МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»


**ІНСТИТУТ ЕЛЕКТРОЕНЕРГЕТИКИ**
**Факультет інформаційних технологій**
**Кафедра безпеки інформації та телекомунікацій**

# МЕТОДИ ПОБУДОВИ ТА АНАЛІЗУ КРИПТОСИСТЕМ

**Методичні рекомендації**
**до виконання лабораторних робіт**
**з дисципліни**

**для студентів-магістрів спеціальності 125 Кібербезпека**
**галузі знань 12 Інформаційні технології**

Дніпро
НТУ «ДП»
2019

**Саксонов Г.М.**

Методи побудови та аналізу криптосистем. Методичні рекомендації до виконання лабораторних робіт з дисципліни для студентів-магістрів спеціальності 125 Кібербезпека галузі знань 12 Інформаційні технології / Упоряд.: Г.М. Саксонов, О.А. Жукова ; М-во освіти і науки України, Нац. техн. ун-т «Дніпровська політехніка». – Дніпро: НТУ, 2019. – 51 с.

Упорядники:
Г.М. Саксонов, ст. викладач;
О.А. Жукова, доцент.

Затверджено методичною комісією зі спеціальності 125 Кібербезпека (протокол № 7 від 07.03.2019) за поданням кафедри безпеки інформації та телекомунікацій (протокол № 7 від 07.03.2019).

Подано методичні рекомендації до виконання лабораторних робіт з дисципліни «Методи побудови та аналізу криптосистем» для студентів-магістрів спеціальності 125 Кібербезпека галузі знань 12 Інформаційні технології.

Відповідальний за випуск зав. кафедри БІТ В.І. Корнієнко, д-р техн. наук, проф.

## ПЕРЕДМОВА

Розвиток інформаційних технологій, їх проникнення в усі сфери людської діяльності призводить до того, що проблеми інформаційної безпеки з кожним роком стають все більш і більш актуальними - і одночасно більш складними.

На практиці інформаційна безпека зазвичай розглядається як сукупність наступних трьох базових властивостей інформації, що захищається: конфіденційність, цілісність і доступність, що гарантує безперешкодний доступ до інформації, що захищається для законних користувачів.Багато операційних систем пропонують програмістам криптографічні бібліотеки, що дозволяють створювати захищені додатки. Наприклад, *ОС Linux* пропонує бібліотеку *Openssl*. *Microsoft* ще в 1996 р починаючи з *Windows95 OSR2* і *WindowsNT SP3* включає в свої операційні системи вбудовані засоби шифрування, доступні прикладного програмісту через бібліотеку Microsoft *CryptoAPI*. Кожна з цих бібліотек мають переваги і недоліки. У цьому методичному виданні ми познайомимося з бібліотекою *CryptoAPI*.

В *ОС Windows 2000* і *Windows XP* функції *CryptoAPI* містяться в модулях *crypt32.dll* і *advapi32.dll*. Починаючи з *Windows Vista* і *Windows Server 2008* підтримується бібліотека *Cryptography API*: *Next Generation (CNG)*, покликана в майбутньому замінити *CryptoAPI*.

Створення лабораторних робіт по курсу «Методи побудови і аналізу криптосистем» ставить собі за мету:

- надання студентам необхідної інформації по виконанню лабораторних робіт;
- підготовка джерел інформації з програмним забезпеченням для установки і настройки всього необхідного для виконання лабораторних робіт;
- надання студентам зразкових алгоритмів виконання лабораторних робіт.
- В результаті виконання даних лабораторних робіт студенти познайомляться з основами використання *Microsoft CryptoAPI* і отримають можливість створювати нескладні програми, які будуть звертатися до *CryptoAPI*.

## Лабораторна робота 1.

## Криптографічний захист даних за допомогою *MICROSOFT CRYPTOAPI*

**Мета роботи:**

Вивчення основних функцій бібліотеки *CRYPTOAPI* необхідних для захисту конфіденційності документа. Ознайомлення з функціями шифрування і дешифрування документів, генерацією ключів і робота з сертифікатами *X509*.

**Результат:**

Створення програмного додатка для захисту документів за допомогою шифрування.

**Завдання для самостійного виконання.** Перед виконанням завдання студент, використовуючи програму *openssl*, створює на своєму комп'ютері локальний центр сертифікації. Потім створює ключову пару *RSA <KO, KC>*. На підставі відкритого ключа *KO* створює запит на сертифікацію і передає його в локальний центр сертифікації. Отримує з локального центру сертифікації сертифікат X509 свого відкритого ключа *KO* і сертифікат *X509* відкритого ключа центру сертифікації. Сертифікат *X509* свого відкритого ключа необхідно розмістити в сховище *WINDOWS*.

### Задача

Використовуючи криптографічні інтерфейси *MS CryptoAPI* 1.0 (2.0), створити додаток, що дозволяє шифрувати і розшифровувати файли за наступною схемою. Щоб зашифрувати файл треба:

- створити сеансовий ключ *KS* симетричного алгоритму (алгоритм визначається варіантом завдання);
- зашифрувати файл на ключі *KS*;
- зашифровать сеансовий ключ відкритим ключем одержувача шифрованого файлу і додати до зашифрованого файлу зашифрований сеансовий ключ. Відкритий ключ одержувача витягується з сертифіката одержувача.

Щоб розшифрувати файл треба:

- витягти з файлу зашифрований сеансовий ключ *KS* і розшифрувати його, використовуючи секретний ключ одержувача. Секретний ключ одержувача витягується з сертифіката одержувача.
- розшифрувати файл сеансовим ключем *KS*.

**Варіанти завдань для роботи №1**

| **№ варіанта** | **Алгоритм и режим шифрування** |
|---|---|
| 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 |

Варіант завдання визначається так: $(n_1n_2+11)$ *mod* 27 +1, де $n_1n_2$- дві останні цифри номера залікової книжки.

**Методичні вказівки до першої частини лабораторної роботи.**

Концепція *CryptoAPI* має на увазі приховування від програміста всіх тонкощів процесу шифрування даних.У CryptoAPI можна виділити п'ять основних функціональних областей. Функції, що входять в кожну з областей, як правило, містять в імені відповідні ключові слова:

- базові криптографічні функції (Crypt). У цю групу входять функції для підключення до кріптопровайдеру, генерації і зберігання ключів, обміну ключами, управління параметрами ключа;
- функції кодування / декодування (Crypt) забезпечують доступ до алгоритмів шифрування / дешифрування і хеширования;
- функції роботи з сховищем сертифікатів (Store) призначені для управління наборами цифрових сертифікатів, використовуваних при цифрового підпису та шифрування з відкритим ключем;
- спрощені функції для роботи з повідомленнями (Message) дозволяють шифрувати / дешифрувати повідомлення і блоки даних, підписувати їх і перевіряти цифровий підпис;
- низькорівневі функції для роботи з повідомленнями (Msg) дозволяють більш гнучко виконати ту ж роботу, що і спрощені функції, але вимагають для цього великих зусиль з боку програміста. Для використання функцій CryptoAPI в додатках, написаних на C / C ++, необхідно використовувати заголовки *WinCrypt.h* і підключити до додатка бібліотеку імпорту *Crypt32.lib*.

