Linux on Z and LinuxONE

*openCryptoki - An Open Source
Implementation of PKCS #11*

IBM

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, "User scenarios," on page 83 documents the basic structure of openCryptoki applications and documents 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.

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," on page 3
- Chapter 2, "Architecture and components of openCryptoki," on page 5
- Chapter 3, "Installing openCryptoki ," on page 15
- Chapter 4, "Adjusting the openCryptoki configuration file," on page 17
- Chapter 5, "openCryptoki environment variables," on page 21

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 called *Cryptoki*, defined by PKCS #11.

openCryptoki is the open source implementation of this API and 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 possibly different tokens. Due to the popularity of PKCS #11, many software products that perform cryptographic operations, provide plug-in mechanisms that 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 to control the number of token slots provided 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:
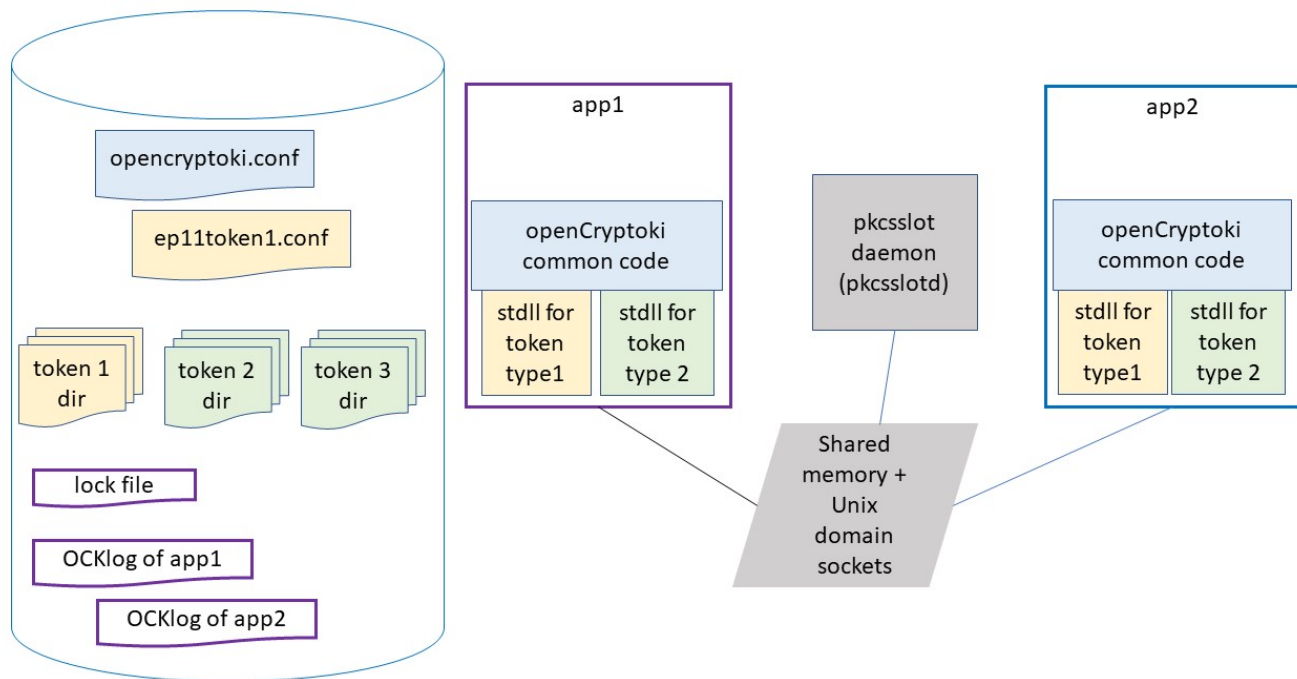
Figure 1. openCryptoki architecture

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 7 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 73) 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;
 }
```

Slot manager

The slot manager daemon (**pkcsslotd**) manages slots (and therefore tokens) 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 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-rw-. 1 root root 82792 Apr  8 12:04 usr.local.var.lib.opencryptoki.ccatok
-rw-rw-rw-. 1 root root 82792 Apr  8 12:04 usr.local.var.lib.opencryptoki.ep11tok
-rw-rw-rw-. 1 root root 82792 Apr  8 12:04 usr.local.var.lib.opencryptoki.lite /* legacy name of
ICA Tok */
-rw-rw-rw-. 1 root root 82792 Apr  8 12:04 usr.local.var.lib.opencryptoki.swtok
...
```

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 25.

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

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 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.
- `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.
- `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.

