/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* EE6032 Communication & Security Protocols

\* Dr. Tom Newe

\* Project

\*

\* Group:

\* Patrick Stapleton 10122834

\* Cian Conway 10126767

\* Rory Burns 10108696

\*

\* Start Date: 13/02/2014

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*#include "stdafx.h"

#include <Windows.h> // needed by CHILKAT e.g. for SystemTime

// added for CHILKAT

#include "..\include\ckimap.h"

//#include "..\include\tchar.h"

//

// need this for various things

extern "C" {

#include "../include/allC.h"

#include "../include/allUnicodeC.h"

}

#include "../include/allUnicode.h"

// for the AES Encryption & Hashing

#include "../include/CkCrypt2.h"

// for RSA

#include "../include/CkRsa.h"

// get strings, because the const char\* causes problems....

#include <iostream>

#include <string>

#include <conio.h>

using namespace std;

void ChilkatAESSample(void)

{

CkCrypt2 crypt;

bool success;

success = crypt.UnlockComponent("T06152014Crypt\_J2uOFPEBkQ3d");

if (success != true) {

printf("%s\n",crypt.lastErrorText());

return;

}

// AES is also known as Rijndael.

crypt.put\_CryptAlgorithm("aes");

// CipherMode may be "ecb" or "cbc"

crypt.put\_CipherMode("cbc");

// KeyLength may be 128, 192, 256

crypt.put\_KeyLength(256);

// The padding scheme determines the contents of the bytes

// that are added to pad the result to a multiple of the

// encryption algorithm's block size. AES has a block

// size of 16 bytes, so encrypted output is always

// a multiple of 16.

// How many bytes are added to the result so that its a

// multiple of block size.

crypt.put\_PaddingScheme(0);

// EncodingMode specifies the encoding of the output for

// encryption, and the input for decryption.

// It may be "hex", "url", "base64", or "quoted-printable".

crypt.put\_EncodingMode("hex");

// An initialization vector is required if using CBC mode.

// ECB mode does not use an IV.

// The length of the IV is equal to the algorithm's block size.

// It is NOT equal to the length of the key.

const char \* ivHex;

ivHex = "000102030405060708090A0B0C0D0E0F";

crypt.SetEncodedIV(ivHex,"hex");

// The secret key must equal the size of the key. For

// 256-bit encryption, the binary secret key is 32 bytes.

// For 128-bit encryption, the binary secret key is 16 bytes.

const char \* keyHex;

keyHex = "000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F";

crypt.SetEncodedKey(keyHex,"hex");

// Encrypt a string...

// The input string is 44 ANSI characters (i.e. 44 bytes), so

// the output should be 48 bytes (a multiple of 16).

// Because the output is a hex string, it should

// be 96 characters long (2 chars per byte).

const char \* encStr;

const char \* encStr1;

encStr = "Temp String";

encStr1 = crypt.encryptStringENC(encStr);

printf("%s\n",encStr);

printf("%s\n",encStr1);

// Now decrypt:

const char \* decStr;

decStr = crypt.decryptStringENC(encStr1);

printf("%s\n",decStr);

}

void ChilkatRSASample(void)

{

CkRsa rsa;

bool success;

success = rsa.UnlockComponent("T06152014RSA\_XFn8tkbGkQQd");

if (success != true) {

printf("RSA component unlock failed\n");

return;

}

// This example also generates the public and private

// keys to be used in the RSA encryption.

// Normally, you would generate a key pair once,

// and distribute the public key to your partner.

// Anything encrypted with the public key can be

// decrypted with the private key. The reverse is

// also true: anything encrypted using the private

// key can be decrypted using the public key.

// Generate a 1024-bit key. Chilkat RSA supports

// key sizes ranging from 512 bits to 4096 bits.

success = rsa.GenerateKey(1024);

if (success != true) {

printf("%s\n",rsa.lastErrorText());

return;

}

// Keys are exported in XML format:

const char \* publicKey;

publicKey = rsa.exportPublicKey();

const char \* privateKey;

privateKey = rsa.exportPrivateKey();

const char \* plainText;

plainText = "Encrypting and decrypting should be easy!";

printf("%s\n", plainText);

// Start with a new RSA object to demonstrate that all we

// need are the keys previously exported:

CkRsa rsaEncryptor;

// Encrypted output is always binary. In this case, we want

// to encode the encrypted bytes in a printable string.

// Our choices are "hex", "base64", "url", "quoted-printable".

rsaEncryptor.put\_EncodingMode("hex");

// We'll encrypt with the public key and decrypt with the private

// key. It's also possible to do the reverse.

rsaEncryptor.ImportPublicKey(publicKey);

bool usePrivateKey;

usePrivateKey = false;

const char \* encryptedStr;

encryptedStr = rsaEncryptor.encryptStringENC(plainText,usePrivateKey);

printf("%s\n",encryptedStr);

// Now decrypt:

CkRsa rsaDecryptor;

rsaDecryptor.put\_EncodingMode("hex");

rsaDecryptor.ImportPrivateKey(privateKey);

usePrivateKey = true;

const char \* decryptedStr;

decryptedStr = rsaDecryptor.decryptStringENC(encryptedStr,usePrivateKey);

printf("%s\n",decryptedStr);

}

String ChilkatSHASample(int na)

{

CkCrypt2 crypt;

// Any string argument automatically begins the 30-day trial.

bool success;

success = crypt.UnlockComponent("T06152014Crypt\_J2uOFPEBkQ3d");

if (success != true) {

printf("Crypt component unlock failed\n");

return;

}

printf(na);

crypt.put\_HashAlgorithm("sha1");

crypt.put\_EncodingMode("hex");

// Other possible EncodingMode settings are:

// "quoted-printable", "base64", and "url"

const char \* hash;

hash = crypt.hashStringENC(na);

printf("SHA1:\n");

printf("%s\n",hash);

return hash

// Hashes for "The quick brown fox jumps over the lazy dog"

// SHA1:

// 2FD4E1C67A2D28FCED849EE1BB76E7391B93EB12

// MD2:

// 03D85A0D629D2C442E987525319FC471

// MD5:

// 9E107D9D372BB6826BD81D3542A419D6

// SHA256:

// D7A8FBB307D7809469CA9ABCB0082E4F8D5651E46D3CDB762D02D0BF37C9E592

// SHA384:

// CA737F1014A48F4C0B6DD43CB177B0AFD9E5169367544C494011E3317DBF9A509CB1E5DC1E85A941BBEE3D7F2AFBC9B1

// SHA512:

// 07E547D9586F6A73F73FBAC0435ED76951218FB7D0C8D788A309D785436BBB642E93A252A954F23912547D1E8A3B5ED6E1BFD7097821233FA0538F3DB854FEE6

// Haval:

// B89C551CDFE2E06DBD4CEA2BE1BC7D557416C58EBB4D07CBC94E49F710C55BE4

}

\*/