AMB Group

Центр Защиты Ресурсов Информационных Систем

ЦЕЗАРИС

Инфраструктура открытых ключей

Руководство программиста

Редакция 1.0

АННОТАЦИЯ

Данный документ содержит методику использования криптопровайдера ***"CESARIS Cryptographic Service Provider ©"***, который входит в состав проекта **Центр Защиты Ресурсов Информационных Систем (ЦЕЗАРIС)**, для встраивания инструментов цифровой подписи в разрабатываемые приложения с применением ***Microsoft Cryptographic Application Programming Interface (CryptoAPI).*** Приведены примеры.

Документ предназначен для прикладных программистов, разрабатывающих собственные приложения.

СОДЕРЖАНИЕ

1. Разработка приложений с использованием криптопровайдера

Криптопровайдер поддерживает стандартный интерфейс - *Microsoft Cryptographic Application Programming Interface (CryptoAPI)*. Подробное описание функций интерфейса дано в библиотеке MSDN (msdn.microsoft.com) в разделе [Security].

На некоторых особенностях CryptoAPI мы остановимся ниже.

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

Функции высокого уровня, предназначенные для реализации криптографических преобразований, вызывают именно функции CryptoAPI 1.0. Таким образом, именно интерфейс CryptoAPI 1.0 является криптографическим ядром прикладного уровня современных операционных систем Microsoft.

1.1. Обзор функции CryptoAPI 1.0

В этом разделе дано краткое описание 31 функции, составляющих интерфейс Microsoft CryptoAPI 1.0. Для удобства функции размещены в разных таблицах, в зависимости от назначения:

- в таблице 1 - функции управления криптопровайдерами и контекстами криптопровайдеров;
- в таблице 2 - функции, которые применяются приложениями для создания, конфигурирования и уничтожения криптографических ключей, а также для передачи ключей другим приложениям;
- в таблице 3 - функции, реализующие операции шифрования и дешифрования с использованием симметричных ключей;
- в таблице 4 - функции, которые используются приложениями для вычисления значений хеш-функций, а также создания и проверки цифровой подписи сообщений.

Таблица 1. Функции управления криптопровайдерами и контекстами криптопровайдеров

Функция	Краткое описание
CryptAcquireContext	Используется для создания дескриптора определенного ключевого контейнера в рамках определенного криптопровайдера.
CrypiContextAddRef	Увеличивает на единицу счетчик ссылок на дескриптор криптопровайдера.

CryptEnumProviders	Используется для получения первого и следующего доступного криптопровайдера.
CryptEnumProviderTypes	Используется для получения первого и следующего типа доступных криптопровайдеров.
CryptGetDefaultProvider	Находит криптопровайдер, используемый по умолчанию, для указанного типа криптопровайдера.
CryptGetProvParam	Возвращает параметры криптопровайдера.
CryptReleaseContext	Используется для освобождения дескриптора криптопровайдера, созданного ***CryptAcquireContext.***
CryptSetProvider ***CryptSetProviderEx***	Используется для задания имени и типа криптопровайдера, используемого по умолчанию.
CryptSetProvParam	Устанавливает параметры криптопровайдера.

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

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

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

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

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

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

1.2. Принципы реализации интерфейса вызовов CryptoAPI 1.0

Общая архитектура CryptoAPI 1.0 показана на рис. 4-1.

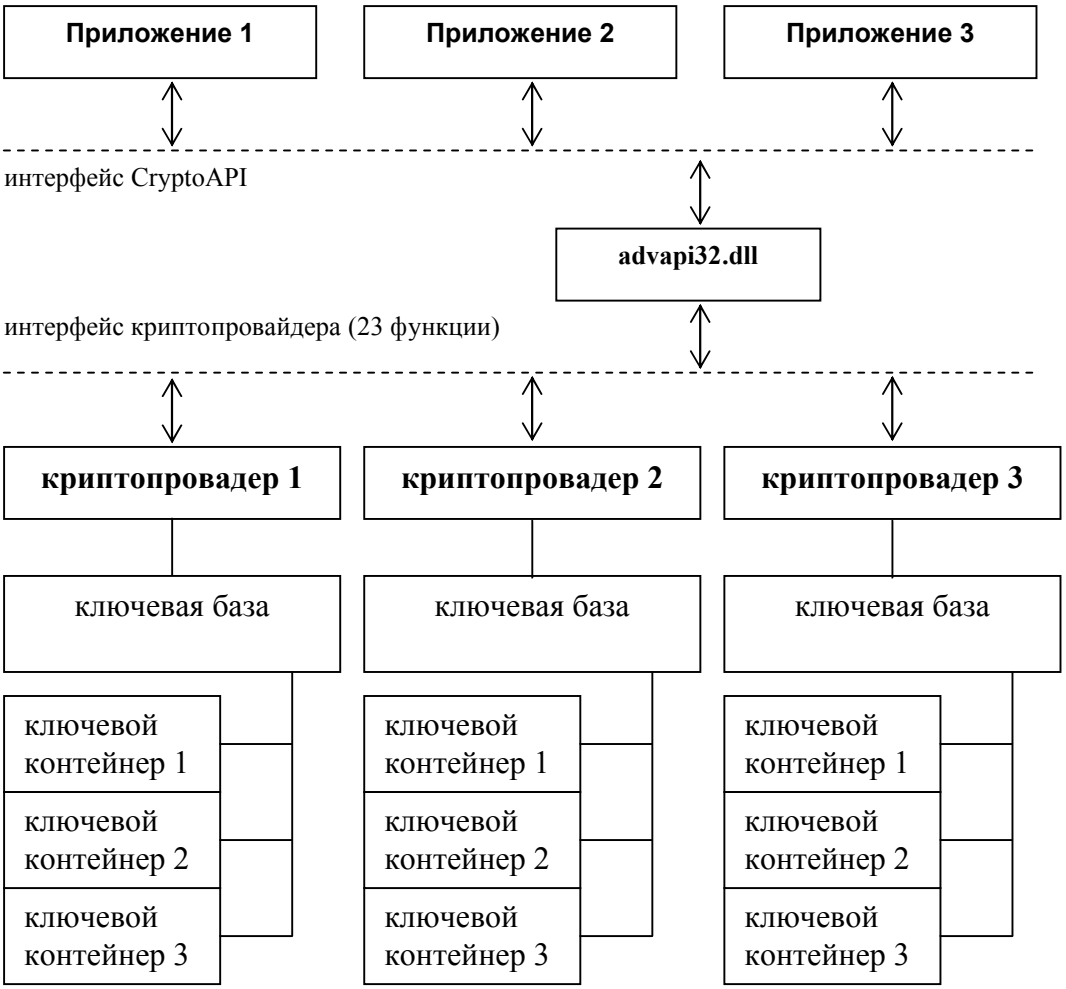

Рис. 4-1. Архитектура CryptoAPI 1.0

Все функции интерфейса, описанные в разделе "Обзор функции CryptoAPI 1.0", содержатся в библиотеке advapi32.dll. За исключением нескольких сервисных функций (например, CryptSetProvider), эти процедуры выполняют ряд вспомогательных операций и вызывают библиотеку, в которой непосредственно реализованы соответствующие криптографические преобразования. Такие библиотеки называются криптопровайдерами (Cryptographic Service Provider, CSP).

Криптопровайдеры имеют стандартный набор функций, который состоит из 23 обязательных и 2 необязательных процедур.

Всю необходимую информацию об определенном криптопровайдере можно получить средствами функции CryptGetProvParam.

Самое главное, что необходимо при этом знать, — наборы криптографических стандартов (шифрования, хеширования, цифровой подписи и несимметричного обмена ключами), которые реализуют криптопровайдеры, установленные в системе (см. "Примеры использования криптопровайдера", пример в приложении 1).

Каждый криптопровайдер характеризуется собственным именем и типом. Его имя - просто строка, по которой система распознает криптопровайдер. Так, базовый

криптопровайдер Microsoft назван "Microsoft Base Cryptographic Provider v1.0". Тип криптопровайдера — целое число (в нотации языка С — DWORD), значение которого идентифицирует набор поддерживаемых алгоритмов цифровой подписи и несимметричного обмена ключей. "Microsoft Base Cryptographic Provider v1.0" имеет тип 1 (в файле WinCrypt.h определена константа PROV_RSA_FULL), этот тип криптопровайдеров реализует в качестве алгоритмов цифровой подписи и несимметричного обмена ключей стандарт RSA.

Криптопровайдер "CESARIS GOST 34.310-95 and RSA Cryptographic Provider" имеет тип 804. Этот тип криптопровайдеров реализует в качестве алгоритма цифровой подписи стандарт ГОСТ 34.310-95, а для реализации ключевого обмена использует алгоритм RSA.

Криптопровайдер "CESARIS DSTU 4145-2002(PB) and ECDH Cryptographic Provider" имеет тип 806. Этот тип криптопровайдеров реализует в качестве алгоритма цифровой подписи стандарт ДСТУ 4145-2002 (полиномиальный базис), а для реализации ключевого обмена использует алгоритм то же стандарт по протоколу Диффи-Хелмана (ECDH).

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

Для работы с набором криптопровайдеров в системном реестре *(HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider)* содержится список имен всех криптопровайдеров. С каждым именем связан тип криптопровайдера и имя библиотеки, реализующей криптопровайдер. Кроме того, в системе содержится информация о том, какой криптопровайдер применять, если пользователь при вызове не определил конкретное имя, а задал только тип необходимого ему криптопровайдера. Такие криптопровайдеры называются криптопровайдерами, используемыми по умолчанию (default cryptographic service provider) для данного типа. Для типа 1 криптопровайдером по умолчанию является «Microsoft Base Cryptographic Provider vl.O», а для типа 13 — «Microsoft Base DSS and Diffie-Hellman Cryptographic Provider».

