МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ  
Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ   
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационных систем и технологий

Специальность 1—98 01 03 «Программное обеспечение информационной

безопасности мобильных систем»

**Отчет**

по дисциплине «Основы информационной безопасности»

**Исполнитель**

студент(ка) 2 курса 7 группы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Банкузов М.О.

подпись, дата

**Руководитель**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ржеутская Н. В.

должность, ученая степень, ученое звание подпись, дата

|  |
| --- |
| **Практическое занятие №8** |
| **Тема «Изучение стандартных средств для реализации приложений, использующих симметричное и ассиметричное шифрование с использованием библиотеки** [**System.Security.Cryptography**](https://docs.microsoft.com/ru-ru/dotnet/api/system.security.cryptography)**»** |
| Цель: Изучить модель криптографии .NET Framework, основные классы и структуры данных, разработать приложение для шифрования файлов использующих симметричные и ассиметричные алгоритмы шифрования |
| **Теоретические сведения** |

**Криптография в .Net**

В .Net Framework присутствует пространство имён для выполнения криптографических операций под названием System.Security.Cryptography. Данное пространство имён предоставляет криптографические службы, включающие безопасное кодирование и декодирование данных, а также другие операции, такие как хеширование сообщений, генерация случайных чисел и проверка подлинности сообщений. Данная библиотека предоставляет доступ для использования различных реализаций алгоритмов в основном это программные интерфейсы CryptoApi (CAPI) и Cryptography Next Generation API (CNG API) помимо этого для некоторых алгоритмов возможно использование реализаций на основе OpenSsl.

CryptoAPI — интерфейс программирования приложений, который обеспечивает разработчиков Windows-приложений стандартным набором функций для работы с криптопровайдером. Входит в состав операционных систем Microsoft. Большинство функций CryptoAPI поддерживается, начиная с Windows 2000.

Cryptography Next Generation стала долгосрочной заменой CAPI. Данный набор интерфейсов поддерживает все алгоритмы предлагаемые CAPI а также другие алгоритмы перечисленные в своде правил Suite B Агентства национальной безопасности США [1]. Данный интерфейс поддерживает следующие длины ключей или размерность хеша.

* RSA 512 бит to 16384 бит, с шагом 64 бит
* DH 512 бит to 16384 бит, с шагом 64 бит
* DSA 512 бит to 1024 бит, с шагом 64 бит
* ECDSA P-256, P-384, P-521 (NIST Curves)
* ECDH P-256, P-384, P-521 (NIST Curves)
* MD2 128 бит
* MD4 128 бит
* MD5 128 бит
* SHA-1 160 бит
* SHA-256 256 бит
* SHA-384 384 бит
* SHA-512 512 бит

Рассматривая структуру наследования для симметричных алгоритмов в .Net стоит упомянуть что SymmetricAlgorithm является абстрактным классом, от который наследуют абстрактные классы для реализаций каждого из алгоритмов. В свою очередь каждая из реализации алгоритма является производной от абстрактного класса алгоритма. Ниже представлена структура наследования.

SymmetricAlgorithm

Aes

AesCng

AesManeged

AesCryptoServiceProvider

Des

DesCng

DesManeged

DesCryptoServiceProvider

TripleDes

TripleDesCng

TripleDesManeged

TripleDesCryptoServiceProvider

Данная структура наследования повторяется для каждого из трех типов поддерживаемых криптографических операций: SymmetricAlgorithm, AsymmetricAlgorithm, HashAlgorithm.

Рассмотрим часть кода для шифрования сообщения используя алгоритм Aes. Как представлено ниже мы создаём объект шифратор на основе созданного экземпляра криптографического объекта aesAlg. Далее мы создаём потоки для шифрования. Стоит отметить CryptoStream который определяет поток, связывающий потоки данный с криптографическим преобразованием.

ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);

// создание потоков используемого для шифрования

using (MemoryStream msEncrypt = new MemoryStream())

{

using (CryptoStream csEncrypt = new CryptoStream(msEncrypt, encryptor, CryptoStreamMode.Write))

//1-используемый поток, 2 - криптографические преобразование, 3 тип доступа

{

using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))

{

swEncrypt.Write(plainText);

}

encrypted = msEncrypt.ToArray();

}

}

}

Дешифрование выполняется аналогично за исключением изменения типа доступа CryptoStream и изменения потока с записи на чтение.

ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);

using (MemoryStream msDecrypt = new MemoryStream(cipherText))

{

using (CryptoStream csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))

{

using (StreamReader srDecrypt = new StreamReader(csDecrypt))

{

plaintext = srDecrypt.ReadToEnd();

}

}

}

Также стоит упомянуть что сложность большинства алгоритмов шифрование имеет сильную зависимость от длины ключа. Изменение длины ключа возможно в большинстве алгоритмов как симметричного, так и ассиметричного шифрования. Для того что бы изменить размер ключа после инициализации алгоритма необходимо изменить свойство KeySize, что представлено в листинге ниже.

using (TripleDES myDes = TripleDES.Create())

{

myDes.KeySize = 128;

EncryptStringToFile\_DES(route, "2 keys usedE.txt", myDes.Key, myDes.IV);

DecryptStringToFile\_DES("2 keys usedE.txt", "2 key deceypted.txt", myDes.Key, myDes.IV);

}

Что касается ассиметричных алгоритмов то они представлены в данной структуре наследования.

AsymmetricAlgorithm

Rsa

RsaCng

RsaOpenSsl

RsaCryptoServiceProvider

Dsa

DsaCng

DsaOpenSsl

DsaCryptoServiceProvider

ECDiffieHellman

ECDiffieHellmanCng

ECDiffieHellmanOpenSsl

Шифрование и дешифрование для асимметричных алгоритмов выполняется проще из за встроенных функций Encrypt и Decrypt. Так же стоит помнить, что Rsa является блочным алгоритмов и если длина данных не совпадает с длиной блока, то данные нужно дополнить до длины блока. Для этого в данном пространстве имён можно использовать разные режимы заполнения.

using (RSA myRsa = RSA.Create())

{

string publickey = myRsa.ToXmlString(false); //получим открытый ключ

string privatekey = myRsa.ToXmlString(true);//получим закрытый ключ

byte[] encrypted = myRsa.Encrypt(data, RSAEncryptionPadding.Pkcs1);

byte[] decrypted = myRsa.Decrypt(encrypted, RSAEncryptionPadding.Pkcs1);

Console.WriteLine("До шифрования: {0}", original);

Console.WriteLine("Зашифрованное: {0}", System.Text.Encoding.UTF8.GetString(encrypted));

Console.WriteLine("После дешифровки: {0}", System.Text.Encoding.UTF8.GetString(decrypted));

}

Как представлено в коде выше после создания экземпляра Rsa возможна сразу же зашифровать и дешифровать данные использую автоматически сгенерированные ключи. Также код представленный выше содержит пример для получения открытого и закрытого ключа в виде строки в формате Xml.

<RSAKeyValue>

<Modulus>6yEjtrItcUq1hoA01xc63EW5/P99kstIobXsxPCUfUODRn2dazzcyhJ5Quhw1oHodlOMvtDN3xJdOTWDbH3xdQ==

</Modulus>

<Exponent>AQAB

</Exponent>

</RSAKeyValue>

Пример сохранённого в файл открытого ключа

<RSAKeyValue>

<Modulus>6yEjtrItcUq1hoA01xc63EW5/P99kstIobXsxPCUfUODRn2dazzcyhJ5Quhw1oHodlOMvtDN3xJdOTWDbH3xdQ==

</Modulus>

<Exponent>AQAB

</Exponent>

<P>9TsbWvgvA2OqPZxUZ96PomUG8rJk2T0SiH6chz65zkc=

</P>

<Q>9XR41YP8/CUoajovRPKWQZou3J23n3usp1acC3v9dGM=

</Q>

<DP>lnjHJ0GD72t3KUjETdu1fKrK4Z5u3RFpTtGJkD7/2b8=

</DP>

<DQ>9Qv2ppNCuigOlR7JmjflslDPgAklDN9XmyhoWT7L5qk=

</DQ>

<InverseQ>gSJ6G275fGFrEMqwSdgJYvmUQhnpCTcX0T3imIVQwOE=

</InverseQ>

<D>z0gMwu+6zehNtP/rFT9eXXd+qgHWAwYAxsapr0hjrzSXv1qS9QJJ+062YbdHc24WZagrKqABfOLQ3hfLXP3JdQ==

</D></RSAKeyValue>

Пример сохранённого в файл открытого ключа

Как видно на представленных выше файлах открытого и закрытого ключа объём ключей отличается, но также совпадают modulus (n) и Exponent (e). Остальные параметры в закрытом ключе совпадают с общепринятыми обозначениями кроме DP=d mod (p-1), DQ=d mode (q-1) и D обозначающего r или закрытый показатель степени.

