SOFTIDEA

SIPKCS

Application programming interface PKCS #11 for GNT USB Token

May 2012

Contents

Tables

1 Overview

GNT USB Token [1] ("*Token*") is a PKI hardware security module dedicated to applications with strong data security demands. *Token* is delivered with SIPKCS provider which implements the PKCS #11 version 2.20 developed by RSA Security Inc [2] ("*standard*"). Purpose of this document is to define the set of *standard* operations and mechanisms supported by *Token* and to disclose implementation details that helps developer to understate how *Token* features are exposed through the SIPKCS interface. In order to understand this document we assume the reader is familiar with documents [1] and [2].

2 Supported functions

Table 1 defines *standard* functions supported by SIPKCS. Any call to function not within this list shall fail with error code *CKR_FUNCTION_NOT_SUPPORTED*.

Table 1: Supported PKCS #11 functions

C_CloseAllSessions	C_CloseSession	C_CreateObject	C_Decrypt
C_DecryptInit	C_DestroyObject	C_Digest	C_DigestFinal
C_DigestInit	C_DigestUpdate	C_Encrypt	C_EncryptInit
C_Finalize	C_FindObjects	C_FindObjectsFinal	C_FindObjectsInit
C_GenerateKey	C_GenerateKeyPair	C_GenerateRandom	C_GetAttributeValue
C_GetFunctionList	C_GetInfo	C_GetMechanismInfo	C_GetMechanismList
C_GetObjectSize	C_GetSessionInfo	C_GetSlotInfo	C_GetSlotList
C_GetTokenInfo	C_Initialize	C_InitPIN	C_Login
C_Logout	C_OpenSession	C_SetAttributeValue	C_SetPIN
C_Sign	C_SignInit	C_UnwrapKey	C_Verify
C_VerifyInit	C_WaitForSlotEvent	C_WrapKey	

3 Token configuration

For use with SIPKCS interface *Token* has to be configured first[1] using the administration application GINIT [3]. GINIT provides the predefined option for such configuration. In this chapter the configuration details of *Token* configured for SIPKCS are explained.

3.1 Users and passwords

Assignment of user types recognized by *Token* to user types recognized by *standard* is as follows. PKCS #11 "*normal user*" is mapped to *Token* user authenticated by password *PIN1*. PKCS #11 "*security officer*" is mapped to *Token* user authenticated by **both** passwords *PIN2* **and** *APW*. Token which has been initialized for SIPKCS has always passwords *PIN2* and *APW* equal. The purpose of this is explained in more details in the next chapter.

1 Depending on the purchase option might be your Token was already configured for SIPKCS. In the next versions of SIPKCS we plan to support Token configuration using *C_InitToken* and GINIT no more shall be necessary.

3.1.1 Role of *PIN2*

Area *MEM3* (see chapter 3.3) is used as a configuration memory containing public accessible data (SIPKCS version supported by *Token*) necessary for SIPKCS operation. This data is written down by the administrator during the configuration process just after the *Token* is issued. *Token* specification does not allow to grant the *APW* authenticated administrator the exclusive write access to any MEM area. For this purpose we identify administrator with both *APW* and *PIN2* and configure *MEM3* for *PIN2* write access. This allows the administrator after committing issue operation to login using *PIN2* and write down the configuration data to the *MEM3* area.

3.2 RSA slots

Token configured for SIPKCS comprises 17 RSA slots logically divided into two groups. Slots 1-9 including are designated as non extractable and are dedicated for RSA keys with private key attribute value *CKA_EXTRACTABLE=FALSE*. Slots 10-17 including are designated as extractable and are dedicated for RSA keys with private key attribute value *CKA_EXTRACTABLE=TRUE*. SIPKCS does not make use of RSA slot 0. Values of RSA slots attributes are defined in Table 2. The maximal count of RSA key pairs which can be stored onto the *Token* is constrained by the count of RSA slots. This count is also constrained by availability of free memory in *MEM* areas for all PKCS #11 attributes of key objects.

Table 2: Attributes of RSA slot

Attribute	RSA slots 1-9	RSA slots 10-17
exportable	no	yes
importable	yes	yes
generatable	yes	yes
exportable only during generation	no	no
deletable	yes	yes
PIN1 accessible	yes	yes
PIN2 accessible	no	no
encryption / decryption allowed	yes	yes
digital signature creation / verification allowed	yes	yes
symmetric key wrap / unwrap allowed	yes	yes

3.3 MEM areas

Token configured for SIPKCS comprises 3 MEM areas. Attributes of each of the MEM areas are summarized in Table 3. *MEM3* contains configuration data mentioned in chapter 3.1.1. *MEM1* and *MEM2* are equipped with a file system and are dedicated to storage of PKCS #11 objects with *CKA_TOKEN=TRUE*. Private objects with *CKA_PRIVATE=TRUE* are stored in *MEM1*. Public objects with *CKA_PRIVATE=FALSE* are stored in *MEM2*. Lengths of MEM areas are optimized so that the typical PKCS #11 enabled application[1] makes the effective use of the storage capacity.