Для определения имени криптопровайдера по умолчанию можно воспользоваться функцией CryptGetDefaultProvider (см. Приложение 2), а для изменения этого параметра — функциями CryptSetProvider или CryptSetProviderEx. Из описания этих функций следует, что криптопровайдер по умолчанию задается как для текущего пользователя (current user), так и для системы в целом (local computer). Вторые задаются для всех типов установленных криптопровайдеров при установке операционной системы и сохраняются в ключе реестра HKEY_LOCAL_MACHINE. В дальнейшем можно изменить эти параметры или назначить криптопровайдер по умолчанию только для текущего пользователя. Параметры для текущего пользователя имеют приоритет перед общесистемными и сохранятся в ключе реестра HKEY_CURRENT_USER. Разумеется, если параметры для текущего пользователя в явном виде отсутствуют, то применяются общесистемные.

Ключевая база каждого криптопровайдера состоит из контейнеров, в которых могут храниться 2 типа ключевых пар:

- для обмена ключами – тип 1 (в файле WinCrypt.h - AT_KEYEXCHANGE)
- для подписи - тип 2 (в файле WinCrypt.h - AT_SIGNATURE)

2. Использование криптопровайдера

В **приложении 1** размещен пример, показывающий, каким образом пользователь начинает работу с криптопровайдером, который ему необходим, и каким образом система вызывает конкретную библиотеку, соответствующую выбранному криптопровайдеру.

Рассматривается действие функции ***CryptAcquireContext***. С ее вызова начинается любая программа, работающая с CryptoAPI. Именно в этой функции пользователь задает имя используемого криптопровайдера, его тип и имя рабочего ключевого контейнера. Функция ***CryptAcquireContext*** возвращает пользователю дескриптор криптопровайдера (handle of a cryptographic service provider), который в дальнейшем пользователь передает в процедуры для проведения всех необходимых операций с криптопровайдером.

На примере функции ***OwnCryptAcquireContextA*** приблизительно показана работа библиотеки **advapi32.dll** при вызове функции ***CryptAcquireContext***.

Также рассмотрена структура и последовательность формирования контекста криптопровайдера. Дескриптор криптопровайдера, который возвращает функция ***CryptAcquireContext,*** является указателем на контекст криптопровайдера. Когда мы говорим "дескриптор криптопровайдера" или "контекст криптопровайдера", это не совсем точно. С помощью функции ***CryptAcquireContext*** создается контекст не всего криптопровайдера, а определенного ключевого контейнера криптопровайдера. Конечно, используя полученный дескриптор, мы можем получать параметры криптопровайдера и устанавливать параметры, которые будут распространяться на все ключевые контексты. Но, по сути, функция ***CryptAcquireContext*** записывает ключевой контейнер в память и возвращает приложению дескриптор, позволяющий работать с этим образом ключевого контейнера.

Далее вызывается функция ***CryptGetProvParam*** для получения параметров криптопровайдера.

Когда вызывается функцию ***CryptReleaseContext,*** то разрушается только образ ключевого контейнера, сам ключевой контейнер остается в физическом хранилище криптопровайдера (см. рис. 8). Для удаления ключевого контейнера из физического хранилища необходимо использовать функцию ***CryptAcquireContext*** *с* флагом CRYPT_DELETEKEYSET. Исключение составляет ключевой контекст, созданный с флагом CRYPT_VERIFYCONTEXT. Он не является отражением ключевого контейнера. В таком контексте нельзя создавать долговременные ключевые пары подписи и обмена. Он предназначен для проверки подписи сообщений, поскольку в этом случае нет необходимости доступа к секретному ключу пары.

При создании объектов ключей (функция ***CryptGenKey)*** и хеш-функций (функция ***CryptCreateHash)*** создается контекст с аналогичной структурой. Однако аналогия не полная, есть несколько отличий. В контексте ключа или хеш-функции сохраняются указатели только на те функции, в работе которых принимают участие дескрипторы этих объектов.

Для решения задач целостности и аутентичности необходимо использовать механизм электронной цифровой подписи. **В приложении 2** на примере показана процедура формирования отправителем подписи к отправляемому сообщению. Текст

сообщения содержится в файле, собственно, как и электронная подпись со всеми необходимыми параметрами.

Криптопровайдеры поддерживают два алгоритма цифровой подписи: ГОСТ 34.310-95 и ДСТУ 4145-2002. Кроме того, поддерживаются алгоритм хеш-функции, используемой в подписи, согласно с ГОСТ 34.311-95.

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

Рекомендуемая литература

1. А.Щербаков, А.Домашев "Прикладная криптография. Использование и синтез криптографических интерфейсов". Москва: Русская редакция, 2003.
2. RSA Laboratories. PKCS#11: Cryptographic Token Interface Standard.
3. ГОСТ 34.311-95 "Информационная технология. Криптографическая защита информации. Функция хеширования".
4. ГОСТ 34.310-95 "Информационная технология. Криптографическая защита информации. Процедура выработки и проверки электронной цифровой подписи на базе ассиметричного криптографического алгоритма ";
5. ДСТУ 4145-2002 "Інформаційні технології. Криптографічний захист інформації. Цифровий підпис, що ґрунтується на еліптичних кривих. Формування та перевіряння"
6. ГОСТ 28147-89 "Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования".
7. RSA Laboratories. PKCS#1: RSA Cryptography Standard.

Приложение 1. Пример использования CryptoAPI 1.0 для определения параметров криптопровайдера

```
#include <windows.h>
#include <stdio.h>

#define PROV_G34310_RSA_NAME            "CESARIS GOST 34.310-95 and RSA Cryptographic Provider"
#define PROV_G34310_RSA _TYPE           804
#define PROV_D4145PB_RSA_NAME           "CESARIS DSTU 4145-2002(PB) and RSA Cryptographic Provider"
#define PROV_D4145PB_RSA _TYPE          805

//данный пример использует ГОСТ 34.310-95
#define PROV_NAME                       PROV_G34310_RSA_NAME
#define PROV_TYPE                       PROV_G34310_RSA _TYPE


typedef struct BKP_CSP_FUNCTION_LIST BKP_CSP_FUNCTION_LIST;

#ifndef CRYPT_VERIFY_IMAGE_A
typedef BOOL (WINAPI *CRYPT_VERIFY_IMAGE_A)(LPSTR  szImage, CONST BYTE *pbSigData);
#endif
#ifndef CRYPT_VERIFY_IMAGE_W
typedef BOOL (WINAPI *CRYPT_VERIFY_IMAGE_W)(LPCWSTR szImage, CONST BYTE *pbSigData);
#endif
#ifndef CRYPT_RETURN_HWND
typedef void (*CRYPT_RETURN_HWND)(HWND *phWnd);
#endif

// Векторы ADVAPI-функций
typedef struct _VTableProvStruc {
        DWORD           Version;
        CRYPT_VERIFY_IMAGE_A FuncVerifyImage;
        CRYPT_RETURN_HWND    FuncReturnhWnd;
        DWORD           dwProvType;
        BYTE            *pbContextInfo;
        DWORD           cbContextInfo;
        LPSTR           pszProvName;
} VTableProvStruc,     *PVTableProvStruc;

//см.комментарий 1
// Константы, определяющие версию, используемой ОС
// От этого зависит структура контекста криптопровайдера
// и его инициализация

// Определяется для Windows NT 4.0
//#define _WIN32_WINNT      0x0400
// Определяется для Windows 2000 и старше
// (Windows 98 и старше)
#define _WIN32_WINNT       0x0500


// Определение максимального значения типа криптопровайдера
#define MAX_PROV_TYPE           999
// Определение идентификатора контекста криптопровайдера
#define PROVIDER_CONTEXT        0x11111111

// Определение массива имен классов алгоритмов
LPTSTR  szAlgClass[]={
                        NULL,
                        "Signature",
                        "Encrypt  ",
                        "Encrypt  ",
                        "Hash     ",
                        "Exchange ",
                        "All      "};


// Определение прототипа функции CSPInCacheCheck
BOOL WINAPI
CSPInCacheCheck(LPTSTR  pszDllName, HMODULE       *phProvDll)
{
        return FALSE;
}

// Определение прототипа функции AddHandleToCSPCache
VOID WINAPI
AddHandleToCSPCache(HMODULE hProvDll)
```

```
{
        return;
}

// Определение прототипа функции CheckSignatureInFile
typedef BOOL (WINAPI * CHECK_SIGNATURE_IN_FILE_FN) (LPWSTR);

CHECK_SIGNATURE_IN_FILE_FN      pfnCheckSignatureInFile = NULL;

// Определение прототипа функции CProvVerifyImage
BOOL WINAPI
CProvVerifyImage(LPSTR     lpszImage,CONST BYTE  *pSigData)
{
        return TRUE;
}

// Определение прототипа функции FuncReturnhWnd
VOID //WINAPI
FuncReturnhWnd(HWND *phWnd)
{
        *phWnd=0;
//      return TRUE;
}

//см.комментарий 2
// Определение типа функции CPAcquireContext
typedef BOOL (WINAPI *PFN_CP_ACQUIRE_CONTEXT)(
                                        HCRYPTPROV      *phProv,
                                        LPCSTR  *pszContainer,
                                        DWORD           dwFlags,
                                        PVTableProvStruc  pVTable
                                        );

//см.комментарий 3
// Определение структуры контекста криптопровайдера
// (структура зависит от операционной системы)
typedef struct _PROV_CONTEXT_ {
        // Массив указателей на функции криптопровайдера
        FARPROC                         CSPFuncPtr[24];

#if (_WIN32_WINNT >= 0x0500)
        // Массив указателей на необязательные функции криптопровайдера
        // (используется начиная с Windows 2000 и Windows 98)
        FARPROC                         CSPOptionalFuncPtr[3];
#endif

        // Дескриптор библиотеки криптопровайдера
        HMODULE                         hCSPDll;
        // Дескриптор ключевого контекста, который вернура функция
        // криптопровайдера CPAcquireContext
        HCRYPTPROV                      hCSP;
        // Идентификатор контекста
        DWORD                           ContextId;
        // Счетчик использования ключевого контекста
        DWORD                           dwContextUseCount;

#if (_WIN32_WINNT >= 0x0500)
        // Дополнительный счетчик использования ключевого
        // контекста, инкрементируется функцией CryptContextAddRef
        // (используется начиная с Windows 2000 и Windows 98)
        DWORD                           dwContextRefCount;
#else
        // Критическая секция ключевого контекста
        CRITICAL_SECTION        CSPLock;
#endif

} PROV_CONTEXT, *PPROV_CONTEXT;

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

