МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
Одеський Національний Університет імені І. І. Мечникова

Шпінарева Ірина Михайлівна

Геренко Ольга Андріївна

Методичний посібник з курсу «Захист інформації в комп'ютерних системах»

для студентів 4 курсу

спеціальності «Комп'ютерні системи і мережі»

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Развитие информационных технологий, их проникновение во все сферы человеческой деятельности приводит к тому, что проблемы информационной безопасности с каждым годом становятся всё более и более актуальными – и одновременно более сложными.

На практике информационная безопасность обычно рассматривается как совокупность следующих трёх базовых свойств защищаемой информации: конфиденциальность, целостность и доступность, гарантирующая беспрепятственный доступ к защищаемой информации для законных пользователей.

Многие операционные системы предлагают программистам криптографические библиотеки, позволяющие создавать защищенные приложения. Например, ОС Linux предлагает библиотеку Openssl. Microsoft еще в 1996 г начиная с Windows95 OSR2 и WindowsNT SP3 включает в свои операционные системы встроенные средства шифрования, доступные прикладному программисту через библиотеку Microsoft CryptoAPI. Каждая из этих библиотек имеют достоинства и недостатки. В этом методическом издании мы познакомимся с библиотекой CryptoAPI.

В ОС Windows 2000 и Windows XP функции CryptoAPI содержатся в модулях crypt32.dll и advapi32.dll

Начиная с Windows Vista и Windows Server 2008 поддерживается библиотека Cryptography API: Next Generation (CNG), призванная в будущем заменить CryptoAPI.

Создание лабораторных работ по курсу «Защита информации в компьютерной системе» ставит своей целью:

- предоставление студентам необходимой информации по выполнению лабораторных работ;
- подготовка источников информации с программным обеспечением для установки и настройки всего необходимого для выполнения лабораторных работ;
- предоставление студентам примерных алгоритмов выполнения лабораторных работ.

В результате выполнения данных лабораторных работ студенты познакомятся с основами использования Microsoft CryptoAPI и получат возможность создавать несложные приложения, которые будут обращаться к CryptoAPI.

Лабораторная работа 1.
Криптографическая защита данных с помощью MICROSOFT CRYPTOAPI

Цель работы:

Изучение основных функций библиотеки CRYPTOAPI необходимых для защиты конфиденциальности документа. Ознакомления с функциями шифрования и дешифрования документов, генерацией ключей и работа с сертификатами X509.

Результат:

Создания программного приложения для защиты документов с помощью шифрования.

Задание для самостоятельного выполнения

Перед выполнением задания студент, используя программу openssl, создает на своем компьютере локальный центр сертификации. Затем создает ключевую пару RSA <KO,KC>. На основании открытого ключа KO создает запрос на сертификацию и передает его в локальный центр сертификации. Получает из локального центра сертификации сертификат X509 своего открытого ключа KO и сертификат X509 открытого ключа центра сертификации. Сертификат X509 своего открытого ключа необходимо разместить в хранилище WINDOWS.

Задача

Используя криптографические интерфейсы MS CryptoAPI 1.0 (2.0), создать приложение, позволяющее шифровать и расшифровывать файлы по следующей схеме.

Чтобы зашифровать файл надо:

- Создать сеансовый ключ K_S симметричного алгоритма (алгоритм определяется вариантом задания).
- Зашифровать файл на ключе K_S.
- Зашифровать сеансовый ключ открытым ключом получателя шифрованного файла и добавить к зашифрованному файлу зашифрованный сеансовый ключ. Открытый ключ получателя извлекается из сертификата получателя.

Чтобы расшифровать файл надо:

- Извлечь из файла зашифрованный сеансовый ключ KS и расшифровать его, используя секретный ключ получателя. Секретный ключ получателя извлекается из сертификата получателя.
- Расшифровать файл сеансовым ключом KS.

Варианты заданий для работы №1

№ Варианта	Алгоритм и режим шифрования
1	RC2 CBC
2	RC4
3	DES CBC
4	Two keys 3DES CBC
5	Three keys 3DES CBC
6	CAST CBC
7	AES CBC
8	RC2 OFB
9	RC4
10	DES OFB
11	Two keys 3DES OFB
12	Three keys 3DES OFB
13	CAST OFB
14	AES OFB
15	RC2 OFB
16	RC4
17	DES OFB
18	Two keys 3DES OFB
19	Three keys 3DES OFB
20	CAST OFB
21	AES OFB

Вариант задания определяется так: $(n_1n_2+11) \bmod 27 +1$, где n_1n_2– две последние цифры номера зачетки.

Методические указания к первой части лабораторной работы

Концепция CryptoAPI подразумевает сокрытие от программиста всех тонкостей процесса шифрования данных.

В CryptoAPI можно выделить пять основных функциональных областей (рис. 1.1). Функции, входящие в каждую из областей, как правило, содержат в имени соответствующие ключевые слова.

- Базовые криптографические функции (Crypt). В эту группу входят функции для подключения к криптопровайдеру, генерации и хранения ключей, обмена ключами, управления параметрами ключа.
- Функции кодирования/декодирования (Crypt) обеспечивают доступ к алгоритмам шифрования/дешифрования и хеширования.

- Функции работы с хранилищем сертификатов (Store) предназначены для управления наборами цифровых сертификатов, используемых при цифровой подписи и шифровании с открытым ключом.
- Упрощенные функции для работы с сообщениями (Message) позволяют шифровать/дешифровать сообщения и блоки данных, подписывать их и проверять цифровую подпись.
- Низкоуровневые функции для работы с сообщениями (Msg) позволяют более гибко выполнить ту же работу, что и упрощенные функции, но требуют для этого больших усилий со стороны программиста.

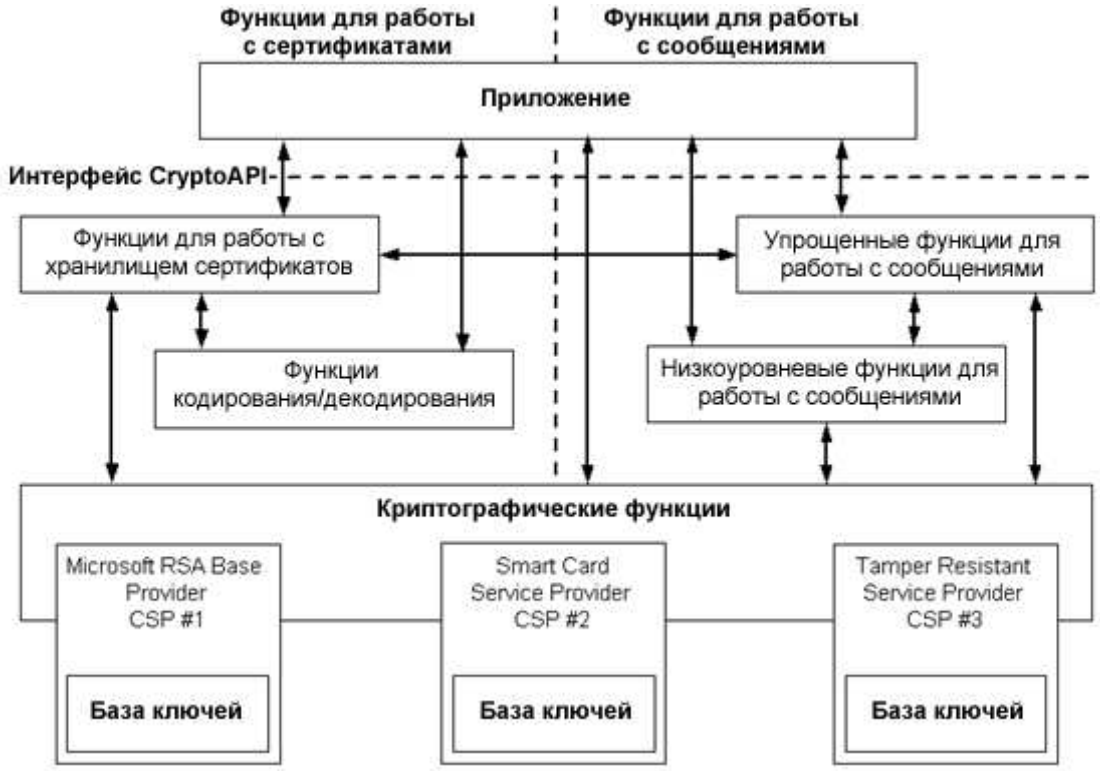

Рисунок 1.1 – Архитектура CryptoAPI

Для использования функций CryptoAPI в приложениях, написанных на C/C++, необходимо использовать заголовочный файл WinCrypt.h и подключить к приложению библиотеку импорта Crypt32.lib.

