Linux on Z and LinuxONE

openCryptoki - An Open Source Implementation of PKCS #11

IBM

SC34-7730-01

Edition notice

This edition applies to the openCryptoki version 3.17 and to all subsequent versions and modifications until otherwise indicated in new editions.

Contents

About this document

openCryptoki is an open source implementation of the *Cryptoki* API defined by the PKCS #11 Cryptographic Token Interface Standard.

This documentation is intended for the following audience:

openCryptoki administrators manage the so called tokens that are plugged into the openCryptoki framework. They are responsible for adding these tokens to the slots of openCryptoki and, if applicable, for configuring these tokens. They also use either openCryptoki common tools or token-specific tools for administrating the configured tokens.

Application programmers write PKCS #11 programs that exploit the cryptographic services provided by a configured openCryptoki token. The services of such a token either exploit IBM Z® cryptographic hardware or they are also backed by software tokens (Soft token).

This documentation is divided into the following parts:

- Part 1, “Common features of openCryptoki,” on page 1 describes the openCryptoki architecture and informs about configuration actions that must be performed for all applications that want to exploit these openCryptoki features. Most of the provided information is of interest for both, **openCryptoki administrators** and **application programmers**.
- Part 2, “Common tools of openCryptoki,” on page 23 documents management and key migration tools that are helpful for **openCryptoki administrators** to enable openCryptoki exploitation through programs.
- Part 3, “Token specifications,” on page 39 presents to **application programmers** the documentation of the token-specific mechanisms and information about the contents of a token directory. If a token needs additional token-specific configuration, before it can be accessed by a cryptographic application, **openCryptoki administrators** find the required information here.
- Part 4, “Programming basics and user scenarios,” on page 99 documents the basic structure of openCryptoki applications and presents a simple, but complete code sample for an RSA key pair generation. This part also provides a user scenario showing how to set up a Soft token on a server for use from an application on a remote client.

Summary of changes

Track the changes of this document for each new edition.

Edition SC34-7730-01: Updates for openCryptoki version 3.17

- A new information unit describing IBM®-specific mechanisms and attributes has been added (see Chapter 16, “IBM-specific mechanisms and features for openCryptoki ,” on page 87).
- Conceptional information on the handling of objects and attributes in PKCS #11 has been added.
- openCryptoki code samples have been transferred from *libica Programmer's Reference*, SC34-2602, and *Exploiting Enterprise PKCS #11 using openCryptoki*, SC34-2713, into this current edition.
- All openCryptoki-related information from *libica Programmer's Reference*, SC34-2602, has been integrated into this current edition.
- Comprehensive extensions have been applied to the documentation of all tokens in Part 3, “Token specifications,” on page 39.
- A new topic to assist you in solving problems while working with openCryptoki has been included (see “Trouble shooting” on page 116).
- The **p11sak** offers a new option to obtain a listing of keys in a long output format.

Part 1. Common features of openCryptoki

Read an introduction to openCryptoki and learn about its features that are relevant for all exploiting applications.

The following topics explain the purpose of openCryptoki within the PKCS #11 standard and introduce the concept of the openCryptoki framework:

Chapter 1. Introducing PKCS #11 and openCryptoki

The Public-Key Cryptography Standards (PKCS) comprise a group of cryptographic standards that provide guidelines and application programming interfaces (APIs) for the usage of cryptographic methods. This document describes the use of openCryptoki, which is an open source implementation of a C/C++ API standard defined by PKCS #11, called *Cryptoki*.

openCryptoki is used for devices that hold cryptographic information and perform cryptographic functions.

Though the openCryptoki standard offers C header files, you can also implement the interface in other programming languages than C or C++.

What is PKCS #11?

PKCS #11 is a popular cryptographic standard for the support of cryptographic hardware. It defines a platform-independent API called *Cryptoki* to access cryptographic devices, such as hardware security modules (HSMs). With this API, applications can address these cryptographic devices through so-called tokens and can perform cryptographic functions as implemented by these tokens. This standard, first developed by the RSA Laboratories in cooperation with representatives from industry, science, and governments, is now an open standard lead-managed by the *OASIS PKCS 11 Technical Committee*.

PKCS #11 can support so called hardware tokens which may be cryptographic accelerators or hardware security modules (HSMs). The PKCS #11 standard is independent of specific cryptographic hardware, yet allows to deal with many hardware specific implementations. It can support the use of multiple different token types. Due to the popularity of PKCS #11, many software products that perform cryptographic operations, provide plug-in mechanisms, which, if configured, will redirect cryptographic functions to a PKCS #11 library of mechanisms. For example, the IBM WebSphere® Application Server and the IBM HTTP Server can be configured to use a PKCS #11 library.

The *Cryptoki* API provides access to a number of so-called slots. A slot is a possibility to connect to a cryptographic device (for example, to an IBM Crypto Express adapter). Typically, a slot contains a token, while a cryptographic device is connected to the slot. An application can connect to multiple tokens in a subset of those slots.

In addition, further cryptographic libraries can call PKCS #11 functions, for example, Java Cryptography Architecture (JCA), IBM Global Security Kit (*GSKit*), GnuTLS, or, in case of OpenSSL, by using for example, a PKCS #11 engine from the OpenSC project.

Cryptoki abstracts from the cryptographic device, that is, it makes each device look logically like every other device, regardless of the implementation technology. Whether it is a specific hardware device that requires a special device driver or a solution completely based on software (for example, a client for a cryptographic service), the *Cryptoki* API looks exactly the same. Hence, applications (application programmers) only interact with *Cryptoki*. The concrete *Cryptoki* implementation takes care of the interaction with the selected token.

Cryptoki is likely to be implemented as a library supporting the functions in the interface, and applications will be linked to the library. It follows an object-based approach, addressing the goals of technology independence (any kind of HW device) and resource sharing. It also presents to applications a common, logical view of the device that is called a cryptographic *token*. PKCS #11 assigns a slot ID to each token. An application using the *Cryptoki* API identifies the token that it wants to access by specifying the appropriate slot ID.

For more information about PKCS #11, refer to this URL:

`PKCS #11 Cryptographic Token Interface Standard`

What is openCryptoki?

The PKCS #11 standard comprises the definition of an API called *Cryptoki* (from *cryptographic token interface*). However, the term PKCS #11 is often used instead, to refer to the API as well as to the standard that defines it. openCryptoki in turn is an open source implementation of *Cryptoki*. As such, openCryptoki provides a standard programming interface between applications and all kinds of portable cryptographic devices.

openCryptoki consists of an implementation of the PKCS #11 *Cryptoki* API, a slot manager, a set of slot token dynamic link libraries (STDLLs), and an API for these STDLLs. For example, the EP11 token type is a STDLL introduced with openCryptoki version 3.1.

openCryptoki provides support for several cryptographic algorithms according to the PKCS #11 standard. The openCryptoki library loads the tokens that provide hardware or software specific support for cryptographic functions.

openCryptoki can be used directly through the openCryptoki shared library (C API) from all applications which are written in a language that provides a foreign language interface for C.

openCryptoki is available for major Linux distributions, for example, Red Hat Enterprise Linux, SUSE Linux Enterprise Server, or Ubuntu.

For more information about the openCryptoki services, or about the interfaces between the openCryptoki main module and its tokens, see

```
https://github.com/opencryptoki/opencryptoki.
```

Chapter 2. Architecture and components of openCryptoki

As implementation of the PKCS #11 API (*Cryptoki*), openCryptoki allows interfacing with devices (such as a Crypto Express adapter) that hold cryptographic information and perform cryptographic functions. openCryptoki provides application portability by isolating the application from the details of the cryptographic device. Isolating the application also provides an added level of security because all cryptographic information stays within the device.

openCryptoki consists of a slot manager and an API for slot token dynamic link libraries (STDLLs). The slot manager runs as a daemon, provides the number of configured tokens to applications, and it interacts with the tokens (that are used by the applications) using Unix domain sockets or a shared memory segment. For each device with which a token should be associated, this token must be defined in a slot in the openCryptoki configuration file (`/etc/opencryptoki/opencryptoki.conf`). The shared memory segment allows for proper sharing of state information between applications to help ensure conformance with the PKCS #11 specification.

Figure 1 on page 5 shows the architecture of openCryptoki:

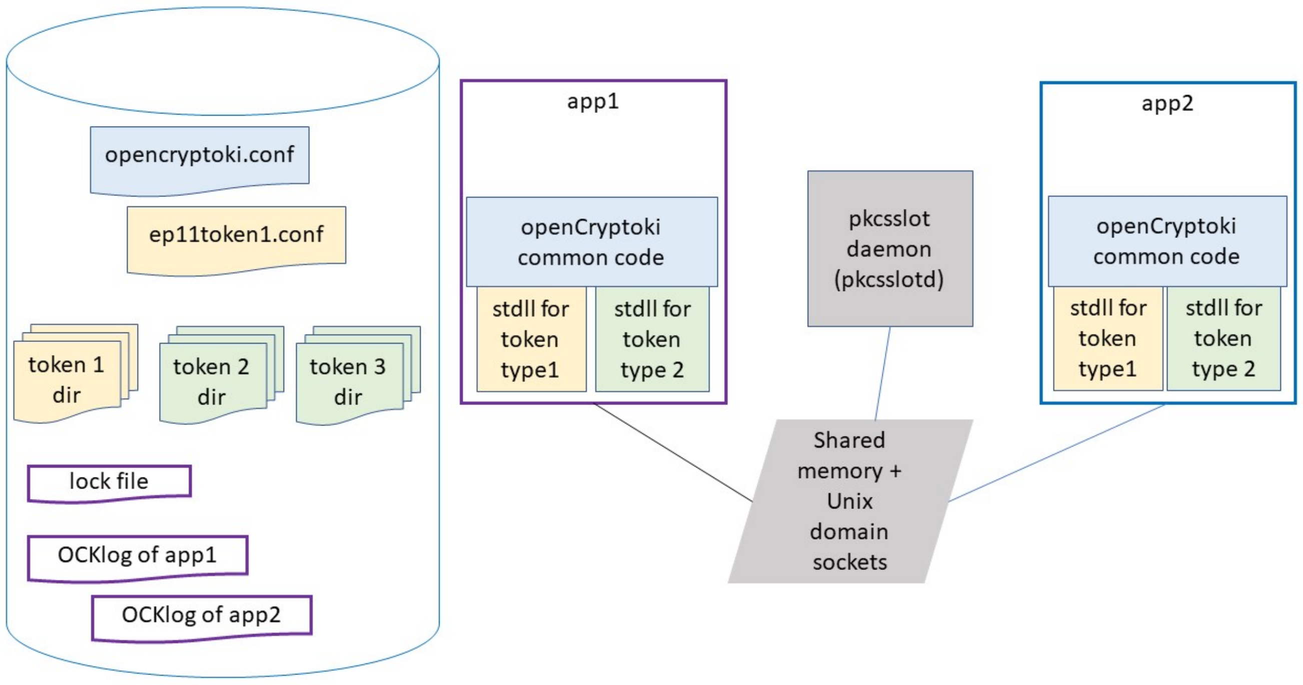


Figure 1. openCryptoki architecture

openCryptoki supports different token types for SW tokens and for various forms of HW support, for example, IBM Crypto Express adapters. openCryptoki allows to manage multiple tokens that can be used in parallel by one or more processes. These multiple tokens can have the same or different types.

Users can configure openCryptoki and in particular, the set of tokens including their respective token types, using the `opencryptoki.conf` file (see Chapter 4, “Adjusting the openCryptoki configuration file,” on page 17). They can define and exploit multiple tokens of any token type, each with a different token name. For multiple EP11 tokens, users can configure them using a token-specific configuration file for each token instance. A sample of such a configuration file is shown in Figure 8 on page 42.

Each token uses a unique token directory. This token directory receives the token-individual information (like for example, key objects, user PIN, SO PIN, or hashes). Thus, the information for a certain token is separated from all other tokens. For example, for most Linux distributions, the CCA token directory is `/var/lib/opencryptoki/ccatok`. The CCA token is called `ccatok`, if there is only one instance of a CCA token, and no explicit name is defined in the openCryptoki configuration file.

Slots and tokens

Imagine the use of smart cards: In the same way a smart card is inserted into a smart card reader, a PKCS #11 token is inserted into a PKCS #11 slot, where a slot is identified by its ID. A token is library code that knows how to interface with the cryptographic hardware. However, there is no requirement for the token to use any hardware at all, and accordingly there is a so called soft token (described in Chapter 13, “Soft token,” on page 75) which represents a pure software library accessible via openCryptoki.

Each token is of a certain token-type, where a token-type is implemented by STDLLs. For example, all EP11 tokens use the libpkcs11_ep11.so STDLL (see also Figure 4 on page 18). A certain instance of a token is implemented by data structures allocated by an STDLL (and a token directory).

All tokens available to openCryptoki are configured in the `opencryptoki.conf` configuration file. Each token configuration in `opencryptoki.conf` defines the token type of the token by specifying the adequate STDLL. The `opencryptoki.conf` configuration file is used by all processes (applications) using (linking to) openCryptoki. Each process may call some or all tokens defined in `opencryptoki.conf`.

The PKCS #11 API provides a set of slot and token management functions. For example:

- `C_GetSlotList()` gets a list of available slots.
- `C_GetSlotInfo()` obtains information on each slot (for example, whether a token is present, or whether the token represents a removable device).
- `C_InitToken()` initializes an inserted token.
- `C_GetTokenInfo()` provides information on such a token (for example, whether a login is required to use the token, a count of failed log-ins, information on whether the token has a random number generator).
- `C_InitPIN()` and `C_SetPIN()` manage the PIN used to protect a token from unauthorized access.

View an example for how to obtain slot information using the `C_GetSlotInfo()` function:

```
CK_RV getSlotInfo(CK_SLOT_ID slotID, CK_SLOT_INFO_PTR slotInfo);

/*  typedef struct CK_SLOT_INFO {         */
/*    CK_UTF8CHAR slotDescription[64];    */
/*    CK_UTF8CHAR manufacturerID[32];     */
/*    CK_FLAGS flags;                     */
/*    CK_VERSION hardwareVersion;         */
/*    CK_VERSION firmwareVersion;         */
/*  } CK_SLOT_INFO;                       */

/* Flags:                      */
/* CKF_TOKEN_PRESENT           */
/* CKF_REMOVABLE_DEVICE        */
/* CKF_HW_SLOT                 */

{
       rc = C_GetSlotInfo(slotID, &slotInfo);
       if (rc != CKR_OK) {
          printf("Error getting slot information: %x \n", rc);
          return rc;
       }

       if ((slotInfo.flags & CKF_TOKEN_PRESENT) == CKF_TOKEN_PRESENT)
           printf("A token is present in the slot.\n");

       if ((slotInfo.flags & CKF_REMOVABLE_DEVICE) == CKF_REMOVABLE_DEVICE)
          printf("The reader supports removable devices.\n");

       if ((slotInfo.flags & CKF_HW_SLOT) == CKF_HW_SLOT)
          printf("The slot is a hardware slot.\n");   /* opposed to a SW slot implementing a soft
token  */

    return CKR_OK;
 }
```

Note: Linux on Z and LinuxONE do not support the Trusted Platform Module (TPM) token library.

Slot manager

The slot manager daemon (**pkcsslotd**) manages slots (and therefore the tokens plugged into these slots) in the system. Its main task is to coordinate token accesses from multiple processes. A fixed number of processes can attach themselves implicitly to **pkcsslotd** during openCryptoki initialization, so a static table in shared memory is used. The current limit of the table is 1000 processes using the subsystem. The daemon sets up this shared memory upon initialization and acts as a garbage collector thereafter, helping to ensure that only active processes remain registered.

The POSIX shared memory segments for the available tokens are located in path `/dev/shm` and are named either by their default names or by the name defined in the `opencryptoki.conf` file. For example, the shared memory segments may be located and named as shown in the following example:

```
[root@system01 shm]# pwd
/dev/shm
[root@system01 shm]# ls -l
...
-rw-rw----. 1 root pkcs11 82792 Apr  8 12:04 var.lib.opencryptoki.ccatok
-rw-rw----. 1 root pkcs11 82792 Apr  8 12:04 var.lib.opencryptoki.ep11tok
-rw-rw----. 1 root pkcs11 82792 Apr  8 12:04 var.lib.opencryptoki.lite /* legacy name: ICA Tok */
-rw-rw----. 1 root pkcs11 82792 Apr  8 12:04 var.lib.opencryptoki.swtok
...
```

The **pkcsslotd** daemon also uses one System V shared memory segment in addition to the mentioned POSIX shared memory segments. This System V shared memory segment can be listed with the **ipcs** command. It is used to share information about the processes currently using openCryptoki services and to share the global session count per slot.

The slot manager also maintains Unix domain sockets between all openCryptoki processes (API layer). There are the following sockets:

```
/var/run/pkcsslotd.socket
/var/run/pkcsslotd.admin.socket
```

When a process attaches to a slot and opens a session, **pkcsslotd** makes future processes aware that a process has a session open and locks out certain function calls, if the process needs exclusive access to the given token. The daemon constantly searches through its shared memory and ensures that when a process is attached to a token, this process is actually running. If an attached process terminates abnormally, **pkcsslotd** cleans up after the process and frees the slot for use by other processes.

Start the slot manager with the commands shown in "Starting the slot manager" on page 26.

Main API

The main API for the STDLLs lies in `/usr/lib/pkcs11/PKCS11_API.so`. This API includes all the functions as outlined in the PKCS #11 API specification. The main API provides each application with the slot management facility. The API also loads token-specific modules (STDLLs) that provide the contained operations (cryptographic operations and session and object management). STDLLs are customized for each token type and have specific functions, such as an initialization routine, to allow the token to work with the slot manager. When an application initializes the subsystem with the `C_Initialize` call, the API loads the STDLL shared objects for all the tokens that exist in the configuration (residing in the shared memory) and invokes the token-specific initialization routines. In addition, a connection to the **pkcsslotd** slot manager and its shared memory segment is established.

Roles and Sessions

PKCS #11 knows two different roles per token. Each role can authenticate itself by a PIN specific to that role and a token.

- **The security officer (SO)** initializes and manages the token and can set the PIN of the ***User***.
- **The (normal) User** can login to sessions, create and access private objects and perform cryptographic operations. ***Users*** can also change their PINs.

A session is a token-specific context for one or more cryptographic operations. It maintains the intermediate state of multi-part functions, like an encryption of a message that is worked on one network packet at a time. Roughly speaking, within one session only one cryptographic operation can be processed at a time, but a program may open multiple sessions concurrently.

There are different types of sessions: Each session is either a read-only session or a read-write session and each session is either a public session or a ***User*** session. Read-only sessions may not create or modify objects. A public session can only access public objects whereas a ***User*** session can access the ***User's*** private and public objects. All sessions of a token become ***User*** sessions after a login to one of the sessions of that token. Access to a specific token is controlled using PINs (***User*** or **security officer** PINs).

The PKCS #11 API provides session functions like:

- `C_OpenSession()` and `C_CloseSession()` are used to open and close a session. A parameter in the `C_OpenSession()` invocation indicates the type of the session.
- `C_Login()` and `C_Logout()` are used to toggle sessions from public to ***User*** sessions and reverse. In order to login, the ***User*** PIN is required.
- `C_GetSessionInfo()` provides information on the type of a session.
- `C_GetOperationState()` and `C_SetOperationState()` are used to checkpoint and restart a multi-part cryptographic operation. Note that not all operations may support saving and restoring the state of operations.

Functions and mechanisms

The PKCS #11 API defines a small set of generic cryptographic functions to do the following tasks:

- encrypting and decrypting messages,
- computing digests (also called *hashes*) of messages,
- signing messages and verifying signatures,
- generating symmetric keys or asymmetric key pairs and deriving keys,
- wrapping and unwrapping keys,
- generating random numbers.

With the exception of the functions to generate random numbers, these generic cryptographic functions accept a mechanism parameter that defines the specific instance of that function. This is depicted in Figure 2 where an encryption function takes an AES_CBC mechanism as argument to encrypt a message with AES encryption in the cipher block chaining (CBC) mode of operation, using a key (and an initialization vector not shown in the figure).

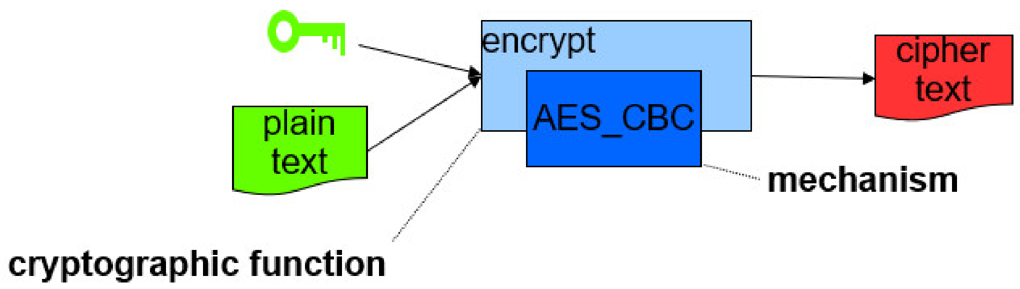


Figure 2. Functions and mechanisms

Each cryptographic function is executed in the context of a session. Most cryptographic functions must be initialized by calling an initialization function that takes a session parameter, a mechanism parameter and function specific parameters like keys. For a cryptographic function `Xyz` the initialization function is called `C_XyzInit()`. Once a function is initialized, the actual function invocation can take place: either as a single part function of the form `C_Xyz()` or as a multi-part function where one or more calls to a function of the form `C_XyzUpdate()` are finalized by a call to `C_XyzFinal()`.

Each token supports token-specific functions. The PKCS #11 API provides the function C_GetFunctionList() to obtain the functions available with a specific token. A mechanism describes a specific set of cryptographic operations. For example, the mechanism `CKM_AES_CBC` refers to AES encryption with the cipher block chaining (CBC) mode of operation. A mechanism may be required for performing one or more cryptographic functions, for example, the `CKM_AES_CBC` mechanism may be used to define encryption, decryption, wrapping and unwrapping functions whereas the mechanism `CKM_ECDSA_KEY_PAIR_GEN` which is a mechanism to generate keys for elliptic curve DSA signatures, only supports the key (pair) generation function.

The list of mechanisms supported depends on the tokens and can be queried using the `C_GetMechanismList()` function. Each mechanism has token-specific attributes like the set of supported functions, minimal and maximal supported key sizes, or a hardware support flag. These attributes are token-specific and can be queried with `C_GetMechanismInfo()`. Some mechanisms have mechanism parameters, for example, `CKM_AES_CBC` has a mechanism parameter to define the initialization vector (IV) required by the CBC mode of operation.

Slot token dynamic link libraries (STDLLs)

STDLLs are plug-in modules to the main API. They provide token-specific functions beyond the main API functions. Specific devices can be supported by building an STDLL for the device. Each STDLL must provide at least a token-type specific initialization function. If the device is an intelligent device, such as a hardware adapter that supports multiple mechanisms, the STDLL can be thin because much of the session information can be stored on the device. If the device only performs a simple cryptographic function, all of the objects and the device status must be managed by the STDLL. This flexibility allows STDLLs to support any cryptographic device.

Shared memory

The slot manager sets up its database in a region of shared memory. Since the maximum number of processes allowed to attach to **`pkcsslotd`** is finite, a fixed amount of memory can be set aside for token management. This fixed memory allocation for token management allows applications easier access to token state information and helps ensure conformance with the PKCS #11 specification. In addition, the slot manager (**`pkcsslotd`**) communicates with the API layer using Unix domain sockets.

Also, each token sets up a shared memory segment to synchronize token objects across multiple processes using the token.

Objects and keys

openCryptoki provides various functions to generate the applicable types of objects, for example, certain types of keys.

In addition, openCryptoki recognizes a number of classes of objects, as defined in the CK_OBJECT_CLASS data type. Object classes are defined with the objects that use them. Typically, objects are generated by function calls that specify an appropriate mechanism used for the generation. Additionally, there is a function `C_CreateObject()` to create objects that are defined by input templates containing appropriate attributes. For example, data objects are defined by a data template `CK_ATTRIBUTE dataTemplate[]`, or certificate objects are defined by a certificate template `CK_ATTRIBUTE certificateTemplate[]`.

An object comprises a set of attributes, each of which has precisely one given value. For an example, have a look at Step 6 of the "Sample openCryptoki program" on page 105, where function `C_GenerateKeyPair()` uses the mechanism CKM_RSA_PKCS_KEY_PAIR_GEN to create an RSA private and public key pair. The attributes are assigned to the public and private keys to be generated with two different templates of type array of CK_ATTRIBUTE (one applicable for the private key and one applicable to the public key), which are input to the function call.

PKCS #11 objects belong to different orthogonal classes depending on their life span, access restrictions and modifiability:

- Session objects exist during the duration of a session whereas token objects are associated with a token and not with any running code. In other words, token objects are stored persistently across sessions and are visible for multiple processes using the token. Session objects disappear when the session is closed, and are only visible within the session that created the object.
- Private objects can only be accessed by the PKCS #11 ***Users*** if they have logged into the token, whereas public objects can always be accessed by both ***User*** and ***security officer***.
- A token object that is read-write for a read-write session is read-only for a read-only session of another application at the same time.

In addition, each object has a set of attributes, especially the CKA_CLASS attribute, which determines which further attributes are associated with an object. Other typical attributes contain the value of an object or determine whether an object is a token object.

The PKCS #11 API provides a set of functions to manage objects, like for example, `C_CreateObject()`, `C_CopyObject()`, `C_DestroyObject()`, `C_GetObjectSize()`, `C_GetAttributeValue()`, `C_SetAttributeValue()`, `C_FindObjects()`.

Note: Token objects are stored in a token-specific object store, the token directory (see also Chapter 14, "Directory content for CCA, ICA, EP11, and Soft tokens," on page 81).

The most important object classes are those that implement keys: private keys (CKO_PRIVATE_KEY), public keys (CKO_PUBLIC_KEY) and secret keys (CKO_SECRET_KEY). Private and public keys are the members of an asymmetric key pair, whereas secret keys are symmetric keys or MAC keys.

There are many key specific attributes. For example, the Boolean attribute CKA_WRAP denotes whether a key may be used to wrap another key. The Boolean attribute CKA_SENSITIVE is only applicable for private and secret keys. If this attribute is TRUE, this means that the value of the key may never be revealed in clear text. There are key-type specific attributes like CKA_MODULUS, which is an attribute specific to RSA keys. Not all tokens support all key types with all possible attributes. CKA_PRIVATE causes the object data to be encrypted when stored in the token directory.

Object management functions to create keys are `C_GenerateKey()`, `C_GenerateKeyPair()` and `C_DeriveKey()`. To import a key not generated by one of the previously mentioned functions, you can use `C_CreateObject()` with all key specific attributes specified in the template. Alternatively, a wrapped key can be imported with `C_Unwrap()`, where a wrapped key is a standard representation of a key specific to the key type (for example, a byte array for secret keys or a BER encoding for other key types) that is encrypted by a wrapping key.

PKCS #11 defines multiple object classes, for example:

- Data objects
- Key objects
- Public key objects
- Private key objects
- Secret key objects
- Certificate objects

Each object class has its own set of attributes, where these attributes define an instance of an object from this class. There is one common attribute called CKA_CLASS for all object classes. This attribute defines the type (or class) of an object. For more information about objects, read "How to create and modify objects" on page 101 and "How to apply attributes to objects" on page 102).

When an object is created or found on a token by an application, openCryptoki assigns it an object handle for that application's sessions to access it (CK_OBJECT_HANDLE, CK_OBJECT_HANDLE_PTR).

PKCS #11 also defines objects for certificates. However, other than functions to operate on generic objects, no functions to operate on certificate objects are part of the PKCS #11 API.

Access control and groups

To properly configure the system, and to be authorized for performing the openCryptoki processes, for example, running the slot manager daemon **`pkcsslotd`**, the *pkcs11* group must be defined in file `/etc/group` of the system. Every user of openCryptoki (including the users configuring openCryptoki and its tokens) must be a member of this *pkcs11* group. Use standard Linux management operations to create the *pkcs11* group if needed, and to add users to this group as required. You may also refer to the man page of openCryptoki (**`man openCryptoki`**) which has a SECURITY NOTE section that is important from the security perspective.

Logging and tracing in openCryptoki

You can enable logging support by setting the environment variable `OPENCRYPTOKI_TRACE_LEVEL`. If the environment variable is not set, logging is disabled by default.

Table 1. openCryptoki log levels

Log level	**Description**
0	Trace off.
1	Log error messages.

Table 1. openCryptoki log levels (continued)

Log level	Description
2	Log warning messages.
3	Log informational messages.
4	Log development debug messages. These messages may help debug while developing openCryptoki applications.
5	Log debug messages that are useful to application programmers. This level must be enabled at build time of the openCryptoki library via option `--enable-debug` in the **`configure`** script.

If a log level > 0 is defined in the environment variable `OPENCRYPTOKI_TRACE_LEVEL`, then log entries are written to file `/var/log/opencryptoki/trace.<pid>`. In this file name specification, `<pid>` denotes the ID of the running process that uses the current token.

The log file is created with ownership *user*, and group *pkcs11*, and permission 0640 (user: read, write; group: read only; others: nothing). For every application, which is using openCryptoki, a new log file is created during token initialization.

A log level > 3 is only recommended for developers and for collecting more information during problem reproductions.

Lock files

As of release 3.8, openCryptoki maintains the following lock files in the system: one global API lock file, one lock file per token instance, except for the TPM token. For the TPM token, openCryptoki keeps one lock file per user.

The lock files are stored in the following directories, if applicable:

```
# ls -lh /var/lock/opencryptoki/

LCK..APIlock
ccatok/LCK..ccatok
ep11tok/LCK..ep11tok
icsf/LCK..icsf
lite/LCK..lite
swtok/LCK..swtok
tpm/<USER>/LCK..tpm
```

The `LCK..APIlock` file serializes access to the shared memory from the **`pkcsslotd`** daemon and from the API calls issued from `libopencryptoki.so`.

The token-specific lock files serialize access to the shared memory and to objects in the token directory from the respective token library (for example, from `libpkcs11_cca.so` for the CCA token).

Thus, each lock file is used to protect the respective shared memory segments and objects in the token directory by letting threads wait for a required lock.

Use and purpose of openCryptoki features

In normal operation, the mentioned shared memory segments, the Unix domain sockets of the slot manager daemon (**`pkcsslotd`**), and the lock files should be transparent to users of openCryptoki. But in case of certain errors, such objects may remain from some previous use of openCryptoki and thus block new operations. In such cases, you need to know the locations of these objects and you must typically use certain operating system tools to remove them and enable a normal use of openCryptoki again.

Figure 3 on page 13 shows the process flow within the Linux on Z and LinuxONE crypto stack. For example, an application sends an encryption request to the crypto adapter. Through various interfaces, such a request is propagated from the application layer down to the target crypto adapter. On its way down, the request passes through the involved layers: the standard openCryptoki interfaces, the adequate IBM Z crypto libraries, and the operating system kernel. The `zcrypt` device driver finally sends the request to the appropriate cryptographic coprocessor. The resulting request output is sent back to the application just the other way round through the layer interfaces.