**Криптопровайдери *CryptoAPI***

Одним з ключових понять *CryptoAPI* є криптографічний провайдер (*CSP*, *Cryptographic Service Provider*), який реалізує базовий набір криптографічних функцій, що відповідають тому чи іншому криптографічному алгоритму (або сімейству алгоритмів). Криптопровайдери реалізовані у вигляді окремих *DLL*, так, що сторонні розробники можуть самостійно включати до складу *CryptoAPI* реалізації своїх алгоритмів (хоча для поширення криптопровайдера потрібно підписати його цифровим підписом в *Microsoft*). Одночасно в операційній системі можна встановити кілька CSP.Вибір конкретного алгоритму задається

параметрами функції і залежить від використовуваного CSP, а універсальний набір функцій визначено для кожного типу криптографічних операцій.

*CryptoAPI* надає наступні стандартні криптопровайдери:

- *Microsoft Base Cryptographic Provider*. Визначення: *MS_DEF_PROV*. Бібліотека: *rsabase.dll;*
- *Microsoft Strong Cryptographic Provider* (поставляється починаючи з *Windows 2000*, є розширенням базового). Визначення: *MS_STRONG_PROV*. Библиотека: *rsaenh.dll;*
- *Microsoft Enhanced Cryptographic Provider* (в порівнянні з базовим забезпечує роботу з ключами більшої довжини). Визначення: *MS_ENHANCED_PROV*. Библиотека: *rsaenh.dll;*
- *Microsoft AES Cryptographic Provider* (підтримка *AES*).
- *Microsoft DSS Cryptographic Provider* (реалізує алгоритми *SHA* і *DSS*). Визначення: *MS_DEF_DSS_PROV*. Бібліотека: *dssbase.dll;*
- *Microsoft Base DSS and Diffie-Hellman Cryptographic Provider* (додана підтримка алгоритма обміну ключей Диффі-Хеллмана). Визначення: *MS_DEF_DSS_DH_PROV*. Бібліотека: *dssbase.dll;*
- *Microsoft DSS and Diffie-Hellman/Schannel Cryptographic Provider* (додана аутентифікація клиентів по захищеному протоколу взаємодії). Визначення: *MS_DEF_DH_SCHANNEL_PROV*. Бібліотека: *dssenh.dll;*
- *Microsoft RSA/Schannel Cryptographic Provider* (реалізація алгоритмів *RSA*).

**Типи криптопровайдерів**

Всі *CSP* відрізняються один від одного своїми типами, які визначаються набором параметрів, що включає:

- алгоритм обміну сесійним (симетричним) ключем;
- алгоритм обчислення цифрового підпису;
- формат цифрового підпису;
- схема генерування сесійного ключа по хешу;
- довжина ключа.

У файлі *WinCrypt.h* дано визначення типів криптопровайдерів, визначених у даний час. Найбільш відомі типи *CSP* представлені в таблиці 1.1.

Таблица 1.1

Описание типов *CSP*

| Тип криптопровайдера | Підтримувані алгоритми |
|---|---|
| PROV_RSA_FULL(1) | ключовий обмін — RSA;<br>цифровий підпис — RSA;<br>шифрування — RC2, RC4;<br>хешування — MD5, SHA. |
| PROV_RSA_SIG (2) | ключовий обмін — не підтримується;<br>цифровий підпис — RSA;<br>шифрування — не підтримується;<br>хешування — MD5. SHA. |
| PROV_RSA_SCHANNEL(12) | ключовий обмін — RSA;<br>цифровий підпис — RSA;<br>шифрування — RC4. DES, Triple DES (не обов'язково всі);<br>хешування — MD5, SHA. |
| PROV_DSS (3)<br>PROV_RSA_SIG | ключовий обмін — не підтримується;<br>цифровий підпис — DSS(DSA);<br>шифрування — не підтримується;<br>хешування — MD5, SHA. |
| PROV_DSS_DH(13) | ключовий обмін — DH;<br>цифровий підпис — DSS;<br>шифрування — CYLINK МЕК;<br>хешування — MD5, SHA. |
| PROV_DH_ SCHANNEL (18) | ключовий обмін — DH (Ephemeral);<br>цифровий підпис — DSS; |

| | |
|---|---|
| | шифрування — DES, Triple DES;<br>хешування — MD5, SHA. |
| PROV_FORTEZZA (4) | ключовий обмін — KEA;<br>цифровий підпис — DSS;<br>шифрування— Skipjack;<br>хешування— SHA. |
| PROV_MS_EXCHANGE (5) | ключовий обмін — RSA;<br>цифровий підпис — RSA;<br>шифрування — CAST;<br>хешування — MD5. |
| PROV_SSL (6) | ключовий обмін — RSA;<br>цифровий підпис — RSA;<br>шифрування — різні алгоритми;<br>хешування — різні алгоритми. |

Функції роботи с криптопровайдерами можна розділити на наступні групи:

- функції ініціалізації контексту і отримання параметрів криптопровайдера;
- функції генерації ключів і роботи з ними;
- функції шифрування / розшифрування даних;
- функції хешування і отримання цифрового підпису даних.

Кожен CSP має свою базу ключів (*key database*), в якій знаходиться один або кілька контейнерів (*key containers*), де зберігаються пари закритих / відкритих ключів. Контейнери бувають двох типів - призначені для користувача (цей тип використовується за умовчанням) і машинні (*CRYPT_MACHINE_KEYSET*). Тип контейнера задається прапором при отриманні контексту провайдера. На комп'ютері зазвичай встановлено кілька провайдерів і відповідних типів криптопровайдерів. Для того, щоб переглянути всі встановлені криптопровайдери і типи треба застосувати відповідні функції