Криптопровайдеры CryptoAPI

Одним из ключевых понятий CryptoAPI является криптографический провайдер (CSP, Cryptographic Service Provider), реализующий базовый набор криптографических функций, соответствующих тому или иному криптографическому алгоритму (или семейству алгоритмов). Криптопровайдеры реализованы в виде отдельных DLL, так что сторонние разработчики могут самостоятельно включать в состав CryptoAPI реализации своих алгоритмов (хотя для распространения

криптопровайдера потребуется подписать его цифровой подписью в Microsoft). Одновременно в операционной системе можно установить несколько CSP.

Выбор конкретного алгоритма задается параметрами функции и зависит от используемого CSP, а универсальный набор функций определен для каждого типа криптографических операций.

CryptoAPI предоставляет следующие стандартные криптопровайдеры:

- Microsoft Base Cryptographic Provider.
  Определение: MS_DEF_PROV, Библиотека: rsabase.dll
- Microsoft Strong Cryptographic Provider (поставляется начиная с Windows 2000, является расширением базового).
  Определение: MS_STRONG_PROV, Библиотека: rsaenh.dll
- Microsoft Enhanced Cryptographic Provider (по сравнению с базовым обеспечивает работу с ключами большей длины).
  Определение: MS_ENHANCED_PROV, Библиотека: rsaenh.dll
- Microsoft AES Cryptographic Provider (поддержка AES).
- Microsoft DSS Cryptographic Provider (реализует алгоритмы SHA и DSS).
  Определение: MS_DEF_DSS_PROV,Библиотека: dssbase.dll
- Microsoft Base DSS and Diffie-Hellman Cryptographic Provider (добавлена поддержка алгоритма обмена ключей Диффи-Хеллмана).
  Определение: MS_DEF_DSS_DH_PROV, Библиотека: dssbase.dll
- Microsoft DSS and Diffie-Hellman/Schannel Cryptographic Provider (добавлена аутентификация клиентов по защищенному протоколу взаимодействия).
  Определение: MS_DEF_DH_SCHANNEL_PROV, Библиотека: dssenh.dll
- Microsoft RSA/Schannel Cryptographic Provider (реализация алгоритмов RSA).

Типы криптопровайдеров

Все CSP отличаются друг от друга своими типами, которые определяются набором параметров, включающим:

- алгоритм обмена сессионным (симметричным) ключом;
- алгоритм вычисления цифровой подписи;
- формат цифровой подписи ;
- схема генерирования сессионного ключа по хешу;
- длина ключа.

В файле WinCrypt.h даны определения типов криптопровайдеров, определенных в настоящее время. Наиболее известные типы CSP представлены в таблице 1.1.

Таблица 1.1 – Описание типов CSP

Тип криптопровайдера	Поддерживаемые алгоритмы
PROV_RSA_FULL(1)	ключевой обмен — RSA; цифровая подпись — RSA; шифрование — RC2, RC4; хеширование — MD5, SHA.
PROV_RSA_SIG (2)	ключевой обмен — не поддерживается: цифровая подпись — RSA; шифрование — не поддерживается; хеширование — MD5. SHA.
PROV_RSA_SCHANNEL(12)	ключевой обмен — RSA; цифровая подпись — RSA; шифрование — RC4. DES, Triple DES (не обязательно все); хеширование — MD5, SHA.
PROV_DSS (3) PROV_RSA_SIG	ключевой обмен — не поддерживается; цифровая подпись — DSS(DSA); шифрование — не поддерживается; хеширование — MD5, SHA.
PROV_DSS_DH(13)	ключевой обмен — DH; цифровая подпись — DSS; шифрование - CYLINK МЕК; хеширование — MD5, SHA.
PROV_DH_ SCHANNEL (18)	ключевой обмен — DH (Ephemeral); цифровая подпись — DSS; шифрование — DES, Triple DES; хеширование — MD5, SHA.
PROV_FORTEZZA (4)	ключевой обмен — KEA; цифровая подпись — DSS; шифрование — Skipjack; хеширование — SHA.
PROV_MS_EXCHANGE (5)	ключевой обмен — RSA; цифровая подпись — RSA; шифрование — CAST; хеширование — MD5.
PROV_SSL (6)	ключевой обмен — RSA; цифровая подпись — RSA; шифрование — различные алгоритмы; хеширование — различные алгоритмы.

Функции работы с криптопровайдерами можно разделить на следующие группы:

- функции инициализации контекста и получения параметров криптопровайдера;

– функции генерации ключей и работы с ними;
– функции шифрования/расшифровывания данных;
– функции хеширования и получения цифровой подписи данных.

Каждый CSP имеет свою базу ключей (*key database*), в которой находится один или несколько контейнеров (*key containers),* где хранятся пары закрытых/открытых ключей.

Контейнеры бывают двух типов – пользовательские (этот тип используется по умолчанию) и машинные (*CRYPT_MACHINE_KEYSET*). Тип контейнера задается флагом при получении контекста провайдера.

На компьютере обычно установлено несколько провайдеров и соответствующих типов криптопровайдеров. Для того, чтобы просмотреть все установленные криптопровайдеры и типы надо применить соответствующие функции *CryptEnumProviders* и *CryptEnumProviderTypes*.

Пример просмотра установленных криптопровайдеров на компьютере:

```
#include <stdio.h>
#include <windows.h>
#include <wincrypt.h>
#include <stdlib.h>
#include <wtypes.h>
#include <iostream>
#include <fstream>
#pragma comment(lib,"crypt32.lib")
using namespace std;

void view_installed_providers()
{
     HCRYPTPROV hProv = NULL;
     LPTSTR      pszName = NULL;
     DWORD       dwType;
     DWORD       cbName;
     DWORD       dwIndex = 0;

     // Цикл по перечисляемым типам провайдеров.
     dwIndex = 0;
     while(CryptEnumProviderTypes(
          dwIndex,
          NULL,
          0,
          &dwType,
          NULL,
          &cbName
          ))
     {
          //  Распределение памяти в буфере для восстановления
этого имени.
          pszName = (LPTSTR) malloc(cbName);
          if(!pszName)
               puts("ERROR - malloc failed!");
          memset(pszName, 0, cbName);
          //  Получение имени типа провайдера.
```

```
          if(CryptEnumProviderTypes(
               dwIndex++,
               NULL,
               0,
               &dwType,
               pszName,
               &cbName))
          {
               printf ("   %4.0d ",dwType);
               wcout<<pszName<<endl;
          }
          else
          {
               puts("ERROR - CryptEnumProviders");
          }
     }
     //   Печать заголовка перечисления провайдеров.
     printf("\n\n          Listing Available Providers.\n");
     printf("Provider type      Provider Name\n");
     printf("_____________________________________\n");

     // Цикл по перечисляемым провайдерам.
     dwIndex = 0;
     while(CryptEnumProviders(
          dwIndex,
          NULL,
          0,
          &dwType,
          NULL,
          &cbName
          ))
     {
          //  Распределение памяти в буфере для восстановления
этого имени.
          pszName = (LPTSTR)malloc(cbName);
          if(!pszName)
               puts("ERROR - malloc failed!");
          memset(pszName, 0, cbName);
          //  Получение имени провайдера.
          if(CryptEnumProviders(
               dwIndex++,
               NULL,
               0,
               &dwType,
               (LPWSTR)pszName,
               &cbName     // pcbProvName -- длина pszName
               ))
          {
               printf ("     %4.0d        ",dwType);
               wcout<<pszName<<endl;
          }
          else
          {
```

```
                puts("ERROR - CryptEnumProviders");
            }
        } // Конец цикла while
        printf("\nProvider types and provider names have been
listed.\n");

}
```

