Secure Patient Health Record – RSA PKCS11 Standard Wrapper

CS448 Capstone

Team B: David Stelter, Dylan Enloe, Holly Decker,

Majed Alhubdaib, Muath Alissa, Vincent Cao

Primary Author(s): Vincent Cao, Mauth Alissa

Date: March 13, 2010

Document Purpose: This document serves as documentation for the PKCS11 Standard wrapper which interfaces with the ACOS5 Dynamic-Link library. This document does not cover details related the the firefox extension Persona which the Graphical User Interface to the Secure Patient Record Store project. Refer too the Medivault Firefox Persona Extention Documentation for details regarding the UI and to doxgen files for detailed developers API calls pertaining PKCS Wrapper and XPCOM components.

Contents **Page**

Question/Answer

Q: Why did we chose to develop support for Windows rather than Linux platform?

It was not neccessarily a technical constraint that we decided to develop under Windows rather than Linux. Both platforms provided a starting point for support with the ACOS5 cryptographic engine, Linux being the OpenSC project; and for Windows platform, the ACOSPKCS11.DLL that come with the ACR100 Development Kit. In the end, it came down to being Windows because of course Windows had a wider user range, from industry to hospitals, to personal usage, which was our target group. Since ACOS complies with the RSA PCKS11 Standard we decided it was the best choice to develop for Windows platform, the code should be portable within any development environment with minimal changes.

Q: Why did we choose to develop using C++ rather than other languages?

The entire process was a learning experience. At first we decided to go with C++ as it was the language everyone in the group was familiar and had experience with or at least would be easier to pick up faster. C++ would also allowed us to interface with the PKCS DLL provided by ACOS within their development kit therefore in theory providing us with some leverage in calling the DLL functions to provide ability to encrypt, decrypt, sign, etc with relative ease. Programming within an Object-Oriented Paradigm would allow us to develope much cleaner, compact, easy to understand code as compared to C or other Non-Object Oriented Languages.

However, as we found out during the development process, in writing actual code we learned the hard way that support for PCKS11 standard went from sparse to nonexistent as compared to many other languages, including Java, Perl, Python just to name a few. However with those languages mentioned we didn't have to knowledge to leverage the acospkcs11.dll provided by ACOS to run the functionality of the ACR100.

Pros C++:

1. Leverage interface with acospkcs11.dll
2. OO Paradigm for easier and cleaner code
3. Platform independence and written to take advantage of platform specific features
4. Able to extend to system level programming if needed, low level communication with device

Cons C++:

1. Zero to sparse support for PKCS11 standard
2. Java and many other OO/Functional language provide greater support for PKCS11
3. Limited scope and functionality than Java language

Q: What knowledge and resources are required to build functionality upon the wrapper?

1. C++ knowledge, Object-Oriented Programming Paradigm
2. Knowledgeable with the RSA PKCS #11: Cryptographic Token Interface Standard
3. Acospkcs11.dll documentation

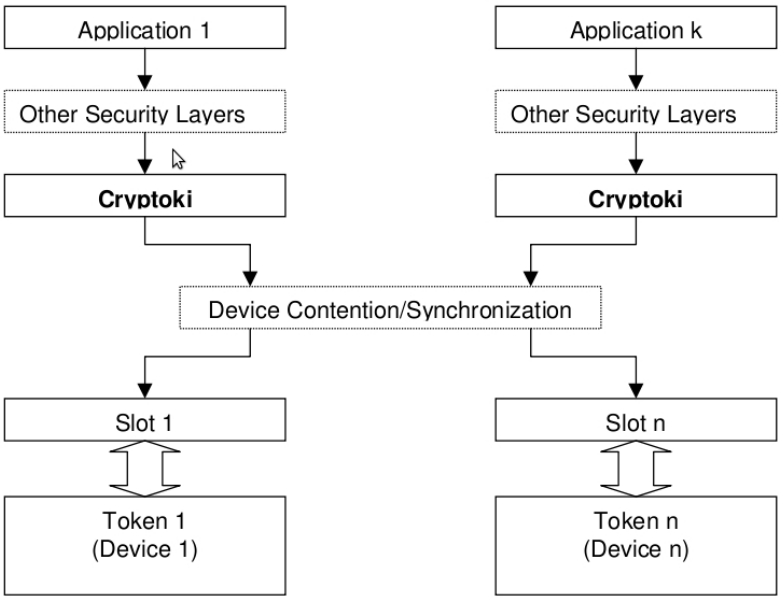
Wrapper Model:

RSA provides developers with abstract Cryptoki libary to be a generic interface to cryptographic operations or security services, although one certainly could build such operations and services with the functions that Cryptoki provides.