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).
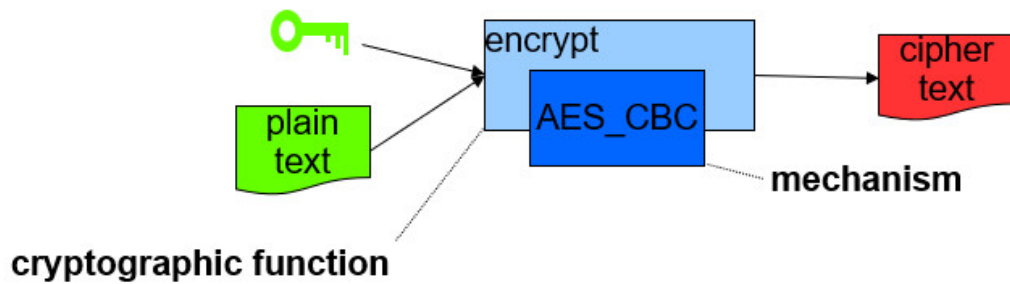

Figure 2. Functions and mechanisms

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()`.

It is token specific which functions are supported by a token. 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 mechanisms 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

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.
- Private objects can only be accessed by the PKCS #11 *Users* if they have logged into the token, whereas public objects can always be access by both *User* and *security officer*.
- Read-only objects cannot be modified whereas read-write objects can.

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 77).

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 three previously mentioned functions, you can use `C_CreateObject()` with all key specific attributes defined. 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 also defines objects for certificates. However no functions to operate on certificates 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 /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.
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 toapplication programmers . This level must be enabled via option `--enable-debug` in the **configure** script at build time.

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.

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 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 by letting threads busily waiting for a required spinlock.

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.

For further information on openCryptoki system resources, read https://github.com/opencryptoki/opencryptoki/blob/master/doc/system_resources

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 appropriatecryptographic coprocessor . The resulting request output is sent back to the application just the other way round through the layer interfaces.
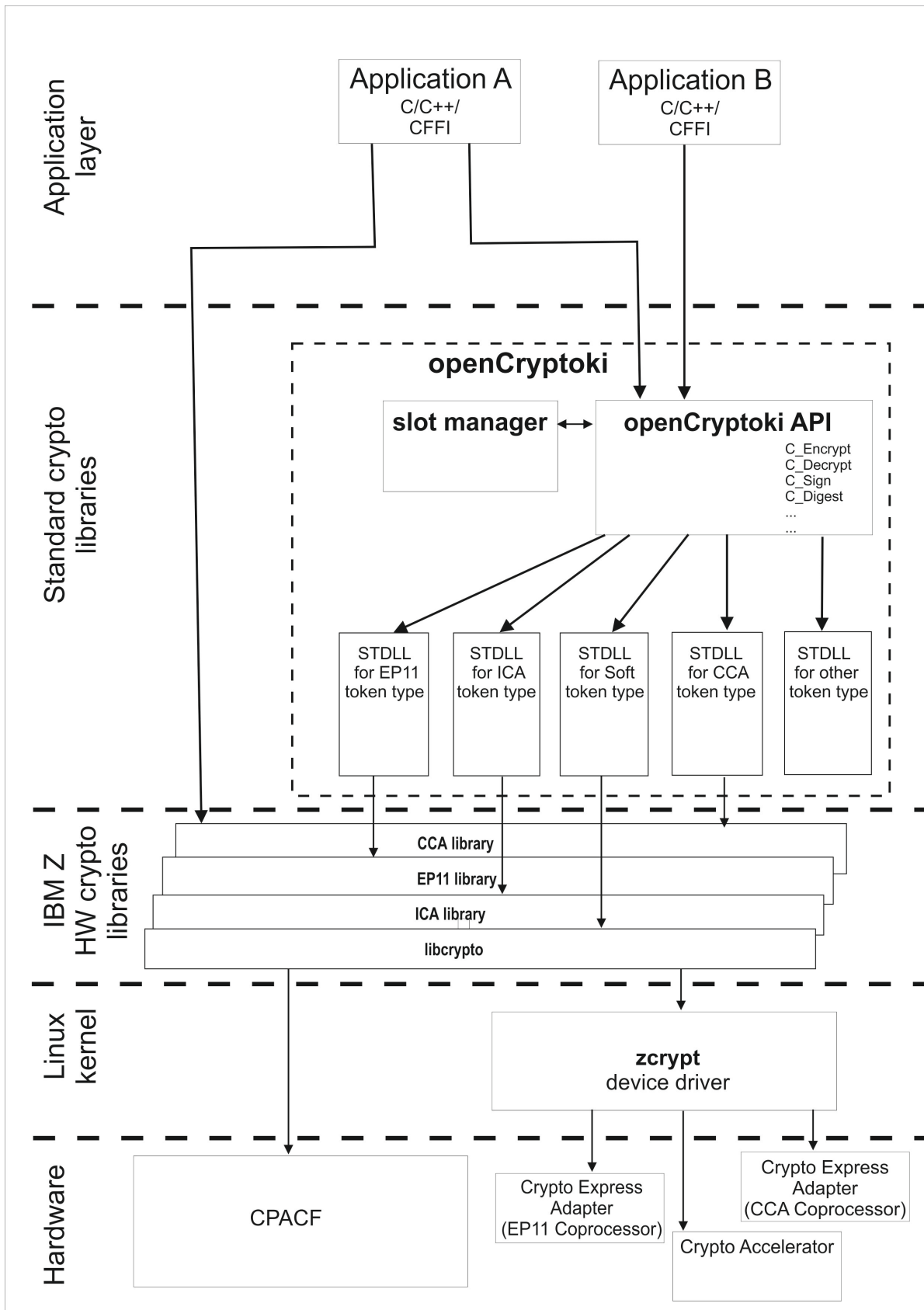
Figure 3. Linux on Z and LinuxONE crypto stack