```
                                "CPGetProvParam",           "CPSetHashParam",
                                "CPGetHashParam",           NULL};

// Определение массива имен необязательных функций криптопровайдера
LPTSTR  OptionalFunctionNames[]={"CPDuplicateKey", "CPDuplicateHash", NULL};
//см.комментарий 5
// Имя ключа реестра (базовый ключ HKEY_CURRENT_USER), в котором
// хранятся имена криптопровайдеров, используемых  по умолчанию, для текущего пользователя
LPTSTR szUserType="SOFTWARE\\Microsoft\\Cryptography\\Providers\\Type ";
// Имя ключа реестра (базовый ключ HKEY_LOCAL_MACHINE), в котором
// хранятся имена криптопровайдеров, используемых  по умолчанию, для системы
LPTSTR szMachineType="SOFTWARE\\Microsoft\\Cryptography\\Defaults\\Provider Types\\Type";
// Имя ключа реестра (базовый ключ HKEY_LOCAL_MACHINE), в котором
// хранятся имена всех криптопровайдеров, установленных на компьтере
LPTSTR szProvider="SOFTWARE\\Microsoft\\Cryptography\\Defaults\\Provider\\";

PBYTE pbContextInfo=NULL;
DWORD cbContextInfo=0;

//см.комментарий 6
// Определение реконструированной функции CryptAcquireContextA
BOOL WINAPI
OwnCryptAcquireContextA(
   HCRYPTPROV  *phProv,
   LPCSTR                   pszContainer,
   LPCSTR                   pszProvider,
   DWORD                    dwProvType,
   DWORD                    dwFlags
   )
{
        BOOL                        Result=TRUE;
        LPTSTR                      pszRegStr=NULL, pszDllName=NULL, pszProvName=NULL;
        LPWSTR                      pwszDllName=NULL;
        PBYTE                       pbSignature=NULL;
        DWORD                       dwRegError=ERROR_SUCCESS;
        DWORD                       dwCryptError=ERROR_SUCCESS;
        HKEY                        hkResult=0;
        DWORD                       dwRegType, dwQueryProvType;
        DWORD                       cbData=0;
        HMODULE                             hProvDll=0, hAdvapiDll=0;
        PPROV_CONTEXTTpProvContext=NULL;
        DWORD                       dwFuncCount=0;
        VTableProvStruc    VTable;


        //Если значение типа криптопровайдера не попадает
        //в заданные границы, то возвращаем ошибку
        if (dwProvType==0 || dwProvType > MAX_PROV_TYPE) {
                Result=FALSE;
                dwCryptError=NTE_BAD_PROV_TYPE;
                goto ReleaseResource;
        }

        if (pszProvider == NULL) {
//см.комментарий 7
                // Если имя криптопровайдера не определено, то
                // пытаемся определить криптопровайдер, используемый по
                // умолчания для указанного типа
                pszRegStr=LocalAlloc(LMEM_ZEROINIT,lstrlen(szUserType)+4);
                if (!pszRegStr) {
                        Result=FALSE;
                        dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
                        goto ReleaseResource;
                }

                // Формируем строку реестра для типа криптопровайдера
                // текущего пользователя
                lstrcpy(pszRegStr,szUserType);
                sprintf(pszRegStr+lstrlen(szUserType),"%03d",dwProvType);

                // Пытаемся открыть ключ реестра для типа криптопровайдера
                // текущего пользователя
                dwRegError=RegOpenKeyEx(
                                        HKEY_CURRENT_USER,
                                        pszRegStr,
                                        0,
                                        KEY_READ,
                                        &hkResult);
                if (dwRegError != ERROR_SUCCESS) {
//см.комментарий 8
                        // Если ключ отсутствует, то обращаемся к системному ключу реестра
```

```
                        LocalFree(pszRegStr);
                        pszRegStr=LocalAlloc(LMEM_ZEROINIT,lstrlen(szMachineType)+4);
                        if (!pszRegStr) {
                                Result=FALSE;
                                dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
                                goto ReleaseResource;
                        }
                        // Формируем строку реестра для типа криптопровайдера,
                        // определенного в системе
                        lstrcpy(pszRegStr,szMachineType);
                        sprintf(pszRegStr+lstrlen(szMachineType),"%03d",dwProvType);

                        // Пытаемся открыть ключ реестра для типа криптопровайдера,
                        // определенного в системе
                        dwRegError=RegOpenKeyEx(
                                        HKEY_LOCAL_MACHINE,
                                        pszRegStr,
                                        0,
                                        KEY_READ,
                                        &hkResult);
                        if (dwRegError != ERROR_SUCCESS) {
                                Result=FALSE;
                                dwCryptError=NTE_PROV_TYPE_NOT_DEF;
                                goto ReleaseResource;
                        }
                }

//см.комментарий 9
                // По открытому ключу определяем имя криптопровайдера,
                // используемого по умолчанию
                dwRegError=RegQueryValueEx(
                                        hkResult,
                                        "Name",
                                        NULL,
                                        &dwRegType,
                                        NULL,
                                        &cbData);
                if (dwRegError != ERROR_SUCCESS) {
                        Result=FALSE;
                        dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
                        goto ReleaseResource;
                }
                // Выделяем память под строку с именем криптопровайдера
                pszProvName=LocalAlloc(LMEM_ZEROINIT,cbData);
                if (!pszProvName) {
                        Result=FALSE;
                        dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
                        goto ReleaseResource;
                }
                // Считываем из реестра строку с именем криптопровайдера
                dwRegError=RegQueryValueEx(
                                        hkResult,
                                        "Name",
                                        NULL,
                                        &dwRegType,
                                        pszProvName,
                                        &cbData);
                if (dwRegError != ERROR_SUCCESS) {
                        Result=FALSE;
                        dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
                        goto ReleaseResource;
                }

                // Освобождаем выделенные ресурсы
                LocalFree(pszRegStr); pszRegStr=NULL;
                RegCloseKey(hkResult); hkResult=0;
        }
        else {
                // Если имя криптопровайдера определено явно, то
                // копируем его в рабочий буфер
                pszProvName=LocalAlloc(LMEM_ZEROINIT,lstrlen(pszProvider)+1);
                if (!pszProvName) {
                        Result=FALSE;
                        dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
                        goto ReleaseResource;
                }
                lstrcpy(pszProvName,pszProvider);
        }

//см.комментарий 10
        // Формируем строку для доступа к записи о выбранном криптопровайдере
```

```
	pszRegStr=LocalAlloc(LMEM_ZEROINIT,lstrlen(szProvider)+lstrlen(pszProvName)+1);
	if (!pszRegStr) {
		Result=FALSE;
		dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
		goto ReleaseResource;
	}
	lstrcpy(pszRegStr,szProvider);
	lstrcpy(pszRegStr+lstrlen(szProvider),pszProvName);

	// Открываем ключ реестра с информацией о выбранном криптопровайдере
	dwRegError=RegOpenKeyEx(
					HKEY_LOCAL_MACHINE,
					pszRegStr,
					0,
					KEY_READ,
					&hkResult);
	if (dwRegError != ERROR_SUCCESS) {
		Result=FALSE;
		dwCryptError=NTE_KEYSET_NOT_DEF;
		goto ReleaseResource;
	}
	LocalFree(pszRegStr); pszRegStr=NULL;

	// По открытому ключу считываем тип криптопровайдера
	cbData=sizeof(DWORD);
	dwRegError=RegQueryValueEx(
					hkResult,
					"Type",
					NULL,
					&dwRegType,
					(PBYTE)&dwQueryProvType,
					&cbData);
	if (dwRegError != ERROR_SUCCESS) {
		Result=FALSE;
		dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
		goto ReleaseResource;
	}

	// Если считанный тип не совпадает с заданным при
	// вызове функции, то возвращаем ошибку
	if (dwQueryProvType != dwProvType) {
		Result=FALSE;
		dwCryptError=NTE_PROV_TYPE_NO_MATCH;
		goto ReleaseResource;
	}

//см.комментарий 11
	// По открытому ключу считываем имя библиотеки криптопровайдера
	dwRegError=RegQueryValueEx(
					hkResult,
					"Image Path",
					NULL,
					&dwRegType,
					NULL,
					&cbData);
	if (dwRegError != ERROR_SUCCESS) {
		Result=FALSE;
		dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
		goto ReleaseResource;
	}
	pszRegStr=LocalAlloc(LMEM_ZEROINIT,cbData);
	if (!pszRegStr) {
		Result=FALSE;
		dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
		goto ReleaseResource;
	}
	dwRegError=RegQueryValueEx(
					hkResult,
					"Image Path",
					NULL,
					&dwRegType,
					pszRegStr,
					&cbData);
	if (dwRegError != ERROR_SUCCESS) {
		Result=FALSE;
		dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
		goto ReleaseResource;
	}

	// По считанному имени определяем полное имя библиотеки
	cbData=ExpandEnvironmentStrings(pszRegStr,NULL,0);
```

