How to use NCryptoki

Author: Ugo Chirico – http://www.ugochirico.com

Data: 2010-09-20

Introduction

PKCS#11 (Public Key Cryptography Standards No. 11) specifications, developed by RSA Data Security labs, defines an high-level, platform-independent API to cryptographic devices (such as smart cards, USB Tokens, HSMs, etc.), that hides the low level operational logic of a cryptographic devices, presenting to the applications a unified abstraction layer for a generic cryptographic token with an higher level set of functions.

The high flexibility and the simple logic of such a model made the PKCS#11 specifications a de-facto standard widely used by applications interacting with smart cards. PKCS#11 is largely adopted to access smart cards, cryptographic tokens and HSMs. Most commercial Certification Authority software uses PKCS#11 to access the CA signing key or to enroll user certificates as well as cross-platform software that needs to use smart cards uses PKCS#11, such as Mozilla Firefox and OpenSSL (using an extension).

Because the API is defined in C language, the PKCS#11 module is implemented in C as native library (a Dinamically Linked Library (.dll) in Windows OS or as Shared Object (.so) on Linux and MacOS) that exports the functions of the API. This means that if your application is in C/C++ you can easily import the API functions in your code, But what if your application is in C# or VB.NET? Or what if you application is in Visual Basic 6 or Delphi? This article addresses this stuffs and explains how to call PKCS#11 API in your .NET application using the library NCryptoki. The first part shows a brief overview about PKCS#11 specifications. The second part describes the NCryptoki library and shows how to accomplish the main procedures of a cryptographic application: key pair generation, certificates creation, encryption and decryption, signature and verification.

Architecture

A typical Cryptoki-based system architecture is depicted in Figure 1.

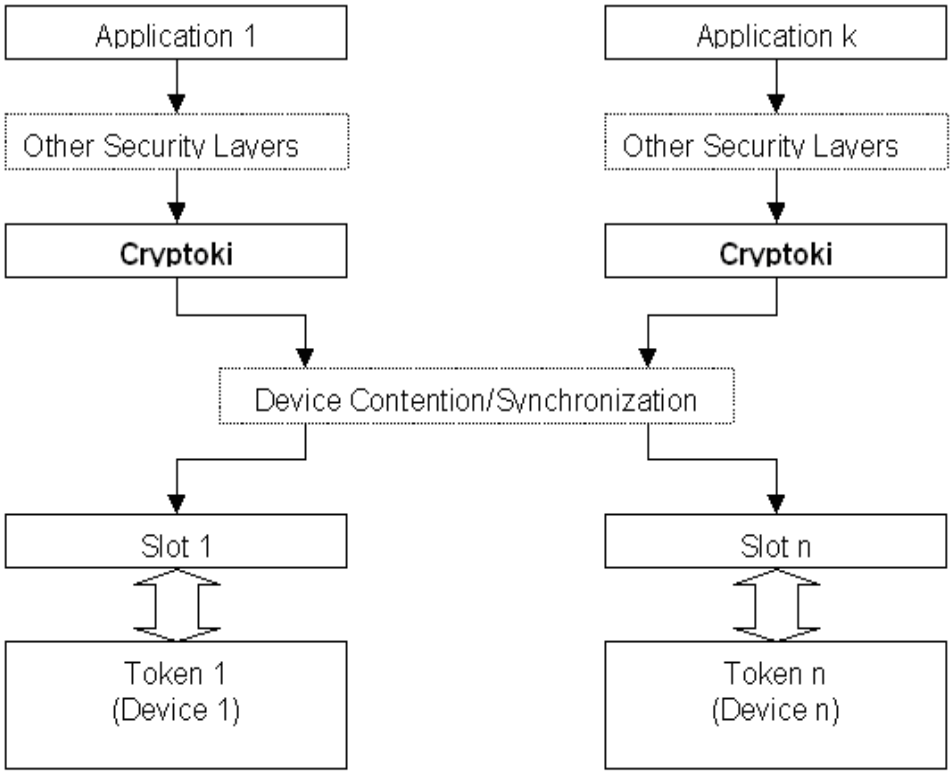

Figure 1

The cryptographic device (aka *token*) is connected to the system via a *slot*. Typically, a slot corresponds to a smart card reader or a specific card terminal. However, because *Cryptoki* offers a purely logical view of the system it could happens that different slots point to the same physical reader device or, viceversa, a single slot could have more than one device.