Cryptoki provides an interface to one or more cryptographic devices that are active in the system through a number of “slots”. Each slot, which corresponds to a physical reader or other device interface, may contain a token. A token is typically “present in the slot” when a cryptographic device is present in the reader. Of course, since Cryptoki provides a logical view of slots and tokens, there may be other physical interpretations. It is possible that multiple slots may share the same physical reader. The point is that a system has some number of slots, and applications can connect to tokens in any or all of those slots.

A cryptographic device can perform some cryptographic operations, following a certain command set; these commands are typically passed through standard device drivers. Cryptoki makes each cryptographic device look logically like every other device, regardless of the implementation technology. Thus the application need not interface directly to the device drivers (or even know which ones are involved); Cryptoki and the acospkcs.dll hides these details.

Our wrapper interface with the provided acospkcs11.dll which interface Cryptoki RSA Standard.



Development Environment:

Windows XP 32bit

- Visual Studio
  - Drivers & ACOS SDK
    - In order to leverage the acospkcs11.dll to communicate with the device, we needed other drivers as well
    - Installation of the SDK will install and provide all the drivers needed.
    - AdminTool
      - The AdminTool allow for developers to connect to card to the smartcard devices connected, provided that you have installed the SDK and it's driver.
      - The tools also allows developers to activate/deactivate of Acos5 card and it's cryptographic engine.
      - Allow for settings of global PIN and configuration of PIN setting.

- Be able to view certificates fields and objects

- Sample codes
  - ACOS provide a few sample codes to showcase simple functionality of the ACOS5 cards by leveraging it's provided DLL.
  - It is semi-useful for new developers to get a basic feel for how the code to call on functionality from ACOS5 is being handled.
- Source Headers
  - Header includes files for functionality of the RSA PKCS11 Standard
    - crytoki.h
      - file containing the top level include directives for building Win32 Cryptoki libraries and applications.
        - pckc11.h
          - defines macro and prototypes used within the Win32 Cryptoki libary.
        - pcks11f.h
          - This header file contains pretty much everything about all the Cryptoki function prototypes.
        - pkcs11t.h
          - contains structs and define variables and function prototypes for RSA PCKS11 Standard.

Current Functionality/API

bool initCrypto(void):
<Initilaizes the library to be able to call the Cryptoki functions>

- Makes sure that the DLL was not loaded before.
- Loads the DLL file.
- Gets the functions list from the DLL file.
- Initializes the Cryptoki library to be able to call them.
- Frees the library if any of the above failed then returns false.

void finalizeCrypto(void):

<Finalizes the Cryptoki>

- Do nothing if the library was not loaded.
- Do nothing if the Cryptoki functions list was not successfully accessed.

- Finalize the Crypto library if it was not loaded.
- Frees the library.

int getTokenCount(void):
<Counts the conneceted tokens>

- Gets only the number of cards which have tokens and store it in *ulSlotCount*.
- Returns *ulSlotCoun*t.

String* enumerateCards(void):
<Lists all the slots with a token present>

- Gets the number of cards which have tokens by calling **getTokenCount()** and store it in *TokenCount*.
- Creates a pointer *SlotsArray**, (return value), to an array, with a size of *TokenCount*, that contains the slots info.
- Search readers and store result in SlotWithTokenList.
- Gets all the slots info.
- Returns *SlotsArray*, a list of all slots with a token present.

bool selectCard(int SlotID, CK_UTF8CHAR* UserPIN, int pinlen):
<Connects the selected card to be use for subsequent operations>

- Lists all slots with a token present by calling **enumerateCards()**.
- Opens a session for the selected card, *SlotID*, to use for subsequent operations.
- Logs into the selected token using *UserPIN* and *pinlen*.
- Returns false if it failed to open a session or to log in.

CK_BYTE* encrypt(string plainText, string keyLabel, int &size):
<Encrypts data>

- Search for label which matches our key, keyLabel by calling **getPublicKey(keyLabel , key)**. *key* is the handler of the encryption
  key.
- Initializes an encryption operation.
- Gets the length of the data.
- Encrypts once to get the size.
- Encrypts the data.
- Returns the encrypted data.

bool encryptFile(string fileToEncrypt, string encryptedFile, string strKeyLabel):
<Reads a file, encrypts it, then writes the encrypted data into a new file>

- Opens the file that wanted to be encrypted, then reads the data, saves it, and then close the file.
- Encrypts the saved data by calling **encrypt()**.
- Creates a new file to write the encryped data with a header and a footer.
- Closes the encrypted file.

string decryptFile(CK_BYTE* cipherText, CK_ULONG size, string keyLabel):
<Decrypts data>