При работе с криптопровайдерами приложения имеют ряд ограничений:

- Приложение не может напрямую воспользоваться реализациями функций из криптопровайдера, и все взаимодействие проходит через базовые криптографические функции. В идеальном случае возможна смена одного криптопровайдера на другого без модификации использующего его приложения, хотя на практике некоторые
- Приложения могут потребовать функциональности определенного криптопровайдера.
- Приложение не имеет прямого доступа к ключам, используемым функциями криптопровайдера.
- Приложения не могут влиять на выполнение криптографических операций иначе как заданием алгоритма и выбором ключа.
- Приложения не отвечают за аутентификацию пользователей, всю эту работу выполняет криптопровайдер.
- Для получения доступа к криптопровайдеру используется функция *CryptAcquireContext*, по завершении работы необходимо вызвать функцию *CryptReleaseContext.*

CryptAcquireContext — выполняет подключение к криптопровайдеру с заданным типом и именем и возвращает его дескриптор (контекст).

Таким образом, сеанс работы начинается с инициализации (получения контекста) с помощью функции *CryptAcquireContext.* В качестве параметров эта функция принимает имя контейнера ключей, имя криптопровайдера, тип провайдера и флаги, определяющие тип и действия с контейнером ключей и режим работы криптопровайдера.

```
BOOL WINAPI CryptAcquireContext(
HCRYPTPROV* phProv,
LPCTSTR pszContainer,//имя контейнера ключей или NULL-по умолчанию
LPCTSTR pszProvider,//указывают имя одного или несколько имен
                    криптопровайдеров
DWORD dwProvType,//тип криптопровайдера
DWORD dwFlags);// 0 или  CRYPT_NEWKEYSET
```

Рассмотрим пример подключения к криптопровайдеру:

```
//указываем имя и тип криптопровайдера и открываем или создаем
контейнер для ключей
    HCRYPTPROV hProv;
```

```
     if(!CryptAcquireContextW(
     &hProv,NULL,MS_ENHANCED_PROV,PROV_RSA_FULL,0) &&
     (!CryptAcquireContextW(
&hProv,NULL,MS_ENHANCED_PROV,PROV_RSA_FULL,CRYPT_NEWKEYSET)))
     {    puts("Не удается получить контекст\n");
          return -1;
      }
```

Возможен другой вариант, когда криптопровайдер выбирается по умолчанию, а тип PROV_DH_SCHANNEL:

```
HCRYPTPROV hProv;
     //пытаемся открыть существующий keyset или создать новый
if(!CryptAcquireContext(&hProv, NULL, NULL,PROV_DH_SCHANNEL,0)
&&
(!CryptAcquireContext(&hProv,NULL,NULL,PROV_DH_SCHANNEL,
CRYPT_NEWKEYSET))
     {
          puts("Не удается создать контекст\n");
          return -1;
     }
```

Функции *CryptSetProvParam* и *CryptGetProvParam* позволяют соответственно задать и узнать ряд характеристик криптопровайдера.

Работа с сертификатами X509

Алгоритмы шифрования с открытым ключом и цифровые подписи тесно связаны с понятием сертификата. Цифровой сертификат выдается некой авторизованной организацией (Certification Authority, CA) и помимо пары публичный/закрытый ключ содержит информацию, позволяющую идентифицировать его владельца. Кроме этого, сертификат имеет срок действия и подписывается с помощью другого ключа CA, который используется для обеспечения гарантии подлинности этих атрибутов и, что наиболее важно, самого открытого ключа.

CryptoAPI поддерживает сертификаты, соответствующие спецификации X.509 (текущая версия 3), описанные в документе RFC 3280. Стандарт X.509 не определяет обязательного типа ключа, встроенного в сертификат, но в настоящее время алгоритм RSA является наиболее известным из применяемых криптографических алгоритмов с асимметричными шифрами.

Существует возможность создавать собственные сертификаты. Метод их создания обычно зависит от способа их применения. Для обычных ситуаций в Интернете, когда пользователь не знает, с кем он связывается, запрашивается, как правило, сертификат от коммерческого центра сертификации. Преимуществом такого подхода является то, что эти известные центры сертификации уже признаны доверенными операционной системой Windows и всеми другими операционными системами (и обозревателями), поддерживающими сертификаты и

протокол SSL. Это позволяет обходиться без обмена ключами центра сертификации.

Можно создавать сертификаты с помощью программы OpenSSL. OpenSSL поддерживает два основных формата, определяющих кодирование криптографических документов: DER - бинарные файлы и PEM - текстовые файлы.

Криптографический документ в формате PEM имеет вид:

-----BEGIN <ТИП ДОКУМЕНТА>-----

<Документ в формате DER, закодированный в base64>

-----END <ТИП ДОКУМЕНТА>-----

Т.е. содержание документа окружается заголовками специального вида, состоящими из пяти знаков «-», слов begin и end и указания типа документа (CERTIFICATE, CRL и т.д.) Благодаря наличию таких заголовков в одном файле может содержаться несколько документов в формате PEM: например, цепочка сертификатов или сертификат и его закрытый ключ. По умолчанию в OpenSSL используется именно формат PEM.

Файлы, содержащие цепочки сертификатов, сертификаты и ассоциированные с ними ключи, могут представляться в нескольких форматах, определяющих состав и строение файлов, содержащих ключевую информацию; сама эта информация кодируется в вышеописанных форматах PEM и DER.

Наиболее распространенные из форматов, определяющих структуру файлов, содержащих криптографическую информацию — PKCS#7 и PKCS#12.

PKCS#7 — формат защищенных сообщений, который, кроме самого сообщения, может содержать необходимые для работы с ним сертификаты и CRL. Многие удостоверяющие центры распространяют свои сертификаты и CRL в виде PKCS#7 документов, в которых нет сообщения, есть только служебная информация. PKCS#7-документ может быть закодирован и в формате DER, и в формате PEM. Расширения файлов сертификатов могут быть .cer или .crt .

PKCS#12 — формат для переноса сертификата и связанного с ним закрытого ключа с машины на машину или для резервного копирования. В этом формате могут также содержаться сертификаты удостоверяющего центра. Файлы формата PKCS#12 всегда кодируются в формате DER. Эти файлы требуют особенно пристального внимания, поскольку содержат закрытый ключ; при неосторожном обращении с таким файлом закрытый ключ легко скомпрометировать. Как правило, эти файлы защищены на пароле. Расширение файлов формата PKCS#12 либо .p12, либо .pfx . В Windows удобно работать с сертификатом данного формата. При выполнении экспорта пар ключей Windows предлагает зашифровать PFX-файл с помощью пароля; при импорте пары ключей пользователь должен снова предоставить этот пароль.

Хранилище сертификатов Windows

Сертификаты и соответствующие им закрытые ключи можно хранить на различных устройствах, например жестких дисках, смарт-картах и в ключах для порта USB. В Windows предусмотрен уровень абстракции, называемый хранилищем сертификатов, который служит для обеспечения единого способа доступа к сертификатам независимо от места их хранения. До тех пор пока у аппаратного устройства имеется поддерживаемый Windows поставщик службы криптографии (CSP), можно получать доступ к данным, хранящимся на этом устройстве, используя интерфейс API хранилища сертификатов.