```
pszDllName=LocalAlloc(LMEM_ZEROINIT,cbData);
if (!pszDllName) {
	Result=FALSE;
	dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
	goto ReleaseResource;
}
cbData=ExpandEnvironmentStrings(pszRegStr,pszDllName,cbData);

// Проверяем наличие данной библиотеки в кэше
if (!CSPInCacheCheck(pszDllName,&hProvDll)) {
	// Если библиотека отсутствует, то загружаем ее...
	hProvDll=LoadLibrary(pszDllName);
	if (!hProvDll) {
		Result=FALSE;
		dwCryptError=NTE_PROVIDER_DLL_FAIL;
		goto ReleaseResource;
	}

	// проверяем цифровую подпись библиотеки
#if (_WIN32_WINNT >= 0x0500)
	hAdvapiDll=LoadLibrary("advapi32.dll");
	if (!hAdvapiDll) {
		Result=FALSE;
		dwCryptError=NTE_FAIL;
		goto ReleaseResource;
	}
	pfnCheckSignatureInFile=(CHECK_SIGNATURE_IN_FILE_FN)GetProcAddress(

	hAdvapiDll,"SystemFunction035");
	if (!pfnCheckSignatureInFile) {
		Result=FALSE;
		dwCryptError=NTE_FAIL;
		goto ReleaseResource;
	}

	pwszDllName=LocalAlloc(LMEM_ZEROINIT,lstrlen(pszDllName)*2+2);
	if (!MultiByteToWideChar(
				CP_ACP,
				MB_PRECOMPOSED,
				pszDllName,
				lstrlen(pszDllName),
				pwszDllName,
				lstrlen(pszDllName))) {
		Result=FALSE;
		dwCryptError=GetLastError();
		goto ReleaseResource;
	}

	if (!pfnCheckSignatureInFile(pwszDllName)) {
		Result=FALSE;
		dwCryptError=NTE_BAD_SIGNATURE;
		goto ReleaseResource;
	}

	// Освобождаем выделенные ресурсы
	LocalFree(pwszDllName); pwszDllName=NULL;
#else
	// По открытому ключу считываем цифровую подпись библиотеки
	dwRegError=RegQueryValueEx(
					hkResult,
					"Signature",
					NULL,
					&dwRegType,
					NULL,
					&cbData);
	if (dwRegError != ERROR_SUCCESS) {
		Result=FALSE;
		dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
		goto ReleaseResource;
	}
	pbSignature=LocalAlloc(LMEM_ZEROINIT,cbData);
	if (!pbSignature) {
		Result=FALSE;
		dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
		goto ReleaseResource;
	}
	dwRegError=RegQueryValueEx(
					hkResult,
					"Signature",
					NULL,
					&dwRegType,
```

```
                                        pbSignature,
                                        &cbData);
                if (dwRegError != ERROR_SUCCESS) {
                        Result=FALSE;
                        dwCryptError=NTE_PROV_TYPE_ENTRY_BAD;
                        goto ReleaseResource;
                }

                if (!CProvVerifyImage(pszDllName,pbSignature)) {
                        Result=FALSE;
                        dwCryptError=NTE_BAD_SIGNATURE;
                        goto ReleaseResource;
                }
        #endif

                // ... и помещаем дескриптор в кеш
                AddHandleToCSPCache(hProvDll);

        }
        // Освобождаем выделенные ресурсы
        LocalFree(pszRegStr); pszRegStr=NULL;
        LocalFree(pszDllName); pszDllName=NULL;
        RegCloseKey(hkResult); hkResult=0;

//см.комментарий 12
        // Выделяем память под структуру контекста криптопровайдера
        (PBYTE)pProvContext=LocalAlloc(LMEM_ZEROINIT,sizeof(PROV_CONTEXT));
        if (!pProvContext) {
                Result=FALSE;
                dwCryptError=ERROR_NOT_ENOUGH_MEMORY;
                goto ReleaseResource;
        }

        // Сохраняем в контексте адреса обязательных
        // функций криптопровадера
        for (dwFuncCount=0; ;dwFuncCount++)
        {
                pProvContext->CSPFuncPtr[dwFuncCount]=GetProcAddress(hProvDll,
                                FunctionNames[dwFuncCount]);
                if (!FunctionNames[dwFuncCount]) break;
                if (!pProvContext->CSPFuncPtr[dwFuncCount])
                {
                        Result=FALSE;
                        dwCryptError=NTE_PROVIDER_DLL_FAIL;
                        goto ReleaseResource;
                }
        }

//см.комментарий 13
#if (_WIN32_WINNT >= 0x0500)
        // Сохраняем в контексте адреса необязательных
        // функций криптопровадера
        // (начиная с Windows 2000 и Windows 98)
        for (dwFuncCount=0; ;dwFuncCount++) {
                (FARPROC)pProvContext->CSPOptionalFuncPtr[dwFuncCount]=
                        GetProcAddress(hProvDll,OptionalFunctionNames[dwFuncCount]);
                if (!OptionalFunctionNames[dwFuncCount]) break;
        }
#endif

        // Сохраняем в контексте дескриптор библиотеки
        // криптопровайдера
        pProvContext->hCSPDll=hProvDll;

        ///!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        //if (hProvDll) FreeLibrary(hProvDll);

//см.комментарий 14
// Заполняем структуру VTableProvStruc версии 3
        VTable.Version=3;
        VTable.FuncVerifyImage = CProvVerifyImage;
        VTable.FuncReturnhWnd=FuncReturnhWnd;
        VTable.dwProvType=dwProvType;
        VTable.pbContextInfo=NULL;
        VTable.cbContextInfo=0;
        VTable.pszProvName=pszProvName;
//--------------------------------------------------------------------
//см.комментарий 15
        // Вызываем функцию CPAcquireContext из выбранного криптопровайдера
        if (!(PFN_CP_ACQUIRE_CONTEXT)(pProvContext->CSPFuncPtr[0])(
                        &pProvContext->hCSP,
```

```
                                (CHAR*)pszContainer,
                                dwFlags,
                                &VTable)) {
                        Result=FALSE;
                        dwCryptError=GetLastError();
                        goto ReleaseResource;
                }
//--------------------------------------------------------------------
                // Если установлен флаг CRYPT_DELETEKEYSET,
                // то освобождаем ресурсы и возвращаемся из функции
                if (dwFlags == CRYPT_DELETEKEYSET) {
                        FreeLibrary(hProvDll);
                        LocalFree(pProvContext);
                        goto ReleaseResource;
                }

                // Сохраняем в контексте идентификатор
                pProvContext->ContextId=PROVIDER_CONTEXT;

//см.комментарий 16
#if (_WIN32_WINNT >= 0x0500)
                // Инициализируем в контексте значение счетчика и
                // дополнительного счетчика ссылок (начиная с Windows 2000 и Windows 98)
                pProvContext->dwContextUseCount=1;
                pProvContext->dwContextRefCount=1;
#else
                // Инициализируем в контексте счетчик ссылок
                pProvContext->dwContextUseCount=0;
                // Инициализируем в контексте критическую секцию
                InitializeCriticalSection(&pProvContext->CSPLock);
#endif

                // Возвращаем указатель на контекст
                *phProv=(HCRYPTPROV)pProvContext;


ReleaseResource:

                // Освобождаем выделенные ресурсы
                if (pszDllName) LocalFree(pszDllName);
                if (pwszDllName) LocalFree(pwszDllName);
                if (pszRegStr) LocalFree(pszRegStr);
                if (pszProvName) LocalFree(pszProvName);
                if (hkResult) RegCloseKey(hkResult);

                if (!Result) {
                        if (pProvContext) LocalFree(pProvContext);
                        if (hProvDll) FreeLibrary(hProvDll);
                }

                // Устанавливаем код ошибки
                SetLastError(dwCryptError);

                return Result;
}


void __cdecl main(int argc, char* argv[])
{
                HCRYPTPROV      hProv;
                BYTE            Data[100];
                DWORD           dwDataLen=0, dwCryptError=0;


                // Переменные определяющий имя криптопровайдера, тип
                // криптопровайдера и имя ключевого контейнера
                LPTSTR          pszProvider = PROV_NAME;
                DWORD           dwProvType =  PROV_TYPE;
                LPTSTR          pszContainer = NULL;

                PROV_ENUMALGS           AlgInfo;
                PROV_ENUMALGS_EX        AlgInfoEx;
                DWORD                   dwFlags=0;
                int dwCount;
   DWORD dwKeySizeInc;

                // Вызываем реконструированную функцию CryptAcquireContextA,
                // для получения дескриптора криптопровайдера

                // Пытаемся создать новый ключевой контейнер
                if (!OwnCryptAcquireContextA(&hProv,pszContainer,pszProvider,dwProvType,CRYPT_NEWKEYSET))
```

```
        {
                dwCryptError=GetLastError();
                if (dwCryptError == (unsigned)NTE_EXISTS) {
                        // Если контейнер уже существует, то
                        // открываем его
                        if (!OwnCryptAcquireContextA(&hProv,pszContainer,pszProvider,dwProvType,0)) {
                                printf("\nError: OwnCryptAcquireContextA=0x%X.\n",GetLastError());
                                return;
                        }
                }
                else {
                        printf("\nError: OwnCryptAcquireContextA=0x%X.\n",GetLastError());
                        return;
                }
        }

        // Тестируем полученный дескриптор

        // Считываем имя криптопровайдера
        dwDataLen=100;
        if (!CryptGetProvParam(hProv,PP_NAME,Data,&dwDataLen,0)) {
                printf("Error: CryptGetProvParam=0x%X.\n",GetLastError());
                return;
        }
        printf("Acquired CSP: %s\n",Data);