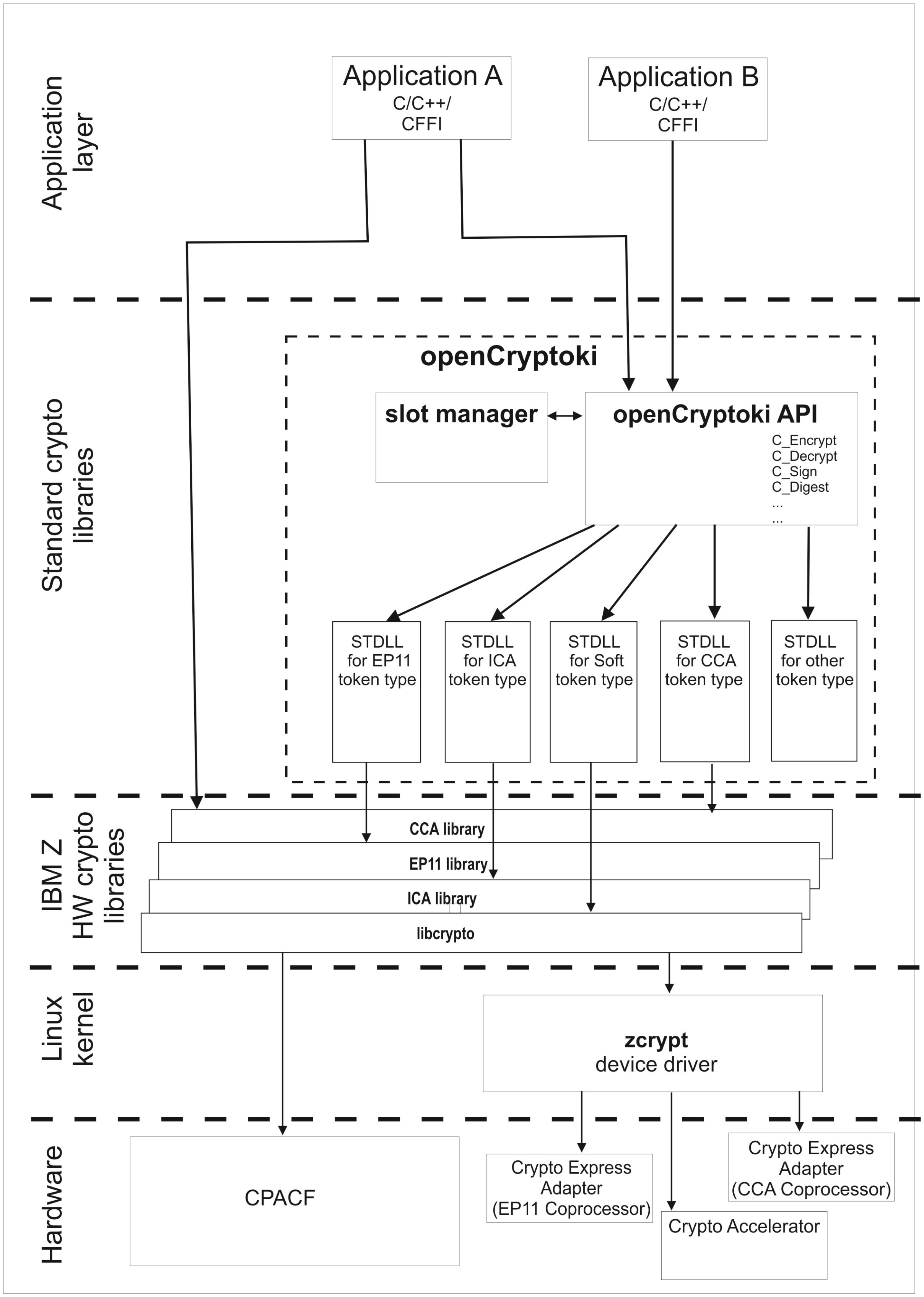


Figure 3. Linux on Z and LinuxONE crypto stack

Chapter 3. Installing openCryptoki

The available tokens are part of the openCryptoki package. The package comes with manual pages (man pages) that describe the usage of the tools and the format of the configuration files. The openCryptoki package in turn is shipped with the Linux on Z distributions. This package might be split into several packages by the distributions, thus allowing to install individual tokens separately.

Check whether you already installed openCryptoki in your current environment, for example:

```
$ rpm -qa | grep -i opencryptoki /* for RPM */
$ dpkg -l | grep -i opencryptoki /* for DEB */
```

Note: The command examples are distribution dependent. `opencryptoki` must in certain distributions be specified as `openCryptoki` (case-sensitive).

You should see all installed openCryptoki packages. If required packages are missing, use the installation tool of your Linux distribution to install the appropriate openCryptoki RPM or DEB.

Notes:

- You can update an installed version of openCryptoki with a package of a newer version. Depending on the tokens to be used, further libraries need to be installed and cryptographic adapters must be enabled. If you had built openCryptoki from the source before, you must remove any previous installation of openCryptoki (`make uninstall`), before you can install a distribution package for a new openCryptoki version.
- Some tokens need a token-specific library to be installed on the system as a prerequisite for usage. These are mentioned for each token in Part 3, "Token specifications," on page 39.

Installing from the RPM or DEB package

The openCryptoki packages are delivered by the distributors. Distributors build these packages as RPM or DEB packages for delivering them to customers.

Customers can install these openCryptoki packages by using the installation tool of their selected distribution.

If you received openCryptoki as an *RPM* package, follow the *RPM* installation process that is described in the *RPM Package Manager* man page. If you received an openCryptoki *DEB* package, you can use the *dpkg - package manager for Debian* described in the *dpkg man page*.

The installation from either an *RPM* or *DEB* package is the preferred installation method.

Installing from the source package

As an alternative, for example for development purposes, you can get the latest version (inclusive latest patches) from the GitHub repository and build it yourself. But this version is not serviced. It is suitable for non-production systems and early feature testing, but you should not use it for production.

In this case, refer to the `INSTALL` file in the top level of the source tree. You can start from the instructions that are provided with the subtopics of this `INSTALL` file and select from the described alternatives. If you use this installation method parallel to the installation of a package from your distributor, then you should keep both installations isolated from each other.

1. Download the latest version of the openCryptoki sources from:

   ```
   https://github.com/opencryptoki/opencryptoki/releases
   ```

2. Decompress and extract the compressed tape archive (tar.gz - file). There is a new directory named like `opencryptoki-3.xx.x`.

3. Change to that directory and issue the following scripts and commands:

```
$ ./bootstrap.sh
$ ./configure
$ make
$ make install
```

The scripts or commands perform the following functions:

bootstrap
Initial setup, basic configurations

configure
Check configurations and build the makefile. You can specify several options here to overwrite the defaults. For example, not all tokens are built as the default. To build the CCA token as an example, specify `./configure --enable_ccatok`

make
Compile and link

make install
Install the libraries

Note: When installing openCryptoki from the source package, the location of some installed files will differ from the location of files installed from an RPM or DEB package.

Chapter 4. Adjusting the openCryptoki configuration file

A preconfigured list of all available tokens that are ready to register to the openCryptoki slot daemon is required before the slot daemon can start. This list is provided by the global configuration file called `opencryptoki.conf`. Read this topic for information on how to adapt this file according to your installation.

Table 2 on page 17 lists the available libraries that may be in place after you successfully installed openCryptoki. It may vary for different distributions and is dependent from the installed packages.

Also, Linux on Z and LinuxONE do not support the Trusted Platform Module (TPM) token library.

A token is only available, if the token library is installed, and the appropriate software and hardware support pertaining to the stack of the token is also installed. For example, the EP11 token is only available if all parts of the EP11 library software are installed and a Crypto Express EP11 coprocessor is detected. For more information, read Exploiting Enterprise PKCS #11 using openCryptoki.

A token needs not be available, even if the corresponding token library is installed. Display the list of available tokens by using the command:

```
$ pkcsconf -t
```

For a sample output, see Figure 5 on page 27. Table 2 on page 17 shows available openCryptoki libraries:

Table 2. openCryptoki libraries

Library	**Explanation**
`/usr/lib64/opencryptoki/libopencryptoki.so`	openCryptoki base library
`/usr/lib64/opencryptoki/stdll/libpkcs11_ica.so`	ICA token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_sw.so`	Soft token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_tpm.so`	TPM token library (not supported by Linux on Z and LinuxONE)
`/usr/lib64/opencryptoki/stdll/libpkcs11_cca.so`	CCA token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_ep11.so`	EP11 token library
`/usr/lib64/opencryptoki/stdll/libpkcs11_icsf.so`	ICSF token library

`libopencryptoki.so` is the openCryptoki shared base library. The main API for the STDLLs `PKCS11_API.so` (mentioned in Chapter 2, “Architecture and components of openCryptoki,” on page 5) is a link, or an alias. Besides the token library, an application needs to load either the `libopencryptoki.so` object or the `PKCS11_API.so` link to be able to exploit a token.

```
lrwxrwxrwx 1 root root     18 May 19 20:05 PKCS11_API.so -> libopencryptoki.so
```

The `/etc/opencryptoki/opencryptoki.conf` file must exist and it must contain an entry for each instance of a token to make these instances available (see the provided sample configuration from Figure 4 on page 18).

You can check the current default `opencryptoki.conf` file on this URL:

```
https://github.com/opencryptoki/opencryptoki/blob/master/usr/sbin/pkcsslotd/opencryptoki.conf
```

```
version opencryptoki-3.17

# The following defaults are defined:
#       hwversion = "0.0"
#       firmwareversion = "0.0"
#       description = Linux
#       manufacturer = IBM
#
# The slot definitions below may be overriden and/or customized.
# For example:
#       slot 0
#       {
#          stdll = libpkcs11_cca.so
#          description = "OCK CCA Token"
#          manufacturer = "MyCompany Inc."
#          hwversion = "2.32"
#          firmwareversion = "1.0"
#       }
# See man(5) opencryptoki.conf for further information.
#

disable-event-support       # not part of the default config file

slot 0
{
stdll = libpkcs11_tpm.so
}

slot 1
{
stdll = libpkcs11_ica.so
}

slot 2
{
stdll = libpkcs11_cca.so
}

slot 3
{
stdll = libpkcs11_sw.so
}

slot 4
{
stdll = libpkcs11_ep11.so
confname = ep11tok.conf
}
```

Figure 4. Default opencryptoki.conf

Note:

- The standard path for slot token dynamic link libraries (STDLLs) is: `/usr/lib64/opencryptoki/stdll/`.
- The standard path for the token-specific configuration file (in our example, `ep11tok.conf`) is `/etc/opencryptoki/`.
- You can specify multiple tokens of any type in different slots. Each token must be specified with a unique token name in each slot. The tokens are located in different file paths in the token directory, but use the same STDLL. For example:

```
slot 4
{
stdll = libpkcs11_ep11.so
confname = ep11tok01.conf
tokname = ep11token01
description = "Ep11 Token"
manufacturer = "IBM"
hwversion = "4.11"
firmwareversion = "2.0"
}

slot 5
{
stdll = libpkcs11_ep11.so
confname = ep11tok02.conf
tokname = ep11token02
}
```

Event notification support

You can enable openCryptoki to deliver notifications about a cryptographic external event to active token instances. External events are for example configuration changes of a cryptographic coprocessor (APQNs becoming available or unavailable), events initiated by an openCryptoki administrator via a command line tool, or HSM master key changes.

Events are delivered to all active token instances. Each process using openCryptoki has access to a set of instances of the configured tokens. Multiple processes that use openCryptoki can exist in a system. Thus, events are delivered to all processes that currently use openCryptoki, and thus to all token instances owned by each of the processes.

Event notification support is enabled by default. You can disable this support by specifying keyword `disable-event-support` in the openCryptoki configuration file as shown in Figure 4 on page 18.

Chapter 5. openCryptoki environment variables

View a table that documents the available openCryptoki environment variables and their purpose.

Table 3. openCryptoki environment variables

Name	**Purpose**
`PKCSLIB`	Used in some tools as path to the openCryptoki libary libopencryptoki.so. You should never change this variable. Maybe useful for debugging information.
`OCK_EP11_LIBRARY`	Specifies the path to the EP11 library. Only of interest, if multiple EP11 libraries are installed. Normally, you should never change this variable. Maybe useful for debugging information.
`OCK_EP11_TOKEN_DIR`	The standard path for the token-specific EP11 token configuration file is `/etc/opencryptoki/`. You can change this path by using this variable.
`PKCS_APP_STORE`	Specifies the path to the main directory. Never change this variable. Maybe useful for debugging information.
`PKCS11_SHMEM_FILE`	Used for pointing to the SHM-Key Files. Only of interest, if you run multiple slot manager daemons (**`pkcsslotd`**).
`PKCS11_USER_PIN`	Provides a possibility to transfer the ***User*** PIN to the **p11sak** utility. It is required for running the openCryptoki tests.
`OPENCRYPTOKI_TRACE_LEVEL`	Enables logging support. If the environment variable is not set, logging is disabled by default.

Part 2. Common tools of openCryptoki

Learn how to use tools provided by openCryptoki that you can use for common purposes.

The following tools are documented:

- Chapter 6, “Managing tokens - pkcsconf utility,” on page 25
- Chapter 7, “Generating and listing keys - p11sak utility,” on page 29
- Chapter 8, “Migrating to FIPS compliance - pkcstok_migrate utility,” on page 37

Chapter 6. Managing tokens - pkcsconf utility

openCryptoki provides a command line program (`/sbin/pkcsconf`) to configure and administer tokens that are supported within the system. The `pkcsconf` capabilities include token initialization, and security officer (SO) PIN and user PIN initialization and maintenance. These PINs are required for token initialization.

`pkcsconf` operations that address a specific token must specify the slot that contains the token with the **`-c`** option. You can view the list of tokens present within the system by specifying the **`-t`** option. Type `pkcsconf --help` or `pkcsconf -h` into a command line to show the options for the `pkcsconf` command:

```
# pkcsconf -h
usage: pkcsconf [-itsmlIupPh] [-c slotnumber -U user-PIN -S SO-PIN -n new PIN]
```

The available options have the following meanings:

-i
: display PKCS #11 info

-t
: display token info

-s
: display slot info

-m
: display mechanism list

-l
: display slot description

-I
: initialize token

-u
: initialize user PIN

-p
: set the user PIN

-P
: set the SO PIN

-h | --help | ?
: show this help

-c
: specify the token slot ID

-U
: the current user PIN (for use when changing the user PIN with -u and -p options). If not specified, user will be prompted.

-S
: the current security officer (SO) PIN (for use when changing the SO PIN with -P option). If not specified, user will be prompted.

-n
: the new PIN (for use when changing either the user PIN or the SO PIN with -u, -p or -P options). If not specified, user will be prompted.

For more information about the `pkcsconf` command, see the *pkcsconf man page*.

Starting the slot manager

A prerequisite for accessing a token is a running slot manager daemon (**pkcsslotd**) .

Use one of the following commands to start the slot daemon, which reads out the configuration information and sets up the tokens:

```
$ service pkcsslotd start
$ systemctl start pkcsslotd.service   /* for Linux distributions providing systemd */
```

For a permanent solution, specify:

```
$ chkconfig pkcsslotd on
$ systemctl enable pkcsslotd.service  /* for Linux distributions providing systemd */
```

Initializing a token with pkcsconf

Once the openCryptoki configuration file and, if applicable, token-specific configuration files are set up, and the **pkcsslotd** daemon is started, the token instances must be initialized. Use the **pkcsconf** command as shown to perform this task.

Note: As mentioned in "Roles and Sessions" on page 8, PKCS #11 defines a ***security officer*** (SO) and a (standard) ***User*** for each token instance. openCryptoki requires that for both a log-in PIN is defined as part of the token initialization.

The following command provides some useful slot information:

```
# pkcsconf -s

Slot #1 Info
        Description: ICA Token
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 4.0
        Firmware Version: 2.11
...
...
Slot #4 Info
        Description: EP11 Token
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 4.0
        Firmware Version: 2.10
```

Receive more detailed information about a token using the following command:

```
# pkcsconf -t
...
...
Token #4 Info:
        Label: ep11tok
        Manufacturer: IBM
        Model: EP11
        Serial Number: 93AABC5H53107366
        Flags: 0x880045 (RNG|LOGIN_REQUIRED|CLOCK_ON_TOKEN|USER_PIN_TO_BE_CHANGED|
SO_PIN_TO_BE_CHANGED)
        Sessions: 0/[effectively infinite]
        R/W Sessions: [information unavailable]/[effectively infinite]
        PIN Length: 4-8
        Public Memory: [information unavailable]/[information unavailable]
        Private Memory: [information unavailable]/[information unavailable]
        Hardware Version: 7.24
        Firmware Version: 3.1
        Time: 2021031912021700
```

Figure 5. Token information with ***pkcsconf -t***

Find the token instance you want to initialize in the output list and note the correct slot number. This number is used in the next initialization steps to identify your token:

```
$ pkcsconf -I -c <slot>   /* Initialize the Token and set up a Token Label */

$ pkcsconf -P -c <slot>   /* change the SO PIN (recommended) */

$ pkcsconf -u -c <slot>   /* Initialize the User PIN (SO PIN required) */

$ pkcsconf -p -c <slot>   /* change the User PIN (optional) */
```

pkcsconf -I
: During token initialization, you are asked for a token label. Provide a meaningful name, because you may need this reference for identification purposes.

pkcsconf -P
: For security reasons, openCryptoki requires that you change the default SO PIN (87654321) to a different value. Use the `pkcsconf -P` option to change the SO PIN.

pkcsconf -u
: When you enter the user PIN initialization you are asked for the newly set SO PIN. The length of the user PIN must be 4 - 8 characters.

pkcsconf -p
: You must at least once change the user PIN with `pkcsconf -p` option. After you completed the PIN setup, the token is prepared and ready for use.

Note: Define a user PIN that is different from 12345678, because this pattern is checked internally and marked as default PIN. A log-in attempt with this user PIN is recognized as *not initialized*.

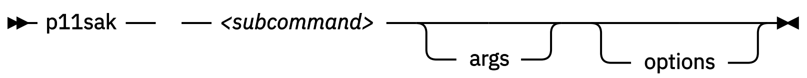
Chapter 7. Generating and listing keys - p11sak utility

Use the **p11sak** tool to list the token keys in an openCryptoki token repository with their PKCS #11 attributes. You can also use the tool to generate and delete symmetric and asymmetric keys in an openCryptoki token repository.

The general invocation scheme of the command line tool is:

```
p11sak COMMAND [ARGS] [OPTIONS]
```

p11sak syntax - General invocation scheme



where

subcommand
is described in the following sections:

- "Generating keys" on page 29
- "Listing keys in the repository" on page 31
- "Remove a key" on page 33

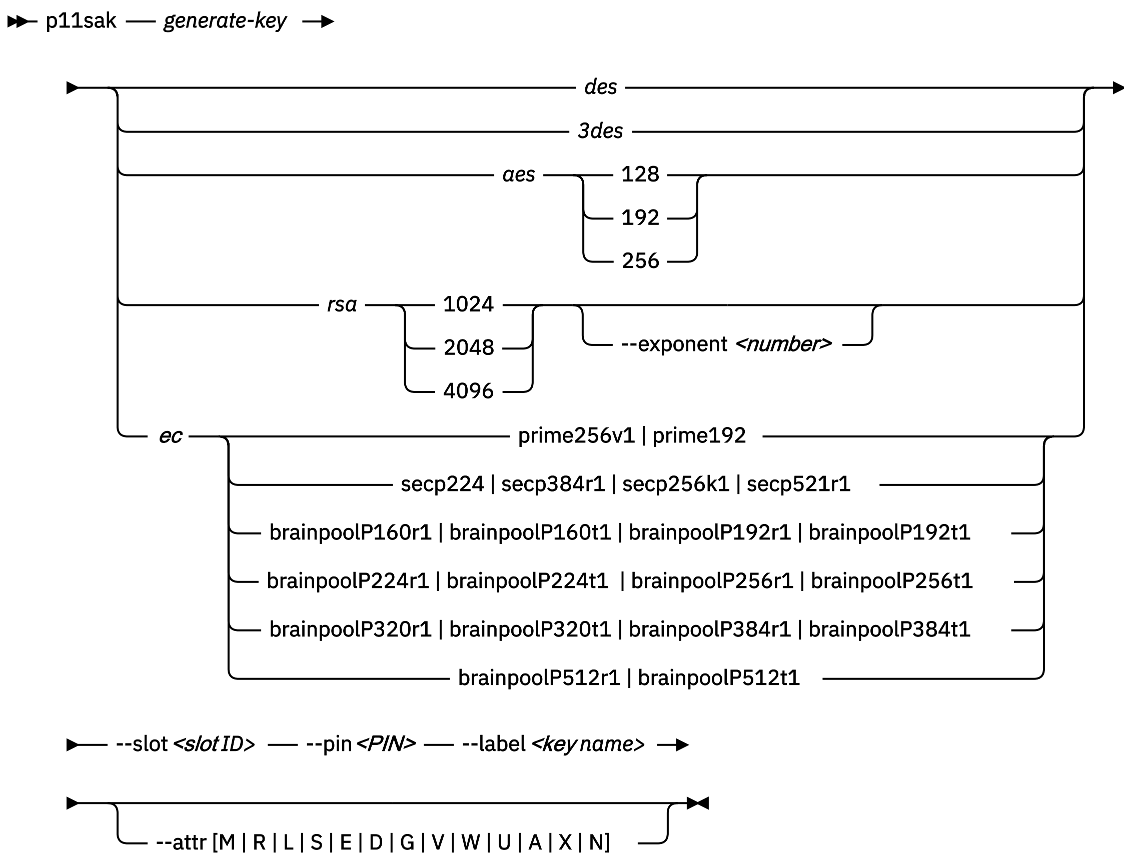
Also, read the information in section "Command help" on page 34 to learn how to use the tool efficiently.

Generating keys

Use the **p11sak generate-key** subcommand to generate keys in the openCryptoki token repository. The tool supports the generation of:

- symmetric keys (AES, 3DES, DES) with PKCS #11 attributes
- asymmetric keys (RSA, EC) with PKCS #11 attributes.

p11sak syntax - Generate a key



p11sak
generate-key
des
3des
aes
128
192
256
rsa
1024
2048
4096
--exponent <number>
ec
prime256v1 | prime192
secp224 | secp384r1 | secp256k1 | secp521r1
brainpoolP160r1 | brainpoolP160t1 | brainpoolP192r1 | brainpoolP192t1
brainpoolP224r1 | brainpoolP224t1 | brainpoolP256r1 | brainpoolP256t1
brainpoolP320r1 | brainpoolP320t1 | brainpoolP384r1 | brainpoolP384t1
brainpoolP512r1 | brainpoolP512t1
--slot <slotID>
--pin <PIN>
--label <key name>
--attr [M | R | L | S | E | D | G | V | W | U | A | X | N]

where

des
specifies a symmetric DES key to be generated.

3des
specifies a symmetric triple DES key.

aes
specifies a symmetric AES key to be generated. You must specify a key length in bits: 128, 192, or 256.

rsa
specifies an asymmetric RSA key pair (private and public) to be generated. You must specify a key length in bits: 1024, 2048, or 4096.

--exponent
specifies an exponent for an RSA key pair generation. This option is optional and only considered for RSA key pair generation. The default is 65537.

ec
specifies an elliptic curve key pair (private and public) to be generated. You must specify an elliptic curve for the ECC key pair. Select one supported curve as shown in the syntax diagram.

--slot
specifies the slot ID of an openCryptoki token with the repository.

--pin
specifies the openCryptoki token repository user PIN. If not specified, the user can enter the PIN on request.

--label
specifies the label (name) of the generated key. This option is mandatory.

--attr

This option is optional and can be used to set one or more of the binary key attributes by specifying one or more of the the respective letters from the subsequent list. With an upper case letter, the respective attribute is set to TRUE and with a lower case letter it is set to FALSE. For multiple attributes add the respective letters without space, for example: `MLD` or `MrS`.

M
CKA_MODIFIABLE

R
CKA_DERIVE

L
CKA_LOCAL

S
CKA_SENSITIVE

E
CKA_ENCRYPT

D
CKA_DECRYPT

G
CKA_SIGN

V
CKA_VERIFY

W
CKA_WRAP

U
CKA_UNWRAP

A
CKA_ALWAYS_SENSITIVE

X
CKA_EXTRACTABLE

N
CKA_NEVER_EXTRACTABLE

The key attributes CKA_TOKEN and CKA_PRIVATE are set by default.

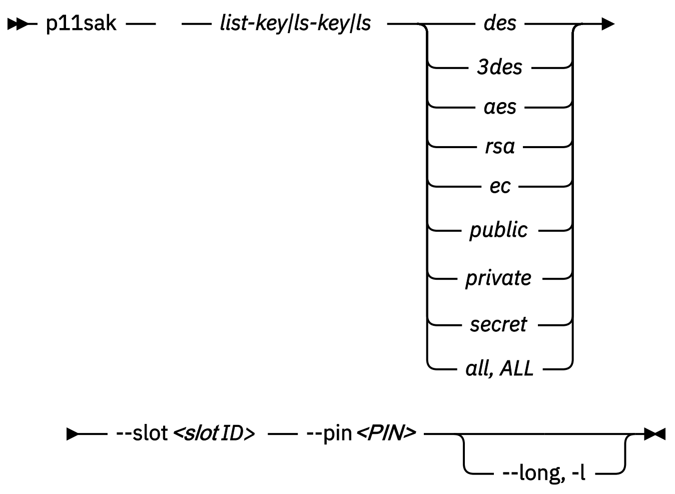
Listing keys in the repository

The tool supports the listing of:

- symmetric keys (AES, 3DES, DES) with PKCS #11 attributes,
- asymmetric keys (RSA, EC) with PKCS #11 attributes,
- public, private and secure keys.
- all keys of any type

With the options in the **`list-key`** subcommand, you can filter the keys that you want to list. They have the same meaning as described in section "Generating keys" on page 29, with additional options to filter for public, private and secure keys. The **`--long`** or **`-l`** parameter produces the long output format. If omitted, you obtain a short output format.

p11sak syntax - Listing keys



Example for a listing output in short format (default):

```
# # p11sak list-key aes --slot 2 --pin 98765432

| M R L S E D G V W U X A N * | KEY TYPE | LABEL
|---------------------------------------+------------+-------------
| 0 0 1 0 0 0 0 0 0 0 0 0 1 1 | AES 128 | 2020-04-29_aes128-M-1.key
| 1 0 1 0 0 0 0 0 0 0 0 0 1 1 | AES 128 | 2020-04-29_aes128-M-2.key
| 0 0 1 0 0 0 0 0 0 0 0 0 1 1 | AES 128 | 2020-04-29_aes128-2.key
| 0 0 1 0 0 0 0 0 0 0 0 0 1 1 | AES 128 | 2020-04-29_aes128-2.key
| 1 1 1 0 0 0 0 0 0 0 0 0 1 1 | AES 128 | 2020-04-29_aes128-MR.key
| 1 1 1 1 1 0 0 0 0 0 0 1 1 1 | AES 128 | 2020-04-29_aes128-MRLSE.key
| 1 1 1 1 1 1 1 1 1 1 1 1 0 1 | AES 128 | 2020-04-29_aes128-MRLSEDGVWUAX
```

Note: The number `1` in column **`*`** indicates whether at least one attribute is defined in the output configuration file. Otherwise, number `0` is shown. To see these attributes, use the long output format as described hereafter.

How to get a listing output in long format

Request a long output format by specifying option `--long` or `-l` with the **`list-key`** command. You need to provide an output configuration file called `p11sak_defined_attrs.conf` to specify attributes to be printed in addition to the ones specified with key generation. You can use the environment variable `P11SAK_DEFAULT_CONF_FILE` to set the full path name for this file. If this variable is not set, the system looks for this configuration file in your user directory. If no configuration file is found there, the default output configuration file `/etc/opencryptoki/p11sak_defined_attrs.conf` is used.

The output configuration file lists the desired attributes in the following format:

```
attribute {
   name = CKA_IBM_ATTRBOUND
   type = CK_BBOOL
   id = 0x80010004
}
attribute {
   name = CKA_IBM_USE_AS_DATA
   type = CK_BBOOL
   id = 0x80010008
}
attribute {
   name = CKA_IBM_PROTKEY_EXTRACTABLE
   type = CK_BBOOL
   id = 0x8001000c
}
attribute {
   name = CKA_MODULUS_BITS
   type = CK_ULONG
```

```
    id = 0x00000121
 }
 attribute {
    name = CKA_MODULUS
    type = CK_BYTE
    id = 0x00000120
 }
```

Example for a listing output in long format:

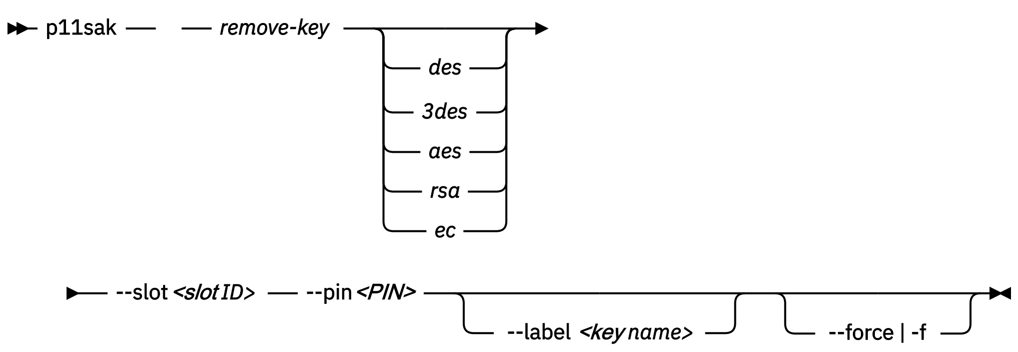
```
# p11sak list-key rsa --slot 1 --pin 11223344 -l

Label: rsa1:pub
       Key: public RSA
       Attributes:
           CKA_TOKEN: CK_TRUE
           CKA_PRIVATE: CK_TRUE
           CKA_MODIFIABLE: CK_TRUE
           CKA_DERIVE: CK_FALSE
           CKA_LOCAL: CK_FALSE
           CKA_ENCRYPT: CK_TRUE
           CKA_VERIFY: CK_TRUE
           CKA_WRAP: CK_TRUE
           CKA_IBM_PROTKEY_EXTRACTABLE: CK_FALSE
           CKA_MODULUS_BITS: 1024 (0x400)
           CKA_MODULUS: len=128 value:
             AD 8E A4 0C 3E 18 62 FE 3F 0B 95 F8 67 73 30 BF
             3B A6 83 18 4B 32 91 AD 37 7D D8 4E 35 6F 97 E3
             EC 22 34 E4 F9 7C 43 E5 6B 96 FE 65 AE 78 1C B2
             A8 81 24 5F 11 B2 8D 0F C6 13 A8 BA 41 75 E7 07
             A4 DC E3 53 95 0E D7 2E 55 7C 6D 1C 78 95 AF 9A
             5B 18 4C AC 43 C7 F4 51 73 8C 4B 07 A6 48 E2 28
             06 65 5A 22 F3 B9 7E C2 F6 5B B5 42 EB E5 CE 81
             29 C4 91 6B 43 3F 19 5C FB 0E 20 47 D4 4C C4 E1

Label: rsa-key:prv
       Key: private RSA
       Attributes:
           CKA_TOKEN: CK_TRUE
           CKA_PRIVATE: CK_TRUE
           CKA_MODIFIABLE: CK_TRUE
           CKA_DERIVE: CK_FALSE
           CKA_LOCAL: CK_FALSE
           CKA_SENSITIVE: CK_FALSE
           CKA_DECRYPT: CK_TRUE
           CKA_SIGN: CK_TRUE
           CKA_UNWRAP: CK_TRUE
           CKA_EXTRACTABLE: CK_TRUE
           CKA_ALWAYS_SENSITIVE: CK_FALSE
           CKA_NEVER_EXTRACTABLE: CK_FALSE
           CKA_IBM_ATTRBOUND: CK_FALSE
           CKA_IBM_USE_AS_DATA: CK_FALSE
           CKA_IBM_PROTKEY_EXTRACTABLE: CK_FALSE
           CKA_MODULUS: len=128 value:
             AD 8E A4 0C 3E 18 62 FE 3F 0B 95 F8 67 73 30 BF
             3B A6 83 18 4B 32 91 AD 37 7D D8 4E 35 6F 97 E3
             EC 22 34 E4 F9 7C 43 E5 6B 96 FE 65 AE 78 1C B2
             A8 81 24 5F 11 B2 8D 0F C6 13 A8 BA 41 75 E7 07
             A4 DC E3 53 95 0E D7 2E 55 7C 6D 1C 78 95 AF 9A
             5B 18 4C AC 43 C7 F4 51 73 8C 4B 07 A6 48 E2 28
             06 65 5A 22 F3 B9 7E C2 F6 5B B5 42 EB E5 CE 81
             29 C4 91 6B 43 3F 19 5C FB 0E 20 47 D4 4C C4 E1
...
```

Remove a key

Use the **p11sak remove-key** subcommand to remove keys from the openCryptoki token repository. The tool supports the deletion of:

- symmetric keys (AES, 3DES, DES)
- asymmetric keys (RSA, EC).

p11sak syntax - Removing keys



With the parameters you can select the key or the keys that you want to delete.