The logical structure of a Token

A specific Cryptoki implementation maps the token's physical structure, typically composed by memory zones in which data, cryptographic keys and their digital certificates are stored, into a logical structure that adheres to the hierarchical model shown in Figure 2.

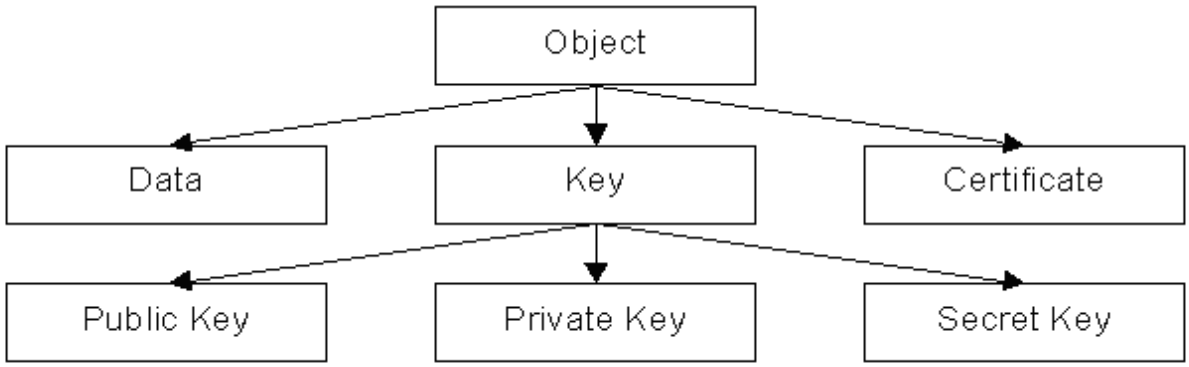

Figure 2

The specifications define three main object classes:

- *Data* objects host generic data which semantics is defined by the application who created them;
- *Certificate* objects store digital certificates;
- *Key* objects contain a public, private or secret cryptographic key.

Cryptoki's objects are classified depending on their visibility in *public objects* (i.e. accessible by all applications), and *private objects* (visible only after granting access permissions typically performed via PIN-verification as described later), and on their persistency in: *token objects* which persist when the token is plugged-out from the slot and in *session objects* which don't persist.

For each class of objects the specifications define a set of attributes (as described later) characterizing all instances of the class, which, are inherited by derived classes, similarly to an object-oriented model (for example, the Private Key class inherits all attributes from the Key class etc.).

The API

PKCS#11 specifications define an API named *Cryptoki* (CRYPtographic TOKen Interface) that implements an API to an abstract model of a cryptographic device, such as a microprocessor-based smart card, a USB cryptographic token or an HSM. The API follows a simple object-based approach, addressing the goals of technology independence (*any kind of device*) and resource sharing (*multiple applications accessing multiple devices*), presenting to applications a common, logical view of the device called a *cryptographic token*. The API defines the most commonly used cryptographic object types (RSA keys, X.509 Certificates, DES/Triple DES keys, etc.) and all the functions needed to use, create/generate, modify and delete those objects:

The addendum A shows the set of the function supplied by Cryptoki API.

The programming language used to define the functions and data types is ANSI C. Along with specifications, RSA Data Security published three C header files (pkcs11.h, pkcs11t.h and pkcs11f.h, available at this page: http://www.rsa.com/rsalabs/node.asp?id=2133) that define function prototypes, Cryptoki-specific data types and a set of macros to manage objects classes and their attributes.