        // Считываем тип криптопровайдера
        dwDataLen=sizeof(DWORD);
        if (!CryptGetProvParam(hProv,PP_PROVTYPE,Data,&dwDataLen,0)) {
                printf("Error: CryptGetProvParam=0x%X.\n",GetLastError());
                return;
        }
        printf("Acquired CSP type: %d\n",*((PDWORD)Data));
//--------------------------------------------------------------------------
   // Перечисляем алгоритмы криптопровайдера
   for (dwCount = 0 ; ; dwCount++)
   {
                // Устанавливаем флаг CRYPT_FIRST при первом обращении в цикле.
     if (dwCount == 0)
                        dwFlags = CRYPT_FIRST;
     else
        dwFlags = 0;
     // Получаем информацию об алгоритмах
     dwDataLen = sizeof(PROV_ENUMALGS_EX);
     dwCryptError=ERROR_SUCCESS;
     // Запрашиваем структуру PROV_ENUMALGS_EX
     if (!CryptGetProvParam(hProv,PP_ENUMALGS_EX,(PBYTE)&AlgInfoEx,&dwDataLen,dwFlags))
     {
                        dwCryptError=GetLastError();
                        if (dwCryptError == ERROR_NO_MORE_ITEMS) break;
        if ((unsigned)dwCryptError == NTE_BAD_FLAGS || (unsigned)dwCryptError == NTE_BAD_TYPE)
        {
                                dwDataLen = sizeof(PROV_ENUMALGS);
           // Если криптопровайдер не поддерживает запрос, то запрашиваем структуру PROV_ENUMALGS
           if (!CryptGetProvParam(hProv,PP_ENUMALGS,(PBYTE)&AlgInfo,&dwDataLen,dwFlags))
           {
                                        dwCryptError=GetLastError();
              if (dwCryptError == ERROR_NO_MORE_ITEMS) break;
              printf("Error reading algorithm = %d",dwCryptError);
              return;
            }
            AlgInfoEx.aiAlgid=AlgInfo.aiAlgid;
            AlgInfoEx.dwDefaultLen=AlgInfo.dwBitLen;
            AlgInfoEx.dwMaxLen=0;
            AlgInfoEx.dwMinLen=0;
            lstrcpy(AlgInfoEx.szLongName,AlgInfo.szName);
        }
        else
                        {
                                printf("Error reading algorithm = %d",GetLastError());
                                return;
        }

                }
     // Распечатываем полученную информацию
     printf("%4.4xh|%s|%-4d |%-5d |%-4d |%s\n",
     AlgInfoEx.aiAlgid,
     szAlgClass[GET_ALG_CLASS(AlgInfoEx.aiAlgid) >> 13],
     AlgInfoEx.dwDefaultLen,
     AlgInfoEx.dwMaxLen,
     AlgInfoEx.dwMinLen,
```

```
      AlgInfoEx.szLongName);
            }

   // Запрашиваем информацию о значении шага инкремента ключа подписи
   dwDataLen=sizeof(DWORD);
   if (CryptGetProvParam(hProv,PP_SIG_KEYSIZE_INC,(PBYTE)&dwKeySizeInc,&dwDataLen,dwFlags))
   {
     if (dwKeySizeInc)
       printf("Number of bits for the increment length of signature key - %d\n",dwKeySizeInc);
   }
   // Запрашиваем информацию о значении шага инкремента ключа обмена
   dwDataLen=sizeof(DWORD);
            if (CryptGetProvParam(hProv,PP_KEYX_KEYSIZE_INC,(PBYTE)&dwKeySizeInc,&dwDataLen,dwFlags))
   {
     if (dwKeySizeInc)
       printf("Number of bits for the increment length of exchange key - %d\n",dwKeySizeInc);
   }
//--------------------------------------------------------------------------------
            // Считываем имя ключевого контейнера
            dwDataLen=100;
            if (!CryptGetProvParam(hProv,PP_CONTAINER,Data,&dwDataLen,0)) {
                        printf("Error: CryptGetProvParam=0x%X.\n",GetLastError());
                        return;
            }
            printf("Current context container: %s\n",Data);
//--------------------------------------------------------------------------------
   // Перечисляем контейнеры
   printf("Names of containers\n");
   for (dwCount = 0 ; ; dwCount++)
   {
                        // Устанавливаем флаг CRYPT_FIRST при первом обращении в цикле.
      if (dwCount == 0)
                                    dwFlags = CRYPT_FIRST;
      else
         dwFlags = 0;
      // Получаем информацию об алгоритмах
      dwDataLen = 100;
      dwCryptError=ERROR_SUCCESS;
      // Запрашиваем имя контейнера
      if (!CryptGetProvParam(hProv,PP_ENUMCONTAINERS,(PBYTE)Data,&dwDataLen,dwFlags))
      {
                                    dwCryptError=GetLastError();
                                    if (dwCryptError == ERROR_NO_MORE_ITEMS){
            break;
         }
         else{
                                                printf("Error reading container name = %d",GetLastError());
                                                return;
         }
                        }
      // Распечатываем полученную информацию
      printf("%d. %s\n",dwCount+1,Data);
            }

            // Освобождаем дескриптор криптопровайдера
            if (!CryptReleaseContext(hProv,0)) {
                        printf("Error: CryptReleaseContext=0x%X.\n",GetLastError());
                        return;
            }
```

Комментарии:

1. Значение константы _WIN32_WINNT определяет версию операционной системы, для которой компилируется программа. От значения константы зависит структура контекста криптопровайдера и некоторые шаги по его формированию.

2. Определяем тип для функции ***CPAcquireContext.*** Функция ***CPAcquireContext*** — одна из 23 обязательных функций, которые должен экспортировать криптопровайдер.

3. Определяем структуру контекста криптопровайдера. Указатель на эту структуру возвращается функцией ***Crypt-Acquire Context через*** параметр *phProv* и используется приложением в качестве дескриптора криптопровайдера. Ини-

циализация и использование переменных структуры будут рассмотрены в соответствующих местах программы. Как уже отмечалось, состав структуры контекста зависит от версии используемой операционной системы.

4. Определяем два массива имен функций. Массив *Function-Names* содержит имена 23 обязательных функций, которые должен экспортировать криптопровайдер. Массив *Optional-FunctionNames* содержит имена 2 необязательных функций.

5. Определяем базовые строки для рабочих ключей реестра.

szUserType содержит строку, на базе которой в программе формируется запрос к ключу реестра HKEY_CURRENT_USER\SOFTWARE\Microsoft\Cryptography\ Providers\TypeNNN.

Ключ содержит информацию о криптопровайдере, используемом по умолчанию текущим пользователем.. NNN — три десятичные цифры, определяющие тип криптопровайдера. Например, информация о криптопровайдере, используемом текущим пользователем, для типа 804, содержится в ключе HKEY_CURRENT_USER\ SOFTWA-RE\Microsoft\Cryptography\Providers\Type804.

szMachineType содержит строку, на базе которой в программе формируется запрос к ключу HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider Types\TypeNNN.

Ключ содержит информацию о криптопровайдере, который по умолчанию используется системой. NNN - три десятичные цифры, определяющие тип криптопровайдера. Например, информация о криптопровайдере, используемом текущим пользователем, для типа 805, содержится в ключе HKEY_CURRENT_ USER\SOFTWA-RE\Microsoft\Cryptography\Providers\Type805.

szProvider содержит строку, на базе которой в программе формируется запрос к ключу HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider\ ProviderName. Ключ содержит информацию о криптопровайдере с именем "Provider Name". Например, информацию о криптопровайдере "CESARIS DSTU 4145-2002(PB) and ECDH Cryptographic Provider" содержит ключ реестра HKEY_LO-CAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults\Provider\CESARIS DSTU4145 -2002(PB) and ECDH Cryptographic Provider.

6. Определяем функцию ***OwnCryptAcquireContextA.*** Она реконструирована из функции ***CryptAcquireContextA*** Windows 2000. Разумеется, реконструкция не является абсолютно точной. Имена функций и глобальных переменных соответствуют отладочной информации для данной версии операционной системы. В функцию также включены особенности, характерные для Windows NT 4.0. Как уже отмечалось, при компиляции версия операционной системы определяется константой _WIN32_WINNT. Поскольку наша задача — показать структуру контекста криптопровайдера и последовательность его формирования, а не полностью восстанавливать функцию, то процедуры ***CSPInCacheCheck, AddHandleToCSPCache, CheckSignatureInFile, FuncVerify-Image*** определены пустыми.

7. Если параметр *pszProvider* нулевой, то приложение запрашивает криптопровайдер, используемый по умолчанию, для типа, определенного параметром *dwProvType.* Поэтому на базе строки *szUserType* в переменной *pszRegStr* формируется имя ключа реестра (см. комментарий 5). Через вызов функции ***RegOpenKeyEx*** *с* параметрами HKEY_CURRENT_USER и *pszRegStr* делается попытка открыть ключ реестра с информацией о криптопровайдере, который назначен по умолчанию, для текущего пользователя.

8. Запрашиваемый ключ реестра не найден, следовательно, явные параметры, заданные для текущего пользователя, отсутствуют, и вызов будет обработан с использованием общесистемных параметров. На базе строки *szMachineType,* в переменной *pszRegStr* формируется имя ключа реестра (см. комментарий 5). Через вызов функции ***RegOpenKeyEx*** с параметрами HKEY_LOCAL_MACHINE *npszRegStr* открываем ключ реестра с информацией о криптопровайдере по умолчанию для системы в целом.

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

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

11. Набор параметров, которые содержит открытый ключ, зависит от используемой операционной системы. Но два наиболее важных параметра присутствуют всегда. Первый параметр (имя "Image Path", тип REG_SZ) — строка с именем библиотеки, в которой реализован данный криптопровайдер. Второй параметр (имя "Type", тип REG_DWORD) - значение типа криптопровайдера. Последнее сохраняем в переменной *dwQueryProvType,* этот параметр необходим для сравнения с запрошенным типом криптопровайдера *dwProvType.* Имя библиотеки криптопровайдера сохраняем по адресу *pszRegStr.* На основании этого параметра производим вызов функции ***LoadLibrary*** и загружаем библиотеку криптопровайдера в память процесса.

12. Получив необходимую информацию о криптопровайдере, начинаем формировать структуру контекста криптопровайдера. На основании массива имен обязательных функций криптопровайдера *FunctionNames* формируем массив указателей на обязательные функции криптопровайдера *pProvContext->CSPFuncPtr.* Обратите внимание, что если какая-либо из функций отсутствует в криптопровайдере, то инициализация прерывается и возвращается ошибка. Именно поэтому данный набор функций является обязательным.

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

14. *И*нициализируем структуру *Viable. И*нициализация производится в соответствии с версией 3 структуры *VTableProvStruc.* Необходимо отметить, что версия 3 используется, только начиная с Windows 2000.

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

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

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

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

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

В функции ***CryptReleaseContext*** счетчики обращений проверяются, и если они находятся не в исходном состоянии, возвращается ошибка ERROR_BUSY (контекст занят). Кроме того, в Windows 2000 функция ***CryptReleaseContext*** декрементирует *c4eT4HKpProvContext->dwContextRef Count.* Поэтому, чтобы освободить контекст, необходимо вызвать функцию ***CryptReleaseContext*** *столько* же раз, сколько вызывается функция ***CryptContextAddRef.***

Приложение 2. Пример использования CryptoAPI 1.0 для реализации схемы цифровой подписи