Either select a specific key for deletion by specifying its name with the **--label** option. Or select multiple keys by specifying a key type together with the slot ID and the user PIN. The parameters and options that define the key type are optional if the **--label** option is set. The **--label** option takes precedence over the key type and deletes all keys with that label.

You are prompted to confirm the deletion of the selected key or keys. To suppress the confirmation, use the **--force | -f** option.

Command help

Request general help with the following command:

```
# p11sak -h
```

to receive the following information:

```
Usage: p11sak COMMAND [ARGS] [OPTIONS]

Commands:
     generate-key        Generate a key
     list-key            List keys in the repository
     remove-key          Delete keys in the repository

Options:
-h, --help          Show this help
```

Request help for a subcommand:

```
# p11sak generate-key -h
# p11sak list-key -h
# p11sak remove-key -h
```

For the **generate-key** subcommand, you receive the following information:

```
# p11sak generate-key -h

Usage: p11sak generate-key [ARGS] [OPTIONS]

Args:
     des
     3des
     aes [128 | 192 | 256]
     rsa [1024 | 2048 | 4096]
      ec [prime256v1 | prime192 | secp224 | secp384r1 | secp521r1 | secp256k1 |
          brainpoolP160r1 | brainpoolP160t1 | brainpoolP192r1 | brainpoolP192t1 |
          brainpoolP224r1 | brainpoolP224t1 | brainpoolP256r1 | brainpoolP256t1 |
          brainpoolP320r1 | brainpoolP320t1 | brainpoolP384r1 | brainpoolP384t1 |
          brainpoolP512r1 | brainpoolP512t1]


Options:
     --slot SLOTID                           openCryptoki repository token SLOTID.
     --pin PIN                               pkcs11 user PIN
     --label LABEL                           key label LABEL to be listed
     --exponent EXP                          set RSA exponent EXP
     --attr [M R L S E D G V W U A X N]      set key attributes
     -h, --help                              Show this help
```

For the **generate-key** subcommand, you can request help for a selected key type, for example, for an AES key:

```
# p11sak generate-key aes -h

 Usage: p11sak generate-key aes [ARGS] [OPTIONS]

 Args:
      128
      192
      256

 Options:
      --slot SLOTID                           openCryptoki repository token SLOTID.
      --pin PIN                               pkcs11 user PIN
      --label LABEL                           key label LABEL to be listed
      --attr [M R L S E D G V W U A X N T]    set key attributes
      -h, --help                              Show this help
```

Chapter 8. Migrating to FIPS compliance - pkcstok_migrate utility

Use the **pkcstok_migrate** tool to migrate the data stores of an EP11 token, a CCA token, an ICA token, or a Soft token to a FIPS compliant format. This FIPS compliant data format is available starting with openCryptoki version 3.12. You can use this tool to migrate tokens created with all versions of openCryptoki, because also for version 3.12 or later, the old non-compliant format is the default. Being FIPS compliant, the token data is stored in a format that is better protected against attacks than the previously used data format.

For further information, read the **pkcstok_migrate** man page.

Parameters

```
# pkcstok_migrate -h

Help:         pkcstok_migrate -h
-h, --help    Show this help

Options:

-s, --slotid SLOTID          PKCS slot number (required)
-d, --datastore DATASTORE    token datastore location (required)
-c, --confdir CONFDIR        location of opencryptoki.conf (required)
-u, --userpin USERPIN        token user pin (prompted if not specified)
-p, --sopin SOPIN            token SO pin (prompted if not specified)
-v, --verbose LEVEL          set verbose level (optional):
                             none (default), error, warn, info, devel, debug
```

Functionality

The utility:

- directly accesses the token objects via file operations;
- assumes that no other action is currently running. It checks if the slot manager **pkcsslotd** is running and asks the user to end it if yes.

Before making any changes to the repository, a temporary copy is created. Migration takes place on this copy. The copied folder is suffixed with _PKCSTOK_MIGRATE_TMP. If the migration fails, the old repository is still available.

Running a migration again, would remove any remaining backups from previous runs, create a new backup, and then do the migration.

- After successfully migrating all token objects, the original repository folder is renamed by appending the suffix _BAK, and the new repository folder gets the name of the original one.
- Also, the `opencryptoki.conf` file is updated by inserting (or updating) the **tokversion** parameter in the token's slot configuration. The old configuration file is still available with the same suffix _BAK.

This makes the new repository immediately usable after restarting the **pkcsslotd** daemon, but also allows the user to switch back manually to the old token format.

Example: To transform a CCA token into the FIPS compliant data format perform a sequence of commands with your adequate input, similar to the following:

```
systemctl stop pkcsslotd.service /* for Linux distributions providing systemd */
/* or */
service pkcsslotd stop

# pkcstok_migrate --slot 2 --sopin 76543210 --userpin 12345678
                  --confdir /etc/opencryptoki
```

```
                   --datastore /var/lib/opencryptoki/ccatok


service pkcsslotd start
```

The output may look similar to the following:

```

pkcstok_migrate:
Summary of input parameters:
  datastore = /var/lib/opencryptoki/ccatok
  confdir = /etc/opencryptoki
  slot ID = 2
  user PIN specified
  SO PIN specified

Slot ID 2 points to DLL name libpkcs11_cca.so, which is a CCA token.
Data store /usr/local/var/lib/opencryptoki/ccatok points to this token info:
  label           : IBM CCA PKCS #11
  manufacturerID  : IBM
  model           : CCA
  serialNumber    :
  hardwareVersion : 0.0
  firmwareVersion : 0.0
Migrate this token with given slot ID? y/n
y
Migrated 2 object(s) out of 2 object(s).
Pre-migration data backed up at '/usr/local/var/lib/opencryptoki/ccatok_BAK'
Config file backed up at '/usr/local/etc/opencryptoki/opencryptoki.conf_BAK'
Remove these backups manually after testing the new repository.
pkcstok_migrate finished successfully.
```

Part 3. Token specifications

Application programmers find documentation about available token mechanisms to be invoked from cryptographic applications. openCryptoki administrators find additional token-specific tools and information about required token-specific configurations, if applicable.

Each token plugged into openCryptoki can implement a selection of the provided PKCS #11 mechanisms to be used in application programs. The names of these mechanisms start with the prefix "CKM_". For example, the CKM_AES_KEY_GEN mechanism generates an AES cryptographic key. This mechanism is offered by the CCA token, the ICA token, the EP11 token, and the Soft token, and can therefore be used to generate an AES key by any application that accesses one or more of these tokens.

Companies which collaborate with the openCryptoki open source community can contribute their company-specific mechanisms to openCryptoki. For example, all mechanisms which IBM adds to openCryptoki in addition to the PKCS #11 standard start with the vendor-specific prefix "CKM_IBM_". An example for an IBM-specific PKCS #11 mechanisms is CKM_IBM_SHA3_384_HMAC which you can use from an EP11 token to sign and verify a message using the SHA3-384 hash function.

Issue the **pkcsconf** command with the -m parameter to display all mechanisms that are supported by the token of interest residing in the slot specified with parameter -c.

```
$ pkcsconf -m -c <slot>
```

For example, if you want to display all supported PKCS #11 mechanisms of an ICA token that resides in slot number 1 in your environment, issue the following command:

```
# pkcsconf -m -c 1
```

The output depends on the supported Crypto Express coprocessors together with the openCryptoki version. The beginning of the output list may look as shown in Figure 6 on page 39. The name corresponds to the PKCS #11 specification. Each mechanism provides its supported key size and some further properties such as hardware support and mechanism information flags. These flags provide information about the PKCS #11 functions that may use the mechanism. Typical functions are for example, *encrypt*, *decrypt*, *wrap key*, *unwrap key*, *sign*, or *verify*. For some mechanisms, the flags show further attributes that describe the supported variants of the mechanism.

Figure 6. List of supported mechanisms for a certain token

```
Mechanism #0
        Mechanism: 0x0 (CKM_RSA_PKCS_KEY_PAIR_GEN)
        Key Size: 512-4096
        Flags: 0x10001 (CKF_HW|CKF_GENERATE_KEY_PAIR)
Mechanism #1
        Mechanism: 0x1 (CKM_RSA_PKCS)
        Key Size: 512-4096
        Flags: 0x67B01 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_SIGN|CKF_SIGN_RECOVER|CKF_VERIFY|
CKF_VERIFY_RECOVER|CKF_WRAP|CKF_UNWRAP)
Mechanism #2
        Mechanism: 0x3 (CKM_RSA_X_509)
        Key Size: 512-4096
        Flags: 0x67B01 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_SIGN|CKF_SIGN_RECOVER|CKF_VERIFY|
CKF_VERIFY_RECOVER|CKF_WRAP|CKF_UNWRAP)
…
```

Chapter 9, "Common token information," on page 41 provides documentation that applies to all openCryptoki token types.

The following token-specific information is provided:

- Chapter 10, "CCA token," on page 45
- Chapter 11, "ICA token," on page 51

- Chapter 12, “EP11 token,” on page 57
- Chapter 13, “Soft token,” on page 75
- Chapter 14, “Directory content for CCA, ICA, EP11, and Soft tokens,” on page 81
- Chapter 15, “ICSF token,” on page 85

Note: Linux on Z and LinuxONE do not support the Trusted Platform Module (TPM) token library.

Chapter 9. Common token information

Read information that applies to all openCryptoki token types.

You can introduce one or multiple token instances of all token types into the openCryptoki framework. For this purpose, you must define a slot entry for each desired token instance in the global openCryptoki configuration file called `opencryptoki.conf`.

If you use multiple token instances, you must specify a unique token directory in the slot entry for each instance, using the `tokname` attribute. This token directory receives the token-individual information (like for example, key objects, user PIN, SO PIN, or hashes). Thus, the information for a certain token instance is separated from other tokens.

Adding tokens to openCryptoki

You can introduce one or multiple token instances of any type into the openCryptoki framework. For this purpose, you must define a slot entry for each desired token instance in the global openCryptoki configuration file called `opencryptoki.conf`.

Note: Currently, for EP11 tokens only, you can configure each token instance differently using a specific EP11 token configuration file (see "Defining an EP11 token-specific configuration file" on page 57).

With multiple tokens of a specific token type configured, you can assign dedicated adapters and domains to different tokens respectively. This ensures data isolation between multiple applications.

If you use multiple token instances of one certain token type, you must specify a unique token directory in the slot entry for each token, using the `tokname` attribute. This token directory receives the token-individual information (like for example, key objects, user PIN, SO PIN, or hashes). Thus, the information for a certain token instance is separated from other token instances.

For example, the default EP11 token directory is `/var/lib/opencryptoki/ep11tok/`. You can use the default only for a single EP11 token. Examples for multiple token directories can be:

```
/var/lib/opencryptoki/ep11token1/
/var/lib/opencryptoki/ep11token2/
/var/lib/opencryptoki/ccatoken1/
/var/lib/opencryptoki/ccatoken2/
```

Note: A certain token type configuration applies to all applications that use tokens of this type.

Adding a slot entry for each EP11 token in `opencryptoki.conf`

As already mentioned, the default openCryptoki configuration file `opencryptoki.conf` provides a slot entry for the EP11 token. It is preconfigured to slot #4. Each slot entry must set the `stdll` attribute to `libpkcs11_ep11.so`. Check this default entry to find out whether you can use it as is.

For each configured EP11 token, you must create a specific EP11 token configuration file. This EP11-specific configuration file defines the target adapters and target adapter domains to which the EP11 token sends its cryptographic requests.

In turn, each slot entry in the global openCryptoki configuration file must specify this EP11 token configuration file. For this purpose, use the `confname` attribute with the unique name of the respective EP11 token configuration file as value.

The example from Figure 7 on page 42 configures two EP11 tokens in slots 4 and 5 in the openCryptoki configuration file. It defines the names of the specific token configuration files to be `ep11tok01.conf` and `ep11tok02.conf`. Per default, these files are searched in the directory where openCryptoki searches its global configuration file. Figure 8 on page 42 shows an example of an EP11 token configuration file.

```
slot 4
  {
     stdll = libpkcs11_ep11.so
     confname = ep11tok01.conf
     tokname = ep11token01
     description = "Ep11 Token"
     manufacturer = "IBM"
     hwversion = "4.11"
     firmwareversion = "2.0"
  }

slot 5
  {
     stdll = libpkcs11_ep11.so
     confname = ep11tok02.conf  /* only for EP11 tokens  */
     tokname = ep11token02
  }
```

Figure 7. Multiple EP11 token instances

```
#
# EP11 token configuration
#
APQN_WHITELIST
0 0
0 1
2 84
END
FORCE_SENSITIVE
STRICT_MODE
VHSM_MODE
CPFILTER /etc/opencryptoki/ep11cpfilter.conf
OPTIMIZE_SINGLE_PART_OPERATIONS
DIGEST_LIBICA DEFAULT
USE_PRANDOM
```

Figure 8. Sample of an EP11 token configuration file

How to recognize tokens

You can use the **`pkcsconf -t`** command to display information about all available tokens. You can check the slot and token information, and the PIN status at any time.

The screen from Figure 9 on page 43 presents excerpts of a **`pkcsconf -t`** command output. The slot number is associated with the shown token number. The ICA token is plugged into slot 1. Therefore, you see information about the ICA token with the label `IBM ICA PKCS #11` in section **`Token #1 Info`**. Accordingly, in Figure 9 on page 43, you see a CCA token in slot 2, a Soft token in slot 3, and an EP11 token in slot 4.

```
$ pkcsconf -t

Token #1 Info:
        Label: IBM ICA PKCS #11
        Manufacturer: IBM
        Model: ICA
        Serial Number:
        Flags: 0x880445 (RNG|LOGIN_REQUIRED|CLOCK_ON_TOKEN|TOKEN_INITIALIZED|
USER_PIN_TO_BE_CHANGED|
SO_PIN_TO_BE_CHANGED)

        Sessions: 0/[effectively infinite]
        R/W Sessions: [information unavailable]/[effectively infinite]
        PIN Length: 4-8
        Public Memory: [information unavailable]/[information unavailable]
        Private Memory: [information unavailable]/[information unavailable]
        Hardware Version: 0.0
        Firmware Version: 0.0
        Time: 2021081811215500
Token #2 Info:
        Label: ccatok
        Manufacturer: IBM
        Model: CCA
        Serial Number:
        Flags: 0x880045 (RNG|LOGIN_REQUIRED|CLOCK_ON_TOKEN|USER_PIN_TO_BE_CHANGED|
SO_PIN_TO_BE_CHANGED)
                                D|SO_PIN_TO_BE_CHANGED)
        Sessions: 0/[effectively infinite]
        R/W Sessions: [information unavailable]/[effectively infinite]
        PIN Length: 4-8
        ...
Token #3 Info:
        Label: softtok
        ...
Token #4 Info:
        Label: ep11tok
        Manufacturer: IBM
        Model: EP11
        Serial Number: 93AABC7X69330380
        Flags: 0x80004D (RNG|LOGIN_REQUIRED|USER_PIN_INITIALIZED|CLOCK_ON_TOKEN|
SO_PIN_TO_BE_CHANGED)
                                SO_PIN_TO_BE_CHANGED)
        ...
        Hardware Version: 7.28
        Firmware Version: 3.1
        Time: 2021081811215500
```

Figure 9. Token information

The most important information is as follows:

- The token **`Label`**, either the default name or a name that you assigned at the initialization phase. In the example, you see that the default name `icatok` was replaced by `IBM ICA PKCS #11` during the initialization phase. You can initialize a token and change a token label by using the `pkcsconf -I` command. As a result, you see the flag **TOKEN_INITIALIZED** in the output.
- The **`Flags`** provide information about the token initialization status, the PIN status, and features such as ***Random Number Generator*** (RNG). They also provide information about requirements, such as ***Login required***, which means that there is at least one mechanism that requires a session log-in to use that cryptographic function.

  The flag USER_PIN_TO_BE_CHANGED indicates that the user PIN must be changed before the token can be used. The flag SO_PIN_TO_BE_CHANGED indicates that the SO PIN must be changed before administration commands can be used.
- The **`PIN Length`** range declared for this token.

For more information about the flags provided in this output, see the description of the **CK_TOKEN_INFO** structure in PKCS #11 Cryptographic Token Interface Base Specification Version 3.0.

ECC header file

For elliptic curve cryptography (ECC), openCryptoki ships the `ec_curves.h` header file. This file defines the curves known by openCryptoki.

View an excerpt of the `ec_curves.h` file.

Note: Whether a curve is actually supported by openCryptoki depends on the utilized token and on the available hardware.

```
/*
 * OIDs and their DER encoding for the EC curves supported by OpenCryptoki:
 */

/* brainpoolP160r1: 1.3.36.3.3.2.8.1.1.1 */
#define OCK_BRAINPOOL_P160R1    { 0x06, 0x09, 0x2B, 0x24, 0x03, 0x03, \
                                  0x02, 0x08, 0x01, 0x01, 0x01 }

/* brainpoolP160t1: 1.3.36.3.3.2.8.1.1.2 */
#define OCK_BRAINPOOL_P160T1    { 0x06, 0x09, 0x2B, 0x24, 0x03, 0x03, \
                                  0x02, 0x08, 0x01, 0x01, 0x02 }

/* brainpoolP192r1: 1.3.36.3.3.2.8.1.1.3 */
#define OCK_BRAINPOOL_P192R1    { 0x06, 0x09, 0x2B, 0x24, 0x03, 0x03, \
                                  0x02, 0x08, 0x01, 0x01, 0x03 }
```

Read the following topics:

- "ECC curves supported by the CCA token" on page 47
- "ECC curves supported by the ICA token" on page 54
- "ECC curves supported by the EP11 token" on page 66.

The selection of curves supported by the Soft token depends on the installed version of OpenSSL.

PKCS #11 Baseline Provider support

openCryptoki implements the *PKCS #11 Baseline Provider* specification. A library implementing PKCS #11 according to the standards of the **Baseline Provider Clause** is called a *PKCS #11 Baseline Provider*. Such a provider has the ability to provide information about its cryptographic services.

A *PKCS #11 Baseline Provider* library can be exploited by an application conforming to the **Baseline Consumer Clause**. Such an application is therefore called a *PKCS #11 Baseline Consumer*. A *Baseline Consumer* calls a *Baseline Provider* implementation of the PKCS #11 API in order to use the cryptographic functionality from that provider. Thus, at run-time, a consumer can query information about a provider, for example, about the offered cryptographic services.

For detailed information about the conformance of a *PKCS #11 Baseline Consumer* and of a *PKCS #11 Baseline Provider* read PKCS #11 Cryptographic Token Interface Profiles Version 3.0.

Chapter 10. CCA token

A CCA token is a secure key token. Any key generation is processed inside an IBM cryptographic coprocessor. A clear key is generated and wrapped by a master key which resides only within the cryptographic coprocessor. The clear key is then deleted and is never visible outside the coprocessor. The wrapped clear key is called a secure key and can only be unwrapped by using the master key within the coprocessor. Secure keys can safely be stored on a system, because they cannot be used for decrypting or encrypting without the master key.

A list of PKCS #11 mechanisms supported by the CCA token is provided, as well as information about the purpose and use of the **pkcscca** tool.

As a prerequisite for an operational CCA token, the CCA library (also called CCA host library in other documentations) must be installed (see Figure 3 on page 13). For information on how to install the CCA library, refer to *Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide*.

PKCS #11 mechanisms supported by the CCA token

View a list of mechanisms provided by PKCS #11 which you can use to exploit the openCryptoki features for the CCA token from within your application. Use the **pkcsconf -m -c <CCA_token_slot>** command to list the mechanisms (algorithms), that are supported by the CCA token.

The command output shown in Table 4 on page 45 lists all mechanisms that are supported by the CCA token in the specified slot.

Table 4. PKCS #11 mechanisms supported by the CCA token

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_DES_KEY_GEN	8-8 bytes	GENERATE	before 3.16
CKM_DES3_KEY_GEN	24-24 bytes	GENERATE	before 3.16
CKM_RSA_PKCS_KEY_PAIR_GEN	512-4096 bits	GENERATE_KEY_PAIR	before 3.16
CKM_RSA_PKCS	512-4096 bits	ENCRYPT, DECRYPT, SIGN, VERIFY, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_OAEP	512-4096	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_PSS	512-4096 bits	ENCRYPT, DECRYPT, SIGN, VERIFY	before 3.16
CKM_MD5_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA224_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA224_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA256_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA512_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	before 3.16

Table 4. PKCS #11 mechanisms supported by the CCA token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_DES_CBC	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES_CBC_PAD	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES3_CBC	24-24 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES3_CBC_PAD	24-24 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_AES_KEY_GEN	16-32 bytes	GENERATE	before 3.16
CKM_AES_ECB	16-32 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_AES_CBC	16-32 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_AES_CBC_PAD	16-32 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_SHA512	n/a	DIGEST	before 3.16
CKM_SHA512_HMAC	256-2048 bits	SIGN, VERIFY	before 3.16
CKM_SHA512_HMAC_GENERAL	256-2048 bits	SIGN, VERIFY	before 3.16
CKM_SHA512_RSA_PKCS	512-4096	SIGN, VERIFY	before 3.16
CKM_SHA384	n/a	DIGEST	before 3.16
CKM_SHA384_HMAC	192-2048 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_HMAC_GENERAL	192-2048 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_RSA_PKCS	512-4096	SIGN, VERIFY	before 3.16
CKM_SHA256	n/a	DIGEST	before 3.16
CKM_SHA256_HMAC	128-2048 bits	SIGN, VERIFY	before 3.16
CKM_SHA256_HMAC_GENERAL	128-2048 bits	SIGN, VERIFY	before 3.16
CKM_SHA224	n/a	DIGEST	before 3.16
CKM_SHA224_HMAC	112-2048 bits	SIGN, VERIFY	before 3.16
CKM_SHA224_HMAC_GENERAL	112-2048 bits	SIGN, VERIFY	before 3.16
CKM_SHA_1	n/a	DIGEST	before 3.16
CKM_SHA_1_HMAC	80-2048 bits	SIGN, VERIFY	before 3.16
CKM_SHA_1_HMAC_GENERAL	80-2048 bits	SIGN, VERIFY	before 3.16
CKM_MD5	n/a	DIGEST	before 3.16
CKM_ECDSA_KEY_PAIR_GEN	160-521 bits	GENERATE_KEY_PAIR, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_ECDSA	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_ECDSA_SHA1	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	GENERATE	before 3.16

Table 4. PKCS #11 mechanisms supported by the CCA token (continued)

Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_ECDSA_SHA224	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	GENERATE	before 3.16
CKM_ECDSA_SHA256	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	GENERATE	before 3.16
CKM_ECDSA_SHA384	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	GENERATE	before 3.16
CKM_ECDSA_SHA512	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	GENERATE	before 3.16

For explanations of the key object properties, see the PKCS #11 Cryptographic Token Interface Standard.

ECC curves supported by the CCA token

View a list of curves supported by the CCA token for elliptic curve cryptography (ECC).

Table 5 on page 47 shows the curves that the CCA token supports for elliptic curve cryptography.

Table 5. Curves supported by the CCA token for elliptic curve cryptography (ECC)

Curve	**Purpose**
brainpoolP160r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP192r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP224r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP256r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP320r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP1384r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP512r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE

Table 5. Curves supported by the CCA token for elliptic curve cryptography (ECC) (continued)

Curve	**Purpose**
prime192v1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
prime256v1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
secp224r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
secp384r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
secp521r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE

Usage notes for CCA library functions

Read important information about the usage and restrictions of CCA library functions.

- Plain CCA key objects (that is, CCA secure key objects generated by the CCA library `libcsulcca.so`) can be extracted from sensitive openCryptoki key objects for the CCA token by accessing the value of the CKA_IBM_OPAQUE attribute.

  Plain CCA key objects can be imported into sensitive openCryptoki key objects for the CCA token by assigning the plain CCA key objects to the CKA_IBM_OPAQUE attribute value. This import is supported for key types:

  - CCA DES key token
  - CCA DES3 key token
  - CCA AES data key token
  - CCA internal RSA private key token (RSA-AESM and RSA-AESC)
  - CCA RSA public key token (RSA-AESM and RSA-AESC)
  - CCA HMAC key token
  - CCA internal EC private key token
  - CCA EC public key token

  CCA AES cipher key token import is not supported and `C_CreateObject()` returns with CKR_TEMPLATE_INCONSISTENT.

  Plain CCA key objects can also be imported into sensitive openCryptoki key objects for the CCA token by assigning the value to the CKA_VALUE attribute or other key-type specific attributes using the PKCS #11 `C_CreateObject()` function. This is supported for RSA private keys and for RSA public keys. Starting with openCryptoki version 3.14, the CCA token supports the C_CreateObject() function for AES DATA, DES, DES3, and generic secret keys in addition to RSA keys.

  Starting with openCryptoki 3.14, the CCA token also supports plain HMAC and EC keys with different curves.

- The default CKA_SENSITIVE setting for generating a key is CK_FALSE although the openCryptoki CCA token handles only secure keys, which correspond to sensitive keys in PKCS #11.

  Setting the value of CKA_SENSITIVE to CK_FALSE does not inhibit inspecting the value of CKA_VALUE. This setting does not compromise security because CKA_VALUE does not contain any sensitive or secret information. Also, CKA_IBM_OPAQUE does not contain any information that can be exploited without the corresponding CCA master key.

- The function `C_DigestKey` is not supported by the CCA token.

Migrate to a new CCA master key - pkcscca utility

If you need to migrate a CCA key to a new wrapping CCA master key (MK), use the **pkcscca** tool.

Before you begin

Prerequisite for using the key migration function is that you have installed openCryptoki version 3.4 or higher.

About this task

There may be situations when CCA master keys must be changed. All CCA secret and private keys are enciphered (wrapped) with a master key (MK). After a CCA master key is changed, the keys wrapped with an old master key need to be re-enciphered with the new master key. Only keys which are marked as CKA_EXTRACTABLE=TRUE can be migrated. However, by default all keys are marked as CKA_EXTRACTABLE. So only those keys where the user explicitly chooses to mark them as non extractable, for example, by setting CKA_EXTRACTABLE=FALSE cannot be migrated.

Use the **pkcscca** tool to migrate wrapped CCA keys.

After a new master key is loaded and set, perform the following steps:

Procedure

1. Stop all processes that are currently using openCryptoki with the CCA token.
   a) Stop all applications that use openCryptoki.
   b) Find out whether the `pkcsslotd` daemon is running by issuing one (or both to cross-check) of the following commands:

   ```
   $ systemctl status pkcsslotd    /* for Linux distributions providing systemd */
   $ ps awx | grep pkcsslotd
   ```

   If the daemon is running, the command output shows a process for `pkcsslotd`.
   c) If applicable, stop the daemon by issuing a command of this form:

   ```
   $ systemctl stop pkcsslotd.service /* for Linux distributions providing systemd */
   ```

2. Back up the token object repository of the CCA token. For example, you can use the following commands:

```
cd /var/lib/opencryptoki/cca/
      tar -cvzf ~/cca/TOK_OBJ_backup.tgz TOK_OBJ
```

3. Migrate the keys of the CCA token object repository with the **pkcscca** migration tool.

```
pkcscca -m keys -s <slotid> -k <aes|apka|asym|sym>
```

The following parameters are mandatory:

-s
    slot number for the CCA token

-k
    master key type to be migrated: `aes`, `apka`, `asym`, or `sym`

The following parameter is optional:

-m keys
re-encipers private keys only with a new CCA master key.

All the specified token objects representing extractable keys that are found for the CCA token are re-encrypted and ready for use. Keys with an attribute CKA_EXTRACTABLE=FALSE are not eligible for migration. The keys that failed to migrate are displayed to the user.

Example:

```
$ pkcscca -m keys -s 2 -k sym
```

migrates all private keys wrapped with symmetric master keys found in the CCA plug-in for openCryptoki in PKCS slot 2.

4. Re-start the previously stopped openCryptoki processes.

   Start or restart pkcsslotd if it was stopped in step 1.

Results

All specified keys, for example, all private and secret keys (for asymmetric and symmetric cryptography) are now re-encrypted with the new CCA master key and are ready for use in CCA verbs.

Chapter 11. ICA token

Read about the tasks to be performed if you want to use the ICA token from within the openCryptoki framework.

The legacy name and therefore the default name of an ICA token is *lite*.

As a prerequisite for an operational ICA token, the ICA library must be installed (see Figure 3 on page 13).

PKCS #11 mechanisms supported by the ICA token

View a list of mechanisms provided by PKCS #11 which you can use to exploit the openCryptoki features for the ICA token from within your application. Use the **`pkcsconf -m -c <ICA_token_slot>`** command to list the mechanisms (algorithms), that are supported by the ICA token.

The command output depends on the libica version, whether libica is running in FIPS mode, and on the processor generation. The output of the **`pkcsconf -m -c <slot>`** command corresponds to the list shown in Table 6 on page 51 which presents all mechanisms supported by the ICA token on an IBM z15™ machine.