Using the Cryptoki API in a .NET application

As we said above, the API is defined in C language and the PKCS#11 modules are implemented in C as native unmanaged libraries. In order to use it in a .NET application we have no chance to avoid from using platform invoke services (*P-Invoke*), supplied by the .NET framework, to import the unmanaged functions of the native API in our C# and/or VB.NET managed code. But importing such functions from an unmanaged dll, expecially from a highly complex PKCS#11 dll, requires very advanced skills in C/C++ and .NET and compels a lot of tedious work to write the declaration of the prototypes related to the functions using the P-Invoke rules and to deal with the marshalling of custom parameters.
NCryptoki library allows to avoid from dealing with P-Invoke declarations and unmanaged code saving a lot of tedious work.

NCryptoki

NCryptoki is a library for .NET framework that implements the PKCS#11 specifications and supplies an API for C#, VB.NET, Visual Basic 6, Delphi and other COM interop languages for integrating a PKCS#11 compliant token in any application. It is available as shareware version with full functionalities from the following url: http://www.ncryptoki.com .
NCryptoki maps the cryptoki's functions defined in PKCS#11 specification in a set of high level classes usable in C#, VB.NET and propose a programming paradigm that allows to integrate your PKCS#11 compliant token in your applications easily with a few lines of code.
NCryptoki supplies also a *COM interface* that allows to use the supplied classes in any language that supports COM interop like Visual Basic 6, Delphi etc.
The main features are:

- Compliant with PKCS#11 2.20 specifications
- Compliant with any PKCS#11 token
- 32 or 64 bit platform
- .NET Framework 2.0, 3.5 and 4.0

The programming paradigm is almost similar to the one in C language described in PKCS#11 specifications: the PKCS#11 C functions are mapped into a set of .NET classes that follows the same classification described above. Figure 3 shows the class hierarchy of NCryptoki.
Cryptoki is the main class that allows to use the library, the classes *Slot* and *Token* enclose the slot-handling and token-handling functions, while the class *CryptokiObject* encapsulates the object-handling functions as well as the definitions related to objects' classes and their attributes. The class *Session* includes the *OpenSession* and *CloseSession* functions, the functions related to login and logout, the search functions to search for PKCS#11 objects and, finally, the cryptographic and hashing functions and the other functions defined in the PKCS#11 specifications. The complete API documentation is available here: http://www.ncryptoki.com/documentation.aspx.