```
#include <windows.h>
#include <stdio.h>

#define PROV_G34310_RSA_TYPE            804 // CESARIS GOST 34.310-95 and RSA Cryptographic Provider
#define PROV_D4145PB_RSA_TYPE           805 // CESARIS DSTU 4145-2002(PB) and RSA Cryptographic Provider

#define ALG_SID_G34311          82
#define CALG_G34311             (ALG_CLASS_HASH|ALG_TYPE_ANY|ALG_SID_G34311)


//см.комментарий 1
//данный пример использует ГОСТ 34.310-95
#define PROV_TYPE                       PROV_G34310_RSA_TYPE

// Определение идентификатора, используемого алгоритма хеширования
#define HASH_ALG                        CALG_G34311
// Определение длины ключа подписи и флагов
#define SIGN_KEY_LEN                    512
#define SIGN_KEY_FLAGS                  0
// Определение строки для подписи и строк описания
#define SIGN_MSG                        "The data that is to be hashed and signed."
#define SIGN_DESCRIPTION                NULL
#define VERIFY_DESCRIPTION              NULL
// Определение флагов подписи и проверки
#define SIGN_FLAGS                      0
#define VERIFY_FLAGS                    0
// Определение имен файлов сообщения и цифровой подписи
#define MESSAGE_FILE                    "Message.txt"
#define SIGNATURE_FILE                  "Message.sig"
// Определение имени ключевого контейнера отправителя
#define ORIGINATOR_CONTEXT              NULL
#define BUFFER_SIZE                     160


// Определение прототипа функции I_CryptGetDefaultCryptProv
// из библиотеки Crypt32.dll
typedef HCRYPTPROV (WINAPI *PFN_I_CRYPT_GET_DEFAULT_CRYPT_PROV)(ALG_ID AlgId);

PFN_I_CRYPT_GET_DEFAULT_CRYPT_PROV pfnCryptGetDefaultCryptProv = NULL;

BOOL
CryptAcquireContextEx(
        HCRYPTPROV      *phProv,
        LPTSTR          pszContainer,
        LPTSTR          pszProvider,
        DWORD           dwProvType,
        DWORD           dwFlags,
        DWORD           dwKeySpec,
        DWORD           KeyFlags
        );

BOOL
WriteFileBlob(
        FILE            *hFile,
        PDATA_BLOB      pDataBlob
        );

BOOL
ReadFileBlob(
        FILE            *hFile,
        PDATA_BLOB      pDataBlob
        );


void main(void) {
        LPTSTR pszProvName = NULL;
        DWORD cbProvName = 0;
        // Определение и инициализация переменных
        BOOL            Result=TRUE;

        FILE            *hMessage=NULL;
        FILE            *hSignature=NULL;
        FILE            *hCert=NULL;
```

```
        fpos_t          fSavePos=0, fSetPos=0;

        HCRYPTPROV      hCryptProv=0;
        HCRYPTKEY       hSigKey=0;
        HCRYPTHASH      hHash=0;
        DWORD           dwKeyFlags=(SIGN_KEY_LEN << 16)|SIGN_KEY_FLAGS;
        ALG_ID          aiHashAlg=HASH_ALG;

        DATA_BLOB       PubKeyBlob={0,NULL};
        DATA_BLOB       Signature={0,NULL};
        DATA_BLOB       SignMsg={strlen((char*)SIGN_MSG)+1,(PBYTE)SIGN_MSG};

    BYTE        pbBuffer[BUFFER_SIZE];
    DWORD       dwCount;


// Первый этап
// Отправитель, генерирует (или открывает) ключ подписи,
// вырабатывает цифровую подпись файла сообщения и
// отправляет ее получателю.

        printf("First phase start.\n");

        // Открываем файл с сообщением
    if((hMessage=fopen(MESSAGE_FILE,"rb"))==NULL) {
            if((hMessage=fopen(MESSAGE_FILE,"wb"))==NULL) {
                printf("Error: Opening %s file.\n",MESSAGE_FILE);
                Result=FALSE;
                goto FirstPhaseReleaseResource;
            }
            fwrite(SignMsg.pbData, SignMsg.cbData, 1, hMessage);
            if(ferror(hMessage)) {
                printf("Error: Writing data to file.\n");
                Result=FALSE;
                goto FirstPhaseReleaseResource;
            }
            fclose(hMessage); hMessage=0;
            if((hMessage=fopen(MESSAGE_FILE,"rb"))==NULL) {
                printf("Error: Opening %s file.\n",MESSAGE_FILE);
                Result=FALSE;
                goto FirstPhaseReleaseResource;
            }
        }
        printf("The message file, %s, is open.\n",MESSAGE_FILE);

        // Открываем файл для сохранения цифровой подписи
        if((hSignature=fopen(SIGNATURE_FILE,"wb"))==NULL) {
            printf("Error: Opening %s file.\n",SIGNATURE_FILE);
            Result=FALSE;
            goto FirstPhaseReleaseResource;
        }
        printf("The signature file, %s, is open.\n",SIGNATURE_FILE);

        // получаем размер имени провайдера по умолчанию
        if (!CryptGetDefaultProvider(
            PROV_TYPE,
            NULL,
            CRYPT_MACHINE_DEFAULT,
            NULL,
            &cbProvName)) {
                Result=FALSE;
                goto FirstPhaseReleaseResource;
        }

        // Выделяем память для открытого ключа подписи
        pszProvName = LocalAlloc(LMEM_ZEROINIT,cbProvName);
        if(!pszProvName) {
            printf("Error: Out of memory.\n");
            Result=FALSE;
            goto FirstPhaseReleaseResource;
        }

//см.комментарий 2
        // получаем размер имени провайдера по умолчанию
        if (!CryptGetDefaultProvider(
            PROV_TYPE,
            NULL,
            CRYPT_MACHINE_DEFAULT,
            pszProvName,
            &cbProvName)) {
                Result=FALSE;
```

```
                goto FirstPhaseReleaseResource;
            }

        printf("Default CSP name acquired.\n");

//см.комментарий 3
        if (!CryptAcquireContextEx(
                &hCryptProv,
                ORIGINATOR_CONTEXT,
                pszProvName,
                PROV_TYPE,
                0,
                AT_SIGNATURE,
                dwKeyFlags)) {
            Result=FALSE;
            goto FirstPhaseReleaseResource;
        }
        printf("Originator CSP context acquired.\n");

        // Получаем дескриптора ключа подписи
        if(!CryptGetUserKey(
                hCryptProv,
                AT_SIGNATURE,
                &hSigKey)) {
            printf("Error: CryptGetUserKey=0x%X.\n",GetLastError());
            Result=FALSE;
            goto FirstPhaseReleaseResource;
        }
    printf("The originator signature key has been acquired.\n");

//см.комментарий 4
        // Определяем необходимый размера памяти для
        // экспорт открытого ключа пары подписи.
        if(!CryptExportKey(
                hSigKey,
                0,
                PUBLICKEYBLOB,
                0,
                NULL,
                &PubKeyBlob.cbData)) {
            printf("Error: CryptExportKey=0x%X.\n",GetLastError());
            Result=FALSE;
            goto FirstPhaseReleaseResource;
        }
    printf("The originator public key blob is %d bytes long.\n",PubKeyBlob.cbData);
        // Выделяем память для открытого ключа подписи
        PubKeyBlob.pbData=LocalAlloc(LMEM_ZEROINIT,PubKeyBlob.cbData);
        if(!PubKeyBlob.pbData) {
            printf("Error: Out of memory.\n");
            Result=FALSE;
            goto FirstPhaseReleaseResource;
        }
    printf("Memory has been allocated for the blob.\n");
        // Экспортируем открытого ключа пары подписи
        if(!CryptExportKey(
                hSigKey,
                0,
                PUBLICKEYBLOB,
                0,
                PubKeyBlob.pbData,
                &PubKeyBlob.cbData)) {
            printf("Error: CryptExportKey=0x%X.\n",GetLastError());
            Result=FALSE;
            goto FirstPhaseReleaseResource;
        }
        printf("Contents have been written to the originator public key blob.\n");