Table 6. Supported mechanism list for the ICA token

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_RSA_PKCS_KEY_PAIR_GEN	512-4096 bits	HW, GENERATE_KEY_PAIR	before 3.16
CKM_RSA_PKCS	512-4096 bits	HW, ENCRYPT, DECRYPT, SIGN, SIGN_RECOVER, VERIFY, VERIFY_RECOVER, WRAP, UNWRAP	before 3.16
CKM_RSA_X_509	512-4096 bits	HW, ENCRYPT, DECRYPT, SIGN, SIGN_RECOVER, VERIFY, VERIFY_RECOVER, WRAP, UNWRAP	before 3.16
CKM_SHA1_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_RSA_PKCS_OAEP	512-4096 bits	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_PSS	512-4096 bits	HW, SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS_PSS	512-4096 bits	HW, SIGN, VERIFY	before 3.16
CKM_SHA256_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_RSA_PKCS	512-4096 bits	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA256_RSA_PKCS_PSS	512-4096 bits	HW, SIGN, VERIFY	before 3.16
CKM_SHA384_RSA_PKCS_PSS	512-4096 bits	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_RSA_PKCS_PSS	512-4096 bits	HW, SIGN, VERIFY	before 3.16
CKM_SHA224_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA224_RSA_PKCS_PSS	512-4096 bits	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_224	n/a	HW, DIGEST	before 3.16

Table 6. Supported mechanism list for the ICA token (continued)

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_SHA512_224_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_224_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_256	n/a	HW, DIGEST	before 3.16
CKM_SHA512_256_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_256_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_DES_KEY_GEN	8-8 bytes	HW, GENERATE	before 3.16
CKM_DES_ECB	8-8 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES_CBC	8-8 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES_CBC_PAD	8-8 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_KEY_GEN	24-24 bytes	HW, GENERATE	before 3.16
CKM_DES3_ECB	24-24 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_CBC	24-24 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_MAC	24-24 bytes	SIGN, VERIFY	before 3.16
CKM_DES3_MAC_GENERAL	24-24 bytes	SIGN, VERIFY	before 3.16
CKM_DES3_CBC_PAD	24-24 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_CMAC_GENERAL	16-24 bytes	SIGN, VERIFY	before 3.16
CKM_DES3_CMAC	16-24 bytes	SIGN, VERIFY	before 3.16
CKM_DES_OFB64	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES_CFB64	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES_CFB8	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_MD5_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_MD5_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA_1	n/a	HW, DIGEST	before 3.16
CKM_SHA_1_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA_1_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA256	n/a	HW, DIGEST	before 3.16
CKM_SHA256_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA256_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA224	n/a	HW, DIGEST	before 3.16

Table 6. Supported mechanism list for the ICA token (continued)

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_SHA224_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA224_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA384	n/a	HW, DIGEST	before 3.16
CKM_SHA384_HMAC	HW, n/a	SIGN, VERIFY	before 3.16
CKM_SHA384_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512	n/a	HW, DIGEST	before 3.16
CKM_SHA512_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	HW, GENERATE	before 3.16
CKM_ECDSA_KEY_PAIR_GEN	160-521 bits	HW, GENERATE_KEY_PAIR, EC_F_P, EC_OID	before 3.16
CKM_ECDSA	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA1	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA224	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA256	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA384	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA512	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDH1_DERIVE	160-521 bits	HW, DERIVE, EC_F_P, EC_F_P, EC_OID	before 3.16
CKM_AES_KEY_GEN	16-32 bytes	HW, GENERATE	before 3.16
CKM_AES_ECB	16-32 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CBC	16-32 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_MAC	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_AES_MAC_GENERAL	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_AES_CBC_PAD	16-32 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CTR	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_GCM	16-32 bytes	ENCRYPT, DECRYPT	before 3.16

Table 6. Supported mechanism list for the ICA token (continued)

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_AES_CMAC_GENERAL	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_AES_CMAC	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_AES_OFB	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CFB64	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CFB8	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CFB128	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_IBM_SHA3_224	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_256	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_384	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_512	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_224_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_256_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_384_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_512_HMAC	n/a	SIGN, VERIFY	before 3.16

Certain mechanisms indicate the `HW` flag in the **Properties** column (short form for the CKF_HW). If for theses mechanisms, the CKF_HW flag is set to TRUE, the pertaining cryptographic operations are performed by the cryptographic hardware. If the flag is not set for these mechanisms, the operations are performed in software.

For a description of mechanisms with a name pattern of `CKM_IBM_...` refer to "IBM-specific mechanisms" on page 87.

Usage notes for the ICA library functions

As of openCryptoki version 3.6, the `C_SeedRandom` function of the ICA token always returns CKR_RANDOM_SEED_NOT_SUPPORTED.

ECC curves supported by the ICA token

View a list of curves supported by the ICA token for elliptic curve cryptography (ECC).

Table 7 on page 55 shows the maximum number of curves that the ICA token can support if all prerequisites are fulfilled at their best conditions. The following dependencies exist:

- Which openCryptoki version (and thus which libica version) is used? Refer to the applicable libica documentation for information about supported curves.
- Which cryptographic coprocessors are available?
- Is the MSA9 component of IBM z15 or later available?

- Is libica or OpenSSL running in FIPS mode? When FIPS mode is active for libica, OpenSSL is also set into FIPS mode. However, note that the case where libica does not run in FIPS mode, but OpenSSL does, may cause errors when software fallbacks are used. If, for example, an elliptic curve is supported by hardware in libica, but not by OpenSSL, because OpenSSL runs in FIPS mode, this software fallback fails.

Table 7. Curves supported by the ICA token for elliptic curve cryptography (ECC)

Curve	**Purpose**
brainpoolP160r1 **(1), (3)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
brainpoolP192r1 **(1), (3)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
brainpoolP224r1 **(1), (3)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
brainpoolP256r1 **(1), (3)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
brainpoolP320r1 **(1), (3)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
brainpoolP1384r1 **(1), (3)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
brainpoolP512r1 **(1), (3)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
prime192v1 **(1), (3)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
prime256v1 **(1), (2)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
secp224r1 **(1)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
secp384r1 **(1), (2)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`
secp521r1 **(1), (2)**	• for Sign/Verify operations with `CKM_ECDSA` and `CKM_ECDSA_SHAnnn` • for ECDH with `CKM_ECDH1_DERIVE`

Notes:

(1) supported via CryptoExpress CCA coprocessor
(2) supported via CPACF on processors with MSA9 component of IBM z15 or later
(3) not available if libica runs in FIPS mode

Chapter 12. EP11 token

An EP11 token is a secure key token. A list of PKCS #11 mechanisms supported by the EP11 token is provided, as well as information about the purpose and use of the tools **`pkcsep11_migrate`** and **`pkcsep11_session`**.

You can read some information about secure keys in Chapter 10, “CCA token,” on page 45.

As a prerequisite for an operational EP11 token, the EP11 library (also called EP11 host library in other documentations) must be installed (see Figure 3 on page 13).

For EP11 tokens, you can introduce one or multiple tokens into the openCryptoki framework (see “Adding tokens to openCryptoki” on page 41) and configure them differently. For information on how to install the EP11 library and on how to configure a certain EP11 token using a token specific configuration file, refer to Exploiting Enterprise PKCS #11 using openCryptoki.

Defining an EP11 token-specific configuration file

One default configuration file for the EP11 token called `ep11tok.conf` is delivered by openCryptoki. You must adapt it according to your installation's system environment. If you use multiple EP11 tokens, you must provide an individual token configuration file for each token. Each slot entry in the global configuration file `opencryptoki.conf` defines these configuration file names.

In the example from “Adding tokens to openCryptoki” on page 41, these names are defined as `ep11tok01.conf` and `ep11tok02.conf`. If the environment variable `OCK_EP11_TOKEN_DIR` is set, then the EP11 token looks for the configuration file or files in the directory specified with this variable. If `OCK_EP11_TOKEN_DIR` is not set, then the EP11 token configuration files are searched in the global openCryptoki directory, for example: `/etc/opencryptoki/ep11tok.conf`.

Example: If a slot entry in `opencryptoki.conf` specifies `confname = ep11tok02.conf`, and you set the environment variable `OCK_EP11_TOKEN_DIR` like:

```
export OCK_EP11_TOKEN_DIR=/home/user/ep11token
```

then your EP11 token configuration file appears here:

```
<root>/home/user/ep11token/ep11tok02.conf
```

You can use the shown example to set your own token directory for test purposes.

Note: The setting of this environment variable is ignored, if a program trying to access the designated EP11 token is marked with file permission `setuid`.

The following is a list of available options for an EP11 token configuration file. A sample of such a file is shown in Figure 12 on page 62.

APQN_ALLOWLIST

Because different EP11 hardware security modules (HSM) can use different wrapping keys (referred to as master keys in the TKE environment), users need to specify which HSM, in practice an adapter/domain pair, can be used by the EP11 token as a target for cryptographic requests. Therefore, an EP11 token configuration file contains a list of adapter/domain pairs to be used.

You start this list of adapter/domain pairs starting with a line containing the keyword `APQN_ALLOWLIST`. Next follows the list which can specify up to 512 adapter/domain pairs, denoted by decimal numbers in the range 0 - 255. Each pair designates an adapter (first number) and a domain (second number) accessible to the EP11 token. Close the list using the keyword END.

Alternatively, you can use the keyword `APQN_ANY` to define that all adapter/domain pairs with EP11 firmware, that are available to the system, can be used as target adapters. This is the default.

Notes:

- The term *APQN* stands for adjunct processor queue number. It designates the combination of a cryptographic coprocessor (adapter) and a domain, a so-called adapter/domain pair. At least one adapter/domain pair must be specified.
- If more than one APQN is used by a token, then these APQNs must be configured with the same master key.
- This attribute used to be APQN_WHITELIST but has been renamed due to the inclusive terminology initiative. For compatibility reasons, you can still use the old name, but this is considered to be deprecated.

An adapter/domain pair is displayed by the **`lszcrypt`** tool or in the `sys` file system (for example, in `/sys/bus/ap/devices`) in the form *card .domain*, where both numbers are displayed in hexadecimal format.

There are two ways to specify the cryptographic adapter:

- either as an explicit list of adapter/domain pairs:

  ```
  APQN_ALLOWLIST
   8 13
   10 13
  END
  ```

  The adapter and domain can be given in decimal, octal (with leading 0), or hexadecimal (with leading 0x) notation:

  ```
  APQN_ALLOWLIST
   8 0x0d
   0x0a 13
  END
  ```

  Valid adapter and domain values are in the range 0 to 255.
- or as any available cryptographic adapters:

  ```
  APQN_ANY
  ```

In the example from Figure 12 on page 62, adapter 0 with domains 0 and 1, and adapter 2 with domain 84 are specified as target for requests from the EP11 token. In Figure 10 on page 59, these adapter/domain pairs are shown in hexadecimal notation as APQNs (00,0000), (00,0001), and (02,0054).

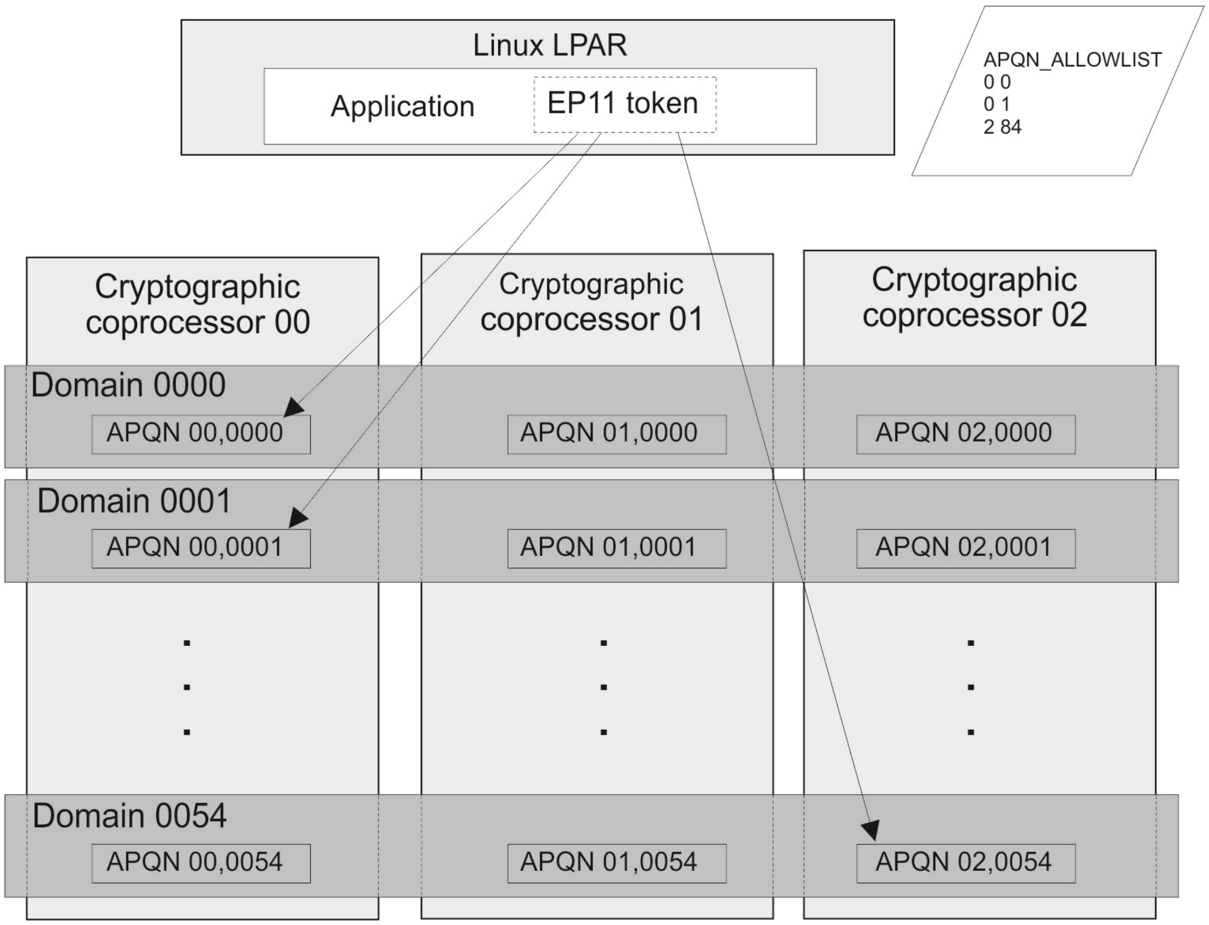


Figure 10. Cryptographic configuration for an LPAR

CPFILTER

The list of mechanisms returned by `C_GetMechanismList` is filtered using the domain or access control point (ACP) settings of the used cryptographic coprocessors. The EP11 access control point filter configuration file (ACP-filter configuration file) is used to associate certain access (domain) control points with mechanisms that are dependent on these access control points. The default ACP-filter configuration file is `ep11cpfilter.conf` located in the same directory as this EP11 token configuration file. You can optionally specify the name or location, or both, of the ACP-filter file:

```
CPFILTER /etc/opencryptoki/ep11cpfilter.conf
```

```
#
# EP11 token ACP-filter configuration
#
# The list of mechanisms returned by C_GetMechanismList is filtered
# using the access control point settings of the used crypto adapters.
# The EP11 ACP-filter config file is used to associate certain access control
# points with mechanisms that are dependent on these access control points.
#
# Syntax:
#       acp: mech1, mech2, ...
#
# Both, acp as well as mech<n> is specified as name or in decimal, octal
# (with leading 0) or hexadecimal (with leading 0x):
#
#       XCP_CPB_SIGN_SYMM: CKM_SHA256_HMAC, CKM_SHA256_HMAC_GENERAL
#       4: 0x00000251, 0x00000252
#

# sign with HMAC or CMAC
XCP_CPB_SIGN_SYMM: CKM_SHA256_HMAC, CKM_SHA256_HMAC_GENERAL, CKM_SHA224_HMAC,
CKM_SHA224_HMAC_GENERAL, CKM_SHA384_HMAC, CKM_SHA384_HMAC_GENERAL, CKM_SHA512_HMAC,
CKM_SHA512_HMAC_GENERAL, CKM_SHA_1_HMAC, CKM_SHA_1_HMAC_GENERAL, CKM_IBM_SHA3_224_HMAC,
CKM_IBM_SHA3_256_HMAC,CKM_IBM_SHA3_384_HMAC, CKM_IBM_SHA3_512_HMAC, CKM_IBM_CMAC,
CKM_DES3_CMAC, CKM_DES3_CMAC_GENERAL, CKM_AES_CMAC, CKM_AES_CMAC_GENERAL

# verify with HMAC or CMAC
XCP_CPB_SIGVERIFY_SYMM: CKM_SHA256_HMAC, CKM_SHA256_HMAC_GENERAL, CKM_SHA224_HMAC,
CKM_SHA224_HMAC_GENERAL, CKM_SHA384_HMAC, CKM_SHA384_HMAC_GENERAL, CKM_SHA512_HMAC,
CKM_SHA512_HMAC_GENERAL, CKM_SHA_1_HMAC, CKM_SHA_1_HMAC_GENERAL, CKM_IBM_SHA3_224_HMAC,
CKM_IBM_SHA3_256_HMAC,CKM_IBM_SHA3_384_HMAC, CKM_IBM_SHA3_512_HMAC, CKM_IBM_CMAC,
CKM_DES3_CMAC, CKM_DES3_CMAC_GENERAL, CKM_AES_CMAC, CKM_AES_CMAC_GENERAL

# encrypt with symmetric keys
XCP_CPB_ENCRYPT_SYMM: CKM_AES_ECB, CKM_AES_CBC, CKM_AES_CBC_PAD, CKM_DES3_ECB,
CKM_DES3_CBC, CKM_DES3_CBC_PAD, CKM_DES_ECB, CKM_DES_CBC

#decrypt with private keys
XCP_CPB_DECRYPT_ASYMM: CKM_RSA_PKCS

# decrypt with symmetric keys
XCP_CPB_DECRYPT_SYMM: CKM_AES_ECB, CKM_AES_CBC, CKM_AES_CBC_PAD, CKM_DES3_ECB,
CKM_DES3_CBC, CKM_DES3_CBC_PAD, CKM_DES_ECB, CKM_DES_CBC

# generate asymmetric keypairs
XCP_CPB_KEYGEN_ASYMM: CKM_RSA_PKCS_KEY_PAIR_GEN, CKM_RSA_X9_31_KEY_PAIR_GEN,
CKM_EC_KEY_PAIR_GEN, CKM_DSA_KEY_PAIR_GEN, CKM_DH_PKCS_KEY_PAIR_GEN

# DSA private-key use
XCP_CPB_ALG_DSA: CKM_DSA_PARAMETER_GEN, CKM_DSA_KEY_PAIR_GEN, CKM_DSA, CKM_DSA_SHA1

# Diffie-Hellman use (private keys)
XCP_CPB_ALG_DH: CKM_ECDH1_DERIVE, CKM_DH_PKCS_PARAMETER_GEN, CKM_DH_PKCS_KEY_PAIR_GEN,
CKM_DH_PKCS_DERIVE

# enable support of curve25519, c448 and related algorithms incl. EdDSA (ed25519 and ed448)
XCP_CPB_ALG_EC_25519: CKM_IBM_EC_C25519, CKM_IBM_EDDSA_SHA512, CKM_IBM_EC_C448,
CKM_IBM_ED448_SHA3

#enable support of Dilithium
XCP_CPB_ALG_PQC_DILITHIUM: CKM_IBM_DILITHIUM
```

Figure 11. Excerpt of a sample ACP-filter configuration file

DIGEST_LIBICA *<libica-path>* | DEFAULT | OFF

To improve the performance of required hash functions, the EP11 token on initialization loads the default libica library. If required, the EP11 token invokes the libica SHA-based hash functions, because the libica library performs these hash functions on the CPACF, thus avoiding hash processing on a cryptographic coprocessor which results in I/O operations to the adapter.

libica provides an OpenSSL based software fall-back, in case CPACF or a certain hashing function of CPACF is not available. In case a libica operation fails, because neither the hardware nor the software support is available, or if libica is not available at all, then the request is passed to the EP11 library instead.

With the DIGEST_LIBICA option, you can control which libica library is loaded:

DEFAULT

The default libica library is loaded. If libica could not be found, a message is issued to `syslog`, and all hash based functions use the EP11 library.

The same behavior is applied if the DIGEST_LIBICA option is not specified at all.

<libica-path>

The specified library is loaded. If it can not be found, a message is issued to `syslog`, and token initialization fails.

OFF

No libica is loaded, and all hash based functions use the EP11 library.

If DIGEST_LIBICA is not specified, then the default libica library is loaded (same behavior as for DIGEST_LIBICA DEFAULT).

FORCE_SENSITIVE

Specify this option to force that the default for CKA_SENSITIVE is CK_TRUE for secret keys. For more information, see "Usage notes for the EP11 library functions" on page 68.

OPTIMIZE_SINGLE_PART_OPERATIONS

Set this option to optimize the performance of single part sign- and verify-operations, as well as of single part encrypt- or decrypt-operations. Then the `init` call is not passed through the EP11 library as long as there is no corresponding multi-part operation.

When this option is enabled, error handling can be slightly different, when errors from the deferred `init` call are presented during the first update call or during the calls to `C_Sign`, `C_Verify`, `C_Encrypt`, or `C_Decrypt` for a single part operation. That is, the first update call on a multi part operation or the mentioned calls for a single part operation may return errors, which are usually not returned by the update call. Such errors may be for example:

CKR_OBJECT_HANDLE_INVALID
CKR_ATTRIBUTE_VALUE_INVALID
CKR_KEY_HANDLE_INVALID
CKR_KEY_SIZE_RANGE
CKR_KEY_TYPE_INCONSISTENT
CKR_MECHANISM_INVALID
CKR_MECHANISM_PARAM_INVALID

PKEY_MODE

Use this option to define that for EP11 secure keys of type AES or EC an additional pertaining protected key shall be created, and the protected keys shall be used whenever possible. This significantly improves performance, because protected keys work on the CPACF feature, and calls to CPACF are much faster than calls to an EP11 cryptographic coprocessor.

Notes:

- Processing of the PKEY_MODE option is performed by a transparent IBM-specific mechanism. This mechanism in turn reads certain attributes of the processed key object to determine the generation of the protected key. The new protected key is bound to the key object by the attribute CKA_IBM_OPAQUE_PKEY. For an explanation of the involved attributes, read "Miscellaneous attributes" on page 96.
- The PKEY_MODE option is effective only on IBM z15 processors or later, with EP11 cryptographic coprocessors starting with CEX7P. It requires the EP11 host library 3.0 or later.
- Keys created before introducing the protected key option are not usable for protected key support, because they do not have the CKA_IBM_PROTKEY_EXTRACTABLE attribute. Only keys created after activating the protected key option with the CREATE4NONEXTR mode are eligible for getting a protected key, depending on their key type (only AES and EC). See also "Miscellaneous attributes" on page 96.

Set the PKEY_MODE option to one of the following modes:

DISABLED

Protected key support is disabled. All keys are used as secure keys. This mode allows to completely disable protected key support, for example, for performance comparisons.

USE_EXISTING

This is the default.

New keys

If the application did not specify attribute CKA_IBM_PROTKEY_EXTRACTABLE in its template for key generation, new keys of any type get CKA_IBM_PROTKEY_EXTRACTABLE=CK_FALSE, and no protected key is created.

Existing keys

Existing secure keys with a valid protected key are used via this protected key, and any invalid protected key is re-created with the help of the current firmware master key.

CREATE4NONEXTR

New keys

If the application did not specify CKA_IBM_PROTKEY_EXTRACTABLE in its template for key generation, new keys of any type with CKA_EXTRACTABLE=CK_FALSE get attribute CKA_IBM_PROTKEY_EXTRACTABLE=CK_TRUE and a protected key is automatically created at first use of the key.

Existing keys

Existing keys with a protected key bound to that key with attribute CKA_IBM_OPAQUE_PKEY are used via this bound protected key. Also, if required, for example, after deactivating and reactivating an LPAR, any invalid protected key is re-created by wrapping with the current firmware master key.

STRICT_MODE

In strict-mode, all session-keys strictly belong to the PKCS #11 session that created it. When the PKCS #11 session ends, all session keys created for this session can no longer be used.

For more information, read topic *Controlling access to cryptographic objects* in *Exploiting Enterprise PKCS #11 using openCryptoki*.

USE_PRANDOM

Set this option to control from where the EP11 token reads random data. With USE_PRANDOM specified, the EP11 token reads random data from `/dev/prandom`, or from `/dev/urandom` if `/dev/prandom` is not available. The default is to read the random data using the `m_GenerateRandom` function from the Crypto Express EP11 coprocessor.

VHSM_MODE

In VHSM-mode (virtual-HSM), all keys generated by the EP11 token strictly belong to the EP11 token that created it. Every EP11 token running in this mode requires a VHSM card-PIN which must be set using the **`pkcsep11_session`** tool.

```
#
# EP11 token configuration
#
APQN_WHITELIST
0 0
0 1
2 84
END
FORCE_SENSITIVE
STRICT_MODE
VHSM_MODE
CPFILTER /etc/opencryptoki/ep11cpfilter.conf
OPTIMIZE_SINGLE_PART_OPERATIONS
DIGEST_LIBICA DEFAULT
USE_PRANDOM
```

Figure 12. Sample of an EP11 token configuration file

PKCS #11 mechanisms supported by the EP11 token

View a list of mechanisms provided by PKCS #11 which you can use to exploit the openCryptoki features for the EP11 token from within your application.

Use the **pkcsconf** command with the shown parameters to retrieve a complete list of mechanisms that are supported by the EP11 token:

```
$ pkcsconf -m -c <slot>
Mechanism #2
        Mechanism: 0x131 (CKM_DES3_KEY_GEN)
        Key Size: 24-24
        Flags: 0x8001 (CKF_HW|CKF_GENERATE)
…
Mechanism #10
        Mechanism: 0x132 (CKM_DES3_ECB)
        Key Size: 24-24
        Flags: 0x60301 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_WRAP|CKF_UNWRAP)
Mechanism #11
        Mechanism: 0x133 (CKM_DES3_CBC)
        Key Size: 24-24
        Flags: 0x60301 (CKF_HW|CKF_ENCRYPT|CKF_DECRYPT|CKF_WRAP|CKF_UNWRAP)
...
```

On an Crypto Express EP11 coprocessor (CEX*P) which is configured to support all applicable PKCS #11 mechanisms from the current openCryptoki version, the EP11 token can exploit the mechanisms listed by the **pkcsconf -m -c <slot>** command output. This output corresponds to the list shown in Table 8 on page 63. Each mechanism provides its supported key size and some further properties such as hardware support and mechanism information flags. These flags provide information about the PKCS #11 functions that may use the mechanism. In some cases, the flags also provide further attributes that describe the supported variants of the mechanism. Typical functions are for example, *encrypt*, *decrypt*, *wrap key*, *unwrap key*, *sign*, or *verify*.

Table 8. PKCS #11 mechanisms supported by the EP11 token

Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_RSA_PKCS_OAEP	1024-4096 bits	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_KEY_PAIR_GEN	1024-4096 bits	GENERATE_KEY_PAIR	before 3.16
CKM_RSA_X9_31_KEY_PAIR_GEN	1024-4096 bits	GENERATE_KEY_PAIR	before 3.16
CKM_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_X9_31	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA256_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA256_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA224_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA224_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA512_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA512_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16

Table 8. PKCS #11 mechanisms supported by the EP11 token (continued)

Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_AES_KEY_GEN	16-32 bytes	GENERATE	before 3.16
CKM_AES_ECB	16-32 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_AES_CBC	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CBC_PAD	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES2_KEY_GEN	16-16 bytes	GENERATE	before 3.16
CKM_DES3_KEY_GEN	24-24 bytes	GENERATE	before 3.16
CKM_DES3_ECB	16-24 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES3_CBC	16-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_CBC_PAD	16-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_SHA256	n/a	DIGEST	before 3.16
CKM_SHA256_KEY_DERIVATION	n/a	DERIVE	before 3.16
CKM_SHA256_HMAC	128-256 bytes	SIGN, VERIFY	before 3.16
CKM_SHA224	n/a	DIGEST	before 3.16
CKM_SHA224_KEY_DERIVATION	n/a	DERIVE	before 3.16
CKM_SHA224_HMAC	112-256 bytes	SIGN, VERIFY	before 3.16
CKM_SHA_1	n/a	DIGEST	before 3.16
CKM_SHA1_KEY_DERIVATION	n/a	DERIVE	before 3.16
CKM_SHA_1_HMAC	80-256 bytes	SIGN, VERIFY	before 3.16
CKM_SHA384	n/a	DIGEST	before 3.16
CKM_SHA384_KEY_DERIVATION	n/a	DERIVE	before 3.16
CKM_SHA384_HMAC	192-256 bytes	SIGN, VERIFY	before 3.16
CKM_SHA512	n/a	DIGEST	before 3.16
CKM_SHA512_KEY_DERIVATION	n/a	DERIVE	before 3.16
CKM_SHA512_HMAC	256-256 bytes	SIGN, VERIFY	before 3.16
CKM_SHA512_256	n/a	DIGEST	before 3.16
CKM_SHA512_256_HMAC	128-256 bytes	SIGN, VERIFY	before 3.16
CKM_SHA512_224	n/a	DIGEST	before 3.16
CKM_SHA512_224_HMAC	112-256 bytes	SIGN, VERIFY	before 3.16
CKM_ECDSA_KEY_PAIR_GEN	192-521 bits	GENERATE_KEY_PAIR, EC_F_P, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16

Table 8. PKCS #11 mechanisms supported by the EP11 token (continued)

Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_ECDSA	192-521 bits	SIGN, VERIFY, EC_F_P, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA1	192-521 bits	SIGN, VERIFY, EC_F_P, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDH1_DERIVE	192-521 bits	DERIVE, EC_F_P, EC_UNCOMPRESS	before 3.16
CKM_DSA_PARAMETER_GEN	1024-3072 bits	GENERATE	before 3.16
CKM_DSA_KEY_PAIR_GEN	1024-3072 bits	GENERATE_KEY_PAIR	before 3.16
CKM_DSA	1024-3072 bits	SIGN, VERIFY	before 3.16
CKM_DSA_SHA1	1024-3072 bits	SIGN, VERIFY	before 3.16
CKM_DH_PKCS_PARAMETER_GEN	1024-3072 bits	GENERATE	before 3.16
CKM_DH_PKCS_KEY_PAIR_GEN	1024-3072 bits	GENERATE_KEY_PAIR	before 3.16
CKM_DH_PKCS_DERIVE	1024-3072 bits	DERIVE	before 3.16
CKM_IBM_DILITHIUM	256-256 bytes	SIGN, VERIFY, GENERATE_KEY_PAIR	before 3.16
CKM_RSA_X9_31	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_PBE_SHA1_DES3_EDE_CBC	24-24 bytes	GENERATE	before 3.16
CKM_IBM_SHA3_224	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_256	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_384	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_512	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_224_HMAC	112-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_256_HMAC	128-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_384_HMAC	192-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_512_HMAC	256-256 bytes	SIGN, VERIFY	before 3.16
CKM_ECDSA_SHA224	192-521 bits	SIGN, VERIFY, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA256	192-521 bits	SIGN, VERIFY, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA384	192-521 bits	SIGN, VERIFY, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA512	192-521 bits	SIGN, VERIFY, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_IBM_EC_C25519	256-256 bytes	DERIVE, EC_F_P, EC_UNCOMPRESS	before 3.16

Table 8. PKCS #11 mechanisms supported by the EP11 token (continued)

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_IBM_EC_X25519		is a synonym for CKM_IBM_EC_C25519	
CKM_IBM_EC_C448	448-448 bytes	DERIVE, EC_F_P, EC_UNCOMPRESS	before 3.16
CKM_IBM_EC_X448		is a synonym for CKM_IBM_EC_C448	
CKM_IBM_ED25519_SHA512	256-256 bytes	SIGN, VERIFY, EC_F_P, EC_UNCOMPRESS	before 3.16
CKM_IBM_EDDSA_SHA512		is a synonym for CKM_IBM_ED25519_SHA512	before 3.16
CKM_IBM_ED448_SHA3	448-448 bytes	SIGN, VERIFY, EC_F_P, EC_UNCOMPRESS	before 3.16
CKM_IBM_CMAC	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_AES_CMAC	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_DES3_CMAC	16-24 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_ATTRIBUTEBOUND_WRAP	0-4096 bits	WRAP, UNWRAP	3.16

For a description of mechanisms with a name pattern of `CKM_IBM_...` refer to Chapter 16, “IBM-specific mechanisms and features for openCryptoki ,” on page 87.