Xеширования в .Net выполняется проще остальных операций из за отсутствия необходимости в обратном преобразовании. В данном пространстве имён поддерживаются следующие алгоритмы:

* MD5
* SHA256
* SHA384
* SHA512

Пример кода представлен ниже.

byte[] hashValue = new byte[256];

using (SHA256 mysha256 = SHA256.Create())

{

hashValue = mysha256.ComputeHash(File.ReadAllBytes("sha.txt"));

File.WriteAllBytes("hash.txt", hashValue);

}

Как видно из представленного кода после инициализации экземпляра для шифрования нужно вызвать только один метод ComputeHash который выполнит хеширование данных.

Пример захешированных данных представлен ниже

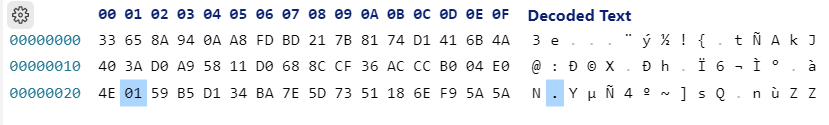


Рисунок 1

**Задание для выполнения:**

1. Ознакомиться с созданием криптографического приложения;
2. Выполнить шифрование, дешифрование и хеширование своей фамилии по указанным алгоритмам. Используя функции работы с файлами сохранить ключи шифрования, результаты шифрования и хеширования.
3. Для выше указанных алгоритмов используя Hex-редактор продемонстрировать ключи шифрования, зашифрованные и захешированные данные.

|  |  |  |
| --- | --- | --- |
| 2 | DES | SHA1 |

using System;

using System.IO;

using System.Security.Cryptography;

using System.Text;

public class CryptoExample

{

private static byte[] EncryptStringToBytes(string plainText, byte[] Key, byte[] IV)

{

using (DESCryptoServiceProvider des = new DESCryptoServiceProvider())

{

des.Key = Key;

des.IV = IV;

using (MemoryStream memoryStream = new MemoryStream())

{

using (CryptoStream cryptoStream = new CryptoStream(memoryStream, des.CreateEncryptor(), CryptoStreamMode.Write))

{

byte[] plaintextBytes = Encoding.UTF8.GetBytes(plainText);

cryptoStream.Write(plaintextBytes, 0, plaintextBytes.Length);

cryptoStream.FlushFinalBlock();

}

return memoryStream.ToArray();

}

}

}

private static string DecryptStringFromBytes(byte[] cipherText, byte[] Key, byte[] IV)

{

using (DESCryptoServiceProvider des = new DESCryptoServiceProvider())

{

des.Key = Key;

des.IV = IV;

using (MemoryStream memoryStream = new MemoryStream(cipherText))

{

using (CryptoStream cryptoStream = new CryptoStream(memoryStream, des.CreateDecryptor(), CryptoStreamMode.Read))

{

byte[] decryptedBytes = new byte[cipherText.Length];

int decryptedByteCount = cryptoStream.Read(decryptedBytes, 0, decryptedBytes.Length);

return Encoding.UTF8.GetString(decryptedBytes, 0, decryptedByteCount);

}

}

}

}

private static byte[] ComputeHash(string input)

{

using (SHA1 sha1 = SHA1.Create())

{

byte[] inputBytes = Encoding.UTF8.GetBytes(input);

return sha1.ComputeHash(inputBytes);

}

}

public static void Main()

{

string originalText = "Банкузов";

// Генерация ключа и вектора инициализации

byte[] key = new byte[8]; // 8 байт для DES

byte[] iv = new byte[8]; // 8 байт для DES

using (DES des = DES.Create())

{

key = des.Key;

iv = des.IV;

}

// Шифрование

byte[] encryptedData = EncryptStringToBytes(originalText, key, iv);

// Дешифрование

string decryptedText = DecryptStringFromBytes(encryptedData, key, iv);

// Хеширование

byte[] hash = ComputeHash(originalText);

// Сохранение ключей, зашифрованных данных и хеша в файлы

File.WriteAllBytes("encryption\_key.bin", key);

File.WriteAllBytes("encryption\_iv.bin", iv);

File.WriteAllBytes("encrypted\_data.bin", encryptedData);

File.WriteAllBytes("hash.bin", hash);

// Демонстрация ключей, зашифрованных данных и хеша в Hex-редакторе

string keyHex = BitConverter.ToString(key).Replace("-", "");

string ivHex = BitConverter.ToString(iv).Replace("-", "");

string encryptedDataHex = BitConverter.ToString(encryptedData).Replace("-", "");

string hashHex = BitConverter.ToString(hash).Replace("-", "");

Console.WriteLine("Encryption Key (Hex): " + keyHex);

Console.WriteLine("Encryption IV (Hex): " + ivHex);

Console.WriteLine("Encrypted Data (Hex): " + encryptedDataHex);

Console.WriteLine("Hash (Hex): " + hashHex);

}

}