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.

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 must remove any previous package of openCryptoki, before you can install a package for a new openCryptoki version. Depending on the tokens to be used further libraries need to be installed and cryptographic adapters must be enabled.
- Each token needs 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 maximum number of 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
```

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
`/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. By default, one such entry is available upon installation (see the `slot 4` entry in the provided sample configuration from Figure 4 on page 18).

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

```
version opencryptoki-3.16

# 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.
#
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
}
```

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.

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

openCryptoki provides a command line program (`/usr/lib/pkcs11/methods/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 ?` into a command line to show the options for the `pkcsconf` command:

```
# pkcsconf ?
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
```

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. Listing and generating 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
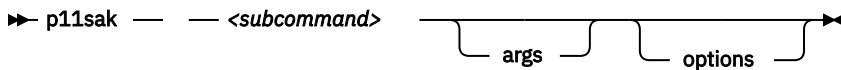
where

subcommand
> is described in the following sections:
>
> - "Generate a key" on page 29
> - "List 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.

Generate a key

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 ──┤       │
                            └─── ec ───┬─ prime256v1 ──────┤
                                       ├─ secp384r1 ───────┤
                                       └─ secp521r1 ───────┘

►─── --slot <slot ID> ─┬──────────────────┬─── --label <key name> ─┬────────────────────────────┬─►
                       └─ --pin <PIN> ─────┘                        └─ --exponent <number> ──────┘

►─┬──────────────────────────────────────────────┬─►◄
  └─ --attr [M | R | L | S | E | D | G | V | W | U | A | X | N] ─┘
```
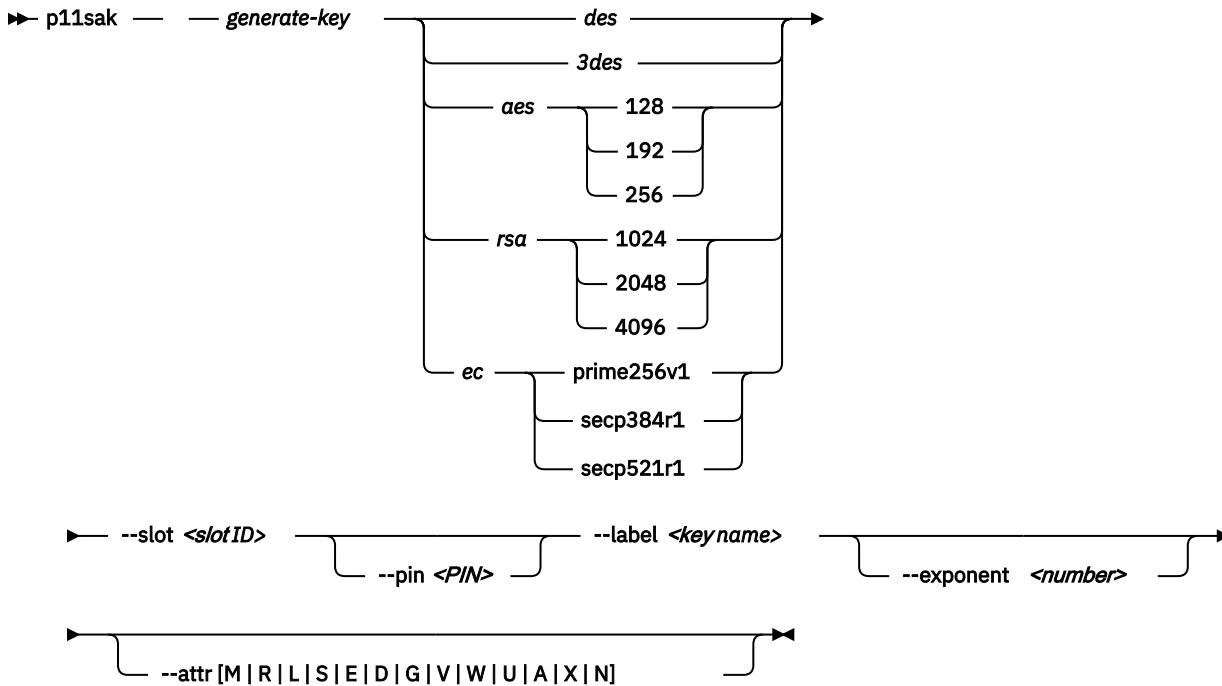
where