For more detailed information on how to use the EP11 token, refer to Exploiting Enterprise PKCS #11 using openCryptoki.

For explanation about the key object properties see the PKCS #11 Cryptographic Token Interface Standard.

ECC curves supported by the EP11 token

View a list of curves that are supported by the EP11 token for elliptic curve cryptography (ECC).

For the support of elliptic curve cryptography, the EP11 token provides standard mechanisms and IBM-specific mechanisms for key derivation and for sign and verify operations. For more information, refer to “PKCS #11 mechanisms supported by the EP11 token” on page 63.

Table 9. Curves supported by the EP11 token for elliptic curve cryptography (ECC)

Curve	Purpose
brainpoolP160r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP160t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP192r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE

Table 9. Curves supported by the EP11 token for elliptic curve cryptography (ECC) (continued)

Curve	**Purpose**
brainpoolP192t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP224r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP224t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP256r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP256t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP320r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP320t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP1384r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP384t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP512r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP512t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
prime192v1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
prime256v1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
secp224r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
secp256k1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
secp384r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE

Table 9. Curves supported by the EP11 token for elliptic curve cryptography (ECC) (continued)

Curve	**Purpose**
secp521r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
Montgomery curves, only for ECDH with certain IBM-specific mechanisms	
X448	ECDH with CKM_IBM_EC_C448
X25519	ECDH with CKM_IBM_EC_C25519
Edwards Curves, only for Sign/Verify (EdDSA) with certain IBM-specific mechanisms	
ed448	Sign/Verify with CKM_IBM_ED448_SHA3
ed25519	Sign/Verify with CKM_IBM_ED25519_SHA512

The EP11 library provides access control point 55 to enable support of curve25519, c448, and related algorithms, including EdDSA:

```
55    XCP_CPB_ALG_EC_25519     enable support of curve25519, c448 and related algorithms
                               incl. EdDSA
```

Usage notes for the EP11 library functions

In this topic, you find information about certain limitations of the EP11 library.

- The EP11 library implements the *secure key concept* as explained in the introduction of Chapter 10, “CCA token,” on page 45.

  Therefore, the EP11 token only knows sensitive secret keys (CKO_SECRET_KEY). However, the PKCS #11 standard defines the default value of attribute CKA_SENSITIVE to be CK_FALSE. Thus, for previous versions of the EP11 token, all applications must have the attribute value of CKA_SENSITIVE explicitly changed to CK_TRUE whenever an EP11 secret key had been generated, unwrapped, or build with C_CreateObject.

  Starting with the EP11 token for openCryptoki version 3.10, an option is implemented to change the default value of attribute CKA_SENSITIVE to be CK_TRUE for all secret keys created with the EP11 token. This applies to functions C_GenerateKey, C_GenerateKeyPair, C_UnwrapKey, and C_DeriveKey when creating key with CKA_CLASS = CKO_SECRET_KEY, if the attribute CKA_SENSITIVE is not explicitly specified in the template.

  To enable this option, you must specify keyword FORCE_SENSITIVE in the EP11 token configuration file, as shown in Figure 13 on page 68. Note that the semantics specified with the FORCE_SENSITIVE keyword matches the semantics used by z/OS for EP11.

  ```
  #
  # EP11 token configuration
  #
  FORCE_SENSITIVE
  #
  APQN_WHITELIST
  5 2
  6 2
  END
  ```

  Figure 13. Sample of an EP11 token configuration file

- Keys leaving the hardware security module (HSM) are encrypted by the HSM master key (wrapping key) and come as binary large object (BLOB). In openCryptoki, objects can have special attributes that describe the key properties. Besides dedicated attributes defined by the application, there are some attributes defined as token-specific by openCryptoki.

Table 10 on page 69 and Table 11 on page 69 show the EP11 token-specific attributes and their default values for private and secure keys.

Table 10. Private key (CKO_PRIVATE_KEY) default attributes of the EP11 token

Private key attributes	**value**
CKA_SENSITIVE	CK_TRUE
CKA_EXTRACTABLE	CK_TRUE

Table 11. Secret key (CKO_SECRET_KEY) default attributes of the EP11 token

Secret key attributes	**value**
CKA_EXTRACTABLE	CK_TRUE

- When you create keys the default values of the attributes CKA_ENCRYPT, CKA DECRYPT, CKA_VERIFY, CKA_SIGN, CKA_WRAP and CKA_UNWRAP are CK_TRUE. Note, no EP11 mechanism supports the Sign/Recover or Verify/Recover functions.

  Even if settings of `CKA_SENSITIVE`, `CKA_EXTRACTABLE`, or `CKA_NEVER_EXTRACTABLE` would allow accessing the key value, then openCryptoki returns `00..00` as key value (due to the secure key concept).

  For information about the key attributes, see the PKCS #11 Cryptographic Token Interface Standard.
- All RSA keys must have a public exponent (CKA_PUBLIC_EXPONENT) greater than or equal to 17.
- The Crypto Express EP11 coprocessor restricts RSA keys (primes and moduli) according to ANSI X9.31. Therefore, in the EP11 token, the lengths of the RSA primes (p or q) must be a multiple of 128 bits. Also, the length of the modulus (`CKA_MODULUS_BITS`) must be a multiple of 256.
- The mechanisms `CKM_DES3_CBC` and `CKM_AES_CBC` can only wrap keys, which have a length that is a multiple of the block size of DES3 or AES respectively. See the mechanism list and mechanism information (**`pkcsconf -m`**) for supported mechanisms together with supported functions and key sizes.
- The EP11 coprocessor adapter can be configured to restrict the cryptographic capabilities in order for the adapter to comply with specific security requirements and regulations. Such restrictions on the adapter impact the capability of the EP11 token.
- The PKCS #11 function `C_DigestKey()` is not supported by the EP11 library.
- Pure EP11 key objects can be extracted from sensitive openCryptoki key objects for the CCA token by accessing the value of the CKA_IBM_OPAQUE attribute value.

Restriction to extended evaluations

For openCryptoki versions up to 3.8, the EP11 token only supported those functions and mechanisms that are available on an adapter that is configured to comply to the extended evaluations. These extended evaluations meet public sector requirements with regard to both FIPS and Common Criteria certifications.

For more details, see the *IBM z14 Technical Guide*.

Starting with the current version of the EP11 enablement, you can control the use of certain mechanisms within a domain of an EP11 cryptographic coprocessor by configuring this coprocessor by means of access control points (ACPs). So except for one restriction, the use of mechanisms is no longer restricted to the limitations imposed by the extended evaluations.

Read the information about filter mechanisms in

```
Exploiting Enterprise PKCS #11 using openCryptoki
```

for information on how to manage the access to PKCS #11 mechanisms using ACPs.
The available mechanisms and their attributes are then reflected by the openCryptoki functions `C_GetMechanismList` and `C_GetMechanismInfo`. However, there is one restriction

on RSA mechanisms that cannot be reflected in the result of `C_GetMechanismInfo`: The `CKA_PUBLIC_EXPONENT` must have a value of at least 17.

Migrating master keys - `pkcsep11_migrate` utility

There may be situations when the master key on (a domain of) a Crypto Express EP11 coprocessor (CEX*P) must be changed, for example, if company policies require periodic changes of all master keys. Simply changing the master keys using the TKE results in all secure keys stored in the EP11 token to become useless. Therefore all data encrypted by these keys are lost. To avoid this situation, you must accomplish a master key migration process, where activities on the TKE and on the Linux system must be interlocked.

All secret and private keys are secure keys, that means they are enciphered (wrapped) with the master key (*MK*) of the CEX*P adapter domain. Therefore, the master key is often also referred to as wrapping key. If master keys are changed in a domain of a CEX*P adapter, all key objects for secure keys in the EP11 token object repository become invalid. Therefore, all key objects for secure keys must be re-enciphered with the new *MK*. In order to re-encipher secure keys that are stored as EP11 key objects in the EP11 token object repository, openCryptoki provides the master key migration tool **`pkcsep11_migrate`**.

How to access the master key migration tool

The *pkcsep11_migrate* key migration utility is part of openCryptoki versions 3.1 or later, which include the EP11 support.

Prerequisites for the master key migration process

The master key migration process for the EP11 token requires a TKE version 7.3 environment. How to set up this environment is described in topic *Setting up the TKE environment* of *Exploiting Enterprise PKCS #11 using openCryptoki*.

To use the *pkcsep11_migrate* migration tool, the EP11 crypto stack including openCryptoki must be installed and configured. For information on how to set up this environment, read Exploiting Enterprise PKCS #11 using openCryptoki.

The master key migration process

Prerequisite for re-encipherment: The EP11 token may be configured to use more than one adapter/domain pair to perform its cryptographic operations. This is defined in the EP11 token configuration file. If the EP11 token is configured to use more than one adapter/domain pair, then all adapter/domain pairs must be configured to each have the same set of master keys. Therefore, if a master key on one of these adapter/domain pairs is changed, it must be changed on all those other adapter/domain pairs, too.

To migrate master keys on the set of adapter/domain pairs used by an EP11 token, you must perform the following steps:

1. On the TKE workstation (TKE), submit and commit the same new master key on all CEX*P adapter/domain combinations used by the EP11 token.
2. On Linux, stop all processes that are currently using openCryptoki with the EP11 token.
3. On Linux, back up the token object repository of the EP11 token. For example, you can use the following commands:

```
cd /var/lib/opencryptoki/ep11
tar -cvzf ~/ep11TOK_OBJ_backup.tgz TOK_OBJ
```

4. On Linux, migrate the keys of the EP11 token object repository with the *pkcsep11_migrate* migration tool (see the invocation information provided at the end of these process steps). The *pkcsep11_migrate* tool must only be called once for one of the adapter/domain pairs that the EP11 token uses. If a failure occurs, restore the backed-up token repository and try this step again.

Attention: Do not continue with step "5" on page 71 unless step "4" on page 70 was successful. Otherwise you will lose your encrypted data.

5. On the TKE, activate the new master keys on all EP11 adapter/domain combinations that the EP11 token uses.
6. On Linux, restart the applications that used openCryptoki with the EP11 token.

In step "1" on page 70 of the master key migration process, the new master key must be submitted and committed via the TKE interface. That means the *new EP11 master key* must be in the state `Full Committed`. The current MK is in the state `Valid`. Now both (current and new) *EP11 master keys* are available and accessible. The utility can now decrypt all relevant key objects within the token and re-encrypt all these key objects with the new master key.

Note: All the decrypt and encrypt operations are done inside the EP11 cryptographic coprocessor, that means that at no time clear key values are visible within memory.

Invocation:

```
pkcsep11_migrate -slot <number> -adapter <number> -domain <number>
```

The following parameters are mandatory:

-slot
- slot number for the EP11 token

-adapter
- the card ID; can be retrieved form the card ID in the *sysfs* (to be retrieved from `/sys/devices/ap/cardxx`, or with **`lszcrypt`**.

-domain
- the decimal card domain number (to be retrieved from `/sys/bus/ap/ap_domain` or with **`lszcrypt -b`**)

All token objects representing secret or private keys that are found for the EP11 token, are re-encrypted.

Note: The adapter and domain numbers can be specified in decimal, octal (with prefix 0), or hexadecimal (with prefix 0x) notation. The **`lszcrypt`** utility displays these fields in hexadecimal values.

Usage: You are prompted for your user PIN.

Examples:

```
pkcsep11_migrate  -slot 2 -adapter 8 -domain 48
pkcsep11_migrate  -slot 0x2 -adapter 010 -domain 0x30
```

Both invocations migrate the master key for the cryptographic coprocessor 8 (octal 010) and domain 48 (hex 0x30) used by the EP11 token from slot 2.

Note: The program stops if the re-encryption of a token object fails. In this case, restore the back-up.

After this utility re-enciphered all key objects, the new master key must be activated. This activation must be done by using the TKE interface command **`Set, immediate`**. Finally, the new master key becomes the current master key and the previous master key must be deleted.

Note: This tool is installed in the users `sbin` path and therefore callable from everywhere.

To prevent token object generation during re-encryption, openCryptoki with the EP11 token must not be running during re-encryption. It is recommended to make a back-up of the EP11 token object directory (`/var/lib/opencryptoki/ep11tok/TOK_OBJ`).

Managing EP11 sessions - pkcsep11_session utility

An EP11 session is a state on the EP11 cryptographic coprocessor and must not be confused with a PKCS #11 session. An EP11 session is generated by the *strict session mode* or the *VHSM mode*. They are implicitly stored and deleted by openCryptoki if the according modes are set. So under normal circumstances, you need not care about the management of these EP11 sessions. But in some cases, for example, when programs crash or when programs do not close their sessions or do not call `C_Finalize` before exiting, some explicit EP11 session management may be required.

EP11 sessions are a limited resources shared by all domains of an EP11 cryptographic coprocessor and must be closed when no longer needed to avoid situations where the EP11 cryptographic coprocessor runs out of session resources. Therefore the usage of the *strict session mode* and the *VHSM mode* is only recommended in environments where all systems accessing EP11 cryptographic coprocessors can be trusted to cooperate to not open or keep open EP11 sessions unnecessarily.

For more information about EP11 sessions, read *Exploiting Enterprise PKCS #11 using openCryptoki*.

The **pkcsep11_session** tool allows to delete an EP11 session from the EP11 cryptographic coprocessors left over by programs that did not terminate normally. An EP11 cryptographic coprocessor supports only a certain number of EP11 sessions at a time. Because of this, it is important to delete any EP11 session, in particular when the program for which it was logged in, terminated unexpectedly. The **pkcsep11_session** tool is also used to set the VHSM-PIN required for the *VHSM mode*.

pkcsep11_session syntax

To see all available sub commands of the **pkcsep11_session** utility, request help with the `pkcsep11_session -h` or `pkcsep11_session --help` command:

```
# pkcsep11_session -h

usage:  pkcsep11_session show|logout|vhsmpin|status [-date <yyyy/mm/dd>] [-pid <pid>] [-id <sess-
id>] [-slot <num>] [-force] [-h]
```

pkcsep11_session sub-command usage examples

- **show:** Show all left over sessions:

  ```
  pkcsep11_session show
  ```

  A sample output for two left-over EP11 sessions could look as shown:

  ```
  # pkcsep11_session show -slot 4
  Using slot #4...

  Enter the USER PIN:
  List of EP11 sessions:

  30D5457762D8DDC158B558FCCC79FAB6:
          Pid:    48196
          Date:   2018/ 7/12
  30D5457762D8DDC158B558FCCC79FAB6:
          Pid:    48196
          Date:   2018/ 7/12

  2 EP11-Sessions displayed
  ```

  Note that only the first 16 bytes of the EP11 session ID are stored in the session object and therefore, the session IDs are displayed only partially. Otherwise, a user would be able to re-login on an EP11 adapter and re-use keys generated with this EP11 session, when the full EP11 session ID would be visible to the outside. Thus there may be identical session IDs when the *strict session mode* and the *virtual HSM (VHSM) mode* are combined for a session, as shown in the example.

- **show:** Show all left over EP11 sessions that belong to a specific process id (pid):

  ```
  pkcsep11_session show -pid 1234
  ```

- **show:** Show all left over EP11 sessions that have been created before a specific date:

  ```
  pkcsep11_session show -date 2018/06/29
  ```

- **logout:** Logout all left over EP11 session:

```
pkcsep11_session logout
```

- **logout:** Logout all left over EP11 session that belong to a specific process id (pid):

```
pkcsep11_session logout -pid 1234
```

- **logout:** Logout all left over EP11 session that have been created before a specific date:

```
pkcsep11_session logout -date 2018/07/27
```

- **logout:** Logout all left over EP11 session even when the logout does not succeed on all adapters:

```
pkcsep11_session logout -force
```

- **vhsmpin:** Every EP11 token running in virtual HSM mode (VHSM_MODE configuration option) requires a VHSM-PIN. The **vhsmpin** subcommand sets this VHSM-PIN used for the virtual HSM mode (VHSM_MODE).

  In VHSM mode, all keys generated by the EP11 token strictly belong to the EP11 token instance that created it.

  Note: When changing the VHSM-PIN, all existing keys stored as token objects become unusable.

  See also “Defining an EP11 token-specific configuration file” on page 57 or read topic *Controlling access to cryptographic objects* in *Exploiting Enterprise PKCS #11 using openCryptoki*.

```
# pkcsep11_session vhsmpin -slot 4

Using slot #4...

Enter the USER PIN:
```

  The VHSM-PIN must contain between 8 and 16 alphanumeric characters.

- **status:** Query the maximum and currently available number of EP11 sessions for each available EP11 APQN. :

```
pkcsep11_session status
```

  See the following sample output:

```
APQN 0b.0024:
  Max Sessions:        1024
  Available Sessions:  234
APQN 0a.0024:
  Max Sessions:        1024
  Available Sessions:  0
```

The **pkcsep11_session** tool provides its own man page that is installed as part of the EP11 package.

Chapter 13. Soft token

The Soft token is often used for test purposes before you let your application access one of the other available tokens in openCryptoki. View a list of PKCS #11 mechanisms supported by the Soft token.

As a prerequisite for an operational Soft token, the OpenSSL library called `libcrypto` must be installed (see Figure 3 on page 13).

With OpenSSL 3.0 there is no way for a Soft token to obtain the intermediate digest state from a digest operation, which was possible with earlier versions of OpenSSL. This leads to the fact that for operations that involve digests, function `C_GetOperationState()` returns CKR_STATE_UNSAVEABLE when built against OpenSSL 3.0. This affects digest operations using `C_DigestInit()`, `C_DigestUpdate()`, and `C_DigestFinal()`, but also sign and verify operations with mechanisms involving digests. This is explicitly allowed by the PKCS #11 standard for function `C_GetOperationState()`: *An attempt to save the cryptographic operations state of a session which is performing an appropriate cryptographic operation (or two), but which cannot be satisfied, for example, because certain necessary state or key information cannot leave the token, should fail with the error CKR_STATE_UNSAVEABLE.*

PKCS #11 mechanisms supported by the Soft token

View a list of mechanisms provided by PKCS #11 which you can use to exploit the openCryptoki features for the Soft token from within your application.

Use the **pkcsconf** command with the shown parameters to retrieve a complete list of algorithms (or mechanisms) that are supported by the Soft token:

```
$ pkcsconf -m -c <slot>

Mechanism #0
        Mechanism: 0x0 (CKM_RSA_PKCS_KEY_PAIR_GEN)
        Key Size: 512-4096
        Flags: 0x10000 (CKF_GENERATE_KEY_PAIR)
Mechanism #1
        Mechanism: 0x120 (CKM_DES_KEY_GEN)
        Key Size: 8-8
        Flags: 0x8000 (CKF_GENERATE)
Mechanism #2
        Mechanism: 0x131 (CKM_DES3_KEY_GEN)
        Key Size: 24-24
        Flags: 0x8000 (CKF_GENERATE)
…
…
```

The command output shown in Table 12 on page 75 displays all mechanisms that are supported by the Soft token. Each mechanism provides its supported key size and some further properties such as hardware support and mechanism information flags. These flags provide information about the PKCS #11 functions that may use the mechanism. In some cases, the flags also provide further attributes that describe the supported variants of the mechanism. Typical functions are for example, *encrypt*, *decrypt*, *wrap key*, *unwrap key*, *sign*, or *verify*.

The **`pkcsconf -m -c <slot>`** command output corresponds to the list shown in "PKCS #11 mechanisms supported by the Soft token" on page 75.

Table 12. PKCS #11 mechanisms supported by the Soft token

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_RSA_PKCS_KEY_PAIR_GEN	512-4096 bits	GENERATE KEY PAIR	before 3.16
CKM_DES_KEY_GEN	8-8 bytes	GENERATE	before 3.16

Table 12. PKCS #11 mechanisms supported by the Soft token (continued)

Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_DES3_KEY_GEN	24-24 bytes	GENERATE	before 3.16
CKM_RSA_PKCS	512-4096 bits	ENCRYPT, DECRYPT, SIGN, SIGN RECOVER, VERIFY, VERIFY RECOVER, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA224_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	3.16
CKM_SHA256_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	3.16
CKM_SHA384_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	3.16
CKM_SHA512_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	3.16
CKM_SHA224_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	3.16
CKM_SHA256_RSA_PKCS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA384_RSA_PKCS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA512_RSA_PKCS	512-4096 bits	SIGN, VERIFY	3.16
CKM_RSA_X_509	512-4096 bits	ENCRYPT, DECRYPT, SIGN, SIGN_RECOVER, VERIFY, VERIFY_RECOVER, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_OAEP	512-4096 bits	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_MD5_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_DH_PKCS_DERIVE	512-2048 bits	DERIVE	before 3.16
CKM_DH_PKCS_KEY_PAIR_GEN	512-2048 bits	GENERATE KEY PAIR	before 3.16
CKM_DES_ECB	8-8 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES_CBC	8-8 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES_CBC_PAD	8-8 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_ECB	24-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_CBC	24-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16

Table 12. PKCS #11 mechanisms supported by the Soft token (continued)

Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_DES3_CBC_PAD	24-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_CMAC	16-24 bytes	SIGN, VERIFY	before 3.16
CKM_DES3_CMAC_GENERAL	16-24 bytes	SIGN, VERIFY	before 3.16
CKM_SHA_1	n/a	DIGEST	before 3.16
CKM_SHA_1_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA_1_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA224	n/a	DIGEST	before 3.16
CKM_SHA224_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA224_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA256	n/a	DIGEST	before 3.16
CKM_SHA256_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA256_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA384	n/a	DIGEST	before 3.16
CKM_SHA384_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA384_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512	n/a	DIGEST	before 3.16
CKM_SHA512_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_224	n/a	DIGEST	before 3.16
CKM_SHA512_224_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_224_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_256	n/a	DIGEST	before 3.16
CKM_SHA512_256_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_256_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_224	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_256	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_384	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_512	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_224_HMAC	112-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_256_HMAC	128-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_384_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_512_HMAC	256-256 bytes	SIGN, VERIFY	before 3.16

Table 12. PKCS #11 mechanisms supported by the Soft token (continued)

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_MD5	n/a	DIGEST	before 3.16
CKM_MD5_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_MD5_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SSL3_PRE_MASTER_KEY_GEN	48-48 bytes	GENERATE	before 3.16
CKM_SSL3_MASTER_KEY_DERIVE	48-48 bytes	DERIVE	before 3.16
CKM_SSL3_KEY_AND_MAC_DERIVE	48-48 bytes	DERIVE	before 3.16
CKM_DES3_MAC	16-24 bytes	HW, SIGN, VERIFY	before 3.16
CKM_DES3_MAC_GENERAL	16-24 bytes	HW, SIGN, VERIFY	before 3.16
CKM_AES_MAC	16-24 bytes	HW, SIGN, VERIFY	before 3.16
CKM_AES_MAC_GENERAL	16-24 bytes	HW, SIGN, VERIFY	before 3.16
CKM_SSL3_MD5_MAC	384-384 bits	SIGN, VERIFY	before 3.16
CKM_SSL3_SHA1_MAC	384-384 bits	SIGN, VERIFY	before 3.16
CKM_AES_CTR	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_OFB	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_CFB8	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_CFB128	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_GCM	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_DES_OFB64	24-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_DES_CFB8	24-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_DES_CFB64	24-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_KEY_GEN	16-32 bytes	GENERATE	before 3.16
CKM_AES_ECB	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CBC	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CBC_PAD	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CMAC	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_AES_CMAC_GENERAL	16-32 bytes	SIGN, VERIFY	before 3.16

Table 12. PKCS #11 mechanisms supported by the Soft token (continued)

Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	GENERATE	before 3.16
CKM_ECDSA_KEY_PAIR_GEN	160-521 bits	GENERATE_KEY_PAIR, EC_F_P, EC_OID	before 3.16
CKM_ECDSA	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA1	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA224	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA256	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA384	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA512	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDH1_DERIVE	160-521 bits	DERIVE, EC_F_P, EC_OID	before 3.16

For a description of mechanisms with a name pattern of `CKM_IBM_...` refer to "IBM-specific mechanisms" on page 87.

For explanation about the key object properties see the PKCS #11 Cryptographic Token Interface Standard.

Chapter 14. Directory content for CCA, ICA, EP11, and Soft tokens

Each token uses a unique token directory. This token directory stores the token-specific objects (like for example, key objects, user PIN, SO PIN, or hashes). Thus, the information for a certain token is separated from all other tokens. Read the information about the format of data and objects stored in these directories. openCryptoki users may need this information for example, when working with containerized applications.

As of openCryptoki version 3.16, CCA, ICA, EP11, and Soft tokens have the same structure and content within their directories.

The path name of the token directories is derived from either the token default name or by the token name as defined in the `opencryptoki.conf` file. The location of the token directories may look similar to the shown examples:

```
[root@t3545033 opencryptoki]# pwd
/var/lib/opencryptoki
[root@t3545033 opencryptoki]# ls -l
total 24
drwxrwx---. 3 root pkcs11 4096 Mar 19 11:47 ccatok
drwxrwx---. 3 root pkcs11 4096 Mar 19 12:35 ep11tok
drwxrwx---. 3 root pkcs11 4096 Mar 19 11:47 lite     /* legacy name of ICA Tok */
drwxrwx---. 3 root pkcs11 4096 Mar 19 11:47 swtok
```

Note: With openCryptoki, you can select from two data store formats:

- Before openCryptoki version 3.12, there is only the one NVTOK.DAT format (the old format).
- Starting with openCryptoki version 3.12, you can choose to use a FIPS compliant data format. Being FIPS compliant, the token data is stored in a format that is better protected against attacks than the previously used data format.

If you want to use the token data format that was generated with FIPS compliant operations, you must explicitly specify the `tokversion` option for the token's slot entry in the openCryptoki configuration file. You must do this before token initialization with the **`pkcsconf`** command, for example:

```
slot 4
{
stdll = libpkcs11_ep11.so
confname = ep11tok01.conf
tokname = ep11token01
tokversion = 3.12
description = "Ep11 Token"
manufacturer = "IBM"
hwversion = "4.11"
firmwareversion = "2.0"
}
```

Figure 14. Slot entry for an EP11 token with FIPS compliant data format in the opencryptoki.conf file

You can use the **`pkcstok_migrate`** utility to transform an EP11 token, a CCA token, an ICA token, or a Soft token created with any version of openCryptoki into a data format that was generated by FIPS compliant operations. For more information, read Chapter 8, "Migrating to FIPS compliance - pkcstok_migrate utility," on page 37.

There are two objects derived from both a token's user PIN and a token's SO PIN:

- a non-secret password hash which is checked at login
- and a secret key encryption key (KEK) which is used to wrap the token's master key (SO KEK or a user KEK, respectively).

Therefore, the token's master key is stored on disk twice:

- The MK_SO file holds the master key wrapped with the SO KEK.
- The MK_USER file holds the master key wrapped with the user KEK.

So both user and SO can access the master key. The password-based key derivation function PBKDF2 is used to derive those four objects. An iteration count of 100000 is used with a salt consisting of different *purpose strings* for each of the four derivations and a random part. Iteration count, purpose strings, and the random parts are non-secret and can be stored on disk with the other non-volatile token data in NVTOK.DAT, together with the corresponding data. The two password hashes are non-secret and can also be stored unencrypted in NVTOK.DAT.

The two key-encryption keys (KEKs) are secret and cannot be stored with a token's non-volatile token data: They must be (re-)derived when needed and are cached on the stack for an application's lifetime.

All following structures are C pseudo code, used to describe data structures at byte level.

The old NVTOK.DAT format (still valid)

```
struct TOKEN_DATA {
    CK_TOKEN_INFO info;
    u8 user_pin_sha1 [24];
    u8 so_pin_sha1 [24];
    u8 next_token_obj_name[8];
    u32 allow_weak_des;
    u32 check_des_parity;
    u32 allow_key_mods;
    u32 netscape_mods;
};
```

The new NVTOK.DAT format

Note: The new NVTOK.DAT is available starting with openCryptoki version 3.12 and is valid in parallel with the old format.

```
struct TOKEN_DATA {
    /* --- old format for compat --- */
    CK_TOKEN_INFO info;
    u8 user_pin_sha1 [24];
    u8 so_pin_sha1 [24];
    u8 next_token_obj_name[8];
    u32 allow_weak_des;
    u32 check_des_parity;
    u32 allow_key_mods;
    u32 netscape_mods;

    /* --- 3.12 additions start here --- */

    u32 version; /* tokversion major<<16|minor */

    /* --- PBKDF2 ---
     * 64b salts are 32b purpose string concat 32b random */
    /* SO PW hash (login) */
    u64 so_login_it;
    u8 so_login_salt[64];
    u8 so_login_key[32];
    /* User PW hash (login) */
    u64 user_login_it;
    u8 user_login_salt[64];
    u8 user_login_key[32];
    /* SO MK KEK (wrap) */
    u64 so_wrap_it;
    u8 so_wrap_salt[64];
    /* User MK KEK (wrap) */
    u64 user_wrap_it;
    u8 user_wrap_salt[64];
};
```

Changes from old to new data store format

In the old format, the multiple-byte-width data types were not serialized before stored to disk, so NVTOK_DAT could not be used on little endian (LE) and big endian (BE) platforms. The new format serializes those data types from host to BE byte order. The *AES Key Wrap* algorithm is used to wrap a token's master key (MK) with the KEKs. The token's master key is randomly generated at token initialization. In the old format, the master key was a 3DES CBC key. The KEK was a 2TDEA CBC key. CBC was used with a per token (fixed) initialization vector (IV). Integrity was intended to be provided by a SHA1 hash of the unencrypted key. The MK master key object had the following format:

```
struct MK {
    u8 MK [24];
    u8 sha1 [20];
    u8 padding[4];
};
```

It was wrapped by the SO KEK and stored to MK_SO and was also wrapped by the user KEK and stored to MK_USER. The new format defines the MK to be an AES-256 key. Its unencrypted format is just the 32 key bytes. Its encrypted format is a 40 byte key blob output by the *AES Key Wrap* algorithm (wrapped with the SO or user KEK). The file names MK_SO and MK_USER are unchanged.

