|  |
| --- |
| BOOL CryptDecryptAndVerifyMesageSignature(PCRYPT\_DECRYPT\_MESSAGE\_PARA pDecryptPara,  PCRYPT\_VERIFY\_MESSAGE\_PARA pVerifyPara,  DWORD dwSignerIndex,  const BYTE\* pbEncryptedBlob,  DWORD cbEncryptedBlob,  BYTE\* pbDecrypted  DWORD\* pcbDecrypted,  PCCERT\_CONTEXT\* ppXchgCert,  PCCERT\_CONTEXT\* ppSignerCert); |

Значения параметров данной функции полностью аналогичны значениям параметров функций **CryptDecryptMessage** и **CryptVerifyMessageSignature**. Результат проверки (правильна или неправильна цифровая подпись) получается как результат функции **CryptDecryptAndVerifyMessageSignature**.

Пример использования данной функции приведен ниже:

|  |
| --- |
| HCERTSTORE hSystemStore;  if(!(hSystemStore = CertOpenSystemStore(NULL, "MY")))  {  Error("CertOpenSystemStore");  return;  }  std::cout << "System store is opened" << std::endl;  // Инициализация структуры, необходимой для расшифровывания  CRYPT\_DECRYPT\_MESSAGE\_PARA DecryptPara;  ZeroMemory(&DecryptPara, sizeof(DecryptPara));  DecryptPara.cbSize = sizeof(DecryptPara);  DecryptPara.dwMsgAndCertEncodingType =  X509\_ASN\_ENCODING | PKCS\_7\_ASN\_ENCODING;  HCERTSTORE StoreArray[] = {hSystemStore};  DecryptPara.cCertStore = 1;  DecryptPara.rghCertStore = StoreArray;  // Инициализация структуры, необходимой для проверки цифровой подписи  CRYPT\_VERIFY\_MESSAGE\_PARA VerifyPara;  ZeroMemory(&VerifyPara, sizeof(VerifyPara));  VerifyPara.cbSize = sizeof(VerifyPara);  VerifyPara.dwMsgAndCertEncodingType =  X509\_ASN\_ENCODING | PKCS\_7\_ASN\_ENCODING;  VerifyPara.hCryptProv = NULL;  BOOL result = CryptDecryptAndVerifyMessageSignature(&DecryptPara,  &VerifyPara, 0, encrypted, count, NULL, NULL, NULL, NULL);  std::cout << "Decryption and verification completed" << std::endl;  std::cout << "Result: string " << ((result)?"verified!":"NOT verified!")  << std::endl;  free(encrypted);  CertCloseStore(hSystemStore, 0); |

**Работа с форматом Base64 (PEM)**

Кроме стандарта *PKCS #7* достаточно часто можно встретить употребление другого стандарта – *PEM* (Privacy-Exchanged Mail). Фактически же в данном стандарте применяются два последовательных кодирования: первичное в *PKCS #7* и вторичное в *Base64*. То есть если мы после завершения шифрования высокоуровневой функцией **CryptEncryptMessage** закодируем полученный результат в кодировке *Base64*, то полученный результат можно принять за образец применения стандарта *PEM*.

Работа с *Base64* крайне проста и реализована во многих библиотеках. В частности, я бы хотел порекомендовать читателям средства, предоставляемые функциями **Base64Encode** и **Base64Decode**. Данные функции объявлены в заголовочном файле библиотеки ATL *“atlenc.h”*. Работа с данными функциями достаточно проста. Ниже приведен пример использования данных функций.

|  |
| --- |
| char data[] = "Test";  //--- Кодирование полученного массива данных в строку формата BASE64int len = 0;  len = Base64EncodeGetRequiredLength(sizeof(data), ATL\_BASE64\_FLAG\_NOCRLF);  LPSTR out = static\_cast<LPSTR>(malloc(len + 1));  ZeroMemory(out, len + 1);  Base64Encode((BYTE\*)data, sizeof(data), out, &len, ATL\_BASE64\_FLAG\_NOCRLF);  std::cout << "Encode completed" << std::endl;  // Вывод на экран результата  std::cout << "Encoded string: " << out << std::endl;  //--- Конвертация входной строки подписи из кодировки BASE64char\* decode = static\_cast<char\*>(malloc(strlen(data) + 1));  ZeroMemory(decode, strlen(data) + 1);  Base64Decode(out, len, (BYTE\*)decode, &len);  std::cout << "Decode completed" << std::endl;  // Вывод на экран результата  std::cout << "Decoded string: " << decode << std::endl;  free(out);  free(decode); |

CAPICOM

Зачастую применение непосредственно функций Crypto API достаточно проблематично. Например, в Web-клиентах, где вызовы процедур напрямую невозможны. Для подобных целей, а также для упрощения работы с Crypto API был создан тип объектов *CAPICOM* (Crypto API COM-object). В своей реализации данный объект почти полностью охватывает все то, о чем говорилось выше в данной статье: от шифрования до работы с сертификатами. Благодаря использованию дуальных интерфейсов, доступ к данному объекту возможен как из клиентов с ранним связыванием (C++), так и с поздним (VBscript).

В основе работы всех функций *CAPICOM* лежит использование высокоуровневых функций работы с криптографическими сообщениями, которые были рассмотрены ранее. Таким образом, базовым стандартом для выходных данных является стандарт *PKCS #7*, вторично кодированный *Bas64*. Так же, как и в случае высокоуровневых функций, работа ведется только с относительно небольшим участком данных, загруженных в память. Работа с данными большого объема или работа с поблочной загрузкой данных в объекте не предусмотрена.

|  |
| --- |
| **ПРЕДУПРЕЖДЕНИЕ**  Несмотря на то, что объект *CAPICOM* в качестве выходных данных использует общепринятый стандарт *PKCS #7*, используемый в функциях Crypto API, полной прямой совместимости между функциями Crypto API и *CAPICOM* нет. То есть выходная информация объекта *CAPICOM* не может быть использована в качестве входной для функций Crypto API. Дело в том, что в объекте CAPICOM происходит некоторое предварительное изменение входных данных перед выполнениями функций. То есть алгоритм выполнения, например, функции подписи, выглядит так:  1. Получить входные данные.  2. Изменить их (для внутреннего использования).  3. Вызвать стандартные функции Crypto API.  В качестве «изменения» входных данных используется простое дополнение к каждому символу исходной информации символа с кодом 0x00. |

Подробно работа с объектом типа *CAPICOM* в данной статье рассматриваться не будет, так как сам по себе объект не входит в состав Crypto API и рассказ о нем несколько выходит за рамки этой статьи.