1 For example PGP

Table 3: Attributes of MEM areas

	MEM1	***MEM2***	***MEM3***
Usage	private objects	public objects	configuration data
Length [B]	9155	16909	16
Authentication required for read	PIN1	free	free
Authentication required for write	PIN1	free	PIN2

4 Conformance with *standard*

SIPKCS implements *standard* PKCS #11 version 2.20 [2]. List of identifiers of mechanism supported by SIPKCS is defined in document [1] in Table 7 Conformance of cryptographic operations with standards.

The following types of objects are supported:

- *CKO_PUBLIC_KEY* and *CKO_PRIVATE_KEY* of type *CKK_RSA*,
- *CKO_SECRET_KEY* of types *CKK_AES*, *CKK_DES*, *CKK_DES3*,
- *CKO_CERTIFICATE* of types *CKC_X_509* and *CKC_X_509_ATTR_CERT*,
- *CKO_DATA*.

Implementation specifics are enumerated in the following list:

- two new mechanisms are supported (*CKM_AES_OFB*, *CKM_AES_CTR*), which will be included in the new version of the standard 2.30 [4],
- use of cryptographic operations in multipart mode is supported only for hash,
- the following object attributes are not supported: *CKA_WRAP_WITH_TRUSTED*, *CKA_WRAP_TEMPLATE, CKA_UNWRAP_TEMPLATE, CKA_CHECK_VALUE*, *CKA_ALLOWED_MECHANISMS, CKA_ALWAYS_AUTHENTICATE, CKA_CERTIFICATE_CATEGORY, CKA_HASH_OF_SUBJECT_PUBLIC_KEY, CKA_HASH_OF_ISSUER_PUBLIC_KEY, CKA_JAVA_MIDP_SECURITY_DOMAIN,*
- some other types not necessary for mechanisms and operations provided by Token are not supported.

5 Arguments of *C_Initialize*

The most simple way how to call *C_Initialize* is with argument *pInitArgs=NULL_PTR*. Doing so application indicates it will not call SIPKCS from multiple threads. In such case SIPKCS internally creates the helper thread for monitoring of *Token* hardware events (insertion and removal of *Tokens*) and each call to *C_GetSlotList* will reflect the actual state. If application denies to create threads setting flag *CKF_LIBRARY_CANT_CREATE_OS_THREADS,* the internal monitoring thread is not created and every call to *GetSlotList* shall return the snapshot of the state retrieved during SIPKCS initialization regardless of the real state. If however the application creates the monitoring thread by its own which blocks in the call *C_WaitForSlotEvent*, after the hardware

event also call to *C_GetSlotList* will reflect the actual state.

If application does not allow to use OS synchronization primitives clearing the flag *CKF_OS_LOCKING_OK*, the method call fails, because SIPKCS requires this. Even if application supplies its own mutexes SIPKCS always makes use of OS synchronization primitives.

6 Multithreading and multiprocessing

SIPKCS in any case can handle multithreaded access. Multiple processes can access the Token in parallel but all of them must access it through SIPKCS. If other process uses different interface that does not rely on SIPKCS (e. g. GNTAPI) the function call could fail.

7 RSA key backup

Private RSA keys are sensitive data. Its unavailability (e. g. if lost, or in the case of device failure) can make troubles. In order to achieve high availability it can be suitable to create backup of key so that it can be restored later if necessary[1]. Even the standard PKCS #11 does not directly provide mechanisms for backup and restore it is possible to implement such system based on the standard. Example implementation is in the following chapters.

Note: For implementation of backup system it is necessary that key at least one time leaves the Token. This can represent the security weakness when compared to the most secure way where RSA key is never extracted from within Token.

7.1.1 Backup

- Call *C_GenerateKeyPair* to generate RSA key pair K1 with atribute *CKA_EXTRACTABLE=TRUE*,
- call *C_GetAttributeValue* to export K1 and store it secure on the backup media,
- call*C_DestroyObject* to destroy K1 on *Token*,
- call *C_CreateObject* to create RSA key pair K2 as a copy of K1 with attribute *CKA_EXTRACTABLE=FALSE.*

7.1.2 Restore

- Call *C_CreateObject* to create RSA key pair Kr as a copy of K1 with attribute *CKA_EXTRACTABLE=FALSE.*

1 Architecture based on root CA allows for simple revocation of lost key and assignment of new key what can in some cases (e. g. for signing keys) eliminate need for backup.