The old format stored non-private token objects unencrypted in the following format:

```
struct OBJECT_PUB {
    u32 total_len;
    u8 private_flag;
    u8 object[object_len];
};
```

The new format is:

```
struct OBJECT_PUB {
    //--------------        <--+
    u32 tokversion;            | 16-byte header
    u8 private_flag;           |
    u8 reserved[7];            |
    u32 object_len;            |
    //--------------        <--+
    u8 object[object_len];     | body
    //--------------        <--+
};
```

The old format encrypted all private token objects under the MK. The unencrypted format was:

```
struct OBJECT_PRIV {
    u32 total_len;
    u8 private_flag;
    //--- enc ---
    u32 object_len;
    u8 object[object_len];
    u8 sha1[20];
    u8 padding[padding_len];
};
```

The new format features authenticated encryption via AES-256-GCM. To avoid initialization vector (IV) uniqueness , instead of using the MK to encrypt all token objects, the MK is used to wrap a per-object key using the *AES Key Wrap* algorithm. The wrapped per-object key is stored together with the IV and other meta-data as additional authenticated data (AAD) in the authenticated object header. The 16 byte GMAC tag is appended to the authenticated and encrypted object body which holds the private token object's data:

```
struct OBJECT_PRIV {
    u32 total_len;
    //- auth -------        <--+
    u32 tokversion;            | 64-byte header
    u8 private_flag;           |
    u8 reserved[3];            |
    u8 key_wrapped[40];        |
    u8 iv[12];                 |
    u32 object_len;            |
    //- auth+enc ---        <--+
    u8 object[object_len];     | body
    //--------------        <--+
    u8 tag[16];                | 16-byte footer
    //--------------        <--+
};
```

In the old format, the multiple-byte-width data types were not serialized before stored to disk, so the token objects could not be moved between LE and BE platforms. The new format serializes all header and footer data from host to BE byte order, so all object data can always be encrypted and decrypted. However, the decrypted object data itself (body) must still be interpreted in host byte order.

So the new format should only be used with fresh setups. The new format can be used by specifying the new **`tokversion`** keyword in the token's slot configuration in `opencryptoki.conf`. For a value of equal or grater 3.12 the new format is used.

Here is a table of all key material that is used per version 3.12 token:

Table 13.

NAME	TYPE	GENERATION	USAGE	STORAGE
MK	256-bit AES	random	wrap tok.obj keys (AES-KW)	wrapped on disk (MK_SO,MK_USER)
SO KEK	256-bit AES	derived from SO PIN (PBKDF2)	wrap MK (AES-KW)	cached on stack/heap
User KEK	256-bit AES	derived from User PIN (PBKDF2)	wrap MK (AES-KW)	cached on stack/heap
SO PW hash	256-bit	derived from SO PIN (PBKDF2)	SO login	on disk (NVTOK.DAT)
User PW hash	256-bit	derived from User PIN (PBKDF2)	User login	on disk (NVTOK.DAT)
Tok.obj keys	256-bit AES	random	auth.enc of tok.obj data	wrapped on disk (in tok.objs)

Chapter 15. ICSF token

ICSF (Integrated Cryptographic Service Facility) is a software element of IBM z/OS. The ICSF token is a clear-key, remote cryptographic token. The actual operations are performed remotely on a server and all the PKCS #11 key objects are stored remotely on the server. Adding an ICSF token requires a running remote z/OS instance.

PKCS #11 mechanisms supported by the ICSF token are not documented in this edition. Instead, read the information about the purpose and use of the **pkcsicsf** tool.

For more information about PKCS #11 and ICSF, refer to *Cryptographic Services ICSF: Writing PKCS #11 Applications*.

Configuring the ICSF token - pkcsicsf utility

Use the **pkcsicsf** utility to add an ICSF token to openCryptoki or to list available ICSF tokens.

Adding an ICSF token to openCryptoki creates an entry in the opencryptoki.conf file for this token. It also creates a token_name.conf configuration file in the same directory as the opencryptoki.conf file, containing ICSF specific information. This information is read by the ICSF token.

The ICSF token must bind and authenticate to an LDAP server. Several SASL authentication mechanisms (Simple Authentication and Security Layer mechanisms) are supported. You must specify one of these mechanisms when listing the available ICSF tokens or when adding an ICSF token. openCryptoki currently supports adding only one ICSF token.

openCryptoki administrators can either allow the LDAP calls to utilize existing LDAP configurations, such as ldap.conf or .ldaprc for bind and authentication information. Or they can set the bind and authentication information within openCryptoki by using this utility and its options. The information is placed in the token_name.conf file to be used in the LDAP calls. When using simple authentication, the user is prompted for the RACF password when listing or adding a token.

```
pkcsicsf  [-h]  [-l|-a token name] [-b BINDDN] [-c client-cert-file] [-C CA-cert-file]
          [-k privatekey] [-m mechanism] [-u URI]
```

Options

-a token_name
adds the specified ICSF token to openCryptoki.

-b bind_name
specifies the distinguished name to bind when using simple authentication.

-c client_cert_file
specifies the client certification file when using SASL authentication.

-C CA_cert_file
specifies the certificate authority (CA) certification file when using SASL authentication.

-k private_key
specifies the client private key file when using SASL authentication.

-m auth_mechanism
specifies the authentication mechanism to use when binding to the LDAP server. Specify either simple or sasl).

-l
lists available ICSF tokens.

-h
shows usage information for this utility.

Chapter 16. IBM-specific mechanisms and features for openCryptoki

Numerous IBM-specific mechanisms and features are available across multiple token-types or for certain tokens only. Scan the content of this topic to find the appropriate information.

IBM-specific mechanisms

In your openCryptoki applications, you can apply IBM-specific mechanisms for special purposes as offered by an exploited token. Information about which token or tokens support a given mechanism is included in the description of each mechanism.

Table 14. IBM-specific mechanisms

Mechanism	Reference
CKM_IBM_DILITHIUM	"CKM_IBM_DILITHIUM" on page 87
CKM_IBM_SHA3_224	"CKM_IBM_SHA3_nnn" on page 90
CKM_IBM_SHA3_256	"CKM_IBM_SHA3_nnn" on page 90
CKM_IBM_SHA3_384	"CKM_IBM_SHA3_nnn" on page 90
CKM_IBM_SHA3_512	"CKM_IBM_SHA3_nnn" on page 90
CKM_IBM_SHA3_224_HMAC	"CKM_IBM_SHA3_nnn_HMAC" on page 91
CKM_IBM_SHA3_256_HMAC	"CKM_IBM_SHA3_nnn_HMAC" on page 91
CKM_IBM_SHA3_384_HMAC	"CKM_IBM_SHA3_nnn_HMAC" on page 91
CKM_IBM_SHA3_512_HMAC	"CKM_IBM_SHA3_nnn_HMAC" on page 91
CKM_IBM_CMAC	"CKM_IBM_CMAC" on page 92
CKM_IBM_EC_C448 (synonym: CKM_IBM_EC_X448)	"Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA" on page 92
CKM_IBM_EC_C25519 (synonym: CKM_IBM_EC_X25519)	"Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA" on page 92
CKM_IBM_ED448_SHA3	"Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA" on page 92
CKM_IBM_ED25519_SHA512 (synonym: CKM_IBM_EDDSA_SHA512)	"Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA" on page 92
CKM_IBM_ATTRIBUTEBOUND_WRAP	"CKM_IBM_ATTRIBUTEBOUND_WRAP" on page 94

CKM_IBM_DILITHIUM

Availability:

The CKM_IBM_DILITHIUM mechanism is available with the EP11 token.

Description:

Dilithium is a post-quantum signature scheme. That is, it can generate signatures that resist attacks from quantum computers, but also from classical computers. *Dilithium* is one of the candidate algorithms submitted to the NIST post-quantum cryptography project. At the time of writing, the NIST selection process was in round 2, so all keys created with the CKM_IBM_DILITHIUM mechanism are internally flagged as `round2`.

Because Dilithium keys can only sign or verify, the EP11 token only provides one single mechanism for all three operations: key generation, sign, and verify.

With the EP11 token, you can also import and export Dilithium keys by wrapping or unwrapping them using AES or TDES key encrypting keys (KEKs). That is, you can protect Dilithium keys that are sent to another system, received from another system, or stored with data in a file. Use any mechanism to wrap or unwrap IBM Dilithium keys that is convenient for your purposes, for example:

- CKM_AES_CBC_PAD
- CKM_DES3_CBC_PAD

On the TKE workstation, you must enable Dilithium processing by setting domain (access) control point 65 on the used cryptographic coprocessors :

```
65    XCP_CPB_ALG_PQC_DILITHIUM           enable support for Dilithium algorithm
```

Prerequisites IBM Dilithium has the following prerequisites:

- EP11 library 3.0 or later. Dilithium support in the EP11 host library has oid = 1.3.6.1.4.1.2.267.1.6.5.
- A Crypto Express7S feature or later, configured as Crypto Express EP11 coprocessor (CEX7P) with firmware level 4.7.15 or later

Key type The IBM-specific key type for an IBM Dilithium key object is:

```
#define CKK_IBM_PQC_DILITHIUM (CKK_VENDOR_DEFINED +0x10023)
```

Key form The EP11 host library uses the following key form for IBM Dilithium public and private keys:

- **Public key**

```
DilithiumPublicKey ::= BIT STRING {
   SEQUENCE {
      rho BIT STRING, [2]  -- nonce
      t1 BIT STRING [3]    -- from vector(L)
   }
}
```

- **Private key**

```
DilithiumPrivateKey ::= SEQUENCE {
   version INTEGER,   -- v0, reserved 0
   rho BIT STRING,    -- nonce
   key BIT STRING,    -- key/seed/D
   tr BIT STRING,     -- PRF bytes ('CRH' in spec)
   s1 BIT STRING,     -- vector(L)
   s2 BIT STRING,     -- vector(K)
   t0 BIT STRING,     -- low bits(vector L)
   t1 [0] IMPLICIT OPTIONAL {
      t1 BIT STRING,  -- high bits(vector L)
                      -- see also public key
   }
}
```

Attributes:

Table 15. Attributes for IBM Dilithium key

Attribute	Data type	Meaning
CKA_IBM_DILITHIUM_KEYFORM	CK_ULONG	The Dilithium key form, currently only *round 2*.
CKA_IBM_DILITHIUM_RHO	CK_CHAR[32]	The private rho key component
CKA_IBM_DILITHIUM_SEED	CK_CHAR[32]	The private seed key component
CKA_IBM_DILITHIUM_TR	CK_CHAR[48]	The private tr key component
CKA_IBM_DILITHIUM_S1	CK_CHAR[480]	The private s1 key component
CKA_IBM_DILITHIUM_S2	CK_CHAR[576]	The private s2 key component
CKA_IBM_DILITHIUM_T0	CK_CHAR[2688]	The private t0 key component
CKA_IBM_DILITHIUM_T1	CK_CHAR[1728]	The public t1 key component

A Dilithium key also contains the usual standard attributes for public and private key objects, such as CKA_CLASS, CKA_KEYTYPE. See tables *Common Key Attributes*, *Common Private Key Attributes*, and *Common Public Key Attributes* in

```
PKCS #11 Cryptographic Token Interface Base Specification Version 3.0.
```

All created keys are internally flagged as `round2` keys via the CKA_IBM_DILITHIUM_KEYFORM attribute. All attributes, except CKA_IBM_DILITHIUM_KEYFORM, are bit strings. The CKA_IBM_DILITHIUM_KEYFORM attribute is a CK_ULONG, indicating the actual key form. The currently only supported key form is:

```
#define IBM_DILITHIUM_KEYFORM_ROUND2              1
```

IBM Dilithium signatures have a length of 3366 bytes.

Functions for IBM Dilithium key creation and unwrapping:

Use function `C_GenerateKeyPair()` to create an IBM Dilithium key object consisting of a public/private key pair. Note that the CKM_IBM_DILITHIUM mechanism has no mechanism-specific parameters. The following is a sample for key object creation:

```
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE public_template[] = {
    {CKA_VERIFY, &true, sizeof(true)},
};
CK_ATTRIBUTE private_template[] = {
    {CKA_SIGN, &true, sizeof(true)},
};

CK_MECHANISM mech;
CK_SESSION_HANDLE session;
CK_OBJECT_HANDLE publ_key = CK_INVALID_HANDLE, priv_key = CK_INVALID_HANDLE;
...
mech.mechanism = CKM_IBM_DILITHIUM;
mech.ulParameterLen = 0;
mech.pParameter = NULL;
CK_RV rv;

rv = C_GenerateKeyPair(session, &mech,
                   public_template, 1,
                   private_template, 1,
                   &publ_key, &priv_key);
if (rv == CKR_OK) {
...
}
```

Use functions `C_WrapKey()` and `C_UnwrapKey()` if you need to transport IBM Dilithium keys. See the following sample of an unwrapping operation:

```
CK_MECHANISM wrap_mech;
CK_OBJECT_HANDLE secret_key = CK_INVALID_HANDLE;
CK_BYTE_PTR wrapped_key = NULL;
CK_ULONG wrapped_keylen;
CK_OBJECT_HANDLE unwrapped_key = CK_INVALID_HANDLE;

CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE key_type = CKK_IBM_PQC_DILITHIUM;
CK_OBJECT_HANDLE priv_key = CK_INVALID_HANDLE;
CK_BYTE unwrap_label[] = "unwrapped_private_Dilithium_Key";
CK_BBOOL true = TRUE;
CK_RV rv;

CK_ATTRIBUTE unwrap_tmpl[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &key_type, sizeof(key_type)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, &unwrap_label, sizeof(unwrap_label)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
};

wrap_mech.mechanism = CKM_AES_CBC_PAD;
wrap_mech.pParameter = "0123456789abcdef";
wrap_mech.ulParameterLen = 16;

/* Generate wrapping key */
...

/* Wrap Dilithium key with wrapping key */
...

/* Unwrap Dilithium key */
rv = funcs->C_UnwrapKey(session, &wrap_mech, secret_key,
                        wrapped_key, wrapped_keylen,
                        unwrap_tmpl,
                        sizeof(unwrap_tmpl) / sizeof(CK_ATTRIBUTE),
                        &priv_key);
if (rv != CKR_OK) {
   ...
}
```

Restrictions for using IBM Dilithium keys:

- IBM Dilithium keys cannot actively be used to transport (wrap and unwrap) other keys, but they can be wrapped and unwrapped for being transported using standard key types (AES, TDES).
- IBM Dilithium keys cannot be derived from given keys. They can only be generated or imported from given key values.
- Only `SignInit()` and `Sign()` (single-part operations) is supported by the EP11 host library. `SignUpdate()` and `SignFinal()` functions are not supported. The same restrictions apply for verify operations accordingly.

CKM_IBM_SHA3_nnn

Availability:

The following SHA3 mechanisms are available for the ICA token, the EP11 token, and the Soft token:

- CKM_IBM_SHA3_224
- CKM_IBM_SHA3_256
- CKM_IBM_SHA3_384
- CKM_IBM_SHA3_512

Description:

Use the listed mechanisms from the supporting tokens to perform the appropriate digest operations using the secure hash algorithm SHA3, which is the latest version of the Secure Hash Algorithm family as released by the *National Institute of Standards and Technology (NIST)*.

CKM_IBM_SHA3_224

Flags and functions:

The CKF_DIGEST flag is set to true for this mechanism. Therefore, the function `C_DigestInit()` accepts the CKM_IBM_SHA3_224 mechanism.

CKM_IBM_SHA3_256

Flags and functions:

Same as for mechanism CKM_IBM_SHA3_224.

CKM_IBM_SHA3_384

Flags and functions:

Same as for mechanism CKM_IBM_SHA3_224.

CKM_IBM_SHA3_512

Flags and functions:

Same as for mechanism CKM_IBM_SHA3_224.

CKM_IBM_SHA3_nnn_HMAC

Availability:

The following SHA3 - HMAC mechanisms are available for the ICA token, the EP11 token, and the Soft token:

- CKM_IBM_SHA3_224_HMAC
- CKM_IBM_SHA3_256_HMAC
- CKM_IBM_SHA3_384_HMAC
- CKM_IBM_SHA3_512_HMAC

Description:

Use the listed mechanisms from the supporting tokens to perform hash-based message authentication using the secure hash algorithm SHA3. The length of the HMAC output is according to the SHA3 hash sizes:

- CKM_IBM_SHA3_224_HMAC: 28 byte
- CKM_IBM_SHA3_256_HMAC: 32 byte
- CKM_IBM_SHA3_384_HMAC: 48 byte
- CKM_IBM_SHA3_512_HMAC: 64 byte

CKM_IBM_SHA3_224_HMAC

Flags and functions:

The CKF_SIGN and CKF_VERIFY flags are set to true for this mechanism. Therefore, the functions `C_SignInit()` and `C_VerifyInit()` accept the CKM_IBM_CMAC mechanism.

Possible processing sequences are:

- for single part sign or verify operations:
  - C_SignInit() followed by C_Sign()
  - C_VerifyInit() followed by C_Verify()
- for multi-part sign or verify operations:
  - `C_SignInit()` followed by multiple `C_SignUpdate()`, then followed by `C_SignFinal()`
  - `C_VerifyInit()` followed by multiple `C_VerifyUpdate()`, then followed by `C_VerifyFinal()`

CKM_IBM_SHA3_256_HMAC

Flags and functions:

Same as for mechanism CKM_IBM_SHA3_224_HMAC.

CKM_IBM_SHA3_384_HMAC

Flags and functions:

Same as for mechanism CKM_IBM_SHA3_224_HMAC.

CKM_IBM_SHA3_512_HMAC

Flags and functions:

Same as for mechanism CKM_IBM_SHA3_224_HMAC.

CKM_IBM_CMAC

Availability:

The CKM_IBM_CMAC mechanism is available with the EP11 token.

Description:

Use the CKM_IBM_CMAC mechanism to produce a block cipher-based message authentication code (CMAC) to assure the authenticity and integrity of a message. You can use this mechanism with AES or TDES keys (according to the PKCS #11 mechanisms CKM_AES_CMAC or CKM_DES3_CMAC).

The length of the output MAC is according to either the AES block size (16 bytes) or the TDES block size (8 bytes).

Flags and functions:

The CKF_SIGN and CKF_VERIFY flags are set to true for this mechanism. Therefore, the functions `C_SignInit()` and `C_VerifyInit()` accept the CKM_IBM_CMAC mechanism.

Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA

Availability:

The following mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA are available with the EP11 token.

- CKM_IBM_EC_C448 (synonym: CKM_IBM_EC_X448)
- CKM_IBM_EC_C25519 (synonym: CKM_IBM_EC_X25519)

- CKM_IBM_ED448_SHA3
- CKM_IBM_ED25519_SHA512 (synonym: CKM_IBM_EDDSA_SHA512)

Description:

The CKM_IBM_EC_* mechanisms are used for key derivation with function `C_DeriveKey()`, and use the CK_ECDH1_DERIVE_PARAMS structure (same as for the CKM_ECDH1_DERIVE mechanism).

Mechanisms CKM_IBM_ED25519_SHA512 and CKM_IBM_ED448_SHA3 are used for sign/verify actions using the functions `C_SignInit()`/`C_Sign()` and `C_VerifyInit()`/`C_Verify()`. These mechanisms do not have a mechanism parameter.

Keys for use with these IBM-specific mechanisms are generated using mechanism CKM_EC_KEY_PAIR_GEN with function `C_GenerateKeyPair()`, just like EC keys. Attribute CKA_EC_PARAMS specifies the OID of the curves to generate keys for. Importing known clear key values works the same as for EC keys.

No special key type is defined for keys for Edwards curves with above mechanisms. Key type CKK_EC is also used for those keys. This is different to what is defined with the PKCS #11 version 3 standard, where additional key types are defined for such keys (CKK_EC_EDWARDS and CKK_EC_MONTGOMERY). Also PKCS #11 version 3 defined new mechanisms for generating such keys (CKM_EC_EDWARDS_KEY_PAIR_GEN and CKM_EC_MONTGOMERY_KEY_PAIR_GEN). Although different to PKCS #11 version 3, key type CKK_EC is used with these IBM-specific mechanisms as long as openCryptoki does not support the key types of PKCS #11 version 3.

CKM_IBM_EC_C448

Flags and functions:

- The CKF_DERIVE flag specifies that the mechanism can be used with `C_DeriveKey()`.
- The CKF_EC_F_P flag specifies that the mechanism can be used with EC domain parameters over the finite field F_p.
- The CKF_EC_UNCOMPRESS flag specifies that the mechanism can be used with elliptic curve point uncompressed.
- This mechanism can also use the CK_ECDH1_DERIVE_PARAMS structure.

CKM_IBM_EC_C25519

Flags and functions:

Same as for mechanism CKM_IBM_EC_C448.

CKM_IBM_ED448_SHA3

Flags, functions, and parameters:

The CKF_SIGN and CKF_VERIFY flags are set to true for this mechanism. Therefore, the functions `C_SignInit()` and `C_VerifyInit()` accept the CKM_IBM_ED448_SHA3 mechanism.

- Possible processing sequences are the same as described with mechanism CKM_IBM_SHA3_224_HMAC.
- The CKF_EC_F_P flag specifies that the mechanism can be used with EC domain parameters over the finite field F_p.
- The CKF_EC_UNCOMPRESS flag specifies that the mechanism can be used with elliptic curve point uncompressed.
- This mechanism also must use the CK_ECDH1_DERIVE_PARAMS structure.

CKM_IBM_ED25519_SHA512

Flags and functions:

Same as for mechanism CKM_IBM_ED448_SHA3.

CKM_IBM_ATTRIBUTEBOUND_WRAP

Availability:

The CKM_IBM_ATTRIBUTEBOUND_WRAP mechanisms is available with the EP11 token.

Description:

Keys may have attributes bound to them that control the usage of these key, for example, restrictions on operations, like exportability. Attribute-bound keys provide a different format for key wrapping and unwrapping, where attributes cannot be separated from the key during transport and thus must be contained in the wrapped format of the key. Therefore, you cannot use an attribute-bound key to wrap a classical key or the other way round. For all operations except key wrapping and unwrapping, attribute-bound keys behave exactly like regular keys. The CKM_IBM_ATTRIBUTEBOUND_WRAP mechanism supports the wrapping of such attribute-bound keys.

As a prerequisite, you must specify attribute CKA_IBM_ATTRBOUND with value CK_TRUE when creating a new key with bound attributes (default is CK_FALSE).

The goal of attribute-bound keys is to bind attributes to the key during transport via wrap and unwrap. Additionally, the CKM_IBM_ATTRIBUTEBOUND_WRAP mechanism adds a signing/verification processing to the wrapping/unwrapping actions. Part of the wrapped blob is signed and the signature is appended to the blob. When unwrapping the transported key, the signature is verified. For both actions, wrapping and unwrapping, the signing and verification key is propageted to the mechanism using a parameter (see the description of the CK_IBM_ATTRIBUTEBOUND_WRAP parameter later in this topic). To this end, you need both, a wrapping/unwrapping key pair and a signing/verification key pair for the key transport. Therefore, to transport an attribute-bound key using asymmetric keys for all involved operations, both sender and receiver create a key pair each, setting CKA_IBM_ATTRBOUND to CK_TRUE for both the public and the private keys.

Thereafter, sender and receiver exchange the public keys via the usual public key transport and then import this key using `C_CreateObject()`. Then, the sender applies wrapping using the CKM_IBM_ATTRIBUTEBOUND_WRAP mechanism on the attribute-bound key and sends the resulting blob to the receiver. The key is wrapped with the public key created by the receiver and signed by the private key created by the sender.

The receiver can unwrap the blob with its own private key and verify the signature with the public key created by the sender. Both, unwrapping and signature verification is done by the attribute-bound unwrapping mechanism CKM_IBM_ATTRIBUTEBOUND_WRAP.

Only symmetric keys and RSA key(pairs) are supported as wrapping keys, also called key encrypting keys (KEKs). The signing/verifying key can be a symmetric key, an RSA key, an EC key, or a DSA key.

Attribute-bound keys contain a number of Boolean attributes, controlling the usage and trust inside the key blob. The attributes bound to the key are the following:

- CKA_EXTRACTABLE
- CKA_NEVER_EXTRACTABLE
- CKA_MODIFIABLE
- CKA_SIGN
- CKA_SIGN_RECOVER
- CKA_DECRYPT
- CKA_ENCRYPT
- CKA_DERIVE
- CKA_UNWRAP
- CKA_WRAP
- CKA_VERIFY
- CKA_VERIFY_RECOVER
- CKA_LOCAL

- CKA_WRAP_WITH_TRUSTED
- CKA_TRUSTED
- CKA_IBM_NEVER_MODIFIABLE (unsupported by openCryptoki)
- CKA_IBM_RESTRICTABLE (unsupported by openCryptoki)
- CKA_IBM_ATTRBOUND
- CKA_IBM_USE_AS_DATA
- CKA_KEY_TYPE
- CKA_VALUE_LEN

Following key types are supported as attribute-bound keys:

- CKK_AES
- CKK_DES2
- CKK_DES3
- CKK_GENERIC_SECRET
- CKK_RSA
- CKK_EC
- CKK_DSA
- CKK_DH
- CKK_IBM_PQC_DILITHIUM

Attempts to create an attribute-bound key of an unsupported type returns CKR_TEMPLATE_INCONSISTENT.

Key and key-pair generation You can specify the attribute CKA_IBM_ATTRBOUND only during key creation (for key pairs in both templates, private and public). If the attribute is specified as TRUE, the attribute CKA_SENSITIVE must be set to CK_TRUE in the attribute template for the public key as well. This can be implicitly done by the EP11 token configuration option `FORCE_SENSITIVE`. If, however, the option has its default value CK_FALSE, key or key pair generation fails with CKA_TEMPLATE_INCONSISTENT. Otherwise, attribute-bound keys behave exactly like non-attribute-bound keys.

Key derivation You can only use attribute-bound input keys to produce attribute-bound derived keys. That is, if the input key is not attribute-bound, but the attribute CKA_IBM_ATTRBOUND is set to CK_TRUE in the template for the derived key, derivation fails with return code CKR_TEMPLATE_INCONSISTENT.

By default, a derived key inherits from the input key. That is, if CKA_IBM_ATTRBOUND is not specified in the derivation template, it defaults to the value of the input key, or, if the input key does not specify CKA_IBM_ATTRBOUND, it defaults to CK_FALSE. Thus, you can derive attribute-bound keys from attribute-bound keys without specifying CKA_IBM_ATTRBOUND in the template for the derived key. However, it is possible to derive classical PKCS #11 keys from attribute-bound keys. To do this, set the attribute CKA_IBM_ATTRBOUND to FALSE in the template for the derived key. If CKA_IBM_ATTRBOUND is specified when deriving asymmetric keys, it must be specified for both templates with the same value. Otherwise, CKR_TEMPLATE_INCONSISTENT is returned.

Object creation Attribute-bound public keys can be created with `C_CreateObject()` by setting the CKA_IBM_ATTRBOUND attribute to TRUE. Attribute-bound private keys or secret keys can only be generated with the following functions:

- `C_GenerateKey()`
- `C_GenerateKeyPair()`
- `C_DeriveKey()`
- `C_UnwrapKey()`

`C_CreateObject()` returns CKR_ATTRIBUTE_VALUE_INVALID on an attempt to create attribute-bound private keys.

Object copy When copying an object, the attribute CKA_IBM_ATTRBOUND cannot be changed. An attempt to change it results in error CKA_ATTRIBUTE_READ_ONLY.

Attribute setting Since some boolean attributes, mostly key usage attributes, are bound to the key, the key blob changes when changing those attributes. This change is reflected in the value of the CKA_IBM_OPAQUE attribute. The attribute CKA_IBM_ATTRBOUND is read-only and cannot be changed. Attempts to change it returns error CKA_ATTRIBUTE_READ_ONLY. All other attributes follow standard PKCS #11 rules. However, attribute-bound keys must have CKA_SENSITIVE set to CK_TRUE and thus have additional restrictions according to PKCS #11.

How to use the CKM_IBM_ATTRIBUTEBOUND_WRAP mechanism:

The mechanism requires a parameter of type CK_IBM_ATTRIBUTEBOUND_WRAP.

```
typedef struct CK_IBM_ATTRIBUTEBOUND_WRAP {
  CK_OBJECT_HANDLE hSignVerifyKey;
} CK_IBM_ATTRIBUTEBOUND_WRAP_PARAMS;
```

This parameter provides the key for signing and verifying the wrapped key.

The CKM_IBM_ATTRIBUTEBOUND_WRAP mechanism has the following restrictions:

- All keys involved (target, wrapping/unwrapping, signature/verification) must be attribute-bound keys (CKA_IBM_ATTRBOUND = CK_TRUE). Otherwise:
  - For the target key on `C_WrapKey()`, CKR_KEY_NOT_WRAPPABLE is returned.
  - For the wrapping and unwrapping keys and the signature and verification keys, CKR_KEY_FUNCTION_NOT_PERMITTED is returned.
- On `C_WrapKey()`, the signing private key must be capable of signing (CKA_SIGN = CK_TRUE). Otherwise CKR_KEY_FUNCTION_NOT_PERMITTED is returned.
- On `C_UnwrapKey()`, the verification public key must be capable of verifying (CKA_VERIFY = CK_TRUE). Otherwise CKR_KEY_FUNCTION_NOT_PERMITTED is returned.

Only RSA and symmetric keys are supported as wrapping keys. Usage of other key types returns CKR_WRAPPING_KEY_TYPE_INCONSISTENT for wrapping, and CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT for unwrapping.

Since the input blob to be unwrapped contains both the key and the attributes, `C_UnwrapKey()` also recreates all attributes bound to the key, except the ones flagged as not supported by openCryptoki (see the previous list of attributes).

For unwrapping, CKA_KEY_TYPE must be specified in the unwrap template and must match the value bound to the key. If the values do not match, unwrapping fails with CKR_TEMPLATE_INCONSISTENT. Any key attribute of the wrapped key that is not specified in the unwrap template, is ignored. After a successful unwrapping of the key and its attributes, the attributes are available via the standard PKCS #11 methods. Remember that attribute-bound keys are sensitive keys and have additional restrictions on attribute visibility.

Flags and functions:

CKF_WRAP and CKF_UNWRAP flags are set to true for this mechanism. Therefore, the functions `C_WrapKey()` and `C_UnwrapKey` accept the CKM_IBM_ATTRIBUTEBOUND_WRAP mechanism.

Miscellaneous attributes

This section list IBM-specific attributes that apply to objects but do not adhere to certain mechanisms or purposes.

CKA_IBM_PROTKEY_NEVER_EXTRACTABLE
  Marks objects that are never importable as protected key. Does conflict with CKA_IBM_PROTKEY_EXTRACTABLE and behaves the same as CKA_NEVER_EXTRACTABLE.

CKA_IBM_STD_COMPLIANCE1
Compliance attribute. For EP11 tokens only and for all types of EP11 keys. Compliance settings correspond to standards-mandated sets of CPs. They are read-only, and are updated when CPs are updated, or a domain changes state. See also *Enterprise PKCS#11 (EP11) Library structure*.