Хранилище сертификатов глубоко запрятано в профиле пользователя. Это позволяет использовать списки управления доступом (ACL) для ключей конкретной учетной записи. Каждое хранилище разделено на контейнеры:

– Контейнер с названием Personal (личный), предназначен для хранения пользовательского сертификата (т.е., у которых есть соответствующий закрытый ключ).

– В контейнере Trusted Root Certification Authorities (доверенные корневые центры сертификации) хранятся сертификаты всех центров сертификации, которым доверяет пользователь.

– В контейнере Other People (Другие) содержатся сертификаты коллег, с которыми имеется безопасная связь. И так далее.

Самый простой способ получить доступ к своему хранилищу сертификатов заключается в выполнении команды *certmgr.msc***.**

Существует также хранилище на уровне компьютера, используемое учетными записями компьютера с ОС Windows (сети, локальной службы и локальной системы) или служащее для совместного использования сертификатов и ключей разными учетными записями.

Любой сертификат X.509 получаемый пользователем по почте или по запросу из центра сертификации необходимо поместить в хранилище сертификатов в WINDOWS. Для этого выполняют двойной щелчок мыши по файлу сертификата и затем в диалоговом окне выбирают имя хранилища (например, личный «МУ»).

Работа с сертификатом X.509 с помощью CryptoAPI

Алгоритм работы с сертификатом состоит из вызова следующих функций:

- *CertOpenStore* - открываем хранилище сертификатов.
- *CertOpenSystemStore* открываем хранилище системных сертификатов.
- *CertFindCertificateInStore* – поиск нужного сертификата.
- *CertEnumCertificatesInStore*- перебирает все сертификаты в хранилище.

– *CertGetNameString*- извлекает имя текущего просматриваемого сертификата.

Пример работы с сертификатом:

```
// Сертификат, представлен в формате PKSC#12
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <wtypes.h>
#include <wincrypt.h>

#define MY_TYPE  (PKCS_7_ASN_ENCODING | X509_ASN_ENCODING)

// Наименование персонального хранилища
#define CERT_STORE_NAME  L"MY"

// Наименование сертификата, установленного в это хранилище
#define SIGNER_NAME  L"EVGENY"

// Открываем хранилище сертификатов
  HCERTSTORE hStore;

  if ( !( hStore = CertOpenStore(
    CERT_STORE_PROV_SYSTEM,
    0,
    NULL,
    CERT_SYSTEM_STORE_CURRENT_USER,
    CERT_STORE_NAME)))
  {
       puts("Нельзя открыть хранилище MY ");

    }

// Получаем указатель на наш сертификат
  PCCERT_CONTEXT pSignerCert=0;

  if(pSignerCert = CertFindCertificateInStore(
    hStoreHandle,
    MY_TYPE,
    0,
    CERT_FIND_SUBJECT_STR,
    SIGNER_NAME,
    NULL))
  {
    printf("Серификат найден\n");
  }
  else
  {

    printf("Сертификат не найден.");
  }
```

```
//С помощью данной функции выведем имя CSP и имя контейнера
ключей
DWORD dwUserNameLen = 100;
  CHAR szUserName[100];

  if(CryptGetProvParam(
    hProv,              // Дескриптор на CSP
    PP_NAME,            // параметр для получения имени CSP
    (BYTE *)szUserName, // Указатель на буфер, содержащий имя
CSP
    &dwUserNameLen,     // длина буфера
    0))
  {
    printf("Name CSP: %s\n",szUserName);
  }
  else
  {
    puts("Ошибка CryptGetProvParam.\n");
  }

  if(CryptGetProvParam(
    hProv,              // Дескриптор на CSP
    PP_CONTAINER,      // параметр для получения имени key
container
    (BYTE *)szUserName, // Указатель на буфер, содержащий имя
key container
    &dwUserNameLen,     // длина буфера
    0))
  {
    printf("Name key container: %s\n",szUserName);
  }
  else
  {
    puts("ErrorCryptGetProvParam.\n");  }
```

Пример, позволяющий просмотреть все сертификаты в хранилище.

```
     //открываем хранилище сертификатов
if(!(hStore=CertOpenSystemStore(NULL," EVGENY ")))
     {cout<<"Error opening store\n";cin.get();return 1;}
char* pszName=new char[128];
int i=0;int certNum=0;
     //перебираем и выводим все имена сертификатов
while(hContext=CertEnumCertificatesInStore(hStore,hContext))
     {
if(!CertGetNameString(hContext,CERT_NAME_SIMPLE_DISPLAY_TYPE,0,
0,pszName,128))
          {cout<<"Error getting name\n";cin.get();return 1;}
          i++;
          cout<<i<<" "<<pszName<<'\n';
          }
     cout<<"Enter number of certificate\n";
     cin>>certNum;hContext=NULL;
```

```
//получаем из хранилища выбраный сертификат
for(i=0;i<certNum;i++)
{
hContext=CertEnumCertificatesInStore(hStore,hContext);
}
delete [] pszName;
//проверяем как получили сертификат
if(hContext==NULL)
{cout<<"Error getting certificate\n";
cin.get();return 1;}
```

Все ключи в CRYPTOAPI управляются и используются при помощи неких идентификаторов, и приложение не получает открытого доступа к ним. При подключении к криптопровайдеру он может предоставить доступ к стандартному хранилищу ключей, либо воспользоваться хранилищем, созданным по запросу приложения.

Получив указатель на сертификат X.509 теперь можно извлечь открытый ключ с помощью функции *CryptImportPublicKeyInfo.* В дальнейшем, получив идентификатор открытого ключа можно использовать его для обмена ключами (экспорта сессионного ключа) или для проверки электронно-цифровой подписи.

Пример импорта открытого ключа из сертификата:

```
// Импортируем public key для последующей верификации подписи
или шифрования сеансового ключа
  HCRYPTKEY hPublicKey;
  if(CryptImportPublicKeyInfo(
    hProv,
    MY_TYPE,
    &(pSignerCert->pCertInfo->SubjectPublicKeyInfo),
    &hPublicKey))
  {
    printf("Import public key.\n");
  }
  else
  {
    printf ("Ошибка CryptAcquireContext.");
   }
```

Закрытый ключ, соответствующий сертификату открытого ключа хранится в специальном контейнере ключей. Для работы с данным ключом необходимо перейти к работе с CSP и контейнеру нашего сертификата через функцию *CryptAcquireCertificatePrivateKey.* А затем с помощью функции *CryptGetUserKey* получить значения закрытого ключа из указанного контейнера ключей. Данный ключ может использоваться для обмена ключами (импорта сессионного ключа) или для создания цифровой подписи.

Рассмотрим пример извлечение секретного ключа, соответствующего сертификату открытого ключа PKSC#12:

```
HCRYPTKEY hPrivateKey=0;
DWORD keySpec=0;
// Извлекаем из сертификата контекст приватного ключа
if(!CryptAcquireCertificatePrivateKey(
     pSignerCert,
     0,
     NULL,
     &hProv,
     &keySpec,
     NULL))
      { cout<<"Error getting private context\n";
      getchar();
      return -1;
      }
//   Извлекаем закрытый ключ
if(!CryptGetUserKey(hProv,keySpec,&hPrivateKey))
          {cout«"Error getting private key\n";
           getchar ();
           return -1;
          }
```

Работа с ключами шифрования с помощью CryptoAPI

CryptoAPI позволяет шифровать данные, размещенные в оперативной памяти в виде последовательности байтов. Для шифрования могут быть использованы как симметричные, так и асимметричные алгоритмы. CryptoAPI предоставляет методы для работы с сессионными (симметричными) ключами и с открытыми ключами. Асимметричные алгоритмы используются для обмена сеансовыми ключами. Зашифровав сообщение электронной почты симметричным алгоритмом, вы добавляете к зашифрованному сообщению сам сеансовый ключ, зашифрованный на открытом ключе получателя с использованием асимметричного алгоритма. Это позволяет вам уничтожить сеансовый ключ сразу же после шифрования сообщения. Даже если ваш компьютер подвергнется взлому, злоумышленник все равно не узнает содержание сообщения.