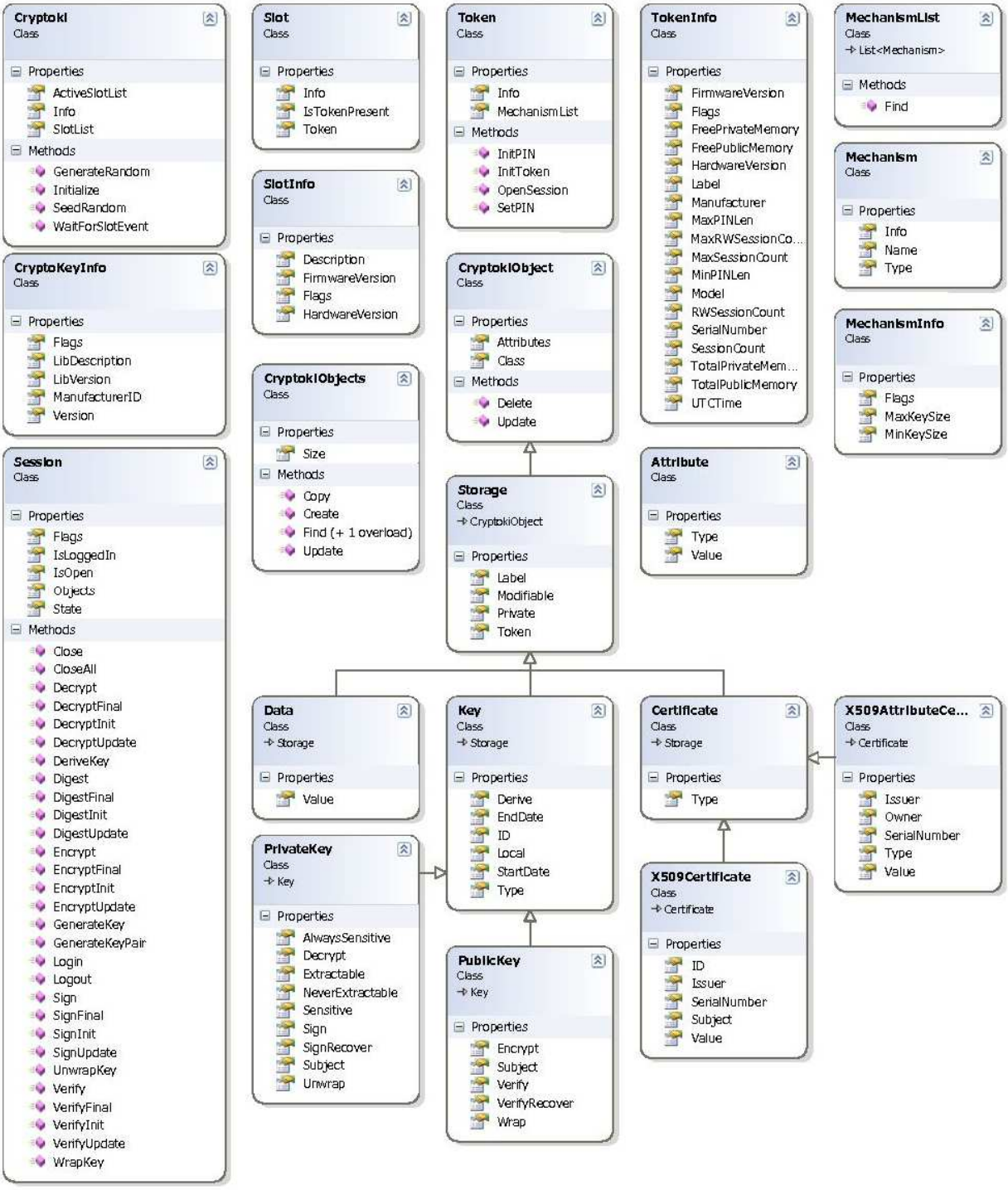

Figure 3

A simple program

Let's analyze a simple program to see hot to use NCryptoki. The attached Visual Studio project shows the typical PKCS#11 procedures: initializing the library, searching for some object, generating a key pair, adding a certificates, encrypting some text and so on.

Instantiate and Initialize a Cryptoki object

Cryptoki constructor takes the path to the native PKCS#11 library. This lines of code create a Cryptoki object and attach it to the PKCS#11 native library *smaoscki.dll*, then initialize the library by calling *Initialize* method:

```
Cryptoki cryptoki = new Cryptoki("smaoscki.dll");
int nRet = cryptoki.Initialize();
if (nRet != 0)
{
    error(nRet);
}
```

Read available slots:

The property *Slots* contains the available slots:

```
SlotList slots = cryptoki.Slots;
if (slots.Count == 0)
    throw new Exception("No slots available");
```

Open a session

To open a session with a toke we have to check if the slots contains a token checking the property *Slot.IsTokenInserted*. If so we get a *Token* object from the slot and open a session with the method *token.OpenSession*:

```
Slot slot = slots[0];
if(!slot.IsTokenInserted)
{
    Console.WriteLine("No token found in the slot: " + slot.Info.Description);
    return;
}

Token token = slot.Token;

// Prints all information relating to the token
TokenInfo tinfo = token.Info;
Console.WriteLine(tinfo.Label);
Console.WriteLine(tinfo.ManufacturerID);
Console.WriteLine(tinfo.Model);
Console.WriteLine(tinfo.SerialNumber);
Console.WriteLine(tinfo.HardwareVersion);

// Opens a read/write serial session
Session session =
    token.OpenSession (Session.CKF_SERIAL_SESSION | Session.CKF_RW_SESSION, null, null);
```

Login

To login to a session we use *Session.Login* method that takes as parameters the type of user: USER (simple user) or SO (Security Officer) and the PIN:

```
int nRes = session.Login((int)Session.CKU_USER, "12345678");
if (nRes != 0)
{
    Console.WriteLine("Wrong PIN");
    return;
}

Console.WriteLine("Logged in:" + session.IsLoggedIn);
```