CKA_IBM_PROTKEY_EXTRACTABLE
This key attribute is internally set to CK_TRUE, if the EP11 configuration option PKEY_MODE is enabled and the key has CKA_EXTRACTABLE set to FALSE. This makes the key eligible for being transformed into a protected key for better performance, if applicable (see also "Defining an EP11 token-specific configuration file" on page 57).

CKA_IBM_OPAQUE_PKEY
If the option PKEY_MODE is enabled in the EP11 token configuration file, a protected key is generated for the applicable key object and is added to the secure key object with this IBM-specific key attribute at first use of the key. A new protected key is generated each time if required, for example, it an LPAR has been deactivated and reactivated and its firmware master key has changed.

CKA_IBM_OPAQUE
Use this attribute for importing and exporting plain CCA key objects into or from sensitive openCryptoki key objects. See also "Usage notes for CCA library functions" on page 48.

CKA_IBM_USE_AS_DATA
Set this attribute for keys where raw key bytes may be used as data of some cryptographic operation, such as hashing (`DigestKey()`) or key derivation (`DeriveKey()`). This restriction further controls key-based operations which do not involve key migration, therefore, are not controlled by EXTRACTABLE or transport-related control points.

CKA_IBM_ATTRBOUND
Set this attribute for attribute-bound keys to enable these keys for being wrapped and unwrapped using the CKM_IBM_ATTRIBUTEBOUND_WRAP mechanism.

Re-encrypting data with a mechanism

The vendor-specific function `C_IBM_ReencryptSingle()` is available in openCryptoki and is supported by all tokens. You can use it to re-encrypt data encrypted with a given key and mechanism with another key and mechanism. This function is useful for secure key encryption with an EP11 token or a CCA token, because during the process, the data is never visible in the clear anywhere outside the cryptographic coprocessor.

The `C_IBM_ReencryptSingle` function has the following signature:

```
CK_RV C_IBM_ReencryptSingle(CK_SESSION_HANDLE hSession,
                            CK_MECHANISM_PTR pDecrMech,
                            CK_OBJECT_HANDLE hDecrKey,
                            CK_MECHANISM_PTR pEncrMech,
                            CK_OBJECT_HANDLE hEncrKey,
                            CK_BYTE_PTR pEncryptedData,
                            CK_ULONG ulEncryptedDataLen,
                            CK_BYTE_PTR pReencryptedData,
                            CK_ULONG_PTR pulReencryptedDataLen);
```

Because the new function is non-standard, it does not appear in the PKCS #11 `CK_FUNCTION_LIST` structure returned by `C_GetFunctionList()`. To invoke this function, you must either locate the desired function in the main DLL using `dlsym()`, or link the application program with the main DLL. You can also use `C_GetInterface()` to get the interface called `Vendor IBM`. This interface also provides the `C_IBM_ReencryptSingle()` function.

Like other PKCS #11 functions, this function returns output in a variable-length buffer, conforming to the convention defined by PKCS #11.

If **`pReencryptedData`** is `NULL_PTR`, then the function only uses parameter **`*pulReencryptedDataLen`** to return a number of bytes which would suffice to hold the cryptographic output produced from the input to the function. This number may exceed the precise number of bytes needed, but not to a very high extent.

If **pReencryptedData** is not NULL_PTR, then ***pulReencryptedDataLen** must contain the size in bytes of the buffer pointed to by **pReencryptedData**. If that buffer is large enough to hold the cryptographic output produced by the function, then that cryptographic output is placed there, and **CKR_OK** is returned. If the buffer is not large enough, then **CKR_BUFFER_TOO_SMALL** is returned. In either case, ***pulReencryptedDataLen** is set to hold the exact number of bytes needed to hold the produced cryptographic output.

The function generally allows to specify any combination of decryption and encryption mechanisms. However, not all combinations work with all data sizes. Mechanisms that do not perform any padding, require that the data to be encrypted is a multiple of the block size. Also some mechanisms have certain size limitations (for example, RSA). If the data size after decryption with the decryption mechanism conflicts with the requirements of the encryption mechanisms, then the re-encrypt operation may fail with **CKR_DATA_LEN_RANGE**. Also, not all tokens may support all mechanism combinations. **CKR_MECHANISM_INVALID** is returned if one of the mechanisms specified is not supported for the re-encrypt operation.

Part 4. Programming basics and user scenarios

Learn about use cases on how to take advantage of openCryptoki from new and existing applications.

The most common openCryptoki use cases fall into one of two types:

- scenarios where application programmers write new security applications using openCryptoki,
- scenarios where openCryptoki administrators want to configure an existing application with a PKCS #11 interface to use a certain openCryptoki token.

There are a variety of benefits why users may want to exploit the standardized openCryptoki cryptographic functions:

- They can start using the cryptographic operations of a soft token and later switch to a hardware security module (HSM) without changing the application code.
- They can switch between hardware security modules (HSM) from different suppliers without changing the code.
- They can use the sophisticated services of a cryptographic library without coping with the complexity of their APIs.
- They can use different openCryptoki tokens from within one application to enforce isolation between the data.
- Many software products that support encryption provide plug-in mechanisms that, if configured, will redirect cryptographic functions to a PKCS #11 library. For example, IBM middleware like the WebSphere Application Server and the HTTP Server including IBM's internal cryptographic library GSKIT can be configured to use a PKCS #11 library.

For developers of new openCryptoki applications, Chapter 17, “Programming with openCryptoki,” on page 101 first documents the basic structure of such applications. Additionally, a simple, but complete and executable code sample for generating an RSA private and public key pair with a specified openCryptoki token is provided.

Chapter 18, “Configuring a remote PKCS #11 service with openCryptoki,” on page 121 provides a user scenario showing how to set up a Soft token on a server for use from an application on a remote client.

Chapter 17. Programming with openCryptoki

Learn how to write applications from scratch that are using the functions from a certain token in the openCryptoki framework.

How it works

Shortly spoken, an openCryptoki application must specify, which token(s) it uses. Then it can invoke `C_...()` functions of openCryptoki. Where required, input to these functions is an appropriate `CKM_` mechanism that must be offered by the selected token. The mechanism defines how to perform the using function. For example, the `CKM_RSA_PKCS` mechanism defines to functions `C_EncryptInit()` and `C_Encrypt()` to encrypt clear text with a (previously generated) private RSA key.

Each mechanism is coded in such a way that it exploits one or more adequate APIs from the connected cryptographic library specified with the token definition in the `opencryptoki.conf` file.

Terminology

openCryptoki application programs deal with the following main items:

- Functions: Prefix `C_`
- Data types or general constants: Prefix `CK_`
- Attributes: Prefix `CKA_`
- Mechanisms: Prefix `CKM_`
- Return codes: Prefix `CKR_`

How to create and modify objects

All openCryptoki functions that create, modify, or copy objects take a template as one of their arguments, where the template specifies attribute values. Cryptographic functions that create objects, for example, `C_GenerateKey`, may also contribute some additional attribute values themselves. This depends on which cryptographic mechanism is being performed. In any case, all the required attributes supported by an object class that do not have default values, must be specified during object creation, either in the template or by the function itself.

An application can use the following functions for creating objects:

- `C_CreateObject`
- `C_GenerateKey`
- `C_GenerateKeyPair`
- `C_UnwrapKey`
- `C_DeriveKey`

In addition, an application can create new objects using the `C_CopyObject` function.

To create an object with any of these listed functions, the application must supply an appropriate template. This template specifies values for valid attributes. An attribute is valid if it is either one of the attributes described in the PKCS #11 specification or it is an additional vendor-specific attribute supported by the library and token. The attribute values supplied by the template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, must fully define the object to create.

Look at the following code example, where function `C_CreateObject` is used to generate an RSA key, using a template `keyTempl` to specify the key attributes. One of these attributes, CKA_KEY_TYPE, defines the RSA type of the key:

```
/*
 * create an RSA key object with C_CreateObject
 */
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;

CK_OBJECT_CLASS
  dataClass = CKO_DATA,
  certificateClass = CKO_CERTIFICATE,
  keyClass = CKO_PUBLIC_KEY;

CK_KEY_TYPE keyType = CKK_RSA;

CK_ATTRIBUTE keyTemplate[] = {
  {CKA_CLASS, &keyClass, sizeof(keyClass)},
  {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
  {CKA_WRAP, &true, sizeof(true)},
  {CKA_MODULUS, modulus, sizeof(modulus)},
  {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
};
CK_RV rc;

rc = C_CreateObject(hSession, keyTemplate, 5, &hKey);

if (rc != CKR_OK) {
    printf("Error creating key object: 0x%X\n", rc);  return rc;
}

if (rc == CKR_OK) {
      printf("RSA key object creation successful.\n");
}
```

How to apply attributes to objects

In openCryptoki, an attribute defines a characteristic of an object, for example, for a generated key. In openCryptoki, there are general attributes, such as whether the object is private or public. There are also attributes that are specific to a particular type of object, such as a modulus or exponent for RSA keys.

Attributes determine the characteristics of an object or how a mechanism is applied to an object.

Attributes are denoted by names starting with the prefix `'CKA_'`. Some attributes are valid for one certain object type, others are valid for multiple object types.

The type is specified on an object through the CKA_CLASS attribute of the object. Especially, the CKA_CLASS attribute determines which further attributes are associated with an object. Other typical attributes contain the value of an object or determine whether an object is a token object.

Structure of an openCryptoki application

The basic structure of an application that uses an openCryptoki token in order to encrypt and decrypt with an RSA key pair is described in this topic. You can use it as a template for general cryptographic applications.

Before you begin

Consider two things:

1. Identify the slot ID of the token you want to utilize.
2. Check whether the mechanisms that you must use or want to use are supported by the selected token.

Procedure

1. Provide the openCryptoki data types, functions, attributes and all other available items for programming via the following ANSI C header files.

   ```
   #include <opencryptoki/pkcs11.h>   /* top-level Cryptoki include file */
   #include <stdlib.h>
   #include <errno.h>
   ```

```
#include <stdio.h>
#include <dlfcn.h>
#include <defs.h>
```

2. Use function `C_Initialize` to initialize *Cryptoki*.

```

CK_RV C_Initialize(CK_VOID_PTR pInitArgs);
```

   An application becomes an openCryptoki application by calling the `C_Initialize` function from one of its threads. After this call, the application can call other openCryptoki functions.

3. Use function `C_InitToken` to initialize the desired token in the specified slot. (if token not yet initialized, for example, with `pkcsconf`).

```

CK_RV C_InitToken(
    CK_SLOT_ID slotID,
    CK_UTF8CHAR_PTR pPin,
    CK_ULONG ulPinLen,
    CK_UTF8CHAR_PTR pLabel
);
```

4. Use function `C_OpenSession` to open a connection between an application and a particular token.

```

CK_RV openSession(CK_SLOT_ID slotID, CK_FLAGS sFlags,
CK_SESSION_HANDLE_PTR phSession) {
    CK_RV rc;
    rc = C_OpenSession(slotID, sFlags, NULL, NULL, phSession);

    printf("Open session successful.\n");
    return CKR_OK;
}
```

5. Use function `C_Login` to log a user into a token. Variable **`userType`** specifies the user role (SO or normal User).

```
CK_RV loginSession(CK_USER_TYPE userType, CK_CHAR_PTR pPin,
CK_ULONG ulPinLen, CK_SESSION_HANDLE hSession) {
    CK_RV rc;
    rc = C_Login(hSession, userType, pPin, ulPinLen);

    printf("Login session successful.\n");
    return CKR_OK;
}
```

6. Use function `C_GenerateKeyPair` to generate an RSA private and public key pair. The used mechanism is `CKM_RSA_PKCS_KEY_PAIR_GEN`.

```

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
CK_MECHANISM mechanism = {
  CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};

CK_ULONG modulusBits = 768;
CK_BYTE publicExponent[] = { 3 };
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BBOOL true = CK_TRUE;

CK_ATTRIBUTE publicKeyTemplate[] = {
  {CKA_ENCRYPT, &true, sizeof(true)},
  {CKA_VERIFY, &true, sizeof(true)},
  {CKA_WRAP, &true, sizeof(true)},
  {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
  {CKA_PUBLIC_EXPONENT, publicExponent, sizeof (publicExponent)}
};

CK_ATTRIBUTE privateKeyTemplate[] = {
  {CKA_TOKEN, &true, sizeof(true)},
  {CKA_PRIVATE, &true, sizeof(true)},
  {CKA_SUBJECT, subject, sizeof(subject)},
  {CKA_ID, id, sizeof(id)},
  {CKA_SENSITIVE, &true, sizeof(true)},
  {CKA_DECRYPT, &true, sizeof(true)},
  {CKA_SIGN, &true, sizeof(true)},
  {CKA_UNWRAP, &true, sizeof(true)}
};

CK_RV rv;

rv = C_GenerateKeyPair(
  hSession, &mechanism,
  publicKeyTemplate, 5,
  privateKeyTemplate, 8,
  &hPublicKey, &hPrivateKey);
if (rv == CKR_OK) {
  .
  .
}
```

7. Use functions `C_EncryptInit` and `C_Encrypt` to initialize an encryption operation and to encrypt data with the previously generated RSA private key.

```

#define PLAINTEXT_BUF_SZ 200
#define CIPHERTEXT_BUF_SZ 256

CK_ULONG firstPieceLen, secondPieceLen;
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM mechanism = {
  CKM_DES_CBC_PAD, iv, sizeof(iv)
};

CK_BYTE data[PLAINTEXT_BUF_SZ];
CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
CK_ULONG ulEncryptedData1Len;
CK_ULONG ulEncryptedData2Len;
CK_ULONG ulEncryptedData3Len;
CK_RV rv;
...
firstPieceLen = 90;
secondPieceLen = PLAINTEXT_BUF_SZ-firstPieceLen;
rv = C_EncryptInit(hSession, &mechanism, hKey);

if (rv == CKR_OK) {
  /* Encrypt first piece */
  ulEncryptedData1Len = sizeof(encryptedData);
  rv = C_EncryptUpdate(
    hSession,
    &data[0], firstPieceLen,
    &encryptedData[0], &ulEncryptedData1Len);


  /* Encrypt second piece */
  ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;
  rv = C_EncryptUpdate(
    hSession,
    &data[firstPieceLen], secondPieceLen,
    &encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);
  if (rv != CKR_OK) { ... }

  /* Get last little encrypted bit */

  ulEncryptedData3Len =
    sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;
  rv = C_EncryptFinal(
    hSession,
    &encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],
    &ulEncryptedData3Len);
  if (rv != CKR_OK) {...}

}
```

8. Use function `C_Logout` to logout from the openCryptoki session and close the session.

```

   rv = C_Logout(session);
     if (rv != CKR_OK) goto err;
```

```
    rv = C_CloseSession(session);
     if (rv != CKR_OK) goto err;
```

9. Use function `C_Finalize` to finalize the operation.

```
 rv = fn->C_Finalize(NULL);
     if (rv != CKR_OK) goto err;
```

Sample openCryptoki program

View a completely coded example of an openCryptoki application that performs an RSA key generation operation.

Note: This sample program does not include the operation to encrypt data with the previously generated RSA private key as described in step "7" on page 104 of "Structure of an openCryptoki application" on page 102.

```
/*
 * Build:
 *   cc -o pkcs11 pkcs11.c -ldl
 *
 * Usage:
 *   pkcs11 <so_name> <slot_id> <user_pin>
 *
 * Description:
 *   Loads the PKCS11 shared library <so_name>,
 *   opens a session with slot <slot_id>,
 *   logs the user in using the PIN <user_pin>,
 *   and performs an RSA key generation operation.
 */
/* Step 1 */
#include <opencryptoki/pkcs11.h>
#include <string.h>
#include <stdlib.h>
#include <dlfcn.h>

#define NELEM(array) (sizeof(array) / sizeof((array)[0]))

int main(int argc, char *argv[])
{
    CK_C_GetFunctionList get_functionlist = {NULL};
    CK_SESSION_HANDLE session = CK_INVALID_HANDLE;
    CK_FUNCTION_LIST *fn = NULL;
    void *pkcs11so = NULL;
    CK_SLOT_ID slot_id;
    CK_FLAGS flags;
    int rc = -1;
    char *ptr;
    CK_RV rv;

    if (argc != 4) goto err;

    pkcs11so = dlopen(argv[1], RTLD_NOW);
    if (pkcs11so == NULL) goto err;

    slot_id = strtoul(argv[2], &ptr, 0);
    if (*(argv[2]) == '\0' || *ptr != '\0') goto err;

    *(void **)(&get_functionlist) = dlsym(pkcs11so, "C_GetFunctionList");
    if (get_functionlist == NULL) goto err;

    rv = get_functionlist(&fn);
    if (rv != CKR_OK || fn == NULL) goto err;

/* Step 2 */
    rv = fn->C_Initialize(NULL);
    if (rv != CKR_OK) goto err;

    flags = CKF_SERIAL_SESSION | CKF_RW_SESSION;
/* Step 4  (Step 3 assumed to be done by pkcsconf) */
    rv = fn->C_OpenSession(slot_id, flags, NULL, NULL, &session);
    if (rv != CKR_OK || session == CK_INVALID_HANDLE) goto err;

/* Step 5 */
    rv = fn->C_Login(session, CKU_USER, (CK_UTF8CHAR *)argv[3], strlen(argv[3]));
    if (rv != CKR_OK) goto err;
 /* Step 6  (Step 7 not coded in this example */
    {
        CK_MECHANISM mechanism = {CKM_RSA_PKCS_KEY_PAIR_GEN, NULL, 0};
        CK_BYTE e[] = {0x01, 0x00, 0x01};
        CK_ULONG modbits = 4096;
        CK_BYTE subject[] = "RSA4096 Test";
        CK_BYTE id[] = {1};
        CK_BBOOL true_ = CK_TRUE;

        CK_ATTRIBUTE template_publ[] = {
            {CKA_ENCRYPT,         &true_,   sizeof(true_)},
            {CKA_VERIFY,          &true_,   sizeof(true_)},
            {CKA_WRAP,            &true_,   sizeof(true_)},
            {CKA_MODULUS_BITS,    &modbits, sizeof(modbits)},
            {CKA_PUBLIC_EXPONENT, e,        sizeof(e)}
        };
        CK_ATTRIBUTE template_priv[] = {
            {CKA_SUBJECT,         subject,  sizeof(subject)},
            {CKA_ID,              id,       sizeof(id)},
            {CKA_TOKEN,           &true_,   sizeof(true_)},
            {CKA_PRIVATE,         &true_,   sizeof(true_)},
            {CKA_SENSITIVE,       &true_,   sizeof(true_)},
            {CKA_DECRYPT,         &true_,   sizeof(true_)},
            {CKA_SIGN,            &true_,   sizeof(true_)},
            {CKA_UNWRAP,          &true_,   sizeof(true_)}
        };

        CK_OBJECT_HANDLE publ, priv;

        rv = fn->C_GenerateKeyPair(session, &mechanism,
                                   template_publ, NELEM(template_publ),
                                   template_priv, NELEM(template_priv),
                                   &publ, &priv);
        if (rv != CKR_OK) goto err;
    }

/* Step 8 */
    rv = fn->C_Logout(session);
    if (rv != CKR_OK) goto err;

    rv = fn->C_CloseSession(session);
    if (rv != CKR_OK) goto err;

 /* Step 9 */
    rv = fn->C_Finalize(NULL);
    if (rv != CKR_OK) goto err;

    rc = 0;
err:
    if (pkcs11so != NULL)
        dlclose(pkcs11so);
    return rc;
}
```

openCryptoki code samples (C)

To develop an application that uses openCryptoki, you need to access the library.

There are two ways to access the library:

- Load shared objects using dynamic library calls (dlopen)
- Link the library to your application during build time

Dynamic library call

View some openCryptoki code samples for a dynamic library call.

```
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <dlfcn.h>
#include <opencryptoki/pkcs11.h>

CK_RV init();
CK_RV cleanup();
CK_RV rc;                                    /* return code */
void *dllPtr, (*symPtr)();                   /* pointer to the ock library */
CK_FUNCTION_LIST_PTR  FunctionPtr = NULL;   /* pointer to function list */

int main(int argc, char *argv[]){
  /* opencryptoki initialization   */
  rc = init("/usr/lib64/opencryptoki/libopencryptoki.so");
  /* further opencryptoki commands */
  rc = cleanup();                            /* cleanup/close shared library */
  return 0;
}

CK_RV init(char *libPath){

  dllPtr = dlopen(libPath, RTLD_NOW);       /* open the PKCS11 library */
  if (!dllPtr) {
     printf("Error loading PKCS#11 library \n");
     return errno;
  }
  /* Get ock function list */
  symPtr = (void (*)())dlsym(dllPtr, "C_GetFunctionList");
  if (!symPtr) {
     printf("Error getting function list \n");
     return errno;
  }
  symPtr(&FunctionPtr);
  rc = FunctionPtr->C_Initialize(NULL);     /* initialize opencryptoki/tokens) */
   if (rc != CKR_OK) {
    printf("Error initializing the opencryptoki library: 0x%X\n", rc);
    cleanup();
    }
    printf("Opencryptoki initialized.\n");
    return CKR_OK;
}

  CK_RV cleanup(void) {
      rc = FunctionPtr->C_Finalize(NULL);
      if (dllPtr)
          dlclose(dllPtr);
      return rc;
}
```

To compile your sample code you need to provide the path of the source/include files. Issue a command of the form:

```
gcc sample_dynamic.c -g -O0 -o sample_dynamic
```

Shared linked library

When you use your sample code with a static linked library you can access the APIs directly.

At the compile time you need to specify the openCryptoki library:

```
gcc sample_shared.c -g -O0 -o sample_shared /usr/lib64/opencryptoki/libopencryptoki.so
```

The presented samples that interact with the openCryptoki API are based on the shared linked openCryptoki library.

Base procedures

View some openCryptoki code samples for base procedures, such as a main program, an initialization procedure, and finalize information.

Main program

```
/* Example program to test opencryptoki
 * build: gcc test_ock.c -g -O0 -o test_ock -lopencryptoki
 * execute: ./test_ock -c <slot> -p <PIN> */
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <dlfcn.h>
#include <opencryptoki/pkcs11.h>
#include <string.h>
#include <unistd.h>

void *lib_ock;
char *pin = NULL;
int count, arg;
CK_SLOT_ID  slotID = 0;
CK_ULONG rsaKeyLen = 2048, cipherTextLen = 0, clearTextLen = 0;
CK_BYTE *pCipherText = NULL, *pClearText = NULL;
CK_BYTE *pRSACipher = NULL, *pRSAClear = NULL;
CK_FLAGS rw_sessionFlags = CKF_RW_SESSION | CKF_SERIAL_SESSION;
CK_SESSION_HANDLE hSession;
CK_BYTE keyValue[] = {0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
                      0xCA,0xFE,0xBE,0xEF,0xCA,0xFE,0xBE,0xEF};
CK_BYTE msg[] = "The quick brown fox jumps over the lazy dog";
CK_ULONG msgLen = sizeof(msg);
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;

/*** <insert helper functions (provided below) here> ***/
/*** usage / help ***/
void usage(void)
{
  printf("Usage:\n");
  printf(" -s <slot number> \n");
  printf(" -p <user PIN>\n");
  printf("\n");
  exit (8);  }

int main(int argc, char *argv[]) {
   while ((arg = getopt (argc, argv, "s:p:")) != -1) {
    switch (arg) {
    case 's':   slotID = atoi(optarg);
             break;
    case 'p':   pin = malloc(strlen(optarg));
                strcpy(pin,optarg);
             break;
    default:    printf("wrong option %c", arg);
             usage();
    }  }

  if ((!pin) || (!slotID)) {
    printf("Incorrect parameter given!\n");
    usage();
    exit (8);   }

  init();
  openSession(slotID, rw_sessionFlags, &hSession);
  loginSession(CKU_USER, pin, 8, hSession);
  createKeyObject(hSession, (CK_BYTE_PTR)&keyValue, sizeof(keyValue));
  AESencrypt(hSession, (CK_BYTE_PTR)&msg, msgLen, &pCipherText, &cipherTextLen);
  AESdecrypt(hSession, pCipherText, cipherTextLen, &pClearText, &clearTextLen);
  generateRSAKeyPair(hSession, rsaKeyLen, &hPublicKey, &hPrivateKey);
  RSAencrypt(hSession, hPublicKey, (CK_BYTE_PTR)&msg, msgLen, &pRSACipher, &rsaKeyLen);
  RSAdecrypt(hSession, hPrivateKey, pRSACipher, rsaKeyLen, &pRSAClear, &rsaKeyLen);
  logoutSession(hSession); closeSession(hSession);
  finalize();
  return 0;
}
```

C_Initialize

```
/*
 * initialize
 */
CK_RV init(void){
  CK_RV rc;
  rc = C_Initialize(NULL);
  if (rc != CKR_OK) {
     printf("Error initializing the opencryptoki library: 0x%X\n", rc);
  }
  return rc;
}
```

C_Finalize

```

/*
 * finalize
 */
CK_RV finalize(void) {
  CK_RV rc;
  rc = C_Finalize(NULL);
  if (rc != CKR_OK) {
         printf("Error during finalize: %x\n", rc);
  }
  if (pCipherText) free(pCipherText);
  if (pClearText)  free(pClearText);
  if (pRSACipher)  free(pRSACipher);
  if (pRSAClear)   free(pRSAClear);
  return rc;
}
```

Session and log-in procedures

When you use your sample code with a static linked library you can access the APIs directly. View some openCryptoki code samples for opening and closing sessions and for log-in.

C_OpenSession:

```
/*
 * opensession
 */

CK_RV openSession(CK_SLOT_ID slotID, CK_FLAGS sFlags,
                                     CK_SESSION_HANDLE_PTR phSession) {
 CK_RV rc;
  rc = C_OpenSession(slotID, sFlags, NULL, NULL, phSession);
  if (rc != CKR_OK) {
     printf("Error opening session: %x\n", rc);
     return rc;
  }
  printf("Open session successful.\n");
  return CKR_OK;
}
```

C_CloseSession:

```
/*
 * closesession
 */
CK_RV closeSession(CK_SESSION_HANDLE hSession) {
  CK_RV  rc;
  rc  = C_CloseSession(hSession);
  if (rc != CKR_OK) {
    printf("Error closing session: 0x%X\n", rc);
    return rc;
  }
  printf("Close session successful.\n");
  return CKR_OK;
}
```

C_Login:

```
/*
 * login
 */
CK_RV loginSession(CK_USER_TYPE userType, CK_CHAR_PTR pPin,
           CK_ULONG ulPinLen, CK_SESSION_HANDLE hSession) {
  CK_RV rc;
  rc = C_Login(hSession, userType, pPin, ulPinLen);
  if (rc != CKR_OK) {
    printf("Error login session: %x\n", rc);
    return rc;
  }
  printf("Login session successful.\n");
  return CKR_OK;
}
```

C_Logout:

```
/*
 * logout
 */
CK_RV logoutSession(CK_SESSION_HANDLE hSession) {
  CK_RV rc;
  rc = C_Logout(hSession);
  if (rc != CKR_OK) {
    printf("Error logout session: %x\n", rc);
    return rc;
  }
  printf("Logout session successful.\n");
  return CKR_OK;
}
```

Object handling procedures

When you use your sample code with a static linked library you can access the APIs directly. View some openCryptoki code samples for procedures dealing with object handling.

C_CreateObject:

```
/*
 * create a key object with C_CreateObject
 */
CK_RV createKeyObject(CK_SESSION_HANDLE hSession, CK_BYTE_PTR key, CK_ULONG keyLength) {
  CK_RV rc;

  CK_OBJECT_HANDLE hKey;
  CK_BBOOL true = TRUE;
  CK_BBOOL false = FALSE;
  CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
  CK_KEY_TYPE keyType = CKK_AES;
  CK_ATTRIBUTE keyTempl[] = {
    {CKA_CLASS, &keyClass, sizeof(keyClass)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_TOKEN, &true, sizeof(true)},         /* token object  */
    {CKA_PRIVATE, &false, sizeof(false)},    /* public object */
    {CKA_VALUE, keyValue, keyLength},    /* AES key       */
    {CKA_LABEL, "My_AES_Key", sizeof("My_AES_Key")}
  };
  rc = C_CreateObject(hSession, keyTempl, sizeof (keyTempl)/sizeof (CK_ATTRIBUTE), &hKey);
  if (rc != CKR_OK) {
    printf("Error creating key object: 0x%X\n", rc); return rc;
  }
  printf("AES Key object creation successful.\n");
  return CKR_OK;
}
```

C_FindObjects:

```
/*
 * findObjects
 */
CK_RV getKey(CK_CHAR_PTR label, int labelLen, CK_OBJECT_HANDLE_PTR hObject,
         CK_SESSION_HANDLE hSession) {
  CK_RV rc;
  CK_ULONG ulMaxObjectCount = 1;
  CK_ULONG ulObjectCount;
  CK_ATTRIBUTE objectMask[] = { {CKA_LABEL, label, labelLen} };
  rc = C_FindObjectsInit(hSession, objectMask, 1);
  if (rc != CKR_OK) {
    printf("Error FindObjectsInit: 0x%X\n", rc); return rc;
  }
  rc = C_FindObjects(hSession, hObject, ulMaxObjectCount, &ulObjectCount);
  if (rc != CKR_OK) {
    printf("Error FindObjects: 0x%X\n", rc); return rc;
  }
  rc = C_FindObjectsFinal(hSession);
  if (rc != CKR_OK) {
    printf("Error FindObjectsFinal: 0x%X\n", rc); return rc;
  }
  return CKR_OK;
}
```

Cryptographic operations

When you use your sample code with a static linked library you can access the APIs directly. View some openCryptoki code samples for procedures that perform cryptographic operations.