В CryptoAPI генерация ключей может выполняться случайным образом с помощью функции *CryptGenKey* или на основе некоторых данных, например пользовательского пароля с помощью функции *CryptDeriveKey*. По завершении работы с ключом он уничтожается функцией *CryptDestroyKey*.

В CryptoAPI алгоритм шифрования сообщения указывается при генерации сессионного ключа в *CryptGenKey*:

```
BOOL  CryptGenKey (
   HCRYPTPROV hProv,
         ALG_ID Algid,  // идентификатор алгоритма шифрования, для
                        которого генерируется ключ (например,
                        CALG_3DES). Для ключевой пары RSA для
                        шифрования и подписи используются
```

AT_KEYEXCHANGE и AT_SIGNATURE.
DWORD dwFlags, //задает различные опции ключа, которые зависят от алгоритма и провайдера. CRYPT_EXPORTABLE-для экспорта сессионного ключа. CRYPT_ENCRYPT-для шифрования.
HCRYPTKEY* phKey); //идентификатор ключа.

Пример генерации случайного сессионного ключа:

```
HCRYPTKEY hKey;
//генерируем сессионный ключ для DES
if(!CryptGenKey(
     hProv,
     CALG_DES,
     CRYPT_EXPORTABLE|CRYPT_ENCRYPT,
     &hKey))
     {
          puts("Не удается создать ключ DES\n");
          return -1;
     }
```

Параметры ключа

После создания ключа его параметрами можно манипулировать с помощью функций *CryptGetKeyParam()* и *CryptSetKeyParam()*. Эти функции предоставляют доступ к таким атрибутам, как алгоритм создания ключа, длина шифруемого блока, инициирующее значение, начальный вектор, режим буферизации и режим шифрования, а также допуски, которые задают операции, допустимые для данного ключа.

Алгоритмы симметричного шифрования имеют следующие режимы шифрования сообщений:

CRYPT_MODE_ECB - режим простой замены;
CRYPT_MODE_CBC - режим сцепления блоков шифра;
CRYPT_MODE_OFB - режим гаммирования;
CRYPT_MODE_CFB - режим гаммирования с обратной связью.

По умолчанию в CryptoAPI блочные шифры используются в режиме сцепления блоков шифра (CBC - cipher block chaining). В этом режиме используется инициализирующий вектор (IV - initialization vector). По умолчанию IV нулевой. Инициализирующий вектор должен генерироваться отдельно с помощью функции *CryptGenRandom* и, как и солт-значение, передаваться вместе с ключом в открытом виде. Размер IV равен длине блока шифра. Например, для алгоритма RC2, поддерживаемого базовым криптопровайдером Microsoft, размер блока составляет 64 бита (8 байтов).

Если режим шифрования данных должен быть не CBC, то надо установить соответствующие параметры с помощью функции *CryptSetKeyParam.*

```
CryptSetKeyParam(HCRYPTKEY hKey,
                 DWORD dwParam,
                 BYTE *pbData,
                 DWORD *pdwDataLen,
                 DWORD dwFlags);
```

Если ключ hKey сессионный, то в качестве параметра можно установить начальный вектор KP_IV, режим шифрования KP_MODE, способ дополнения KP_PADDING.

```
// Устанавливаем режим шифрования сообщения OFB

        DWORD dwMode = CRYPT_MODE_OFB;
    if(!CryptSetKeyParam(hKey, KP_MODE, (BYTE*)&dwMode, 0))
    {
      puts ("Error CryptSetKeyParam!\n");
      return -1;
    }
```

Пересылка ключей в CryptoAPI

В качестве ключей экспорта/импорта могут использоваться либо ключевая пара RSA (с типом AT_KEYEXCHANGE), либо симметричный сеансовый ключ. Обмен ключами реализуется с помощью функций *CryptExportKey* и *CryptImportKey***.**

Функция экспорта ключа для его передачи по каналам информации

```
BOOL WINAPI CryptExportKey(
HCRYPTKEY hKey,
HCRYPTKEY hExpKey,
DWORD dwBlobType,// PUBLICKEYBLOB-экспорт открытого ключа
                 // PRIVATEKEYBLOB –экспорт закрытого ключа
                 // SIMPLEBLOB-экспорт сеансового ключа
DWORD dwFlags,
BYTE* pbData,       //буфер, содержащий ключевой блоб
                      (зашифрованный ключ)
DWORD* pdwDataLen); // размер ключевого блоба.
```

Выполним экспорт сессионного ключа с помощью функции *CryptExportKey* и запишем результат в файл:

```
//Определяем размер Bloba сессионного ключа
DWORD dwBlobLenght = 0;
if(CryptExportKey(hKey,hPublicKey,SIMPLEBLOB,0,0,&dwBlobLenght)
)
{
      printf("size of the Blob");
}
else
{
      printf("error computing Blob length");
```

```
      getchar();
      return -1;
}
//Распределяем память для сессионного ключа
      BYTE *ppbKeyBlob;
      ppbKeyBlob = NULL;
  if(ppbKeyBlob = (LPBYTE)malloc(dwBlobLen))
      {
      printf("memory has been allocated for the Blob");
      }
else
      {
        printf ("Error memory for key length!!!");
        getchar();
        return -1;
         }
//Зашифруем сессионный ключ hKey открытым ключом hPublicKey
if(CryptExportKey(hKey, hPublicKey, SIMPLEBLOB, 0, ppbKeyBlob,
&dwBlobLenght))
      {
      printf("contents have been written to the Blob");
      }
else
      {
      printf("Could not get exporting key.");
      free(ppbKeyBlob);
      ppbKeyBlob=NULL;
      getchar();
      return -1;   }

//Записываем экспортированный ключ в файл out.
if(fwrite(ppbKeyBlob,sizeof byte, dwBlobLenght,out))
{
printf("the session key has been written to the file\n");
      free(ppbKeyBlob);
}
else
{
printf("the session key could not be written to the file\n");
      getchar();
      return -1;
      }
```

Функция, предназначенная для получения из каналов информации значения ключа.

```
BOOL CryptImportKey
    (HCRYPTPROV hProv,
          BYTE* pbData,   // ключевой блоб (зашифрованный ключ,
                          полученный в результате действия функции
                          CryptExportKey)
       DWORD dwDataLen,   // длина ключевого блоба (в байтах)
```

HCRYPTKEY hImpKey, //дескриптор ключа, которым дешифруем
DWORD dwFlags,
HCRYPTKEY* phKey); // адрес, по которому функция копирует дескриптор импортированного ключа

Пример импорта сессионного ключа:

```
//Распределяем память для сессионного ключа
      BYTE *ppbKeyBlob;
      ppbKeyBlob = NULL;
  if(ppbKeyBlob = (LPBYTE)malloc(dwBlobLen))
      {
      printf("memory has been allocated for the Blob");
      }
else
      {
        printf ("Error memory for key length!!!");
      getchar();
      return -1;
            }
//Считываем сессионный  ключ из файла in.
if(fread(ppbKeyBlob,sizeof byte, dwBlobLenght,in))
{
      printf("the session key has been read to the file\n");
      free(ppbKeyBlob);
}
else
{
printf("the session key could not be read from the file\n");
      getchar();
      return -1;
      }
//Импортируем сессионный ключ с помощью закрытого ключа
ассиметричного алгоритма
hkey=0;
if(CryptImportKey(hProv,ppbKeyBlob,dwBlobLenght,hPrivateKey,0,
&hKey))
{
printf(" the key has been imported.\n");
CryptDestroyKey(hPrivateKey); //очищаем ресурсы
free(ppbKeyBlob);
}
else
{
      printf("the session key import failed.\n");
      getchar();
      return -1;}
```

Симметричное шифрование

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 контекста.