8 Examples

In this chapter we will show sample code making calls of Token through SIPKCS in ANSI C. Sample assume that the user password is "user". More samples are available on the installation media or on the manufacturer web site http://www.softidea.sk. Code of the main program is as follows.

```
#define MAX_SLOTS 32
#define INPUT_DATA_LEN 64
#define MAX_DATA_LEN 256

#include <stdio.h>
#include <memory.h>
#include "CRYPTOKI.H"

CK_SESSION_HANDLE hSession = CK_INVALID_HANDLE;
CK_OBJECT_HANDLE  hPublicKey = CK_INVALID_HANDLE;
CK_OBJECT_HANDLE  hPrivateKey = CK_INVALID_HANDLE;
CK_OBJECT_HANDLE  hSecretKey = CK_INVALID_HANDLE;
CK_OBJECT_HANDLE  hUnwrappedKey = CK_INVALID_HANDLE;

CK_BYTE inData[INPUT_DATA_LEN];

int main()
{
   CK_RV rv;
   CK_SLOT_ID slotList[MAX_SLOTS];
   CK_ULONG ulSlots = MAX_SLOTS;
   CK_UTF8CHAR pin[] = {'u','s','e','r'};
   //Initialize SIPKCS
   rv = C_Initialize(0);
   //Get slots with token
   rv = C_GetSlotList(1,slotList, &ulSlots);
   if(0 == ulSlots)
   {
      printf("no token present\n");
      goto finish;
   }
   //Open session with first token
   rv = C_OpenSession(slotList[0],CKF_SERIAL_SESSION | CKF_RW_SESSION, 0,0,&hSession);
   //Generate random input data
   rv = C_GenerateRandom(hSession, inData, INPUT_DATA_LEN);
   //Login user
   rv = C_Login(hSession, CKU_USER, pin,sizeof(pin));
   //Commit tests
   rv = FindOrGenerateKeys();
   rv = CryptRSA();
   rv = SignRSA();
   rv = CryptAES();
   rv = Wrap();
finish:
   //Logout, Close session, Finalize SIPKCS
   if(CK_INVALID_HANDLE != hSession)
   {
      rv = C_Logout(hSession);
      rv = C_CloseSession(hSession);
   }
   rv = C_Finalize(0);
   printf("press enter\n");
   fgetc(stdin);
        return 0;
}
```

8.1 Search and key generation

This example demonstrates search for objects stored on the Token and generation of cryptographic keys.

```
int FindOrGenerateKeys()
{
   CK_RV rv;
   CK_MECHANISM mechanism = {0,0,0};
   CK_ULONG modulusBits = 2048;
   CK_ULONG aesBytes = 16;
   CK_BYTE publicExponent[] = {1,0,1};
   CK_BYTE subject[] = {'t','e','s','t'};
   CK_BYTE id[] = {'1','2','3'};
   CK_BBOOL bTrue = CK_TRUE;
   CK_ULONG ulObjectsFound = 0;

   //Prepare templates
   CK_ATTRIBUTE publicKeyTemplate[] = {
      {CKA_TOKEN, &bTrue, sizeof(bTrue)},
     {CKA_ENCRYPT, &bTrue, sizeof(bTrue)},
     {CKA_VERIFY, &bTrue, sizeof(bTrue)},
     {CKA_WRAP, &bTrue, sizeof(bTrue)},
     {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
     {CKA_PUBLIC_EXPONENT, publicExponent, sizeof (publicExponent)}
   };
   CK_ATTRIBUTE privateKeyTemplate[] = {
     {CKA_TOKEN, &bTrue, sizeof(bTrue)},
     {CKA_PRIVATE, &bTrue, sizeof(bTrue)},
     {CKA_SUBJECT, subject, sizeof(subject)},
     {CKA_ID, id, sizeof(id)},
     {CKA_SENSITIVE, &bTrue, sizeof(bTrue)},
     {CKA_DECRYPT, &bTrue, sizeof(bTrue)},
     {CKA_SIGN, &bTrue, sizeof(bTrue)},
     {CKA_UNWRAP, &bTrue, sizeof(bTrue)}
   };
   CK_ATTRIBUTE secretKeyTemplate[] = {
     {CKA_TOKEN, &bTrue, sizeof(bTrue)},
     {CKA_ENCRYPT, &bTrue, sizeof(bTrue)},
     {CKA_DECRYPT, &bTrue, sizeof(bTrue)},
     {CKA_VALUE_LEN, &aesBytes, sizeof(aesBytes)},
     {CKA_ID, id, sizeof(id)},
     {CKA_PRIVATE, &bTrue, sizeof(bTrue)}
   };
   //search for public key
   rv = C_FindObjectsInit(hSession, publicKeyTemplate, 6);
   rv = C_FindObjects(hSession, &hPublicKey, 1, &ulObjectsFound);
   rv = C_FindObjectsFinal(hSession);
   //search for private key
   rv = C_FindObjectsInit(hSession, privateKeyTemplate, 8);
   rv = C_FindObjects(hSession, &hPrivateKey, 1, &ulObjectsFound);
   rv = C_FindObjectsFinal(hSession);
   //search for AES key
   rv = C_FindObjectsInit(hSession, secretKeyTemplate, 6);
   rv = C_FindObjects(hSession, &hSecretKey, 1, &ulObjectsFound);
   rv = C_FindObjectsFinal(hSession);
   //if not both RSA keys found, generate new keypair
   if(CK_INVALID_HANDLE == hPrivateKey || CK_INVALID_HANDLE == hPublicKey)
   {
      mechanism.mechanism = CKM_RSA_PKCS_KEY_PAIR_GEN;
      rv = C_GenerateKeyPair( hSession, &mechanism, publicKeyTemplate, 6, privateKeyTemplate, 8, &hPublicKey,
        &hPrivateKey);
   }
   //if no AES key found, generate new one
   if(CK_INVALID_HANDLE == hSecretKey)
   {
      mechanism.mechanism = CKM_AES_KEY_GEN;
      rv = C_GenerateKey( hSession, &mechanism, secretKeyTemplate, 6, &hSecretKey);
   }
```

```
   return rv;
}
```