//см.комментарий 5
        // Создаем объект хеш-функции
        if(!CryptCreateHash(
                hCryptProv,
                HASH_ALG,
                0,
                0,
                &hHash)) {
            printf("Error: CryptCreateHash=0x%X.\n",GetLastError());
            Result=FALSE;
            goto FirstPhaseReleaseResource;
        }
        printf("Hash object created.\n");
```

```
        do {
                // Читаем из файла сообщения данные размером BUFFER_SIZE байт
                dwCount=fread(pbBuffer,1,BUFFER_SIZE,hMessage);
                if(ferror(hMessage)) {
             printf("Error: Reading data from file.\n");
                        Result=FALSE;
                        goto FirstPhaseReleaseResource;
                }

                // Хешируем буфер данных
                if(!CryptHashData(
                                hHash,
                                pbBuffer,
                                dwCount,
                                0)) {
                        printf("Error: CryptHashData=0x%X.\n",GetLastError());
                        Result=FALSE;
                        goto FirstPhaseReleaseResource;
                }

        } while(!feof(hMessage));
   printf("The message file data has been hashed.\n");

//см.комментарий 6
        // Определяем неоходимый размер памяти для  буфера цифровой подписи
        if(!CryptSignHash(
                        hHash,
                        AT_SIGNATURE,
                        SIGN_DESCRIPTION,
                        SIGN_FLAGS,
                        NULL,
                        &Signature.cbData)) {
                printf("Error: CryptSignHash=0x%X.\n",GetLastError());
                Result=FALSE;
                goto FirstPhaseReleaseResource;
        }
   printf("The hash signature is %d bytes long.\n",Signature.cbData);
        // Выделяем память под буфер цифровой подписи
        Signature.pbData=LocalAlloc(LMEM_ZEROINIT,Signature.cbData);
        if(!Signature.pbData) {
                printf("Error: Out of memory.\n");
                Result=FALSE;
                goto FirstPhaseReleaseResource;
        }
        printf("Memory allocated for the signature.\n");
        // Вычисляем цифровую подпись и записываем значение
        // подписи в выделенный буфер
        if(!CryptSignHash(
                        hHash,
                        AT_SIGNATURE,
                        SIGN_DESCRIPTION,
                        SIGN_FLAGS,
                        Signature.pbData,
                        &Signature.cbData)) {
                printf("Error: CryptSignHash=0x%X.\n",GetLastError());
                Result=FALSE;
                goto FirstPhaseReleaseResource;
        }
        printf("Hash signature have been written to the buffer.\n");

//см.комментарий 7
        // Сохраняем в файле алгоритм хеш-функции
        fwrite(&aiHashAlg,1,sizeof(ALG_ID),hSignature);
        if(ferror(hSignature)) {
           printf("Error: Writing data to file.\n");
                Result=FALSE;
           goto FirstPhaseReleaseResource;
        }
        printf("The hash algirithm has been written to the file.\n");

        // Сохраняем в файле открытый ключ подписи отправителя
        if (!WriteFileBlob(hSignature,&PubKeyBlob)) {
                printf("Error: Writing data to file.\n");
                Result=FALSE;
                goto FirstPhaseReleaseResource;
        }
        printf("The originator public key blob has been written to the file.\n");

        // Сохраняем в файле значение цифровой подписи
        if (!WriteFileBlob(hSignature,&Signature)) {
                printf("Error: Writing data to file.\n");
```

```
                Result=FALSE;
                goto FirstPhaseReleaseResource;
        }
        printf("The message file signature has been written to the file.\n");

FirstPhaseReleaseResource:
        // Освобождаем открытые на первом этапе ресурсы и обнуляем переменный

        // Уничтожаем объект хеш-функции
        if (hHash){
                CryptDestroyHash(hHash);
                hHash=0;
        }
        // Уничтожаем ключ подписи
        if (hSigKey){
                CryptDestroyKey(hSigKey);
                hSigKey=0;
        }
        // Освобождаем дескриптор криптопровайдера
        if (hCryptProv){
                CryptReleaseContext(hCryptProv, 0);
                hCryptProv=0;
        }
        // Освобождаем память буфера цифровой подписи
        if (Signature.pbData){
                LocalFree(Signature.pbData);
                Signature.pbData=NULL;
        }
        // Освобождаем память блоба открытого ключа
        if (PubKeyBlob.pbData){
                LocalFree(PubKeyBlob.pbData);
                PubKeyBlob.pbData=NULL;
        }
        // Закрываем дескрипторы открытых файлов
        if (hMessage){
                fclose(hMessage);
                hMessage=0;
        }
        if (hSignature){
                fclose(hSignature);
                hSignature=0;
        }
        if (!Result) return;

        printf("The first phase is completed.\n\n");



// Второй этап
// Получатель, используя открытый ключ отправителя, проверяет
// цифровую подпись сообщения.

        printf("Second phase start.\n");

//см.комментарий 8
        // Открываем файл с сообщением
        if((hMessage=fopen(MESSAGE_FILE,"rb"))==NULL) {
                printf("Error: Opening %s file.\n",MESSAGE_FILE);
                Result=FALSE;
                goto SecondPhaseReleaseResource;
        }
        printf("The message file, %s, is open.\n",MESSAGE_FILE);

        // Открываем файл с цифрой подписью
        if((hSignature=fopen(SIGNATURE_FILE,"rb"))==NULL) {
                printf("Error: Opening %s file.\n",SIGNATURE_FILE);
                Result=FALSE;
                goto SecondPhaseReleaseResource;
        }
        printf("The signature file, %s, is open.\n",SIGNATURE_FILE);

        // Считываем из файла алгоритм хеш-функции
        fread(&aiHashAlg,1,sizeof(DWORD),hSignature);
        if(ferror(hSignature) || feof(hSignature)) {
                printf("Error: Reading data from file.\n");
                Result=FALSE;
                goto SecondPhaseReleaseResource;
        }
        printf("The hash algorithm has been read from the file.\n");
```

```
		// Считываем из файла открытый ключ отправителя
		if (!ReadFileBlob(hSignature,&PubKeyBlob)) {
			printf("Error: Reading data from file.\n");
			Result=FALSE;
			goto SecondPhaseReleaseResource;
		}
		printf("The originator public key blob has been read from the file.\n");

		// Считываем из файла значение цифровой подписи
		if (!ReadFileBlob(hSignature,&Signature)) {
			printf("Error: Reading data from file.\n");
			Result=FALSE;
			goto SecondPhaseReleaseResource;
		}
		printf("The message file siganture blob has been read from the file.\n");

		// Открываем дескриптор ключевого контекста получателя
		if (!CryptAcquireContext(
			&hCryptProv,
			ORIGINATOR_CONTEXT,
			pszProvName,
			PROV_TYPE,
			0)) {
				Result=FALSE;
				goto FirstPhaseReleaseResource;
			}
		printf("Recipient verify CSP context acquired.\n");

//см.комментарий 9
		// Импортируем открытый ключ отправителя криптопровайдер
		if(!CryptImportKey(
				hCryptProv,
				PubKeyBlob.pbData,
				PubKeyBlob.cbData,
				0,
				0,
				&hSigKey)) {
			printf("Error: CryptImportKey=0x%X.\n",GetLastError());
			Result=FALSE;
			goto SecondPhaseReleaseResource;
		}
   printf("The originator public key has been imported.\n");

//см.комментарий 10
		// Создаем объект хеш-функции
		if(!CryptCreateHash(
				hCryptProv,
				aiHashAlg,
				0,
				0,
				&hHash)) {
			printf("Error: CryptCreateHash=0x%X.\n",GetLastError());
			Result=FALSE;
			goto SecondPhaseReleaseResource;
		}
   printf("The hash object has been created.\n");

		do {
			// Читаем из файла отправителя данные размером BUFFER_SIZE байт
			dwCount=fread(pbBuffer,1,BUFFER_SIZE,hMessage);
			if(ferror(hMessage)) {
		      printf("Error: Reading data from file.\n");
				Result=FALSE;
				goto SecondPhaseReleaseResource;
			}

			// Хешируем буфер данных
			if(!CryptHashData(
					hHash,
					pbBuffer,
					dwCount,
					0)) {
				printf("Error: CryptHashData=0x%X.\n",GetLastError());
				Result=FALSE;
				goto SecondPhaseReleaseResource;
			}

		} while(!feof(hMessage));
		printf("The message file data has been hashed.\n");
```

```
//см.комментарий 11
        // Проверяем цифровую подпись
        if(!CryptVerifySignature(
                        hHash,
                        Signature.pbData,
                        Signature.cbData,
                        hSigKey,
                        VERIFY_DESCRIPTION,
                        VERIFY_FLAGS)) {
                if(GetLastError() == NTE_BAD_SIGNATURE) {
                        printf("File signature failed to validate!\n");
                }
                else {
                        printf("Error: CryptVerifySignature=0x%X.\n",GetLastError());
                        Result=FALSE;
                }
        }
        else {
                printf("File signature validated Ok!\n");
        }

SecondPhaseReleaseResource:
        // Освобождаем открытые на втором этапе ресурсы и обнуляем переменный

        // Уничтожаем объект хеш-функции
        if (hHash) CryptDestroyHash(hHash);

        if (hSigKey) CryptDestroyKey(hSigKey);
        // Освобождаем дескриптор криптопровайдера
        if (hCryptProv) CryptReleaseContext(hCryptProv, 0);

        // Освобождаем память имени провайдера
        if (pszProvName) LocalFree(pszProvName);
        // Освобождаем память буфера подписи
        if (Signature.pbData) LocalFree(Signature.pbData);
        // Освобождаем память память блоба открытого ключа отправителя
        if (PubKeyBlob.pbData) LocalFree(PubKeyBlob.pbData);