*CryptEnumProviders* і *CryptEnumProviderTypes*. Приклад перегляду встановлених криптопровайдерів на комп'ютері:

```
#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 типам провайдерів, що перераховуються.
	dwIndex = 0;
	while(CryptEnumProviderTypes(
		dwIndex,
		NULL,
		0,
		&dwType,
		NULL,
		&cbName
		))
	{
		//Розподіл пам'яти в буфері для відновлення цього імені.
		pszName = (LPTSTR) malloc(cbName);
		if(!pszName)
			puts("ERROR - malloc failed!");
```

```
        memset(pszName, 0, cbName);
        //  Отримання імені типу провайдера.
        if(CryptEnumProviderTypes(
             dwIndex++,
             NULL,
             0,
             &dwType,
             pszName,
             &cbName))
        {
             printf ("   %4.0d ",dwType);
             wcout<<pszName<<endl;
        }
        else
        {
             puts("ERROR - CryptEnumProviders");
        }
}
//   Друк заголовка перерахунка провайдерів.
printf("\n\n          Listing Available Providers.\n");
printf("Provider type      Provider Name\n");
printf("_____________________________________\n");

// Цикл по провайдерам, що перераховуються.
dwIndex = 0;
while(CryptEnumProviders(
        dwIndex,
        NULL,
        0,
        &dwType,
        NULL,
        &cbName
        ))
{
        //Розподіл пам'яти в буфері для відновлення цього імені.
        pszName = (LPTSTR)malloc(cbName);
```

```
            if(!pszName)
                puts("ERROR - malloc failed!");
            memset(pszName, 0, cbName);
            //  Отримання імені провайдера.
            if(CryptEnumProviders(
                dwIndex++,
                NULL,
                0,
                &dwType,
                (LPWSTR)pszName,
                &cbName     // pcbProvName -- длина pszName
                ))
            {
                printf ("     %4.0d        ",dwType);
                wcout<<pszName<<endl;
            }
            else
            {
                puts("ERROR - CryptEnumProviders");
            }
        } // Кінець цикла while
        printf("\nProvider types and provider names have been
listed.\n");

    }
```

При роботі з криптопровайдерами додатки мають ряд обмежень:

Додаток не може безпосередньо скористатися реалізаціями функцій з криптопровайдера, і все взаємодія проходить через базові криптографічні функції. В ідеальному випадку можлива зміна одного криптопровайдера на іншого без модифікації використовує його застосування, хоча на практиці деякі додатки можуть зажадати функціональності певного криптопровайдера.

- Додаток не має прямого доступу до ключів, які використовуються функціями криптопровайдера.

- Додатки не можуть впливати на виконання криптографічних операцій інакше як завданням алгоритму і вибором ключа.

- Додатки не відповідають за аутентифікацію користувачів, всю цю роботу виконує криптопровайдер.

- Для отримання доступу до кріптопровайдеру використовується функція *CryptAcquireContext*, по завершенні роботи необхідно викликати функцію *CryptReleaseContext*.

*CryptAcquireContext* виконує підключення до кріптопровайдеру з заданим типом і ім'ям і повертає його дескриптор (контекст). Таким чином, сеанс роботи починається з ініціалізації (отримання контексту) за допомогою функції *CryptAcquireContex*t. Як параметри ця функція приймає ім'я контейнера ключів, ім'я криптопровайдера, тип провайдера і прапори, що визначають тип і дії з контейнером ключів і режим роботи криптопровайдера.

*BOOL WINAPI CryptAcquireContext*(

*HCRYPTPROV* phProv*,

*LPCTSTR pszContainer*, // им'я контейнера ключів або *NULL* - за замовчуванням

*LPCTSTR pszProvider*, // вказують им'я одного або кілька имен криптопровайдерів

*DWORD dwProvType*, // тип криптопровайдера

*DWORD dwFlags*);// 0 або *CRYPT_NEWKEYSET*

Розглянемо приклад підключення до кріптопровайдеру:

```
//вказуємо им'я і тип криптопровайдера і відчиняємо або
ствоюємо контейнер для ключів
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 вдається отримати контекст\n");
          return -1;
```

```
}
```

Можливий інший варіант, коли криптопровайдер вибирається за замовчуванням, а тип *PROV_DH_SCHANNEL*:

```
HCRYPTPROV hProv;
     //намагаємося відкрити існуючий keyset або створити новий
if(!CryptAcquireContext(&hProv, NULL,
NULL,PROV_DH_SCHANNEL,0) &&
(!CryptAcquireContext(&hProv,NULL,NULL,PROV_DH_SCHANNEL,CRYPT
_NEWKEYSET))
     {
          puts("Не вдається створити контекст\n");
          return -1;
     }
```

Функції *CryptSetProvParam* и *CryptGetProvParam* дозволяют відповідно задати та дізнатися ряд характеристик криптопровайдера.

### Робота з сертифікатами *X509*

Алгоритми шифрування з відкритим ключем і цифрові підписи тісно пов'язані з поняттям сертифіката. Цифровий сертифікат видається якоюсь авторизованої організацією (*Certification Authority*, *CA*) і крім пари публічний / закритий ключ містить інформацію, що дозволяє ідентифікувати його власника. Крім цього, сертифікат має термін дії і підписується за допомогою іншого ключа *CA*, який використовується для забезпечення гарантії достовірності цих атрибутів і, що найбільш важливо, самого відкритого ключа. *CryptoAPI* підтримує сертифікати, відповідні специфікації *X.509* (поточна версія 3), описані в документі *RFC 3280*. Стандарт *X.509* не визначає обов'язкового типу ключа, вбудованого в сертифікат, але в даний час алгоритм *RSA* є найбільш відомим з застосовуваних криптографічних алгоритмів з асиметричними шифрами.Існує можливість створювати власні сертифікати. Метод їх створення зазвичай залежить від способу їх застосування. Для звичайних ситуацій в Інтернеті, коли користувач не знає, з ким він зв'язується, запитується, як правило, сертифікат від комерційного центру сертифікації. Перевагою такого підходу є те, що ці

відомі центри сертифікації вже визнані довіреними операційною системою Windows і всіма іншими операційними системами (і оглядачами), що підтримують сертифікати і протокол *SSL*. Це дозволяє обходитися без обміну ключами центру сертифікації.Можна створювати сертифікати за допомогою програми *OpenSSL*. *OpenSSL* підтримує два основні формати, які визначають кодування криптографічних документів: *DER* - бінарні файли і *PEM* - текстові файли.

Криптографічний документ в форматі *PEM* має вигляд:

*----- BEGIN <ТИП ДОКУМЕНТА>*

*-----<Документ в форматі DER, закодований в base64>-*

*---- END <ТИП ДОКУМЕНТА> -----*

Тобто зміст документа оточується заголовками спеціального виду, що складаються з п'яти знаків «-», слів *begin* і *end* і вказівок типу документа (*CERTIFICATE, CRL* і т.д.) Завдяки наявності таких заголовків в одному файлі може міститися кілька документів в форматі *PEM*: наприклад, ланцюжок сертифікатів або сертифікат і його закритий ключ. За замовчуванням в *OpenSSL* використовується саме формат *PEM*. Файли, що містять ланцюжка сертифікатів, сертифікати та асоційовані з ними ключі, можуть представлятися в декількох форматах, що визначають склад і будова файлів, що містять ключову інформацію; сама ця інформація кодується в вищеописаних форматах *PEM* і *DER*.