8.2 Encryption using RSA

This example demonstrates data encryption and decryption using RSA.

```
int CryptRSA()
{
   CK_RV rv;
   CK_BYTE outData[MAX_DATA_LEN];
   CK_BYTE roundtripData[MAX_DATA_LEN];
   CK_ULONG ulOutDataLen = MAX_DATA_LEN;
   CK_MECHANISM mechanism = {0,0,0};

   //Define mechanism
   mechanism.mechanism = CKM_RSA_PKCS;
   //Encrypt
   rv = C_EncryptInit(hSession, &mechanism, hPublicKey);
   rv = C_Encrypt(hSession, inData, INPUT_DATA_LEN, outData, &ulOutDataLen);
   //Decrypt
   rv = C_DecryptInit(hSession, &mechanism, hPrivateKey);
   rv = C_Decrypt(hSession, outData, ulOutDataLen, roundtripData, &ulOutDataLen);
   //Compare
   if(ulOutDataLen != INPUT_DATA_LEN)
      printf("length differ !\n");
   else if(memcmp(roundtripData, inData, INPUT_DATA_LEN))
      printf("data differ !\n");
   else
      printf("success.\n");
   return rv;
}
```

8.3 Digital signature

This example demonstrates data signing and verification of digital signature.

```
int SignRSA()
{
   CK_RV rv;
   CK_BYTE signature[MAX_DATA_LEN];
   CK_ULONG ulSignatureLen = MAX_DATA_LEN;
   CK_MECHANISM mechanism = {0,0,0};

   //Define mechanism
   mechanism.mechanism = CKM_SHA1_RSA_PKCS;
   //Sign
   rv = C_SignInit(hSession, &mechanism, hPrivateKey);
   rv = C_Sign(hSession, inData, INPUT_DATA_LEN, signature, &ulSignatureLen);
   //Verify
   rv = C_VerifyInit(hSession, &mechanism, hPublicKey);
   rv = C_Verify(hSession, inData, INPUT_DATA_LEN, signature, ulSignatureLen);
   //Evaluate
   if(CKR_OK != rv)
      printf("signature invalid !\n");
   else
      printf("success.\n");
   return rv;
}
```