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.

ec
> specifies an elliptic curve key pair (private and public) to be generated. You must specify an elliptic curve for the ECC key pair: prime256v1, secp384r1, or secp521r1.

--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.

--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.

--attr
> This option is optional and sets one or more of the binary key attributes to TRUE if the respective letters from the subsequent list is specified. If an attribute is not set explicitly, the default values are used. For multiple attributes add the respective letters without white space, for example: MLD.

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.

List 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.

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 "Generate a key" on page 29. The **--long** or **-l** parameter produces the long output format. If omitted, you obtain a short output format.

Also, you can select either a short or a long listing format.

p11sak syntax - Listing keys

```
▶▶─ p11sak ── ── list-key ───────────────────────────▶
                              ├─── des ───┤
                              ├─── 3des ──┤
                              ├─── aes ───┤
                              ├─── rsa ───┤
                              ├─── ec ────┤
                              ├── public ─┤
                              ├── private ┤
                              └── secret ─┘

▶─── --slot <slotID> ── --pin <PIN> ───────────────────◀◀
                                    └── --long, -l ──┘
```
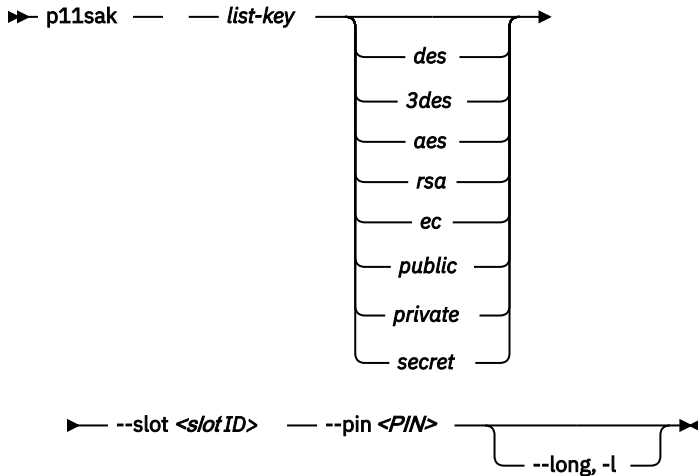
In addition to the **generate-key** subcommand, there are the following options:

public
> selects only public keys to be listed.

private
> selects only private keys to be listed.

secret
> selects only secret keys to be listed.

Example for a listing output in short format (default)

```
# p11sak list-key aes --slot 1

| 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 | AES 128 | 2020-04-29_aes128-M-1.key
| 1 0 1 0 0 0 0 0 0 0 0 0 1 | AES 128 | 2020-04-29_aes128-M-2.key
| 0 0 1 0 0 0 0 0 0 0 0 0 1 | AES 128 | 2020-04-29_aes128-2.key
| 0 0 1 0 0 0 0 0 0 0 0 0 1 | AES 128 | 2020-04-29_aes128-2.key
| 1 1 1 0 0 0 0 0 0 0 0 0 1 | AES 128 | 2020-04-29_aes128-MR.key
| 1 1 1 1 1 0 0 0 0 0 0 1 1 | AES 128 | 2020-04-29_aes128-MRLSE.key
| 1 1 1 1 1 1 1 1 1 1 1 1 0 | AES 128 | 2020-04-29_aes128-MRLSEDGVWUAX
```