**Найбільш поширені з форматів, що визначають структуру файлів, що містять криптографічний інформацію - *PKCS # 7* і *PKCS # 12*. *PKCS # 7* - формат захищених повідомлень, який, крім самого повідомлення, може містити необхідні для роботи з ним сертифікати і *CRL*. Багато з центрів, що засвідчують, поширюють свої сертифікати і *CRL* у вигляді *PKCS # 7* документів, в яких немає повідомлення, є тільки службова інформація. *PKCS # 7* - документ може бути закодований і в форматі *DER*, і в форматі *PEM*. Розширення файлів сертифікатів можуть бути *.cer* або *.crt*. *PKCS # 12* - формат для перенесення сертифіката і пов'язаного з ним закритого ключа**

**з машини на машину або для резервного копіювання. У цьому форматі можуть також міститися сертифікати центру, що засвідчує. Файли формату *PKCS # 12* завжди кодуються в форматі *DER*. Ці файли вимагають особливо пильної уваги, оскільки містять закритий ключ; при необережному поводженні з таким файлом закритий ключ легко скомпрометувати. Як правило, ці файли захищені на паролі. Розширення файлів формату *PKCS # 12* або *.p12*, або *.pfx*. У Windows зручно працювати з сертифікатом даного формату. При виконанні експорту пари ключів *Windows* пропонує зашифрувати *PFX*-файл за допомогою пароля; при імпорті пари ключів користувач повинен знову надати цей пароль.**

**Сховище сертифікатів Windows**

Сертифікати та відповідні їм закриті ключі можна зберігати на різних пристроях, наприклад жорстких дисках, смарт-картах і в ключах для порту *USB*. У *Windows* передбачений рівень абстракції, званий сховищем сертифікатів, який служить для забезпечення єдиного способу доступу до сертифікатів незалежно від місця їх зберігання. До тих пір поки у апаратного пристрою є підтримуваний *Windows* - постачальник служби криптографії (*CSP*) - можна отримувати доступ до даних, що зберігаються на цьому пристрої, використовуючи інтерфейс *API* сховища сертифікатів. Сховище сертифікатів глибоко заховане в профілі користувача. Це дозволяє використовувати списки управління доступом (*ACL*) для ключів конкретного облікового запису. Кожне сховище розділено на контейнери:

Контейнер з назвою *Personal* (особистий), призначений для зберігання призначеного для користувача сертифіката (тобто, у яких є відповідний закритий ключ).

У контейнері *Trusted Root Certification Authorities* (довірені кореневі центри сертифікації) зберігаються сертифікати всіх центрів сертифікації, яким довіряє користувач.

У контейнері *Other People* (інші) містяться сертифікати колег, з якими є безпечний зв'язок. І так далі.

Найпростіший спосіб отримати доступ до свого сховища сертифікатів полягає у виконанні команди *certmgr.msc*. Існує також сховище на рівні комп'ютера, що використовується обліковими записами комп'ютера з ОС *Windows* (мережі, локальної служби і локальної системи) або служить для спільного використання сертифікатів і ключів різними обліковими записами.

Будь-який сертифікат *X.509* одержуваний користувачем поштою або за запитом з центру сертифікації необхідно помістити в сховище сертифікатів в *WINDOWS*. Для цього виконують подвійне клацання миші по файлу сертифіката і потім в діалоговому вікні вибир

## Робота з сертифікатом X.509 за допомогою CryptoAPI

Алгоритм роботи з сертифікатом складається з виклика наступних функцій:

- *CertOpenStore* - відкриваєм сховище сертифікатів.
- *CertOpenSystemStore* відкриваєм сховище системних сертифікатів.
- *CertFindCertificateInStore* – пошук потрібного сертифікату.
- *CertEnumCertificatesInStore*- перебирає всі сертифікати в сховищі.
- *CertGetNameString*- витягує им'я поточного сертифікату.

Приклад роботи з сертифікатом:

```
// Сертификат, наведений в форматі 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)

//Назва персонального сховища
#define CERT_STORE_NAME  L"MY"
```

```
//Назва сертифікату, встановленого в це сховище
#define SIGNER_NAME  L"EVGENY"

//Відкриваємо сховище сертифікатів
  HCERTSTORE hStore;

  if ( !( hStore = CertOpenStore(
    CERT_STORE_PROV_SYSTEM,
    0,
    NULL,
    CERT_SYSTEM_STORE_CURRENT_USER,
    CERT_STORE_NAME)))
  {
       puts("Неможна відкрити сховище MY ");

    }

//Отримуємо покажчик на наш сертифікат
  PCCERT_CONTEXT pSignerCert=0;

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

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

```
    //За допомогою даної функції виведемо ім'я CSP та ім'я
контейнера ключів
    DWORD dwUserNameLen = 100;
      CHAR szUserName[100];

      if(CryptGetProvParam(
        hProv,              // Дескриптор на CSP
        PP_NAME,            // параметр для отримання імені CSP
        (BYTE *)szUserName, // Покажчик на буфер, що містить ім'я
CSP
        &dwUserNameLen,     // длина буферу
        0))
      {
        printf("Name CSP: %s\n",szUserName);
      }
      else
      {
        puts("Помилка CryptGetProvParam.\n");
      }

      if(CryptGetProvParam(
        hProv,              // Дескриптор на CSP
        PP_CONTAINER,      // параметр для отримання імені key
container
        (BYTE *)szUserName, // Покажчик на буфер, що містить ім'я
key container
        &dwUserNameLen,     // длина буферу
        0))
      {
        printf("Name key container: %s\n",szUserName);
      }
      else
      {
        puts("ErrorCryptGetProvParam.\n");  }
```

Приклад, що дозволяє проглянути всі сертифікати в сховищі.

```
     //Відкриваємо сховище сертификатів
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;
     //перебираємо і виводимо всі імена сертификатів
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;
     //отримуємо зі сховища выбраный сертификат
     for(i=0;i<certNum;i++)
     {
     hContext=CertEnumCertificatesInStore(hStore,hContext);
     }
     delete [] pszName;
     //перевіряємо як отримали сертификат
if(hContext==NULL){cout<<"Error getting certificate\n";
     cin.get();return 1;}
```

Всі ключі в *CRYPTOAPI* управляються і використовуються за допомогою якихось ідентифікаторів, і додаток не дістає відкритого доступу до них. При підключенні до криптопровайдеру він може надати доступ до стандартного сховища ключів, або скористатися сховищем, створеним по запиту додатку.

Отримавши покажчик на сертифікат X.509, тепер можна витягувати відкритий ключ за допомогою функції *CryptImportPublicKeyInfo.* Надалі, отримавши ідентифікатор відкритого ключа можна використовувати його для

обміну ключами (експорту сесійного ключа) або для перевірки електронно-цифрового підпису.

Приклад імпорту відкритого ключа з сертифікату:

```
    // Імпортуємо public key для подальшої верифікації підпису
або шифрування сеансового ключа
      HCRYPTKEY hPublicKey;
      if(CryptImportPublicKeyInfo(
        hProv,
        MY_TYPE,
        &(pSignerCert->pCertInfo->SubjectPublicKeyInfo),
        &hPublicKey))
      {
        printf("Import public key.\n");
      }
      else
      {
        printf ("Помилка CryptAcquireContext.");
    }
```

Закритий ключ, відповідний сертифікату відкритого ключа, зберігається в спеціальному контейнері ключів. Для роботи з даним ключем необхідно взятися до роботи з CSP і контейнером нашого сертифікату через функцію *CryptAcquireCertificatePrivateKey*. А потім за допомогою функції *CryptGetUserKey* набути значень закритого ключа з вказаного контейнера ключів. Даний ключ може використовуватися для обміну ключами (імпорту сесійного ключа) або для створення цифрового підпису.

Розглянемо приклад витягання секретного ключа, відповідного сертифікату відкритого ключа PKSC#12:

```
HCRYPTKEY hPrivateKey=0;
DWORD keySpec=0;
// Витягуємо з сертифікату контекст приватного ключа
```

```
if(!CryptAcquireCertificatePrivateKey(
     pSignerCert,
     0,
     NULL,
     &hProv,
     &keySpec,
     NULL))
      { cout<<"Error getting private context\n";
      getchar();
      return -1;
      }
//   Витягуємо закритий ключ
if(!CryptGetUserKey(hProv,keySpec,&hPrivateKey))
       {cout«"Error getting private key\n";
        getchar ();
        return -1;
       }
```

**Робота з ключами шифрування за допомогою *CRYPTOAPI***

*CRYPTOAPI* дозволяє шифрувати дані, розміщені в оперативній пам'яті у вигляді послідовності байтів. Для шифрування можуть бути використані як симетричні, так і асиметричні алгоритми. *CRYPTOAPI* надає методи для роботи з сесійними (симетричними) ключами і з відкритими ключами. Асиметричні алгоритми використовуються для обміну сеансовими ключами. Зашифрувавши повідомлення електронної пошти симетричним алгоритмом, ви додаєте до зашифрованого повідомлення сам сеансовий ключ, зашифрований на відкритому ключі одержувача з використанням асиметричного алгоритму. Це дозволяє вам знищити сеансовий ключ відразу ж після шифрування повідомлення. Навіть якщо ваш комп'ютер піддасться злому, зловмисник все одно не дізнається зміст повідомлення.

У *CRYPTOAPI* генерація ключів може виконуватися випадковим чином за допомогою функції *CryptGenKey* або на основі деяких даних, наприклад,

призначеного для користувача пароля за допомогою функції *CryptDeriveKey*. Після закінчення роботи з ключем він знищується функцією *CryptDestroyKey*.

У *CRYPTOAPI* алгоритм шифрування повідомлення указується при генерації сесійного ключа в *CryptGenKey*:

BOOL
CryptGenKey (
HCRYPTPROV
hProv,
ALG_ID Algid, // ідентифікатор алгоритму шифрування, для якого генерується ключ (наприклад, *CALG_3DES*). Для ключової пари *RSA* для шифрування і підпису використовуються *AT_KEYEXCHANGE і AT_SIGNATURE*.
DWORD dwFlags, // задає різні опції ключа, які залежать від алгоритму і провайдера. *CRYPT_EXPORTABLE*- для експорту сесійного ключа. *CRYPT_ENCRYPT* - для шифрування.
HCRYPTKEY* //ідентифікатор ключа.
phKey);

Приклад генерації випадкового сесійного ключа:

```
HCRYPTKEY hKey;
//генеруємо сесійний ключ для DES
if(!CryptGenKey(
     hProv,
     CALG_DES,
     CRYPT_EXPORTABLE|CRYPT_ENCRYPT,
     &hKey))
     {
          puts("Не вдається створити ключ DES\n");
          return -1;
     }
```

## Параметри ключа

Після створення ключа його параметрами можна маніпулювати за допомогою функцій *CryptGetKeyParam()* і *CryptSetKeyParam()*. Ці функції надають доступ до таких атрибутів, як алгоритм створення ключа, довжина шифрованого блоку, що ініціює значення, початковий вектор, режим буферизації і режим шифрування, а також допуски, які задають операції, допустимі для даного ключа.

Алгоритми симетричного шифрування мають наступні режими шифрування повідомлень:

CRYPT_MODE_ECB - режим простій заміни;

CRYPT_MODE_CBC - режим зчеплення блоків шифру;

CRYPT_MODE_OFB - режим гамміровання;

CRYPT_MODE_CFB - режим гамміровання із зворотним зв'язком.

За умовчанням в *CRYPTOAPI* блокові шифри використовуються в режимі зчеплення блоків шифру (*CBC - cipher block chaining*). У цьому режимі використовується вектор, що ініціалізував (*IV - initialization vector*). За умовчанням *IV* нульовий. Вектор, що ініціалізував, повинен генеруватися окремо за допомогою функції *CryptGenRandom* і, як і солт-значення, передаватися разом з ключем у відкритому вигляді. Розмір *IV* рівний довжині блоку шифру. Наприклад, для алгоритму *RC2*, підтримуваного базовим криптопровайдером *Microsoft*, розмір блоку складає 64 бита (8 байтів).

Якщо режим шифрування даних повинен бути не *CBC,* то треба встановити відповідні параметри за допомогою функції *CryptSetKeyParam.*

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

DWORD dwFlags);

Якщо ключ *hKey* сесійний, то як параметр можна встановити початковий вектор *KP_IV*, режим шифрування *KP_MODE*, спосіб доповнення *KP_PADDING*.

```
// Встановлюємо режим шифрування повідомлення OFB

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

## Пересилка ключів в CRYPTOAPI

Як ключі експорту/імпорту можуть використовуватися або ключова пара *RSA* (з типом *AT_KEYEXCHANGE*), або симетричний сеансовий ключ. Обмін ключами реалізується за допомогою функцій *CryptExportKey* і *CryptImportKey***.**

Функція експорту ключа для його передачі по каналах інформації

BOOL WINAPI CryptExportKey(

HCRYPTKEY hKey,

HCRYPTKEY hExpKey,

DWORD dwBlobType,// PUBLICKEYBLOB-експорт відкритого ключа

// PRIVATEKEYBLOB –експорт закритого ключа

// SIMPLEBLOB-експорт сеансового ключа

DWORD dwFlags,

BYTE* pbData, //буфер, що містить ключовий блоб

(зашифрований ключ)

DWORD* pdwDataLen); // розмір ключового блоба.