8.4 Encryption using symmetric cipher

This example demonstrates data encryption and decryption using symmetric cipher.

```
int CryptAES()
{
   CK_RV rv;
   CK_BYTE iv[16];
   CK_BYTE outData[MAX_DATA_LEN];
   CK_BYTE roundtripData[MAX_DATA_LEN];
   CK_ULONG ulOutDataLen = MAX_DATA_LEN;
   CK_MECHANISM mechanism = {0,0,0};

   //Generate random IV
   rv = C_GenerateRandom(hSession, iv, sizeof(iv));
   //Define mechanism
   mechanism.mechanism = CKM_AES_OFB;
   mechanism.pParameter = iv;
   mechanism.ulParameterLen = sizeof(iv);
   //Encrypt
   rv = C_EncryptInit(hSession, &mechanism, hSecretKey);
   rv = C_Encrypt(hSession, inData, INPUT_DATA_LEN, outData, &ulOutDataLen);
   //Decrypt
   rv = C_DecryptInit(hSession, &mechanism, hSecretKey);
   rv = C_Decrypt(hSession, outData, ulOutDataLen, roundtripData, &ulOutDataLen);
   //Compare
   if(ulOutDataLen != INPUT_DATA_LEN)
      printf("length differ !\n");
   else if(memcmp(roundtripData, inData, INPUT_DATA_LEN))
      printf("data differ !\n");
   else
      printf("success.\n");
        return 0;
}
```

8.5 Wrapping of symmetric key

This example demonstrates wrapping and unwrapping of the symmetric key using RSA.

```
int Wrap()
{
   CK_RV rv;
   CK_BYTE iv[16];
   CK_KEY_TYPE ktAES = CKK_AES;
   CK_BYTE wrappedKey[MAX_DATA_LEN];
   CK_ULONG ulWrappedKeyLen = MAX_DATA_LEN;
   CK_MECHANISM mechanism = {0,0,0};
   CK_BYTE id[] = {'3','2','1'};
   CK_BBOOL bTrue = CK_TRUE;
   CK_BYTE outData[MAX_DATA_LEN];
   CK_BYTE roundtripData[MAX_DATA_LEN];
   CK_ULONG ulOutDataLen = MAX_DATA_LEN;
   CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;

   //Prepare template for the unwrapped key
   CK_ATTRIBUTE secretKeyTemplate[] = {
     {CKA_CLASS, &keyClass, sizeof(keyClass)},
     {CKA_KEY_TYPE, &ktAES, sizeof(ktAES)},
     {CKA_TOKEN, &bTrue, sizeof(bTrue)},
     {CKA_ENCRYPT, &bTrue, sizeof(bTrue)},
     {CKA_DECRYPT, &bTrue, sizeof(bTrue)},
     {CKA_PRIVATE, &bTrue, sizeof(bTrue)},
     {CKA_ID, id, sizeof(id)}
   };

   //Define mechanism
   mechanism.mechanism = CKM_RSA_PKCS;
```

```
   //Wrap
   rv = C_WrapKey(hSession, &mechanism, hPublicKey, hSecretKey, wrappedKey, &ulWrappedKeyLen);
   //Unwrap
   rv = C_UnwrapKey(hSession, &mechanism, hPrivateKey, wrappedKey, ulWrappedKeyLen, secretKeyTemplate, 7,
&hUnwrappedKey);
   //Encrypt using original key
   rv = C_GenerateRandom(hSession, iv, sizeof(iv));
   mechanism.mechanism = CKM_AES_OFB;
   mechanism.pParameter = iv;
   mechanism.ulParameterLen = sizeof(iv);
   rv = C_EncryptInit(hSession, &mechanism, hSecretKey);
   rv = C_Encrypt(hSession, inData, INPUT_DATA_LEN, outData, &ulOutDataLen);
   //Decrypt using unwrapped key
   rv = C_DecryptInit(hSession, &mechanism, hUnwrappedKey);
   rv = C_Decrypt(hSession, outData, ulOutDataLen, roundtripData, &ulOutDataLen);
   //Compare
   if(ulOutDataLen != INPUT_DATA_LEN)
      printf("length differ !\n");
   else if(memcmp(roundtripData, inData, INPUT_DATA_LEN))
      printf("data differ !\n");
   else
      printf("success.\n");
   return 0;
}
```

9 References

[1] GNT USB Token - datasheet, SoftIdea, s.r.o. , May 2011, http://www.softidea.sk/gnt_datasheet_en.pdf

[2] PKCS #11 v2.20: Cryptographic Token Interface Standard, RSA Laboratories, June 2004, http://www.rsasecurity.com

[3] GINIT - User manual, SoftIdea, s.r.o. , May 2011, http://www.softidea.sk/ginit_manual_en.pdf

[4] PKCS #11 Mechanisms v2.30: Cryptoki – Draft 7, July 2009, http://www.rsasecurity.com

SoftIdea s.r.o.
Sliačska 10, 831 02 Bratislava
tel.: +421 2 444 60 444
fax.: +421 2 446 40 441
http://www.softidea.sk
info@softidea.sk