- Gets the private key.A
- Initializes a decryption operation.
- Decrypts once to get the size.
- Decrypts the data.

string LoadFile(string filename):
<Reads the encrypted file then decrypts it>

- Opens the file that wanted to be Decryped, reads the data and saves it, then closes the file.
- Skips over the first encoding of the data.
- Skips over the cipherText of the data.
- Decrypts the data by calling **decryptFile()**.

bool getPublicKey(string keyName, CK_OBJECT_HANDLE &pubKey):
<Gets the public key>

bool getPrivateKey(string keyName, CK_OBJECT_HANDLE &privKey):
<Gets the private key>

bool getKey(string keyName, CK_OBJECT_CLASS keyClass, CK_OBJECT_HANDLE &key):
<Gets the key>

- Sets up the template for the key and the certificate.
- Initializes the certificate search operation.
- Starts searching for certificate.
- Gets the subject field for each certificate it finds.
- Compares if it is the desired key.
- If no, Keep searching for another certificate.
- If yes, it finishes the certificate search operation.
- Gets the Key ID.
- Initializes the key search operation.
- Search for the key.

CK_BYTE* sign(string plainText, string keyLabel, int &size):
<Signs data>

- Gets private keys for signing, verify using public key.
- Initializes a message-digesting operation.
- Digests once to get the size.
- Starts digesting the data.
- Initializes a signature operation.
- Signs once to get the size.
- Starts signing the data.

bool signFile(string fileToSign, string signedFile, string strKeyLabel):
<Reads a file, signs it, then writes the signed date into a new file>

- Opens the file that wanted to be signed, then reads the data, saves it, and then close the file.
- Signs the saved data by calling the sign function.
- Creates a new file to write the signed data with a header and a footer.
- Closes the signed file.

string* listKeys(void):
<List all the keys in the token>

- Sets up the template for the certificate.
- Initializes the certificate search operation.
- Starts and keeps searching for certificate.
- Gets the subject field for each certificate it finds.
- Decrypts the subject line and get the information that we want.
- Adds the key to the list.
- Finishes the certificate search operation.
- Turns vector into an array.

CK_BYTE* Digest(string plainText, CK_ULONG &size):
<Digests data>

- Initialize the digesting operation.
- Gets the length of the data.
- Digests once to get the size.
- Digests the data.
- Returns the digeted data.

bool Verify(string plainText, CK_BYTE* signature, CK_ULONG sigSize, string keyLabel):
<Digests data>

- Gets the private keys for signing, verify using public key.
- Digests data by calling **Digest()**.
- Initializes the verification operation
- Verifies data.
- Returns false if it failed to verify, otherwise returns true.

DER Encoding:

Distinguished Encoding Rules, is a message transfer syntax specified by the ITU in X.690. The Distinguished Encoding Rules of ASN.1 is an International Standard drawn from the constraints placed on basic encoding rules (BER) encodings by X.509. DER encodings are valid BER encodings. DER is the same thing as BER with all but one sender's options removed.

DER is a subset of Basic Encoding Rules (BER) providing for exactly one way to encode an ASN.1 value. DER is intended for situations when a unique encoding is needed, such as in in cryptography and ensures that a data structure that needs to be digitally signed produces a unique serialized representation. DER can be seen as a canonical form of BER. DER is widely used for digital certificates such as X.509, which is the type of certificate that we as a group decided on using.

Putting all this together for the attribute and value *countryName = "US"* , you get this sequence. *CountryName* is defined as an attribute type in the X.520 standard. The attribute definition of *countryName* is written in ASN.1 like this: *countryName OBJECT IDENTIFIER ::= { attributeType 6 }*. Encoding this attribute and the value *US* into DER format you end up with a load of unfriendly octets like this: *06 03 55 04 06 13 02 55 53*.

We found late in the development process that C++ does not provide a easy to use or full support for DER. This meant that we had to come up with a makeshift process in order to parse the information out of the generated X.509 certificates. This bought about the simple Subject class within our source tree. Through reverse engineering, we wrote the class to be reponsible for parsing the DER encoded subject. In the DER encoded string the value 0x04 followed by any value followed by 0x13 means that it found a string literal. The next value is the length of that string literal and following the length there is a string proper of that length.

We know that there are exactly 6 of these in a proper subject field, no more no less. So this function searchs through the data stream for 6 of these strings. Knowing that they are always stored in a certain order we can then load them into the class.

This is by no means a good solution, but given the time constrants of the project and the shear arcaneness of the specs for this, it is the best that we could come up with

Future Functionality/Ideas:

1. Type and extension recognition
2. Ease of use from within Persona Interface
3. Certificate generation
4. Leveraging Firefox PKCS11 supported API