Example for a listing output in long format

```
# ./p11sak list-key 3des --slot 1 --pin 11223344 -l
Label: 2020-04-04_3DES.key
Key: 3DES
Attributes:
CKA_TOKEN: CK_TRUE
CKA_PRIVATE: CK_TRUE
CKA_MODIFIABLE: CK_TRUE
CKA_DERIVE: CK_FALSE
CKA_LOCAL: CK_TRUE
CKA_SENSITIVE: CK_FALSE
CKA_ENCRYPT: CK_TRUE
CKA_DECRYPT: CK_TRUE
CKA_SIGN: CK_TRUE
CKA_VERIFY: CK_TRUE
CKA_WRAP: CK_TRUE
CKA_UNWRAP: CK_TRUE
CKA_EXTRACTABLE: CK_TRUE
CKA_ALWAYS_SENSITIVE: CK_FALSE
CKA_NEVER_EXTRACTABLE: CK_FALSE

Label: 2020-04-16_2_des.key
Key: 3DES
Attributes:
CKA_TOKEN: CK_TRUE
CKA_PRIVATE: CK_TRUE
CKA_MODIFIABLE: CK_TRUE
CKA_DERIVE: CK_FALSE
CKA_LOCAL: CK_TRUE
CKA_SENSITIVE: CK_FALSE
CKA_ENCRYPT: CK_TRUE
CKA_DECRYPT: CK_TRUE
CKA_SIGN: CK_TRUE
CKA_VERIFY: CK_TRUE
CKA_WRAP: CK_TRUE
CKA_UNWRAP: CK_TRUE
CKA_EXTRACTABLE: CK_TRUE
CKA_ALWAYS_SENSITIVE: CK_FALSE
CKA_NEVER_EXTRACTABLE: CK_FALSE

...
```
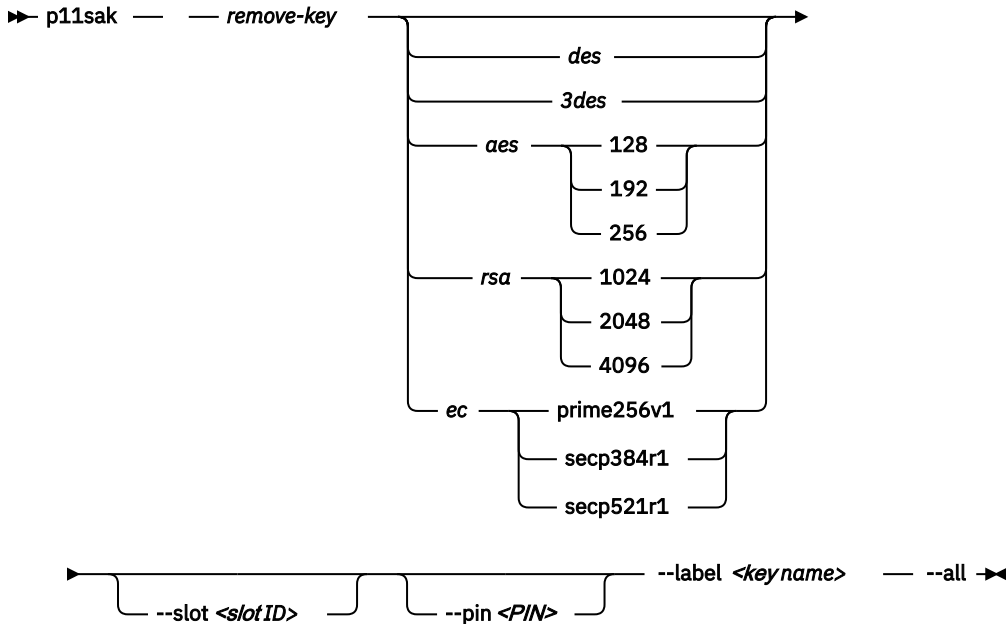
Remove a key

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.

The arguments 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. If the key type is given, it must be used with the **--all** option to delete all keys of the respective cipher key type. That is, the parameter **--all** cannot be used on its own but must always be specified together with a key type.

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

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

Request help for a subcommand, for example:

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

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

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

Args:
    des
    3des
    aes [128 | 192 | 256]
    rsa [1024 | 2048 | 4096]
    ec [prime256v1 | secp384r1 | secp521r1]

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, that you want to specify with the **args** parameter, for example, for a DES key:

```
# p11sak generate-key des -h

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

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]  set key attributes
    -h, --help                        Show this help
```

Chapter 8. Migrating to FIPS compliance - `pkcstok_migrate` utility

Use the **pkcstok_migrate** tool 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.