Search for some objects

In order to search for some object we have to specify the template of the object we want to search for. Such a template is a sort of filter that allows to get only the objects that match the values in the template. The following piece of code searches for RSA private keys having the label equals to “MyRSAKey”:

```
// Sets the template with its attributes
CryptokiCollection template = new CryptokiCollection();
template.Add(new ObjectAttribute(ObjectAttribute.CKA_CLASS,
CryptokiObject.CKO_PRIVATE_KEY));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_KEY_TYPE, Key.CKK_RSA));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_LABEL, " MyRSAKey "));

// Launchs the search with the template just created
CryptokiCollection objects = session.Objects.Find(template, 10);
// If the private keys is found continue
if (objects.Count > 0)
{
    foreach (Object obj in objects)
    {
         Console.WriteLine(((PrivateKey)obj).Label);
    }

    //Gets the first object found
    RSAPrivateKey privateKey;
    privateKey = (RSAPrivateKey)objects[objects.Count - 1];
    Console.WriteLine(privateKey.Label);
}
```

Generate a key pair

In order to generate a key pair we have to specify the key pair’s attributes needed to generate the key pair such as: key type, generation algorithm, label, and so on. This can be done by using a template in which we save all key’s attributes:

```
// Prepares the templates for public key
CryptokiCollection templatePub = new CryptokiCollection();
templatePub.Add(new ObjectAttribute(ObjectAttribute.CKA_CLASS,
CryptokiObject.CKO_PUBLIC_KEY));
templatePub.Add(new ObjectAttribute(ObjectAttribute.CKA_TOKEN, true));
templatePub.Add(new ObjectAttribute(ObjectAttribute.CKA_PRIVATE, false));
templatePub.Add(new ObjectAttribute(ObjectAttribute.CKA_LABEL, "Ugo's new Key"));
templatePub.Add(new ObjectAttribute(ObjectAttribute.CKA_ID, "1"));
templatePub.Add(new ObjectAttribute(ObjectAttribute.CKA_MODULUS_BITS, 1024));
templatePub.Add(new ObjectAttribute(ObjectAttribute.CKA_PUBLIC_EXPONENT, 0x010001));

// Prepares the templates for private key
CryptokiCollection templatePri = new CryptokiCollection();
templatePri.Add(new ObjectAttribute(ObjectAttribute.CKA_CLASS,
CryptokiObject.CKO_PRIVATE_KEY));
templatePri.Add(new ObjectAttribute(ObjectAttribute.CKA_TOKEN, true));
templatePri.Add(new ObjectAttribute(ObjectAttribute.CKA_PRIVATE, true));
templatePri.Add(new ObjectAttribute(ObjectAttribute.CKA_LABEL, "Ugo's new Key"));
templatePri.Add(new ObjectAttribute(ObjectAttribute.CKA_ID, "1"));

//generate the key pair objects
Key[] keys = session.GenerateKeyPair(Mechanism.RSA_PKCS_KEY_PAIR_GEN, templatePub,
templatePri);

// gets the two generated keys
RSAPrivateKey privateKey = (RSAPrivateKey)keys[1];
RSAPublicKey publicKey = (RSAPublicKey)keys[0];
```

Create a Certificate object

To create a *Certificate* object we have to use a template again to specify the certificate’s attributes:

```
// Load a X509 certificate from a file
X509Certficate2 cert = new X509Certificate2("cert.cer");

// Creates the template
CryptokiCollection template = new CryptokiCollection();
template.Add(new ObjectAttribute(ObjectAttribute.CKA_CLASS,
CryptokiObject.CKO_CERTIFICATE));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_SUBJECT, cert.SubjectName.RawData));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_ISSUER, cert.Issuer));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_SERIAL_NUMBER, cert.SerialNumber));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_ID, id));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_LABEL, label));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_TOKEN, true));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_VALUE, cert.RawData));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_MODIFIABLE, modifiable));

// Creates a Certificate object
CryptokiObject certificate = CurrentSession.Objects.Create(template);
```