C_Encrypt (AES):

```
/*
 * AES encrypt
 */
CK_RV AESencrypt(CK_SESSION_HANDLE hSession,
         CK_BYTE_PTR pClearData,  CK_ULONG ulClearDataLen,
         CK_BYTE **pEncryptedData, CK_ULONG_PTR pulEncryptedDataLen) {
  CK_RV rc;
  CK_MECHANISM myMechanism = {CKM_AES_CBC_PAD, "01020304050607081122334455667788", 16};
  CK_MECHANISM_PTR pMechanism = &myMechanism;
  CK_OBJECT_HANDLE hKey;
  getKey("My_AES_Key", sizeof("My_AES_Key"), &hKey, hSession);
  rc = C_EncryptInit(hSession, pMechanism, hKey);
  if (rc != CKR_OK) {
    printf("Error initializing encryption: 0x%X\n", rc);
    return rc;
  }
  rc = C_Encrypt(hSession, pClearData, ulClearDataLen,
         NULL, pulEncryptedDataLen);
  if (rc != CKR_OK) {
    printf("Error during encryption (get length): %x\n", rc);
    return rc;
  }
  *pEncryptedData = (CK_BYTE *)malloc(*pulEncryptedDataLen * sizeof(CK_BYTE));

  rc = C_Encrypt(hSession, pClearData, ulClearDataLen,
         *pEncryptedData, pulEncryptedDataLen);
  if (rc != CKR_OK) {
    printf("Error during encryption: %x\n", rc);
    return rc;
  }
  printf("Encrypted data: ");
  CK_BYTE_PTR tmp = *pEncryptedData;
  for (count = 0; count < *pulEncryptedDataLen; count++, tmp++) {
    printf("%X", *tmp);
  }
  printf("\n");

  return CKR_OK;
}
```

C_Decrypt (AES):

```
/*
 * AES decrypt
 */
CK_RV AESdecrypt(CK_SESSION_HANDLE hSession,
         CK_BYTE_PTR pEncryptedData, CK_ULONG ulEncryptedDataLen,
         CK_BYTE **pClearData, CK_ULONG_PTR pulClearDataLen) {
  CK_RV rc;
  CK_MECHANISM myMechanism = {CKM_AES_CBC_PAD, "01020304050607081122334455667788", 16};
  CK_MECHANISM_PTR pMechanism = &myMechanism;
  CK_OBJECT_HANDLE hKey;
  getKey("My_AES_Key", sizeof("My_AES_Key"), &hKey, hSession);
  rc = C_DecryptInit(hSession, pMechanism, hKey);
  if (rc != CKR_OK) {
    printf("Error initializing decryption: 0x%X\n", rc);
    return rc;
  }
  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen, NULL, pulClearDataLen);
  if (rc != CKR_OK) {
    printf("Error during decryption (get length): %x\n", rc);
    return rc;
  }
  *pClearData = malloc(*pulClearDataLen * sizeof(CK_BYTE));
  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen, *pClearData,
      pulClearDataLen);
  if (rc != CKR_OK) {
    printf("Error during decryption: %x\n", rc);
    return rc;
  }
  printf("Decrypted data: ");
  CK_BYTE_PTR tmp = *pClearData;
  for (count = 0; count < *pulClearDataLen; count++, tmp++) {
    printf("%c", *tmp);
  }
  printf("\n");
  return CKR_OK;
}
```

C_GenerateKeyPair (RSA):

```
/*
 * RSA key generate
 */
CK_RV generateRSAKeyPair(CK_SESSION_HANDLE hSession, CK_ULONG keySize,
             CK_OBJECT_HANDLE_PTR phPublicKey, CK_OBJECT_HANDLE_PTR phPrivateKey ) {
  CK_RV rc;
  CK_BBOOL true = TRUE;
  CK_BBOOL false = FALSE;
  CK_OBJECT_CLASS keyClassPub = CKO_PUBLIC_KEY;
  CK_OBJECT_CLASS keyClassPriv = CKO_PRIVATE_KEY;
  CK_KEY_TYPE keyTypeRSA = CKK_RSA;
  CK_ULONG modulusBits = keySize;
  CK_BYTE_PTR pModulus = malloc(sizeof(CK_BYTE)*modulusBits/8);
  CK_BYTE publicExponent[] = {1, 0, 1};
  CK_MECHANISM rsaKeyGenMech = {CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0};
  CK_ATTRIBUTE pubKeyTempl[] = {
    {CKA_CLASS, &keyClassPub, sizeof(keyClassPub)},
    {CKA_KEY_TYPE, &keyTypeRSA, sizeof(keyTypeRSA)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_PRIVATE, &true, sizeof(true)},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_WRAP, &true, sizeof(true)},
    {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
    {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)},
    {CKA_LABEL, "My_Private_Token_RSA1024_PubKey",
    sizeof("My_Private_Token_RSA1024_PubKey")},
    {CKA_MODIFIABLE, &true, sizeof(true)},
  };
  CK_ATTRIBUTE privKeyTempl[] = {
    {CKA_CLASS, &keyClassPriv, sizeof(keyClassPriv)},
    {CKA_KEY_TYPE, &keyTypeRSA, sizeof(keyTypeRSA)},
    {CKA_EXTRACTABLE, &true, sizeof(true)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_PRIVATE, &true, sizeof(true)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DECRYPT, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_UNWRAP, &true, sizeof(true)},
    {CKA_LABEL, "My_Private_Token_RSA1024_PrivKey",
    sizeof("My_Private_Token_RSA1024_PrivKey")},
    {CKA_MODIFIABLE, &true, sizeof(true)},
  };
  rc = C_GenerateKeyPair(hSession, &rsaKeyGenMech ,
             &pubKeyTempl, sizeof(pubKeyTempl)/sizeof (CK_ATTRIBUTE),
             &privKeyTempl, sizeof(privKeyTempl)/sizeof (CK_ATTRIBUTE),
             phPublicKey, phPrivateKey);
  if (rc != CKR_OK) {
    printf("Error generating RSA keys: %x\n", rc);
    return rc;
  }
  printf("RSA Key generation successful.\n");
  return CKR_OK;
}
```

C_Encrypt (RSA):

```
/*
 * RSA encrypt
 */
CK_RV RSAencrypt(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hKey,
         CK_BYTE_PTR pClearData, CK_ULONG ulClearDataLen,
         CK_BYTE **pEncryptedData, CK_ULONG_PTR pulEncryptedDataLen) {
  CK_RV rc;
  CK_MECHANISM rsaMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};
  rc = C_EncryptInit(hSession, rsaMechanism, hKey);
  if (rc != CKR_OK) {
    printf("Error initializing RSA encryption: %x\n", rc);
    return rc;
  }
  rc = C_Encrypt(hSession, pClearData, ulClearDataLen,
         NULL, pulEncryptedDataLen);
  if (rc != CKR_OK) {
    printf("Error during RSA encryption: %x\n", rc);
    return rc;
  }

  *pEncryptedData = (CK_BYTE *)malloc(rsaKeyLen * sizeof(CK_BYTE));
  rc = C_Encrypt(hSession, pClearData, ulClearDataLen,
         *pEncryptedData, pulEncryptedDataLen);
  if (rc != CKR_OK) {
    printf("Error during RSA encryption: %x\n", rc);
    return rc;
  }

  printf("Encrypted data: ");
  CK_BYTE_PTR tmp = *pEncryptedData;
  for (count = 0; count < *pulEncryptedDataLen; count++, tmp++) {
    printf("%X", *tmp);
  }
  printf("\n");
  return CKR_OK;
}
```

C_Decrypt (RSA):

```
/*
 * RSA decrypt
 */
CK_RV RSAdecrypt(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hKey,
         CK_BYTE_PTR pEncryptedData, CK_ULONG ulEncryptedDataLen,
         CK_BYTE **pClearData, CK_ULONG_PTR pulClearDataLen) {
  CK_RV rc;
  CK_MECHANISM rsaMechanism = {CKM_RSA_PKCS, NULL_PTR, 0};
  rc = C_DecryptInit(hSession, rsaMechanism, hKey);
  if (rc != CKR_OK) {
    printf("Error initializing RSA decryption: %x\n", rc);
    return rc;
  }
  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen,
         NULL, pulClearDataLen);
  if (rc != CKR_OK) {
    printf("Error during RSA decryption: %x\n", rc);
    return rc;
  }

  *pClearData = malloc(rsaKeyLen*sizeof(CK_BYTE));
  rc = C_Decrypt(hSession, pEncryptedData, ulEncryptedDataLen,
         *pClearData, pulClearDataLen);
  if (rc != CKR_OK) {
    printf("Error during RSA decryption: %x\n", rc);
    return rc;
  }
  printf("Decrypted data: ");
  CK_BYTE_PTR tmp = *pClearData;
  for (count = 0; count < *pulClearDataLen; count++, tmp++) {
    printf("%c", *tmp);
  }
  printf("\n");
  return CKR_OK;
}
```

Trouble shooting

When you run into troubles while working with openCryptoki, the information provided in this topic may help you to resolve your problem.

To react on error messages related to openCryptoki, perform a series of checks:

1. “Walk through a check list” on page 116
2. “Checking the syslog messages” on page 117
3. “Apply tracing” on page 119
4. “Final check” on page 120

Walk through a check list

1. Is the current user authorized in the *pkcs11* group?

   The *pkcs11* group must be defined in file `/etc/group` of the system. An entry in this file, like for example: `pkcs11:x:989:root`, indicates that the *root* user is a member of the *pkcs11* group. To add a user to this group, issue the following command: `usermod -aG pkcs11 <user>`.

   Every user of openCryptoki must be a member of this pkcs11 group. For more information, read “Access control and groups” on page 11.
2. Is the slot manager daemon **pkcsslotd** up and running?

   The **pkcsslotd** daemon must be active in the system to coordinate token accesses from multiple processes. For more information, read “Slot manager” on page 7 and “Starting the slot manager” on page 26.
3. Is the host library of the token installed that you want to exploit?

- For the CCA token, the host library and its default distribution location is `/usr/lib64/libcsulcca.so.<v>.<r>.<m>`.
- For the ICA token, the library and its default distribution location is `/usr/lib64/libica.so.<v>.<r>.<m>`.
- For the EP11 token, the library and its default distribution location is `/usr/lib64/libep11.so.<v>.<r>.<m>`.
- For the Soft token, the library and its default distribution location is `/usr/lib64/libcrypto.so.<v>.<r>.<m>`.

4. Are the required cryptographic coprocessors available and online?

   Use the **`lszcrypt`** command to display a list of available cryptographic devices and their online status.

```
# lszcrypt

CARD.DOMAIN TYPE  MODE        STATUS  REQUESTS
----------------------------------------------
08          CEX7C CCA-Coproc  online     10484
08.0031     CEX7C CCA-Coproc  online     10484
09          CEX7C CCA-Coproc  online        34
09.0031     CEX7C CCA-Coproc  online        34
0a          CEX7P EP11-Coproc online     24766
0a.0031     CEX7P EP11-Coproc online     24766
0b          CEX7P EP11-Coproc online     15420
0b.0031     CEX7P EP11-Coproc online     15420
```

5. Are the slot definitions correctly specified in the openCryptoki configuration file (`/etc/opencryptoki/opencryptoki.conf`)?

   A list of all available tokens is required before you can use openCryptoki. This list is provided by the global configuration file called `opencryptoki.conf`. For more information, read Chapter 4, “Adjusting the openCryptoki configuration file,” on page 17.

Checking the syslog messages

openCryptoki issues the following syslog messages (alphabetically sorted). Numbers (placeholder x) and names like for example <file>, will be replaced in the real output. Check the messages and correct the mentioned error.

```
C_Initialize: Invalid number of functions passed in argument structure.
C_Initialize: Application specified that OS locking is invalid.
              PKCS11 Module requires OS locking.
C_Initialize: Module failed to attach to shared memory. Verify that the slot management daemon
              is running, errno=x
C_Initialize: Module failed to create a socket.
              Verify that the slot management daemon is running.
C_Initialize: Module failed to retrieve slot infos from slot deamon.
C_Initialize: Application specified that library can't create OS threads. PKCS11 Module
              requires to create threads when event support is enabled.

Cannot read size
Cannot read boolean
Cannot malloc x bytes to read in token object <file> (ignoring it)
Cannot read token object <file> (ignoring it)
Cannot restore token object <file> (ignoring it)
Cannot read header

chmod(<file>): <strerror>

connect_socket: failed to find socket file, errno=x
connect_socket: pkcs11 group does not exist, errno=x
connect_socket: incorrect permissions on socket file
connect_socket: failed to create socket, errno=x
connect_socket: failed to connect to slotmanager daemon, errno=x

Could not open <lockfilename>
```

```
<dlerror>

Directory(<dir>) missing: <strerror>

DL_Load: dlopen() failed for [<dll_location>]; dlerror = <dlerror>

ep11_load_host_lib: Error loading shared library <file>' [<strerror>]
ep11_load_host_lib: Error loading shared library 'libep11.so[.3|.2|.1]' [<strerror>]
ep11_login_handler: Error: VHSM-Pin blob of adapter <apqn> is not equal to other adapters
                    for same session
ep11_login_handler: Error: Pin blob of adapter <apqn> is not equal to other adapters
                    for same session
ep11_resolve_lib_sym: Error: <dlerror>

ep11tok_load_libica: Error loading shared library '<file>' [<strerror>]
ep11tok_load_libica: Failed to initialize the target lock
ep11tok_load_libica: Error: EP 11 library initialization failed
ep11tok_load_libica: Failed to get the EP11 library version rc=x
ep11tok_load_libica: Failed to get the target info rc=x
ep11tok_load_libica: Error: CKR_IBM_WK_NOT_INITIALIZED occurred, no master key set ?
ep11tok_load_libica: Error: CKR_FUNCTION_CANCELED occurred, control point 13
                     (generate or derive symmetric keys including DSA parameters) disabled ?
ep11tok_load_libica: Warning: Could not get mk_vp, protected key support not available.
ep11tok_login_session: Error: A VHSM-PIN is required for VHSM_MODE.
ep11tok_handle_apqn_event: Failed to get the target info rc=x


fchmod(<file>): <strerror>
fchown(<file>): <strerror>
fchown(<file>,-1,pkcs11) failed: <strerror>. Tracing is disabled.

getgrnam() failed: <strerror>
getgrnam(pkcs11) failed: <strerror>. Tracing is disabled.
getgrnam(): <strerror>
getpwuid(): <strerror>

init_socket_data: read error on daemon socket, errno=x
init_socket_data: read returned with eof but we still expect x bytes from daemon

Invalid strength configuration in policy!

lock directory path too long
lock file path too long

mkdir(<file>): <strerror>

OPENCRYPTOKI_TRACE_LEVEL '<string>' is invalid. Tracing disabled.

open(<file>) failed: <strerror>. Tracing disabled.
open(<file>): <strerror>

Parsing policy configuration failed!

POLICY: Could not retrieve "pkcs11" group!
POLICY: Could not stat configuration file <file>: <strerror>
POLICY: Configuration file <file> should be owned by "root"
POLICY: Configuration file <file> should have group "pkcs11"!
POLICY: Configuration file <file> has wrong permissions!
POLICY: Unknown curve "<curve>" in line <line>
POLICY: allowedmechs has wrong type!
POLICY: allowedcurves has wrong type!
POLICY: allowedmgfs has wrong type!
POLICY: allowedkdfs has wrong type!
POLICY: allowedprfs has wrong type!
POLICY: Failed to open <file>: <strerror>
POLICY: Could not allocate policy private data!
POLICY: Strength definition <file> failed to parse!
POLICY: Failed to open <file>: <strerror>
POLICY: Policy definition <file> failed to parse!

POLICY VIOLATION: CKM_AES_KEY_GEN needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_AES_KEY_WRAP needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_AES_GCM needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_PKCS5_PBKD2 needed by Token-Store for slot <slot>
POLICY VIOLATION: CKP_PKCS5_PBKD2_HMAC_SHA512 needed by Token-Store for slot <slot>
POLICY VIOLATION: Token-Store encryption method not allowed for slot <slot>!
POLICY VIOLATION: Token-Store requires SHA1 for slot <slot>!
POLICY VIOLATION: Token-Store requires MD5 for slot <slot>!
POLICY VIOLATION: CKM_DES3_KEY_GEN needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_AES_KEY_GEN needed by Token-Store for slot <slot>
POLICY VIOLATION: Unknown Token-Store encryption method for slot <slot>!
POLICY VIOLATION: CKM_AES_KEY_GEN needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_AES_CBC needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_PKCS5_PBKD2 needed by Token-Store for slot <slot>
POLICY VIOLATION: CKP_PKCS5_PBKD2_HMAC_SHA256 needed by Token-Store for slot <slot>
POLICY VIOLATION: Token-Store encryption key too weak for slot <slot>!

read_adapter_config_file: Error: EP 11 config file ''<file>' not found
read_adapter_config_file: Error: EP 11 config file '<file>' is too large
read_adapter_config_file: Error: Expected APQN_ALLOWLIST, APQN_ANY, LOGLEVEL, FORCE_SENSITIVE,
                          CPFILTER, STRICT_MODE, VHSM_MODE, OPTIMIZE_SINGLE_PART_OPERATIONS,
                          PKEY_MODE, DIGEST_LIBICA, or USE_PRANDOM keyword, found '<token>
                          in config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>',
                          expected 'END' or adapter number
read_adapter_config_file: Error: Expected valid adapter number, found '<token>'
                          in config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>,
                          expected domain number (2nd number)
read_adapter_config_file: Error: Expected valid domain number (2nd number), found '<token>'
                          in config file <file>
read_adapter_config_file: Error: Too many APQNs in config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>',
                          expected LOGLEVEL value
read_adapter_config_file: Error: Invalid LOGLEVEL value '<token>' in config file <file>
read_adapter_config_file: Warning: LOGLEVEL setting is not supported any more. Use opencryptoki
                          logging/tracing facilities instead.
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>',
                          expected CP-Filter file name
read_adapter_config_file: Error: CP-Filter config file name '<file>' is too long in
                          config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>,
                          expected libica path, 'DEFAULT', or 'OFF'
read_adapter_config_file: Error: libica path '<token>' is too long in config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>',
                          expected pkey_mode 0 .. 3
read_adapter_config_file: Error: unsupported pkey mode '<token> in config file '<file>'
read_adapter_config_file: Error: At least one APQN mode needs to be present in config file
                          '<file>': APQN_ALLOWLIST or APQN_ANY
read_adapter_config_file: Error: Only one APQN mode can be present in config file '<file>':
                          APQN_ALLOWLIST or APQN_ANY
read_adapter_config_file: Error: At least one APQN needs to be defined in config file '<file>'
read_cp_filter_config_file: Warning: EP 11 CP-filter config file '<file>' does not exist,
                            no filtering will be used
read_cp_filter_config_file: Error: Expected valid control point name or number,
                            found '<ftoken>' in CP-filter config file '<file>'
read_cp_filter_config_file: Error: Expected valid mechanism name or number, found '<token>'
                            in CP-filter config file '<file>'
read_cp_filter_config_file: Error: Out of memory while parsing CP-filter config file '<file>'

SHM segment has wrong gid/mode combination (expected: x/0x; got: x/0x)

start_event_thread: pthread_create failed, errno=x

token_specific_init: Error loading library: 'libcsulcca.so' [<dlerror>]

Trace level x is out of range. Tracing disabled.

Token object <file> appears corrupted (ignoring it)

Tspi_Key_GetPubKey failed: rc=x

Username(<name>) too long

Warning: CCA symmetric master key is not yet loaded
Warning: CCA asymmetric master key is not yet loaded
Warning: Your TPM is not configured to allow reading the public SRK by anyone but the owner.
         Use tpm_restrictsrk -a to allow reading the public SRK
Warning: Adapter <apqn1> has different control points than adapter <apqn2>, using minimum
Warning: Adapter <apqn1> has a different number of control points than adapter <apqn2>,
         using maximum
Warning: Adapter <apqn1> has a different API versionversion than the previous CEXxP adapters: x
Warning: Adapter <apqn1> has a different firmware version than the previous CEXxP adapters: x.x
```

Apply tracing

If you successfully checked all issues as described in the previous sections, you may start to exploit the openCryptoki tracing capabilities. If a log level > 0 is activated by setting the environment variable OPENCRYPTOKI_TRACE_LEVEL, then log entries are written to file `/var/log/opencryptoki/trace.<process_id>`. Application programmers may apply the higher tracing log levels 4 and 5.

For detailed information, read “Logging and tracing in openCryptoki” on page 11.

Final check

Finally, if your openCryptoki environment is successfully set up, you can check your settings using the **pkcsconf** utility as described in Chapter 6, “Managing tokens - pkcsconf utility,” on page 25.

Chapter 18. Configuring a remote PKCS #11 service with openCryptoki

A user scenario shows how to set up a Soft token on a server for use from an application on a remote client.

The user scenario presented in this topic describes how you can set up a remote token on an IBM z15 system with an installed Ubuntu 21.04 Linux environment. This token is accessed and exploited from an application running on an x86 client with an installed Red Hat Enterprise Linux 7.9 environment. The Ubuntu 21.04 setup is selected, because at the time of writing, this distribution shipped all required packages and package versions. An analogous setup is possible with subsequent distributions.

Information about the required set up on the server and client side is presented in the contained subtopics:

- "Server side setup" on page 121
- "Client side setup" on page 123

Server side setup

The user scenario describes how to set up a Soft token on a server, which is an IBM z15 system running a Linux operating system.

Before you begin

The server can be set up on various IBM Z systems and with various versions of Linux. For the scenario illustrated here, it is assumed that you have an Ubuntu 21.04 installation on an IBM z15 machine. Open a Linux command line on the server to set up an openCryptoki Soft token.

Procedure

1. Install the **`p11-kit`** package.

   This tool provides a way to load and enumerate PKCS #11 modules and also provides a standard configuration setup for installing PKCS #11 modules in such a way that they are discoverable.

   To install the **`p11-kit`** package and the **`p11tool`**, enter the following command:

   ```
   # apt install p11-kit p11-kit-modules gnutls-bins
   ```

2. Create and edit an `opencryptoki.module` configuration file in the shown filepath: `/etc/pkcs11/modules/opencryptoki.module`

   Enter the following line into this configuration file:

   ```
   module: /lib64/opencryptoki/libopencryptoki.so
   ```

3. To list the available PKCS #11 modules, enter the following command:

   ```
   # p11-kit list-modules
   ```

   You will see an output similar to the following:

```
p11-kit-trust: p11-kit-trust.so
    library-description: PKCS#11 Kit Trust Module
    library-manufacturer: PKCS#11 Kit
    library-version: 0.23
    token: System Trust
        manufacturer: PKCS#11 Kit
        model: p11-kit-trust
        serial-number: 1
        hardware-version: 0.23
        flags:
               write-protected
               token-initialized
opencryptoki: /lib64/opencryptoki/libopencryptoki.so
    library-description: openCryptoki
    library-manufacturer: IBM
    library-version: 3.16
    token: soft
        manufacturer: IBM
        model: Soft
        serial-number:
        flags:
               rng
               login-required
               user-pin-initialized
               clock-on-token
               token-initialized
```

4. To list the available tokens using the **p11tool** utility, enter the following command:

```
# p11tool --list-tokens
```

You will see an output similar to the following:

```
Token 0:
        URL: pkcs11:model=p11-kit-
trust;manufacturer=PKCS%2311%20Kit;serial=1;token=System%20Trust
        Label: System Trust
        Type: Trust module
        Flags: uPIN uninitialized
        Manufacturer: PKCS#11 Kit
        Model: p11-kit-trust
        Serial: 1
        Module: p11-kit-trust.so

Token 1:
        URL: pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft
        Label: soft
        Type: Generic token
        Flags: RNG, Requires login
        Manufacturer: IBM
        Model: Soft
        Serial:
        Module: opencryptoki: /lib64/opencryptoki/libopencryptoki.so
```

As you can see in the example, the Soft token is available now as **Token 1**. With the shown URL, you can access this token.

5. To start the **p11-kit** server to allow remote clients to access the token, enter the following command:

```
# p11-kit server --provider opencryptoki: /lib64/opencryptoki/libopencryptoki.so
"pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft"
```

As output, you will see the following generated commands:

```
P11_KIT_SERVER_ADDRESS=unix:path=/run/user/0/p11-kit/pkcs11-1296159; export
P11_KIT_SERVER_ADDRESS;
P11_KIT_SERVER_PID=1296160; export P11_KIT_SERVER_PID;
```

6. To set and export the following two environment variables, copy and paste the commands from the output from step 5 and enter them into a command line:

```
# P11_KIT_SERVER_ADDRESS=unix:path=/run/user/0/p11-kit/pkcs11-1296159; export
P11_KIT_SERVER_ADDRESS;
# P11_KIT_SERVER_PID=1296160; export P11_KIT_SERVER_PID;
```

Results

You can now continue to set up the client as described in "Client side setup" on page 123.

Client side setup

Learn how to set up an x86 client in the client-server environment illustrated in this user scenario, so that you can exploit a Soft token, previously installed on a remote server.

Before you begin

It is assumed that you want to access and exploit the functions of the remote Soft token from an x86 client running under a Linux system from a Red Hat Enterprise Linux 7.9 distribution.

Procedure

1. Open a Linux command line. To install the `p11-kit` utility, enter the following command:

```
$ sudo yum install p11-kit
```

2. To query the user run-time path, enter the following command:

```
$ systemd-path user-runtime
```

   You will see an output similar to the following:

```
/run/user/1000
```

3. To forward the local UNIX socket to the remote socket, enter the following commands, using the information from step 2 and then log in as a root user into the remote server:

```
$ mkdir /run/user/1000/p11-kit/
$ ssh -L /run/user/1000/p11-kit/pkcs11-1296159:/run/user/0/p11-kit/pkcs11-1296159
root@<remote_server_name>
```

4. To export the **`p11-kit`** server address environment variable, enter the following command:

```
$ P11_KIT_SERVER_ADDRESS=unix:path=/run/user/1000/p11-kit/pkcs11-1296159; export
P11_KIT_SERVER_ADDRESS;
```

5. As the Red Hat Enterprise Linux 7.9 distribution does not package the `p11-kit-client.so` file, you need to build it from the source. Therefore, clone the shown GitHub repository. To achieve this, enter the following command sequence:

```
$ git clone https://github.com/p11-glue/p11-kit.git
$ cd p11-kit
$ git checkout 0.23.10
$ ./autogen.sh
$ ./configure
$ make
```

6. To view a list of available tokens, use the **p11tool**:

```
$ p11tool --provider /<path>/p11-kit-client.so --list-tokens
```

   You will see an output similar to the following, showing that the Soft token is remotely available.

```
Token 0:
        URL: pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft
        Label: soft
        Type: Generic token
        Manufacturer: IBM
        Model: Soft
        Serial:
```

7. To view a list of available mechanisms of the Soft token, use the **p11tool** utility:

```
$ p11tool --provider /<path>/p11-kit-client.so --list-mechanisms
"pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft"
```

   You will see an output list similar to the following (see also "PKCS #11 mechanisms supported by the Soft token" on page 75):

```
[0x0000] CKM_RSA_PKCS_KEY_PAIR_GEN
[0x0120] CKM_DES_KEY_GEN
[0x0131] CKM_DES3_KEY_GEN
[0x0001] CKM_RSA_PKCS
[0x0006] CKM_SHA1_RSA_PKCS
[0x0040] CKM_SHA256_RSA_PKCS
[0x0041] CKM_SHA384_RSA_PKCS
[0x0042] CKM_SHA512_RSA_PKCS
[0x000d] CKM_RSA_PKCS_PSS
[0x0003] CKM_RSA_X_509
[0x0009] CKM_RSA_PKCS_OAEP
[0x0005] CKM_MD5_RSA_PKCS
[0x0006] CKM_SHA1_RSA_PKCS
[0x0020] CKM_DH_PKCS_KEY_PAIR_GEN
[0x0121] CKM_DES_ECB
[0x0132] CKM_DES3_ECB
[0x0134] CKM_DES3_MAC
[0x0220] CKM_SHA_1
[0x0221] CKM_SHA_1_HMAC
[0x0250] CKM_SHA256
[0x0251] CKM_SHA256_HMAC
[0x0260] CKM_SHA384
[0x0261] CKM_SHA384_HMAC
[0x0270] CKM_SHA512
[0x0271] CKM_SHA512_HMAC
[0x0210] CKM_MD5
[0x0211] CKM_MD5_HMAC
[0x0370] CKM_SSL3_PRE_MASTER_KEY_GEN
[0x0380] CKM_SSL3_MD5_MAC
[0x0381] CKM_SSL3_SHA1_MAC
[0x1080] CKM_AES_KEY_GEN
[0x1081] CKM_AES_ECB
[0x1083] CKM_AES_MAC
[0x0350] CKM_GENERIC_SECRET_KEY_GEN
[0x1040] CKM_ECDSA_KEY_PAIR_GEN
[0x1041] CKM_ECDSA
[0x1042] CKM_ECDSA_SHA1
```

8. Use the **p11tool** utility to issue the following command to generate an RSA private and public key pair of a length of 2048 bits:

```
$ p11tool --provider /<path>/p11-kit-client.so --generate-rsa --bits 2048 --login
"pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft"
```

You will see an output similar to the following;

```
warning: no --outfile was specified and the generated public key will be printed on screen.
note: in some tokens it is impossible to obtain the public key in any other way after
generation.
warning: Label was not specified. Label: my-rsa-key Token 'soft' with URL
'pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft' requires user PIN
Enter PIN:
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzwrYewbV0LybCcb9inQ4
1n/jReFtjrYGx2M4B373em+gMiaDlc+T8Y9yvofDoEwZkjN2OOkUPD2GFb8P88a5
jGF8M+FlkZe+E7XlcHvttFPlULHDpAIXK0UnZJrbAR1ncP8O9lKqhV3CdrXw8dwm
ovdG/FVCyaKv4IlGVj4OKwx5IL0L9JBoSluRRtPNqwSYrXKGEYUjfko+PXm7MVuu
DQv2Ckr6KDEnIsk8U7W9hOHWfjZ4OVKSpbqPlRmG5whWL/hYoGQ181IDXeMajH/1
KgQAI7ree8JS2R4/Os0fzR7+Rp6AvpE4BQ6rXZOkO/7EQLbiCSq930TWsE9IEbMT
xQIDAQAB
-----END PUBLIC KEY-----
```

9. Issue the following command to list all available objects in the token:

```
$ p11tool --provider /<path>/p11-kit-client.so --list-all --login
"pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft"
```

You are prompted for your user PIN:

```

Token 'soft' with URL 'pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft' requires user PIN
 Enter PIN: <USER PIN>
 [...]

Object 6:
         URL:
pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft;id=%8a%b8%84%b3%f0%60%1c%32%2e%19%6e%f1%5
5%7f%30%e3%bf%6c%f3%82;object=my-rsa-key;type=private
         Type: Private key
         Label: my-rsa-key
         Flags: CKA_WRAP/UNWRAP; CKA_PRIVATE; CKA_SENSITIVE;
         ID: 8a:b8:84:b3:f0:60:1c:32:2e:19:6e:f1:55:7f:30:e3:bf:6c:f3:82

Object 7:
         URL:
pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft;id=%8a%b8%84%b3%f0%60%1c%32%2e%19%6e%f1%5
5%7f%30%e3%bf%6c%f3%82;object=my-rsa-key;type=public
         Type: Public key
         Label: my-rsa-key
         Flags: CKA_WRAP/UNWRAP;
         ID: 8a:b8:84:b3:f0:60:1c:32:2e:19:6e:f1:55:7f:30:e3:bf:6c:f3:82
```

Results

On your client, you can now write cryptographic applications that exploit the mechanisms of the Soft token using the openCryptoki API (see also Chapter 17, “Programming with openCryptoki,” on page 101).

References

To learn more about the use and features of openCryptoki, you can read the referenced literature.

- `PKCS #11 openCryptoki for Linux HOWTO`
- `Exploiting Enterprise PKCS #11 using openCryptoki`
- `Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide`
- `libica Programmer's Reference`
- `PKCS #11 Cryptographic Token Interface Usage Guide Version 2.40`

OASIS Standards

- `PKCS #11 Cryptographic Token Interface Base Specification Version 3.0`
- `PKCS #11 Cryptographic Token Interface Profiles Version 3.0`
- `PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0`
- `PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 3.0`

Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Documentation accessibility

The Linux on Z and LinuxONE publications are in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when you use the PDF file and want to request a Web-based format for this publication send an email to eservdoc@de.ibm.com or write to:

IBM Deutschland Research & Development GmbH
Information Development
Department 3282
Schoenaicher Strasse 220
71032 Boeblingen
Germany

In the request, be sure to include the publication number and title.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility at

```
www.ibm.com/able
```

Notices

Trademarks

Index

SC34-7730-01