– *CryptDecrypt.* Основная базовая функция расшифровывания данных. В качестве параметров используются ранее полученные контекст криптопровайдера и идентификатор сессионного ключа.

Базовая функция шифрования данных имеет следующее объявление:

```
BOOL CryptEncrypt(
  HCRYPTKEY hKey,    //идентификатор сессионного ключа
 HCRYPTHAS hHash,    // используется для получения хеша данных.
      BOOL Final,    //позволяет обрабатывать данные блоками
    DWORD dwFlags,   //служит указателем на массив входных/выходных
                     данных. Обычно устанавливается в 0.
   BYTE* pbData,     //порция данных для шифрования
       DWORD*        //служит для возврата размера данных,
  pdwDataLen,        возвращаемых функцией.
 DWORD dwBufLen);    // служит для указания длины входного буфера
                     данных
```

Пример использования функции CryptEncrypt приведен ниже:

```
# define BLOCK_SIZE 15
BYTE     *pCryptBuf = 0;
DWORD   buflen;
BOOL      bRes;
DWORD    datalen;

// Определяем размер буфера необходимого для блоков длины BLOCK
SIZE
buflen=BLOCKSIZE;
if(!CryptEncrypt(hKey, 0,TRUE,0,NULL,&buflen, 0 ))
{
cout<<" Crypt Encrypt (bufSize) failed."«endl;
getchar ();
 return -1;
}

//Выделим память под буфер
pCryptBuf = (BYTE*)malloc( buflen );
int t=0;

//   Шифруем файл in
while((t=fread(pCryptBuf,sizeof byte,BLOCK_SIZE,in)))
{
     datalen = t;
bRes=CryptEncrypt( hKey,0, TRUE,0,pCryptBuf,&datalen,buflen);

     if( !bRes ) {
     cout«"CryptEncrypt (encryption) failed, "«endl;
```

```
     getchar(); return -1;
          }
          fwrite(pCryptBuf,sizeof byte,datalen,out);
     }
     cout«"File encryption completed successfully"«endl;
     fclose(in);
     fclose(out);
     free(pCryptBuf);
     CryptDestroyKey(hKey);
     CryptReleaseContext(hProv,0);
```

Пример расшифровки файла с помощью функции CryptDecrypt.

```
// Определяем размер буфера необходимого для блоков длины BLOCK
SIZE

buflen=BLOCK_SIZE;
bRes = CryptEncrypt( hKey,0, TRUE,0, NULL, &buflen,0);
pCryptBuf = (BYTE*)malloc( buflen );

// Расшифровываем файл
while((t=fread(pCryptBuf,sizeof byte,datalen,in)))
{
     buflen = t;
     bRes = CryptDecrypt( hKey, 0,TRUE,0,pCryptBuf,&buflen ) ;
     if( !bRes )
     {
     cout<<"CryptEncrypt (buffer size) failed, "«endl;
     getchar(); return -1;
     }
     fwrite(pCryptBuf,sizeof byte,buflen,out);
}
cout«"File decryption completed successfully"«endl;
```

Лабораторная работа 2. Электронно-цифровая подпись.

Цель работы:

Изучение основных функций библиотеки CRYPTOAPI необходимых для обеспечения целостности документа и защиты авторских прав.

Уметь пользоваться функциями хеширования, функциями создания и проверки электронно-цифровой подписи документа.

Результат:

Создание программного приложения для защиты документов с помощью электронно-цифровой подписи (ЭЦП).

Задание для самостоятельного выполнения

Перед выполнением задания студент, используя программу openssl, создает на своем компьютере локальный центр сертификации. Затем

создает ключевую пару RSA <KO,KC>. На основании открытого ключа KO создает запрос на сертификацию и передает его в локальный центр сертификации. Получает из локального центра сертификации сертификат X509 своего открытого ключа KO и сертификат X509открытого ключа центра сертификации.

Задача 1

Используя криптографические интерфейсы MS CryptoAPI 1.0 (2.0), создать приложение, позволяющее подписывать файлы и проверять подпись подписанных файлов по следующей схеме.

- Секретный ключ KC для создания ЭЦП извлекается из сертификата субъекта, подписывающего файл.
- ЭЦП хранится в отдельном файле вместе с сертификатом открытого ключа подписавшего.
- Чтобы проверить ЭЦП, вначале надо проверить сертификат открытого ключа подписавшего.

Задача 2

Используя криптографические интерфейсы MS CryptoAPI 1.0 (2.0), создать приложение, позволяющее подписывать файлы и проверять подпись подписанных файлов по следующей схеме.

- Секретный ключ KC для создания ЭЦП извлекается из сертификата субъекта, подписывающего файл.
- ЭЦП хранится в отдельном файле.
- Чтобы проверить ЭЦП, надо иметь сертификат открытого ключа подписавшего.

Варианты заданий

Вариант	Задача	Алгоритм
1	Задача 2	MD2
2	Задача 2	MD4
3	Задача 2	MD5
4	Задача 2	SHA
5	Задача 1	MD2
6	Задача 1	MD4
7	Задача 1	MD5
8	Задача 1	SHA

Вариант задания определяется так: $(n_1n_2+11) \bmod 27 +1$, где n_1n_2– две последние цифры номера зачетки.

Замечание. Для успешного выполнения работы используем сертификаты формата PKCS#12 , т.е. с расширением .p12 либо .pfx

Схема создания ЭЦП состоит :

– создание хеш-образа сообщения;
– шифрование хеш-образа асимметричным алгоритмом с помощью закрытого ключа.

Проверка ЭЦП заключается в следующих этапах:

– расшифрование ЭЦП открытым ключом;
– создание хеш – образа сообщения;
– сравнения хеш-образов, полученных на 1 и 2 этапах.

Вычисление хешей с помощью CryptoAPI

CryptoAPI позволяет использовать ряд алгоритмов хеширования для работы с цифровыми подписями. Доступны следующие алгоритмы:

– MD2, MD4, MD5 — хорошо известные и популярные алгоритмы хэширования, разработанные RSA. Генерируют 128-битные хэши, к использованию не рекомендуются.

– Secure Hash Algorithm (SHA) — алгоритм, разработанный Национальным институтом стандартов и технологий (National Institute of Standards and Technology, NIST) и Агентством национальной безопасности (National Security Agency, NSA) США. Генерирует 160-битный хэш, используется в сочетании с алгоритмами DSA (Digital Signature Algorithm) и DSS (Digital Signature Standard).

– SSL3 Client Authorization Algorithm. Используется для аутентификации клиентов, реализован как конкатенация хэшей MD5 и SHA, подписанная приватным RSA-ключом

Таблица - ID алгоритмов хеширования

ID алгоритма	Описание
CALG_MD2	Алгоритм хеширования MD2
CALG_MD4	Алгоритм хеширования MD4
CALG_MD5	Алгоритм хеширования MD5
CALG_SHA	Безопасный алгоритм хеширования (Secure Hash Algorithm — SHA) US DSA
CALG_SHA1	Алгоритм SHA (то же, что и calg sha)

Для хеширования данных сначала необходимо создать объект хеширования с помощью функции *CryptCreateHash*, после чего наполнить его хешируемыми данными с помощью функции *CryptHashData*. После передачи последней порции данных, получить значение хеша можно функцией *CryptGetHashParam*. По окончании работы объект хеширования удаляется функцией *CryptDestroyHash.* Для проверки цифровой подписи схема работы аналогична, она проводится с помощью вызова функции *CryptVerifySignature* после заполнения объекта хеширования данными.