Create a Data object

To create a *Data* object we have to use a template to specify the object's attributes:

```
CryptokiCollection template = new CryptokiCollection();
template.Add(new ObjectAttribute(ObjectAttribute.CKA_CLASS, CryptokiObject.CKO_DATA));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_LABEL, label));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_APPLICATION, application));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_TOKEN, true));
template.Add(new ObjectAttribute(ObjectAttribute.CKA_PRIVATE, true);
template.Add(new ObjectAttribute(ObjectAttribute.CKA_MODIFIABLE, true);
template.Add(new ObjectAttribute(ObjectAttribute.CKA_VALUE, value));

// Creates the Data object
Data data = (Data)session.Objects.Create(template);
```

Encrypt and decrypt

To encrypt some text we use *Session.EncryptInit* and *Session.Encrypt* methods. To decrypt some cipher text we use *Session.DecryptInit* and *Session.Decrypt* methods.
EncryptInit takes as parameters the encryption algorithm and the key object (obtained by a call to *Find* method - see above) to use in encryption.
DecryptInit takes as parameters the decryption algorithm and the key object (obtained by a call to *Find* method - see above) to use in decryption.

```
string helloworld1 = "Hello World to encrypt";
byte[] text = Encoding.ASCII.GetBytes(helloworld1);

// launches the encryption operation DES mechanism
nRes = session.EncryptInit(Mechanism.DES, deskey);

// computes the encryption
byte[] encrypted = session.Encrypt(text);

// launches decryption
nRes = session.DecryptInit(Mechanism.DES, key);

// computes the decryption
byte[] decrypted = session.Decrypt(encrypted);
```

Sign and Verify

To sign some text we use *Session.SignInit* and *Session.Sign* methods. To verify the signature we use *Session.VerifyInit* and *Session.Verify*.
SignInit takes as parameters the signature algorithm and the private key object (obtained by a call to *Find* method - see above) to use to apply the signature.

VerifyInit takes as parameters the signature algorithm and the public key object (obtained by a call to *Find* method - see above) to use to verify the signature.

```
string helloworld2 = "Hello World to sign";
byte[] text = Encoding.ASCII.GetBytes(helloworld2);

// launches the digital signing operation with a RSA_PKCS mechanism
nRes = session.SignInit(Mechanism.SHA1_RSA_PKCS, privateKey);

// computes the signature
byte[] signature = session.Sign(text);


// Initializes the verification function
nRes = session.VerifyInit(Mechanism.SHA1_RSA_PKCS, publicKey);

// verifies the signature
nRes = session.Verify(text, signature);
if(nRes == 0)
    Console.Write("Verify " + nRes);
```

COM Interop

All NCryptoki classes are exported under COM interop, this means the they can be also used in a Visual Basic 6 (and VBA and VBScript) application as COM objects and in any of the other languages that support COM interop like Delphi, etc. In this paper we don't cover this stuff explicitly because the COM classes exported by the library are the same as described above and must be used in the same way. For more info and samples about using NCryptoki classes as COM objects visit: http://www.ncryptoki.com.

Conclusion

NCryptoki makes easy to use HSMs and smart cards in any .NET application and allows to save a lot of development time. A bit of knowledge of PKCS#11 is needed, of course, but a beginner knowledge of C# is enough to use the library. As of this writing, the current version 1.5 is compliant with the version 2.20 of the PKCS#11 specifications. Next version will be compliant with the new PKCS#11 specification 2.30, published in April 2009.
News, updates, more detailed documentation can be found at http://www.ncryptoki.com.

Bibliography:

[1] Programming Smart Cards - Part 1, Ugo Chirico, 2009 - http://www.ugosweb.com/publications.aspx
[2] Programming Smart Cards - Part 2, Ugo Chirico, 2009 - http://www.ugosweb.com/publications.aspx
[3] PKCS #11 v2.20: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf

Addendum A:

List of cryptoki functions:

Function/Category	Description
Library Management Functions	
C_Initialize	Initializes the library
C_Finalize	Frees all resources allocated by the library
C_GetInfo	Reads all information relating to the library
C_GetFunctionList	Returns the list of library-implemented functions
Slot and Token Management Functions	
C_GetSlotList	Returns the list of connected slots
C_GetSlotInfo	Reads all information relating to a specific slot
C_GetTokenInfo	Reads all information relating to the token inserted in the specified slot
C_WaitForSlotEvent	Waits for an event on the slot (i.e. token insertion or removal, etc.)
C_GetMechanismList	Returns the list of library-supported cryptographic mechanisms
C_GetMechanismInfo	Returns all information relating to the specified cryptographic mechanism
C_InitToken	Initializes the token
C_InitPIN	Initializes the token's PIN
C_SetPIN	Sets the token's PIN