Walk through

A walk through to perform a transformation of a token into the FIPS compliant data format may consist of the following commands:

```
killall pkcsslotd
echo -e "y\n" | pkcstok_migrate --verbose debug --slot $SLOT --sopin $PKCS11_SO_PIN
```

```
                                --userpin $PKCS11_USER_PIN
                                --confdir $OCK_CONFDIR --datastore $OCK_DATASTORE
pkcsslotd
```

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.

Each company which collaborates with the openCryptoki open source community can contribute their own specific mechanisms to openCryptoki. For example, all mechanisms added to the PKCS #11 standard by IBM 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 5 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 5. 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 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 in the global openCryptoki configuration file called `opencryptoki.conf`. This section describes in detail how to add tokens of type EP11 (EP11 tokens), because this covers all possible features. Perform the same actions analogously for other token types. Wherever a feature is only available for EP11 tokens, this is indicated.

Note: Currently, for EP11 tokens only, you can configure each token instance differently using a specific EP11 token configuration file (see).

For example, if you want to configure multiple EP11 tokens, you can assign dedicated adapters and domains to different tokens respectively. This ensures data isolation between multiple applications.

If you use multiple EP11 tokens, 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 EP11 token is separated from other EP11 tokens.

The default EP11 token directory is `/var/lib/opencryptoki/ep11tok/`. You can use the default only for a single EP11 token . Resulting examples for multiple EP11 token directories can be:

```
/var/lib/opencryptoki/ep11token1/
/var/lib/opencryptoki/ep11token2/
```

Note: A certain token configuration applies to all applications that use this EP11 token.

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 configures two EP11 tokens in slots 4 and 5. 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 7 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 6. 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 7. Sample of an EP11 token configuration file

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 ICA token" on page 54

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

Chapter 10. CCA token

A CCA token is a secure key token. Any key generation is processed in a 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

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_DES_CBC	8-8 bytes	ECNCRYPT, DECRYPT	before 3.16
CKM_DES_CBC_PAD	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES3_CBC	24-24 bytes	ENCRYPT, ECRYPT	before 3.16
CKM_DES3_CBC_PAD	24-24 bytes	ENCRYPT, ECRYPT	before 3.16
CKM_AES_KEY_GEN	16-32 bytes	GENERATE	before 3.16
CKM_AES_ECB	16-32 bytes	ENCRYPT, ECRYPT	before 3.16
CKM_AES_CBC	16-32 bytes	ENCRYPT, ECRYPT	before 3.16
CKM_AES_CBC_PAD	16-32 bytes	ENCRYPT, ECRYPT	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)

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.

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
  – CCA RSA public key token
  – 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 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 command output shown in Table 5 on page 51 displays all mechanisms supported by the ICA token on an IBM z15™ machine.

Table 5. 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	GENERATE_KEY_PAIR	before 3.16
CKM_RSA_PKCS	512-4096 bits	ENCRYPT, DECRYPT, SIGN, SIGN_RECOVER, VERIFY, VERIFY_RECOVER, WRAP, UNWRAP	before 3.16
CKM_RSA_X_509	512-4096 bits	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	ENCRYPT, DECRYPT, 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	before 3.16
CKM_SHA256_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_RSA_PKCS	512-4096 bits	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	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
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_SHA512_224	n/a	DIGEST	before 3.16

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
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_DES_KEY_GEN	8-8 bytes	GENERATE	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_KEY_GEN	24-24 bytes	GENERATE	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
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	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	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_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

Table 5. Supported mechanism list for the ICA token (continued)

Table 5. Supported mechanism list for the ICA token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
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_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_GENERIC_SECRET_KEY_GEN	80-2048 bits	GENERATE	before 3.16
CKM_KEY_WRAP_LYNKS	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_ECDSA_SHA224	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_ECDSA_SHA256	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_ECDSA_SHA384	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_ECDSA_SHA512	160-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE	before 3.16
CKM_ECDH1_DERIVE	160-521 bits	_DERIVE|CKF_EC_F_P| CKF_EC_NAMEDCURVE	before 3.16
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_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	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16

Table 5. Supported mechanism list for the ICA token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_AES_CTR	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_GCM	16-32 bytes	ENCRYPT, DECRYPT	before 3.16
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

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 6 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 running in FIPS mode?