Для первичной инициализации «пустого» хеш-объекта применяют функцию *CryptCreateHash.* Данная функция имеет следующее описание:

```
BOOL CryptCreateHash(   //инициализированный контекст
                        криптопровайдера
HCRYPTPROV hProv,
ALG_ID AlgId,           //алгоритм получения значения хеша
HCRYPTKEY hKey,         //необходим лишь в случае применения
                        алгоритмов типа MAC и HMAC.
DWORD dwFlags,          // должен быть всегда равен 0
HCRYPTHASH* phHash) ;   //функция возвращает хендл созданного ей
                        хеш-объекта
```

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

```
BOOL CryptHashData(
HCRYPTHASH hHash,//ранее инициализированный хендл хеш-объекта
        BYTE* pbData,    // порция данных для хеш-функции
        DWORD dwDataLen, // длина передаваемых данных
        DWORD 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,      // ранее инициализированный хендл хеш-
                         объекта
    DWORD dwParam,       // функции определяет тип запрашиваемого
                         значения. Для получения хеш-значения
                         необходимо передать вторым аргументом
                         значение HP_HASHVAL
      BYTE* pbData,      //отвечают за блок памяти, используемый под
 DWORD* pdwDataLen,      возвращаемое значение
    DWORD dwFlags);      //должен быть равен нулю.
```

Рассмотрим пример создания хеш-образа:

```
//открываем файл, содержимое которого подписываем и дальше
 создаем дайджест
HCRYPTHASH hHash;
 DWORD dwLen;
 DWORD fSize=GetFileSize(hInFile,&fSize);
 //создаем хеш-объект
 if{!CryptCreateHash(hProv, CALG_MD5,0,0, &hHash))
 {
```

```
     Error("CryptCreateHash");
    return 0;
}
//чтение файла
BYTE*read=new BYTE[fsize+8];
if(!::ReadFile(hInFile,read,fSize,&dwLen,NULL))
{

     puts("Error reading file\n"); return 0;
 }
// Передача хешируемых данных хэш-объекту.
if{!CryptHashData(hHash, read, dwLen, 0))
{
   Error("CryptHashData");
   return 0;
}
std::cout << "Hash data loaded" « std::endl;
// Получение хеш-значения
 count = 0;
if(!CryptGetHashParam(hHash, HP_HASHVAL, NULL, &count, 0))
{
   Error("CryptGetHashParam");
   return 0;
}
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 0;
}
std::cout « "Hash value is received" « std::endl;
```

Дополнительные функции:

CryptSetHashParam.	Функция используется для установки параметров хеша. Может быть использована, для изменения алгоритма формирования хеша.
CryptDestroyHash.	Функция используется для освобождения хеш-объекта.
CryptDuplicateHash.	Функция позволяет получить копию хеша. Используется при передаче хеша между функциями.

Создание и проверка ЭЦП с помощью CryptoAPI

Базовая функция получения подписи хеша данных имеет следующее описание:

BOOL CryptSignHash(	//значение хендла хеш-объекта, уже инициализированного данными (с помощью
HCRYPTHASH hHash,	функции CryptHashData)
DWORD dwKeySpec,	//определяет, какая именно пара ключей будет использована для формирования подписи. AT_KEYEXCHANGE (пара для обмена ключами), AT_SIGNATURE (пара для формирования цифровой подписи).
LPCTSTR sDescription,	//значение должно всегда быть установлено в NULL
DWORD dwFlags,	//обычно устанавливают в 0
BYTE* pbSignature,	//используют для корректного указания ссылки на
DWORD* pdwSigLen);	массив //выходных данных и его размера

Пример создания цифровой подписи хеш-значения

```
count = 0;
if(!CryptSignHash(hHash, 1, NULL, 0, NULL, &count))
 {
   Error("CryptSignHash");   '
   return 0;
}
char* sign_hash = static_cast<char*>(malloc(count + 1));
ZeroMemory(sign_hash, count + 1);
if(!CryptSignHashW(hHash, 1, NULL, 0, (BYTE*)sign_hash,
   &count))
{
   Error("CryptSignHash");
   return 0;
}
std::cout « "Signature created" « std::endl;
```

Проверка цифровой подписи

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

BOOL CryptVerifySignature(	
HCRYPTHASH hHash,	/передается хендл хеш-объекта, предварительно инициализированный данными посредством функции *CryptHashData/*
BYTE* pbSignature,	/отвечают за передачу значения
DWORD dwSigLen,	проверяемой подписи./
HCRYPTKEY hPubKey,	/используется для указания хендла публичного ключа отправителя подписи (того, кто собственно сформировал цифровую подпись)/

LPCTSTR sDescription,	/ значение должно быть установлено в *NULL*/
DWORD dwFlags);	/обычно не несет полезной нагрузки и устанавливается в 0/

Пример проверки подписи приведен ниже:

```
hlnFile = CreateFileA("res.txt", // имя файла
            GENERIC_READ,    // чтение из файла
            FILE_SHARE_READ, // совместный доступ к файлу
            NULL,  // защиты нет
            OPEN_EXISTING,   // открываем существующий файл
          FILE_ATTRIBUTE_NORMAL,//FILE_FLAG_OVERLAPPED
          NULL // шаблона нет);

       HCRYPTKEY hECP_Key;
       fSize=GetFileSize(hlnFile, &fSize);
       read=new BYTE [fSize+8 ] ;
if CryptCreateHash(hProv, CALG_MD5, 0, 0, &hHash))
 {
       Error "CryptCreateHash") ;
       return 0;
 }
std::cout « "Hash created" « std::endl;
if(!::ReadFile(hlnFile,read,fSize,&dwLen,NULL)) //читаем блок
данных
{
          puts("Error reading file\n"); return 0;
}
// Передача хешируемых данных хэш-объекту.
if(!CryptHashData(hHash, read, dwLen, 0))
{
         Error("CryptHashData"); return 0;
}
std::cout « "Hash data loaded" « std::endl;
// Получение хеш-значения
count = 0;
//определение размера хеш-объекта
if(!CryptGetHashParam(hHash, HP_HASHVAL, NULL, &count, 0))
{
         Error("CryptGetHashParam"); return 0;
}
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 0;
}
std::cout « "Hash value is received" << std::endl;
if(!CryptGetUserKey(hProv, 1, &hECP_Key))
```

```
 {
        Error("CryptGetUserKey"); return 0;
}
std::cout « "Public key is received" « std::endl;

//Проверка цифровой подписи
BOOL result = CryptVerifySignatureW(hHash, (BYTE*)sign_hash,
count, hECP_Key, NULL, 0);
std::cout « "Check is completed" « std::endl;
 std::cout « "Check
result:"«((result)?"Verified!":"NOTverified!") « std::endl;
 ::CryptReleaseContext(hProv, 0);
  CryptReleaseContext(hProv, 0) ;
   return 0;
```

Лабораторная работа 3. Аутентификация сообщения

Цель работы:

Изучение алгоритмов библиотеки CRYPTOAPI необходимых для обеспечения аутентификации документа.

Результат:

Создания программного приложения для защиты аутентификации документов с помощью HMAC.

Задание для самостоятельного выполнения

I. Реализуйте аутентификацию сообщений на основе алгоритма HMAC средствами CryptoAPI двумя способами:
   1) используя криптоинтерфейс только для хеширования значений (конкатенацию сообщения и секретного ключа выполнять самостоятельно);
   2) используя вызов функции CryptCreateHash с ключевым значением для вычисления кода аутентификации HMAC средствами CryptoAPI
II. Реализуйте аутентификацию сообщений на основе алгоритма MAC-CBC средствами CryptoAPI.

Варианты заданий

Вариант	Задача	Алгоритм	Задача	Алгоритм
1	Задача I.1	MD2	Задача I.2	MD2
2	Задача I.2	MD4	Задача II	DES
3	Задача I.1	MD5	Задача I.2	MD5
4	Задача I.2	SHA	Задача II	3DES
5	Задача I.1	SHA	Задача I.2	SHA
6	Задача I.2	MD5	Задача II	AES
7	Задача I.1	MD4	Задача I.2	MD4

8	Задача I.2	MD2	Задача II	3DES 2 ключа
9	Задача I.1	MD2	Задача II	AES
10	Задача I.1	SHA	Задача II	DES

Вариант задания определяется так: $(n_1n_2+11) \bmod 27 +1$, где n_1n_2– две последние цифры номера зачетки.