Session Management Functions	
C_OpenSession	Opens a session with a token in the specified slot
C_CloseSession	Closes a session
C_CloseAllSessions	Closes all opened sessions
C_GetSessionInfo	Reads all information relating to the specified session
C_GetOperationState	Returns the session's cryptographic status
C_SetOperationState	Sets the session's cryptographic status
C_Login	Executes the user login
C_Logout	Executes the user logout
Object Management Functions	
C_CreateObject	Creates a new object on the token
C_CopyObject	Creates a new copy of the specified object
C_DestroyObject	Deletes an object on the token
C_GetObjectSize	Returns the size of the specified object
C_GetAttributeValue	Reads the value of an attribute of the specified object
C_SetAttributeValue	Sets the value of an attribute of the specified object
C_FindObjectsInit	Initializes the object lookup mechanism
C_FindObjects	Returns the next object found
C_FindObjectsFinal	Terminates the current objects lookup
Ciphering Functions	
C_EncryptInit	Initializes the ciphering operation
C_Encrypt	Encrypts a single data set
C_EncryptUpdate	Continues the current ciphering operation adding new data being ciphered
C_EncryptFinal	Terminates the current ciphering operation
Deciphering Functions	
C_DecryptInit	Initializes the deciphering operation
C_Decrypt	Deciphers a single set of data
C_DecryptUpdate	Continues the current deciphering operation adding new data being deciphered
C_DecryptFinal	Terminates the current deciphering operation
Hashing Functions	
C_DigestInit	Initializes the hashing operation
C_Digest	Calculates the hash value of a single data set
C_DigestUpdate	Continues the current hashing operation adding new data to the hash calculation
C_DigestKey	Calculates the hash value of the specified Key object
C_DigestFinal	Terminates the current hashing operation
Digital Signing Functions	
C_SignInit	Initializes the digital signing operation
C_Sign	Signs a single set of data
C_SignUpdate	Continues the current digital signing operation adding new data being signed
C_SignFinal	Terminates the current digital signing operation
C_SignRecoverInit	Initializes the sign operation where data being signed are retrieved in the given digital signature
C_SignRecover	Signs a single data set retrieved using the specified digital sign
Digital-Sign-Check Functions	
C_VerifyInit	Initializes the digital signature-verification operation
C_Verify	Verifies the signature applied to a single data set
C_VerifyUpdate	Continue the current digital-sign-check operation adding new data being checked
C_VerifyFinal	Terminates the current digital-signature-verification operation
C_VerifyRecoverInit	Init the sign-verify operation where data being checked are retrieved in the given signature
C_VerifyRecover	Verifies the digital signature applied to a single data set retrieved by the given digital signature
Cryptographic Keys Management Functions	
C_GenerateKey	Generates a symmetric key
C_GenerateKeyPair	Generates a cryptographic key pair
C_WrapKey	Exports a key encrypted by using the specified key
C_UnwrapKey	Imports a key encrypted by using the specified key
C_DeriveKey	Generates a new symmetric key derived from a master key
Radom Numbers Generation Functions	
C_SeedRandom	Sets the seed value for random numbers generation
C_GenerateRandom	Generates a new random number
Advanced Cryptographic Functions	
C_DigestEncryptUpdate	Continues the current hash and encrypt operations adding new data being processed
C_DecryptDigestUpdate	Continues the current decrypt and hash operations adding new data being processed
C_SignEncryptUpdate	Continues the current sign and encrypt operations updating the data being processed
C_DecryptVerifyUpdate	Continues the current decrypt and verify operations updating the data being processed
Parallel Management Functions	
C_GetFunctionStatus	Backward compatibility, always returns CKR_FUNCTION_NOT_PARALLEL
C_CancelFunction	Backward compatibility, always returns CKR_FUNCTION_NOT_PARALLEL