        // Закрываем дескрипторы открытых файлов
        if (hMessage) fclose(hMessage);
        if (hSignature) fclose(hSignature);

        if (Result) printf("The second phase is completed.\n\n");

        return;

}


// Функция CryptAcquireContextEx открывает контекст криптопровайдера,
// соответсвующий указанным параметрам. Если контекст отсутствует, то функция
// создает новый контекст. Функция также проверяет на соответствие параметры
// созданных в контексте ключевых пар и если необходимая пара отсутствует, то
// создает новую.
BOOL
CryptAcquireContextEx(
        HCRYPTPROV      *phProv,
        LPTSTR          pszContainer,
        LPTSTR          pszProvider,
        DWORD           dwProvType,
        DWORD           dwFlags,
        DWORD           dwKeySpec,
        DWORD           dwKeyFlags
        )
{
        BOOL            Result=TRUE;
        DWORD           dwError;
        HCRYPTPROV      hCryptProv=0;
        HCRYPTKEY       hKey = 0;
        DWORD           dwDataLen=0, dwKeyLen=0;


        // Пытаемся открыть ключевой контейнер
        if (!CryptAcquireContext(
                        &hCryptProv,
                        pszContainer,
                        pszProvider,
                        dwProvType,
                        dwFlags)) {
                dwError=GetLastError();
                if (dwError!= NTE_BAD_KEYSET) {
```

```
                printf("Error: CryptAcquireContext=0x%X.\n",dwError);
                Result=FALSE;
                goto ReleaseResource;
            }
            else {
                // Если контейнер не существует, создаем новый
                if (!CryptAcquireContext(
                        &hCryptProv,
                        pszContainer,
                        pszProvider,
                        dwProvType,
                        dwFlags | CRYPT_NEWKEYSET)) {
                    printf("Error: CryptAcquireContext=0x%X.\n",GetLastError());
                    Result=FALSE;
                    goto ReleaseResource;
                }
            }
        }

        if (dwKeySpec) {
            // Пытаемся открыть ключевую пару, заданную параметром dwKeySpec
            if (!CryptGetUserKey(hCryptProv,dwKeySpec,&hKey)) {
                // Если ключевая пара в контейнере отсутствует, то создаем ее
                if (!CryptGenKey(hCryptProv,dwKeySpec,dwKeyFlags,&hKey)) {
                    printf("Error: CryptGenKey=0x%X.\n",GetLastError());
                    Result=FALSE;
                }
                goto ReleaseResource;
            }
            // Если удалось открыть существующую ключевую пара, то
            // сравниваем длину ключа с определенной в старшем слове
            // параметра dwKeyFlags
            dwDataLen=sizeof(DWORD);
            if (!CryptGetKeyParam(hKey,KP_KEYLEN,(PBYTE)&dwKeyLen,&dwDataLen,0)) {
                printf("Error: CryptGetKeyParam=0x%X.\n",GetLastError());
                Result=FALSE;
                goto ReleaseResource;
            }
            if (dwKeyLen != (dwKeyFlags & KEY_LENGTH_MASK) >> 16) {
                CryptDestroyKey(hKey); hKey=0;
                // Если длина ключа не соответствует запрашиваемой, то
                // создаем новую ключевую пару
                if (!CryptGenKey(hCryptProv,dwKeySpec,dwKeyFlags,&hKey)) {
                    printf("Error: CryptGenKey=0x%X.\n",GetLastError());
                    Result=FALSE;
                    goto ReleaseResource;
                }
            }
        }
ReleaseResource:

        // Освобождаем дескриптор ключа
        if (hKey) CryptDestroyKey(hKey);

        if (!Result) {
            // Освобождаем ключевой контекст
            if (hCryptProv) CryptReleaseContext(hCryptProv,0); hCryptProv=0;
        }

        // Возвращаем дескриптор ключевого контейнера
        *phProv=hCryptProv;

        return Result;
}

// Функция сохраняет в файле, описанном парметром hFile,
// структуру DATA_BLOB, описанную параметром pDataBlob.
BOOL
WriteFileBlob(
        FILE            *hFile,
        PDATA_BLOB      pDataBlob
        )
{
        // Сохраняем в файле длину блока данных
        fwrite(&pDataBlob->cbData, 1, sizeof(DWORD), hFile);
        if(ferror(hFile)) {
            return FALSE;
        }
        // Сохраняем в файле блок данных длиной pDataBlob->cbData
        if (pDataBlob->cbData) {
            fwrite(pDataBlob->pbData,1,pDataBlob->cbData,hFile);
```

```
                    if(ferror(hFile)) {
                        return FALSE;
                    }
            }

            return TRUE;
}

// Функция считывает из файла, описанного парметром hFile,
// структуру DATA_BLOB, сохраняя ее в по адресу pDataBlob.
BOOL
ReadFileBlob(
            FILE                *hFile,
            PDATA_BLOB      pDataBlob
            )
{
            BOOL    Result=TRUE;

            pDataBlob->cbData=0;
            pDataBlob->pbData=NULL;

            // Считаем длину блока данных
            fread(&pDataBlob->cbData,sizeof(DWORD),1,hFile);
            if(ferror(hFile) || feof(hFile)) {
                Result=FALSE;
                goto ReleaseResource;
            }
            // Если длина блока не нулевая, то выделяем блок памяти
            if (pDataBlob->cbData)
                        pDataBlob->pbData=LocalAlloc(LMEM_ZEROINIT,pDataBlob->cbData);
            else goto ReleaseResource;
            if (!pDataBlob->pbData) {
                        printf("Error: Out of memory.\n");
                        Result=FALSE;
                        goto ReleaseResource;
            }
            // Считываем блок данных в выделенный буфер
            fread(pDataBlob->pbData,1,pDataBlob->cbData,hFile);
            if(ferror(hFile) || feof(hFile)) {
                        Result=FALSE;
                        goto ReleaseResource;
            }

ReleaseResource:
            if (!Result)
                        // Освобождаем выделенный буфер
                        if (pDataBlob->pbData) LocalFree(pDataBlob->pbData);

            return Result;
}
```

Теперь прокомментируем показанный фрагмент.

1. Определение констант, используемых в программе. Помимо криптографических параметров, здесь определены следующие константы:

* SIGN_MSG — строка с текстом сообщения. Эта строка используется, если отправитель заранее е создал подписываемый файл;
* MESSAGE_FILE — строка, содержащая имя файла с подписываемым сообщением;
* SIGNATURE_FILE — строка, содержащая имя файла, в котором сохраняются подпись файла и другие параметры криптографического сообщения.

2. Получаем имя по умолчанию для указанного типа криптопровайдера. В данном примере тип – 804.

3. Генерируем ключевую пару подписи. Алгоритм ключевой пары зависит от используемого криптопровайдера, который определяется константами PROV_NAME и PROV_TYPE. Длина генерируемого ключа определяется константой SIGN_KEY_LEN.

4. Экспортируем открытый ключ ЭЦП отправителя. Для этого используем функцию ***CryptExportKey*** с параметром PUBLICKEYBLOB. Блоб открытого ключа сохраняем в структуре PubKeyBlob типа DATA_BLOB.

5. Создаем объект хеш-функции и хешируем содержимое файла MESSAGE_FILE. Считывание и хеширование производится блоками размером BUFFER_SIZE байт. Алгоритм, создаваемой хеш-функции, определяется константой HASH_ALG.

6. Вычисляем цифровую подпись для объекта хеш-функции. Для этого используется функция ***CryptSignHash***. Получение значения цифровой подписи производится так же, как и при получении других данных неизвестного размера. В первый раз вызываем функцию для определения необходимого размера буфера, при этом указатель на буфер равен NULL. Затем выделяем необходимую память и снова вызываем функцию ***CryptSignHash*** с указателем на буфер для копирования цифровой подписи. Остановимся подробней на некоторых параметрах функции и значениях, которые передаются в процедуру:

- dwKeySpec — параметр идентифицирует секретный ключ, на котором будет производиться подпись (AT_SIGNATURE или AT_KEYEXCHANGE);
- sDescription — параметр определяется, как описатель подписываемого сообщения. Функция построена так, что если при вызове функции CryptSignHash указан параметр sDescription, то его необходимо передать и в функцию проверки ***CryptVerifySignature*** иначе подпись не пройдет проверку. В нашем примере строка, передаваемая в качестве параметра sDescription в функцию подписи, определяется константой SIGN_DESCRIPTION;
- dwFlags — в настоящее время определено одно значение флагов CRYPT_NOHASHOID. Флаг относится только к криптопровайдерам, реализующим подпись по алгоритму RSA.

7. Сохраняем в файле все необходимые для проверки целостности сообщения параметры: алгоритм, используемого хэша, блоб открытого ключа отправителя, значение цифровой подписи.

8. Получатель начинает свой этап с открытия файлов сообщения и цифровой подписи и считывания алгоритма хеш-функции, блоба открытого ключа отправителя, значения цифровой подписи.

9. Для использования открытого ключа отправителя необходимо импортировать его к ключевой контейнер получателя. Используем для этого функцию ***CryptImportKey***. Таким образом, в переменной hSigKey сохраняется дескриптор открытого ключа проверки подписи.

10. В соответствии с идентификатором алгоритма хэша, полученным от отправителя, создаем объект хеш-функции. Затем производим хеширование содержания файла сообщения. Так же как и на первом этапе, считывание и хеширование выполняют блоки размером BUFFER_SIZE байт.

11. Получатель проверяет цифровую подпись сообщения. Для этого используется функция ***CryptVerifySignature***. Применение параметров sDescription и dwFlags рассмотрены в комментарии 6. Значения этих параметров, которые передаются в функцию, определяются константами -VERIFY_DESCRIPTION и VERIFY_FLAGS соответственно.

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