Table 6. 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
ed448 **(2)**	• Sign/Verify with CKM_IBM_ED448_SHA3
ed25519 **(2)**	• Sign/Verify with CKM_IBM_ED25519_SHA512
X448 **(2)**	• ECDH with CKM_IBM_EC_C448
X25519 **(2)**	• ECDH with CKM_IBM_EC_C25519

Table 6. Curves supported by the ICA token for elliptic curve cryptography (ECC) (continued)	
Curve	**Purpose**
Notes:	

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.

APQN_WHITELIST

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_WHITELIST. 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.

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_WHITELIST
 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_WHITELIST
 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 , adapter 0 with domains 0 and 1, and adapter 2 with domain 84 are specified as target for requests from the EP11 token. In , these adapter/domain pairs are shown in hexadecimal notation as APQNs (00,0000), (00,0001), and (02,0054).
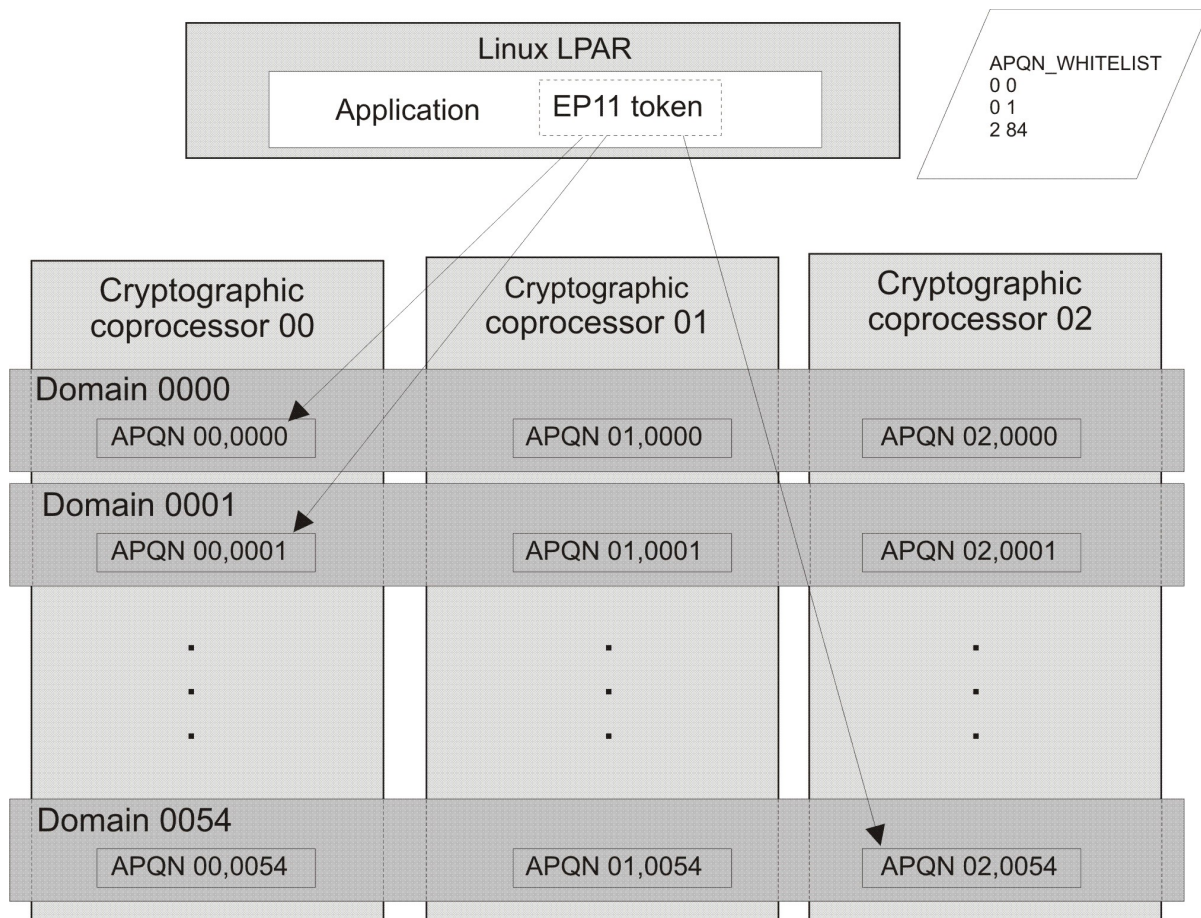
Figure 8. Cryptographic configuration for an LPAR

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 66.

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 Exploiting Enterprise PKCS #11 using openCryptoki.

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.

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
```

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
```

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).

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.