Виконаємо експорт сесійного ключа за допомогою функції *CryptExportKey* і запишемо результат у файл:

```
//Визначуваний розмір Bloba сесійного ключа
DWORD dwBlobLenght = 0;
if(CryptExportKey(hKey,hPublicKey,SIMPLEBLOB,0,0,&dwBlobLengh
t))
{
 printf("size of the Blob");
}
else
{
 printf("error computing Blob length");
 getchar();
 return -1;
}
//Розподіляємо пам'ять для сесійного 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;
          }
//Зашифруємо сесійний ключ hKey відкритим 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;   }
//Записуємо експортований ключ у 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;
 }
```

Функція, призначена для отримання з каналів інформації значення ключа.

```
BOOL
CryptImportKey
        (HCRYPTPROV
              hProv,
        BYTE* pbData,        //ключовий блоб (зашифрований ключ,
                             отриманий в результаті дії функції
                             CryptExportKey)
  DWORD dwDataLen,           // довжина ключового блоба (у байтах)
        HCRYPTKEY            //дескриптор ключа, яким дешифруємо
              hImpKey,
     DWORD dwFlags,
        HCRYPTKEY*           // адреса, по якій функція копіює
              phKey);        дескриптор мпортованого ключа
```

Приклад імпорту сесійного ключа:

```
//Розподіляємо пам'ять для сесійного 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;
                 }
     //Прочитуємо сесійний  ключ з файлу 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;
       }
     //Імпортуємо сесійний ключ за допомогою закритого ключа
ассиметричного алгоритму
     hkey=0;
     if(CryptImportKey(hProv,ppbKeyBlob,dwBlobLenght,hPrivateKey,0
,
     &hKey))
     {
     printf(" the key has been imported.\n");
     CryptDestroyKey(hPrivateKey); //очищаємо ресурси
     free(ppbKeyBlob);
     }
     else
     {
```

```
printf("the session key import failed.\n");
getchar();
return -1;}
```

**Симетричне шифрування**

До групи функцій шифрування/розшифровки даних входять:

– *CryptEncrypt*. Основна базова функція шифрування даних. Як параметри використовує раніше отримані контексти криптопровайдера і сесійного ключа. Дані, що генеруються на виході цієї функції, не є форматованими і не містять ніякий іншій інформації, крім шифрованого контексту.

– *CryptDecrypt*. Основна базова функція розшифровки даних. Як параметри використовуються раніше отримані контекст криптопровайдера і ідентифікатор сесійного ключа.

Базова функція шифрування даних має наступне оголошення:

```
BOOL
CryptEncrypt(
    HCRYPTKEY       //ідентифікатор сесійного ключа
        hKey,
    HCRYPTHAS       // використовується для отримання хеш-
        hHash,  кодування даних.
    BOOL Final,     //дозволяє обробляти дані блоками
    DWORD           //служить покажчиком на масив
        dwFlags,  входных/выходных даних. Зазвичай встановлюється
                  в 0.
    BYTE* pbData,   //порція даних для шифрування
    DWORD*          //служить для повернення розміру даних,
        pdwDataLen,  повертаних функцією.
    DWORD           // служить для вказівки довжини вхідного
        dwBufLen);  буфера даних
```

Приклад використання функції CryptEncrypt приведений нижче:

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

// Визначуваний розмір буфера необхідного для блоків довжини BLOCK SIZE
buflen=BLOCKSIZE;
if(!CryptEncrypt(hKey, 0,TRUE,0,NULL,&buflen, 0 ))
{
cout<<" Crypt Encrypt (bufSize) failed."«endl;
getchar ();
 return -1;
}
//Виділимо пам'ять під буфер
pCryptBuf = (BYTE*)malloc( buflen );
int t=0;

//   Шифруємо 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);
```

Приклад розшифровки файлу за допомогою функції *CryptDecrypt*.

```
// Визначуваний розмір буфера необхідного для блоків довжини BLOCK SIZE

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

// Розшифровуємо 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;
```

## Лабораторна робота 2. Електронно-цифровий підпис.

**Мета роботи:**

Вивчення основних функцій бібліотеки *CRYPTOAPI*, необхідних для забезпечення цілісності документа і захисту авторських прав.

Уміти користуватися функціями хешування, функціями створення і перевірки електронно-цифрового підпису документа.

**Результат:**

Створення програмного застосування для захисту документів за допомогою електронно-цифрового підпису (ЕЦП).

**Завдання для самостійного виконання**

Перед виконанням завдання студент, використовуючи програму *openssl*, створює на своєму комп'ютері локальний центр сертифікації. Потім створює ключову пару *RSA <KO,KC>*. На підставі відкритого ключа *KO* створює запит на сертифікацію і передає його в локальний центр сертифікації. Отримує з локального центру сертифікації сертифікат *X509* свого відкритого ключа *KO* і сертифікат *X509* откритого ключа центру сертифікації.

**Задача 1**

Використовуючи криптографічні інтерфейси *MS CRYPTOAPI 1.0 (2.0),* створити додаток, що дозволяє підписувати файли і перевіряти підпис підписаних файлів по наступній схемі.

– Секретний ключ КС для створення ЕЦП витягується з сертифікату суб'єкта, що підписує файл.

– ЕЦП зберігається в окремому файлі разом з сертифікатом відкритого ключа того, що підписав.

– Щоб перевірити ЕЦП, спочатку треба перевірити сертифікат відкритого ключа того, що підписав.

**Задача 2**

Використовуючи криптографічні інтерфейси *MS CRYPTOAPI 1.0 (2.0)*, створити додаток, що дозволяє підписувати файли і перевіряти підпис підписаних файлів по наступній схемі.

– Секретний ключ КС для створення ЕЦП витягується з сертифікату суб'єкта, що підписує файл.

– ЕЦП зберігається в окремому файлі.

– Щоб перевірити ЕЦП, треба мати сертифікат відкритого ключа того, що підписав.

**Варіанти завдань**

| 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 |

Варіант завдання визначається так*:* $(n_1n_2+11)\ mod\ 27\ +1$, де $n_1n_2$ – дві останні цифри номера залікової книжки.

Зауваження. Для успішного виконання роботи використовуємо сертифікати формату *PKCS#12*, тобто з розширенням *.p12* або *.pfx.*

**Схема створення ЕЦП полягає :**

– створення хеш-образа повідомлення;

– шифрування хеш-образа асиметричним алгоритмом за допомогою закритого ключа.

**Перевірка ЕЦП полягає в наступних етапах:**

– розшифрування ЕЦП відкритим ключем;

– створення хеш-кодування – образу повідомлення;

– порівняння хеш-образов, отриманих на 1 і 2 етапах.

**Обчислення хеш-кодувань за допомогою CRYPTOAPI**

*CRYPTOAPI* дозволяє використовувати ряд алгоритмів хешування для роботи з цифровими підписами. Доступні наступні алгоритми:

– *MD2*, *MD4*, *MD5* — добре відомі і популярні алгоритми хешування, розроблені *RSA*. Генерують 128-бітові хэши, до використання не рекомендуються.

– *Secure Hash Algorithm* (*SHA*) — алгоритм, розроблений Національним інститутом стандартів і технологій (*National Institute of Standards and Technology*, *NIST*) и Агентством національної безпеки (*National Security Agency*, *NSA*) США. Генерує 160-бітовий хэш, використовується у поєднанні з алгоритмами *DSA* (*Digital Signature Algorithm*) та *DSS* (*Digital Signature Standard*).

– *SSL3 Client Authorization Algorithm*. Використовується для аутентифікації клієнтів, реалізований як конкатенація хэшей MD5 і SHA, підписана приватним *RSA*-ключом

Таблиця - ID алгоритмів хешування

| ID алгоритму | Опис |
|---|---|
| CALG_MD2 | Алгоритм хешування MD2 |
| CALG_MD4 | Алгоритм хешування MD4 |
| CALG_MD5 | Алгоритм хешування MD5 |
| CALG_SHA | Безпечний алгоритм хешування (Secure Hash Algorithm — SHA) US DSA |
| CALG_SHA1 | Алгоритм SHA (те ж, що і calg sha) |

Для хешування даних спочатку необхідно створити об'єкт хешування за допомогою функції *CryptCreateHash*, після чого наповнити його даними, що хешируються за допомогою функції *CryptHashData*. Після передачі останній порції даних, набути значення хеш-кодування можна функцією *CryptGetHashParam*. Після закінчення роботи об'єкт хешування віддаляється функцією *CryptDestroyHash.* Для перевірки цифрового підпису схема роботи аналогічна, вона проводиться за допомогою виклику функції *CryptVerifySignature* після заповнення об'єкту хешування даними.

Для первинної ініціалізації «порожніх» хеш-объєкту застосовують функцію *CryptCreateHash.* Дана функція має наступний опис:

```
                BOOL            //контекст, що ініціалізував
CryptCreateHash(    криптопровайдера

  HCRYPTPROV
      hProv,
    ALG_ID AlgId,           //алгоритм отримання значення хеша
  HCRYPTKEY hKey,           //необхідний лише у разі застосування
                        алгоритмів типу MAC и HMAC.
  DWORD dwFlags,            // повинен бути завжди рівний 0
    HCRYPTHASH*             //функція повертає хендл створеного їй
      phHash) ;     хеш-объекта
```

Після ініціалізації хеш-объекта можна почати передачу даних хеш-функции за допомогою виклику *CryptHashData*. Дана функція має наступний опис:

```
BOOL CryptHashData(
HCRYPTHASH hHash, //хендл хеш-объекта, що раніше ініціалізував
    BYTE* pbData,    // порція даних для хеш-функции
    DWORD dwDataLen, // довжина передаваних даних
    DWORD dwFlags);  // зазвичай рівний нулю
```

Після повної передачі всього масиву вхідних даних функції *CryptHashData* виникає необхідність в набутті значення хеш-функції. Дане завдання вирішується із застосуванням функції *CryptGetHashParam*. Дана функція має наступний опис:

```
BOOL
CryptGetHashParam(          // хендл хеш-объекта, що раніше
    HCRYPTHASH              ініціалізував
hHash,
    DWORD                   // функції визначає тип запрошуваного
dwParam,                    значення. Для отримання хеш-значения
                            необхідно передати другим аргументам
                            значення HP_HASHVAL
    BYTE* pbData,           //відповідають за блок пам'яті,
    DWORD*                  використовуваний під повертане  значення
pdwDataLen,
    DWORD                   //повинен бути рівний нулю.
dwFlags);
```

Розглянемо приклад створення хеш-образу:

```
//відкриваємо файл, вміст якого підписуємо і далі створюємо
дайджест
HCRYPTHASH hHash;
DWORD dwLen;
DWORD fSize=GetFileSize(hInFile,&fSize);
//стрвюємо хеш-об’єкт
if{!CryptCreateHash(hProv, CALG_MD5,0,0, &hHash))
{
Error("CryptCreateHash");
return 0;
}
```

```
//читання файлу
BYTE*read=new BYTE[fsize+8];
if(!::ReadFile(hInFile,read,fSize,&dwLen,NULL))
{

puts("Error reading file\n"); return 0;
 }
// Передача даних хэш-объекту.
if{!CryptHashData(hHash, read, dwLen, 0))
{
Error("CryptHashData");
return 0;
}
std::cout << "Hash data loaded" « std::endl;
// Отримання хеш-значення
 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;
```

Додаткові функції:

CryptSetHashParam. Функція використовується для установки параметрів хеш-кодування. Може бути

використана для зміни алгоритму формування хеш-кодування.

CryptDestroyHash. Функція використовується для звільнення хеш-объекта.

CryptDuplicateHash. Функція дозволяє отримати копію хеш-кодування. Використовується при передачі хеш-кодування між функціями.

**Створення і перевірка ЕЦП за допомогою *CRYPTOAPI***

Базова функція отримання підпису хеш-кодування даних має наступний опис:

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

//значення хендла хеш-об’єкту, що вже ініціалізував даними (за допомогою функції CryptHashData)

//визначає, яка саме пара ключів буде використана для формування підпису. AT_KEYEXCHANGE (пара для обміну ключами), AT_SIGNATURE (пара для формування цифрового підпису).

//значення повинно завжди бути встановлено в NULL

//зазвичай встановлюють в 0

//використовують для коректної вказівки посилання на масив вихідних даних і його розміру

Приклад створення цифрового підпису хеш-значення

```
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;
```

Для перевірки цифрового підпису хеш-значення використовується функція *CryptVerifySignature*, що має наступний опис:

BOOL CryptVerifySignature(

HCRYPTHASH hHash, //передається хендл хеш-об’єкту, що заздалегідь ініціалізував дані за допомогою функції *CryptHashData/*

BYTE* pbSignature, //відповідають за передачу значення підпису, що перевіряється

DWORD dwSigLen,

HCRYPTKEY hPubKey, //використовується для вказівки хендла публічного ключа відправника підпису (того, хто власне сформував цифровий підпис)

LPCTSTR // значення повинне бути встановлене
sDescription, в *NULL*
DWORD //зазвичай не несе корисного
dwFlags); навантаження і встановлюється в 0

Приклад перевірки підпису приведений нижче:

```
hlnFile = CreateFileA("res.txt", // им’я файлу
            GENERIC_READ,    // читання з файлу
            FILE_SHARE_READ, // сумісний доступ до файлу
            NULL,  // захисту немає
            OPEN_EXISTING,   // відкриваємо існуючий файл
          FILE_ATTRIBUTE_NORMAL,//FILE_FLAG_OVERLAPPED
          NULL // шаблону немає);