Вычисление хеширования с использованием секретного ключа (HMAC)

CryptoAPI позволяет использовать ряд алгоритмов хэширования для создания HMAC и MAC CBC. Доступны следующие алгоритмы:

– Message Authentication Code (MAC). Алгоритмы MAC схожи с обычными алгоритмами хэширования, но используют в своей работе шифрование с симметричным ключом.

– Cipher Block Chaining (CBC) MAC. Использует алгоритм блочного шифрования и берет его последний блок в качестве значения хэша.

– HMAC. Более сложная модификация CBC.

Таблица - ID алгоритмов аутентификации сообщений

ID алгоритма	Описание
CALG_MAC	Алгоритм хеширования Message Authentication Code (MAC)
CALG_SSL3_SHAMD5	Алгоритм аутентификации сообщения SSL3
CALG_HMAC	Алгоритм хеширования HMAC

Процесс создания HMAC очень похож на процесс хеширования документа, но для алгоритма HMAC необходимо использовать секретный ключ. Секретный ключ будет создаваться на основе пароля. Итак, первоначально создается хеш-объект вызовом функции *CryptCreateHash*, принимающей на входе контекст криптопровайдера, идентификатор алгоритма (CALG_MD5 или CALG_SHA) . После этого вычисляем хеш парольной фразы с помощью функции *CryptHashData*. А затем формируем ключ, который будет производным от хеша -объекта пароля, функцией *CryptDeriveKey*. Ниже представлен код:

```
HCRYPTPROV  hProv       = NULL;
HCRYPTHASH  hHash       = NULL;
HCRYPTKEY   hKey        = NULL;
PBYTE       pbHash      = NULL;
DWORD       dwDataLen   = 0;
BYTE        Data1[]     =
{0x70,0x61,0x73,0x73,0x77,0x6F,0x72,0x64};
// Создаем ключ хеширования.
if (!CryptCreateHash(
    hProv,
```

```
    CALG_SHA1,
    0,
    0,
    &hHash))
{
printf("Error in CryptCreateHash 0x%08x \n",GetLastError());
   goto ErrorExit;
}

if (!CryptHashData(
    hHash,
    Data1,
    sizeof(Data1),
    0))
{
   printf("Error in CryptHashData 0x%08x \n",
          GetLastError());
   goto ErrorExit;
}

if (!CryptDeriveKey(
    hProv,
    CALG_RC4,
    hHash,
    0,
    &hKey))
{
   printf("Error in CryptDeriveKey 0x%08x \n",GetLastError());
   goto ErrorExit;
}
```

Следующий этап, заключается в создании HMAC, для этого вычисляем пустой хеш-объект вызовом функции *CryptCreateHash,* принимающей на входе контекст криптопровайдера, идентификатор алгоритма (CALG_MD5 или CALG_SHA тот же, что и для секретного ключа) и дескриптор ключа (который получили ранее для HMAC). Функция *CryptSetHashParam* – устанавливает параметры хэш-объекта.

BOOL CryptSetHashParam(

HCRYPTHASH *hHash*,	//дескриптор используемого хеш-объекта
DWORD *dwParam*,	//константа, определяющая характер передаваемой в объект информации. Значение типа HMAC_INFO устанавливает алгоритм кодирования, входную и выходную строки, которые используются алгоритмом HMAC. HP_HASHVAL- устанавливается значение хеша
CONST BYTE * *pbData*,	//представляет собой передаваемые данные. Значение хеша передается в форме строковой переменной.

DWORD *dwFlags*);

В нашем случае в качестве второго параметра выбираем HMAC_INFO, т.е. передаются данные об алгоритме кодирования, то в качестве третьего параметра нужно объявить переменную типа HMAC_INFO и передать данные по ссылке. После этого хеш можно вычислять, используя функцию *CryptHashData*. Функция *CryptGetHashParam* позволяет получить размер и значение хэша HMAC.

```
/ / HKEY: симметричным ключ для алгоритма RC4.
/ / HHmacHash: Дескриптор хэш HMAC.
/ / DwDataLen: длина хэша в байтах,
/ / Data2: Сообщение строка для хэш-таблицы.
/ / HmacInfo: Экземпляр HMAC_INFO структуру, которая содержит
        Информация о HMAC хэш.
HCRYPTHASH  hHash       = NULL;
HCRYPTHASH  hHmacHash   = NULL;
DWORD       dwDataLen   = 0;
BYTE        Data2[]     =
{0x6D,0x65,0x73,0x73,0x61,0x67,0x65};
HMAC_INFO   HmacInfo;
```

//Обнуление HMAC_INFO структуры и использовать алгоритм хеширования SHA1.

```
ZeroMemory(&HmacInfo, sizeof(HmacInfo));
HmacInfo.HashAlgid = CALG_SHA1;
;
}
```

// Создаем хеш -объект.

```
if (!CryptCreateHash(
    hProv,
    CALG_HMAC,
    hKey,
    0,
    &hHmacHash))
{
   printf("Error in CryptCreateHash 0x%08x \n",
          GetLastError());
   goto ErrorExit;
}

if (!CryptSetHashParam(
    hHmacHash,
    HP_HMAC_INFO,
    (BYTE*)&HmacInfo,
    0))
{
   printf("Error in CryptSetHashParam 0x%08x \n",
          GetLastError());
   goto ErrorExit;
}

if (!CryptHashData(
```

```
    hHmacHash,
    Data2,                    // message to hash
    sizeof(Data2),
    0))
{
   printf("Error in CryptHashData 0x%08x \n",
          GetLastError());
   goto ErrorExit;
}
```

// Выделяем память и получаем HMAC.

```
if (!CryptGetHashParam(
    hHmacHash,
    HP_HASHVAL,
    NULL,
    &dwDataLen,
    0))
{
   printf("Error in CryptGetHashParam 0x%08x \n",
          GetLastError());
   goto ErrorExit;
}

pbHash = (BYTE*)malloc(dwDataLen);
if(NULL == pbHash)
{
   printf("unable to allocate memory\n");
   goto ErrorExit;
}

if (!CryptGetHashParam(
    hHmacHash,
    HP_HASHVAL,
    pbHash,
    &dwDataLen,
    0))
{
   printf("Error in CryptGetHashParam 0x%08x \n",
GetLastError());
   goto ErrorExit;
}

// Print the hash to the console.

printf("The hash is:  ");
for(DWORD i = 0 ; i < dwDataLen ; i++)
{
   printf("%2.2x ",pbHash[i]);
}
printf("\n");

// Освобожение ресурсов
……
```

Литература

1. Щербаков Л. Ю., Домашен А. В. Прикладная криптография. Использование и синтез криптографических интерфейсов. — М: Издательско-торговый дом ,Русская Редакциям, 2003.
2. Юрий Николаев Использование Crypto API, RSDN Magazine #5-2004
3. Евгений Грищенко Создание и верификация цифровой подписи в CryptoAPI через сертификат открытого ключа, RSDN Magazine #1
4. Алексей Остапенко Хеширование, шифрование и цифровая подпись с использованием CryptoAPI и .NET, RSDN Magazine #1, http://rsdn.ru/article/crypto/cryptoapi.xml

УЧЕБНОЕ ИЗДАНИЕ

Шпинарева Ирина Михайловна
Геренко Ольга Андреевна

Методическое пособие по курсу «Защита информации в компьютерных системах»

для студентов специальности «Компьютерные системы и сети»

Издано в авторской редакции

Підп. до друку 20.05.2010. Формат 60х84/16.
Гарн. Таймс. Тираж 50 прим.

Редакційно-видавничий Центр
Одеського національного університету
імені І.І. Мечникова,
65082, м. Одеса, вул. Єлісаветинська, 12, Україна
Тел.: (048) 723 28 39