Sample of an EP11 token configuration file

```
#
# 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 9. 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 algorithms (or 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 command output shown in Table 7 on page 61. The mechanism ID and 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. 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 7. 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

Table 7. PKCS #11 mechanisms supported by the EP11 token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
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
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

Table 7. PKCS #11 mechanisms supported by the EP11 token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
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_NAMEDCURVE, EC_UNCOMPRESS	before 3.16
CKM_ECDSA	192-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA1	192-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE, 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

Table 7. PKCS #11 mechanisms supported by the EP11 token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
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_NAMEDCURVE, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA256	192-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA384	192-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA512	192-521 bits	SIGN, VERIFY, EC_F_P, EC_NAMEDCURVE, EC_UNCOMPRESS	before 3.16
CKM_IBM_EC_C25519	256-256 bytes	DERIVE, EC_F_P, EC_UNCOMPRESS	before 3.16
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

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 61.

Table 8. 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
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

Table 8. Curves supported by the EP11 token for elliptic curve cryptography (ECC) (continued)	
Curve	**Purpose**
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
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 10 on page 67. 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 10. 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 9 on page 67 and Table 10 on page 67 show the EP11 token-specific attributes and their default values for private and secure keys.

Table 9. Private key (CKO_PRIVATE_KEY) default attributes of the EP11 token

Private key attributes	value
CKA_SENSITIVE	CK_TRUE
CKA_EXTRACTABLE	CK_TRUE

Table 10. 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 CCA 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 EP11cryptographic 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. For more information about EP11 session, read *Exploiting Enterprise PKCS #11 using openCryptoki*.

Per default, the EP11 token stores all token key objects in the Linux file system in the /var/lib/ opencryptoki/ep11tok/TOK_OBJ directory.

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 69 unless step "4" on page 69 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 69 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 embedded 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 (/usr/local/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.

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 card-PIN required for the *VHSM mode*.

pkcsep11_session **usage examples**

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

```
pkcsep11_session show -pid 1234
```

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

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

- Logout all left over EP11 session:

```
pkcsep11_session logout
```

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

```
pkcsep11_session logout -pid 1234
```

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

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

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

```
pkcsep11_session logout -force
```

- Set a card-PIN:

```
pkcsep11_session cardpin
```

The card-PIN must be between 8 and 16 characters in length.

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).

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 11 on page 73 displays all mechanisms that are supported by the Soft token. The mechanism ID and 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. 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 11. 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
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	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS_PSS	1024-4096 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

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
Table 11. PKCS #11 mechanisms supported by the Soft token (continued)			
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_RSA_X_509	512-4096 bits	ENCRYPT, DECRYPT, SIGN, SIGN RECOVER, VERIFY, VERIFY RECOVER, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_OAEP	1024-4096 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
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

Table 11. PKCS #11 mechanisms supported by the Soft token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
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_256	n/a	DIGEST	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
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_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_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
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	GENERATE	before 3.16

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
/usr/local/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 11. 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 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 12.				
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.

Part 4. 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 16, "Programming with openCryptoki," on page 85 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 17, "Configuring a remote PKCS #11 service with openCryptoki," on page 91 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 16. 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_

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 pkcs11.h    /* top-level Cryptoki include file, in turn includes pkcs11t.h and
   pkcs11f.h */
   #include <pkcs11types.h>
   #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);
   );
   ```

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

 rv = fn->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 of .

```
/*
 * 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;
}
```

Chapter 17. 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 selected 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

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: /usr/local/lib/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: /usr/local/lib/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: /usr/local/lib/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 /usr/local/lib/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

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 <u>"PKCS #11 mechanisms supported by the Soft token" on page 73</u>):

```
[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%55%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%55%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 16, "Programming with openCryptoki," on page 85).

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

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at
www.ibm.com/legal/copytrade.shtml

Adobe is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Index

STRICT_MODE 57

T

template
    openCryptoki application 85
token
    initializing 26
token-specific configurations 39, 83
tools
    common for openCryptoki 23
    pkcscca 48
    pkcsconf 25
    pkcsep11_migrate 68
    pkcsep11_session 70
    pkcsficsf 81
    pkcstok_migrate 37

U

Ubuntu 21.04 91
usage notes
    ICA library 54
usage notes for CCA library functions 47
User
    log-in PIN 26
user scenario
    client side setup 93
    server side setup 91
utilities
    pkcsep11_migrate 68
    pkcsep11_session 70
    pkcstok_migrate 37
utility
    pkcscca 48
    pkcsconf 25

V

VHSM_MODE 57

W

wrapping key 68

IBM ®