        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)) //читаємо
блок даних
{
          puts("Error reading file\n"); return 0;
}
// Передача хешируемых даних хэш-объекту.
if(!CryptHashData(hHash, read, dwLen, 0))
{
         Error("CryptHashData"); return 0;
}
```

```
    std::cout « "Hash data loaded" « std::endl;
    // Отримання хеш-значения
    count = 0;
    //визначення розміру 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;

    //Перевірка цифрового підпису
    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;
```

# Лабораторна робота 3. Аутентифікація повідомлення

**Мета роботи:**

Вивчення алгоритмів бібліотеки *CRYPTOAPI* необхідних для забезпечення аутентифікації документа.

**Результат:**

Створення програмного застосування для захисту аутентифікації документів за допомогою *HMAC*.

**Завдання для самостійного виконання**

I. Реалізуйте аутентифікацію повідомлень на основі алгоритму *HMAC* засобами *CRYPTOAPI* двома способами:

1) використовуючи криптоинтерфейс тільки для хешування значень (конкатенацію повідомлення і секретного ключа виконувати самостійно);

2) використовуючи виклик функції *CryptCreateHash* з ключовим значенням для обчислення коди аутентифікації *HMAC* засобами *CRYPTOAPI*.

II. Реалізуйте аутентифікацію повідомлень на основі алгоритму *MAC-CBC* засобами *CRYPTOAPI*.

**Варианти завдань**

| **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 | SHA | Задача | 3DES |

| | | | | |
|---|---|---|---|---|
| | I.2 | | II | |
| 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 |

Варіант завдання визначається так: $(n_1n_2+11) \ mod \ 27 \ +1$, де $n_1n_2$ – дві останні цифри номера залікової книжки.

## Обчислення хешування з використанням секретного ключа (*HMAC*)

*CRYPTOAPI* дозволяє використовувати ряд алгоритмів хешування для створення *HMAC* і *MAC CBC*. Доступні наступні алгоритми:

– *Message Authentication Code* (*MAC*). Алгоритми *MAC* схожі із звичайними алгоритмами хешування, але використовують в своїй роботі шифрування з симетричним ключем.

– *Cipher Block Chaining* (*CBC*) *MAC*. Використовує алгоритм блокового шифрування і бере його останній блок як значення хэша.

– *HMAC*. Складніша модифікація *CBC*.

Таблиця - *ID* алгоритмів аутентифікації повідомлень

| ID алгоритма | Опис |
|---|---|
| CALG_MAC | Алгоритм хешування *Message Authentication Code* (*MAC*) |
| CALG_SSL3_SHAMD5 | Алгоритм аутентифікації повідомлення *SSL3* |
| CALG_HMAC | Алгоритм хешування *HMAC* |

Процес створення *HMAC* дуже схожий на процес хешування документа, але для алгоритму *HMAC* необхідно використовувати секретний ключ. Секретний ключ створюватиметься на основі пароля. Отже, спочатку створюється хеш-кодування-об'єкт викликом функції *CryptCreateHash*, що приймає на вході контекст криптопровайдера, ідентифікатор алгоритму (*CALG_MD5* або *CALG_SHA*). Після цього обчислюємо хеш-кодування парольної фрази за допомогою функції *CryptHashData*. А потім формуємо ключ, який буде похідним від хеш-кодування - объекта пароля, функцією *CryptDeriveKey*. Нижче представлений 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};
    // Створюємо ключ хешування.
    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;
    }
```

Наступний етап полягає в створенні *HMAC*, для цього обчислюємо порожнє хеш-кодування-об'єкт викликом функції *CryptCreateHash,* що приймає на вході контекст криптопровайдера, ідентифікатор алгоритму (*CALG_MD5* або *CALG_SHA* той же, що і для секретного ключа) і дескриптор ключа (який отримали раніше для *HMAC*). Функція *CryptSetHashParam* встановлює параметри хэш-объекта.

BOOL
CryptSetHashParam(
HCRYPTHASH *hHash*, //дескриптор використовуваного хеш-об’єкту.
DWORD *dwParam*, //константа, що визначає характер інформації, яка передається в об'єкт. Значення типу *HMAC_INFO* встановлює алгоритм кодування, вхідний і вихідний рядки, які використовуються алгоритмом *HMAC*. *HP_HASHVAL* - встановлюється значення хеш-кодування.
CONST BYTE * *pbData*, //є передаваними даними. Значення хеш-кодування передається у формі строкової змінної.
DWORD *dwFlags*);

У нашому випадку як другий параметр вибираємо *HMAC_INFO*, тобто передаються дані про алгоритм кодування, то як третій параметр потрібно оголосити змінну типу *HMAC_INFO* і передати дані по посиланню. Після цього хеш-кодування можна обчислювати, використовуючи функцію *CryptHashData*. Функція *CryptGetHashParam* дозволяє отримати розмір і значення хеша *HMAC*.

```
    / / HKEY: симетричним ключ для алгоритму RC4.
  / / HHmacHash: Дескриптор хеш HMAC.
  / / DwDataLen: длина хеша в байтах,
  / / Data2: Повідомлення-рядок для хеш-таблиці.
  / / HmacInfo: Екземпляр HMAC_INFO структура, яка містить
    інформацію про HMAC хеш.
     HCRYPTHASH  hHash       = NULL;
     HCRYPTHASH  hHmacHash   = NULL;
     DWORD       dwDataLen   = 0;
```

```
    BYTE        Data2[]     =
{0x6D,0x65,0x73,0x73,0x61,0x67,0x65};
    HMAC_INFO   HmacInfo;
    //Обнулення HMAC_INFO структури і використовувати
алгоритм хешування SHA1.
    ZeroMemory(&HmacInfo, sizeof(HmacInfo));
    HmacInfo.HashAlgid = CALG_SHA1;
    ;
    }
    // Створюємо об’єкт хеш-кодування.
    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;
}
// Виділяємо пам'ять і отримуємо 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");

     // Звільнення ресурсів
     ……
```

# ЛІТЕРАТУРА

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

Упорядники:

**Саксонов** Геннадій Михайлович

**Жукова** Олена Андріївна

# МЕТОДИ ПОБУДОВИ ТА АНАЛІЗУ КРИПТОСИСТЕМ

**Методичні рекомендації**

**до виконання лабораторних робіт**

**з дисципліни**

**для студентів-магістрів спеціальності 125 Кібербезпека**

**галузі знань 12 Інформаційні технології**

Видано в редакції упорядників

Комп'ютерний дизайн, верстка та обробка – О.А. Жукова

Підписано до друку 25.04.2019. Формат 30х42/4.
Папір офсетний. Ризографія. Ум. друк. арк. 2,9.
Обл.-вид. арк. 2,9. Тираж 6 пр. Зам. №

Національний технічній університет «Дніпровська політехніка»

49005, м. Дніпро, просп. Д. Яворницького, 19