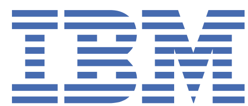
Linux on IBM Z and IBM LinuxONE

openCryptoki - An Open Source Implementation of PKCS #11



SC34-7730-02

Edition notice

This edition applies to openCryptoki version 3.22 and to all subsequent versions and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2021, 2023.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this document..........vii

Summary of changes..........ix

Edition SC34-7730-02: Updates for openCryptoki versions 3.18 - 3.22..........ix

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

Part 1. Common features of openCryptoki..........1

Chapter 1. Introducing PKCS #11 and openCryptoki..........3

What is PKCS #11?..........3

What is openCryptoki?..........4

Chapter 2. Architecture and components of openCryptoki..........5

Part 2. Preparing openCryptoki..........15

Chapter 3. Fastpath to openCryptoki..........17

Chapter 4. Installing openCryptoki..........19

Chapter 5. Adjusting the openCryptoki configuration file..........21

Chapter 6. Supporting cryptographic policies for openCryptoki..........25

Strength configuration file..........25

Policy configuration file..........28

Processing configuration files..........32

Chapter 7. openCryptoki environment variables..........35

Part 3. Common tools of openCryptoki..........37

Chapter 8. Managing tokens - **pkcsconf** utility..........39

Chapter 9. Managing token keys - **p11sak** utility..........43

Chapter 10. Migrating to FIPS compliance - **pkcstok_migrate** utility..........65

Chapter 11. Displaying usage statistics - **pkcsstats** utility..........67

Chapter 12. Managing a concurrent master key change - **pkcshsm_mk_change** utility..........71

Part 4. Token specifications..........79

Chapter 13. Common token information..........81

Adding tokens to openCryptoki..........81

How to recognize tokens..........82

ECC header file..........84

OID file for post-quantum algorithms..........84

How and why to exploit protected keys..........85

How to enable AES XTS support for CCA and EP11 tokens..........87

PKCS #11 Baseline Provider support.......... 88
Dual-function cryptographic functions support.......... 88
Supported features of PKCS #11 3.0 and 3.1.......... 89

Chapter 14. CCA token.......... 91
Defining a CCA token configuration file.......... 91
PKCS #11 mechanisms supported by the CCA token.......... 94
ECC curves supported by the CCA token.......... 96
Usage notes for CCA library functions.......... 97
Migrate to a new CCA master key - **pkcscca** utility.......... 98

Chapter 15. ICA token.......... 101
PKCS #11 mechanisms supported by the ICA token.......... 101
Usage notes for the ICA library functions.......... 104
ECC curves supported by the ICA token.......... 105

Chapter 16. EP11 token.......... 107
Defining an EP11 token configuration file.......... 107
PKCS #11 mechanisms supported by the EP11 token.......... 113
ECC curves supported by the EP11 token.......... 117
Migrating master keys - **pkcsep11_migrate** utility.......... 119
Managing EP11 sessions - **pkcsep11_session** utility.......... 121
Usage notes for the EP11 host library functions.......... 123
Restriction to extended evaluations.......... 124

Chapter 17. Soft token.......... 125
PKCS #11 mechanisms supported by the Soft token.......... 125

Chapter 18. Directory content for CCA, ICA, EP11, and Soft tokens.......... 131

Chapter 19. ICSF token.......... 135
Configuring the ICSF token - **pkcsicsf** utility.......... 135

Part 5. IBM-specific mechanisms and features for openCryptoki.......... 137

Chapter 20. IBM-specific mechanisms.......... 139

Chapter 21. Re-encrypting data with a mechanism.......... 157

Part 6. Programming basics and user scenarios.......... 159

Chapter 22. Programming with openCryptoki.......... 161
How to create and modify objects.......... 162
How to apply attributes to objects.......... 163
Structure of an openCryptoki application.......... 163
Sample openCryptoki program.......... 165
openCryptoki code samples (C).......... 167

Chapter 23. Trouble shooting.......... 179

Chapter 24. Configuring a remote PKCS #11 service with openCryptoki.......... 183
Server side setup.......... 183
Client side setup.......... 185
Support of IBM-specific mechanisms - **p11-kit**.......... 187

References.......... 191

Accessibility... 193

Notices.. 195
Trademarks.. 195

Index.. 197

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 3, “Common tools of openCryptoki,” on page 37 documents management and key migration tools that are helpful for **openCryptoki administrators** to enable openCryptoki exploitation through programs.
- Part 4, “Token specifications,” on page 79 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 5, “IBM-specific mechanisms and features for openCryptoki ,” on page 137 documents numerous mechanisms and features contributed by IBM to the openCryptoki framework. The described mechanisms are available across multiple token-types or for certain tokens only. Described features are normally applicable to all token-types or openCryptoki applications.
- Part 6, “Programming basics and user scenarios,” on page 159 documents the basic structure of openCryptoki applications and presents a simple, but complete code sample for an RSA key pair generation. This part also provides a user scenario showing how to set up a Soft token on a server for use from an application on a remote client.

© Copyright IBM Corp. 2021, 2023

Summary of changes

Track the changes of this document for each new edition.

Edition SC34-7730-02: Updates for openCryptoki versions 3.18 - 3.22

This edition has a new structure. There is a new topic Part 5, “IBM-specific mechanisms and features for openCryptoki ,” on page 137 which lets you find IBM-specific add-ons to openCryptoki at a glance. Existing and new information is integrated into this part as applicable.

- For CCA tokens and EP11 tokens, openCryptoki ensures that all APQNs assigned to such a token are configured with the same master key. In either a CCA token or an EP11 token configuration file, you can optionally specify an expected master key or wrapping key verification pattern (MKVP or WKVP). If specified, all master or wrapping keys must match the pattern, If not specified, they just must be the same. Additionally, the openCryptoki functions that create a key or key pair, for example `C_GenerateKey()`, or `C_DeriveKey()`, now automatically check whether the resulting key or keys are created with this specified expected MKVP or WKVP. If no pattern is specified, openCryptoki checks against the verification pattern of the APQNs as determined during token initialization.
- openCryptoki is extended with a possibility to restrict usage of mechanisms and keys via a global policy. The policy is guided by the notion of cryptographic strength. The user specifies a minimal desired strength, allowed mechanisms, and a way to derive the strength for a given key. openCryptoki then blocks all keys that are not strong enough and mechanisms that are not allowed. The policy is set globally for all applications using openCryptoki.
- The usage of mechanisms is counted based on individual users and accumulated for all users on the respective system. The usage is also counted on the basis if slot-IDs, mechanisms, and key-sizes. Counting is applied transparently to the calling applications.

  A new command line tool `pkcsstats` can be used to display the counter values for slot-IDs and mechanisms, based on individual users or accumulated for all users on the respective system, and also broken down to different key-size values.
- openCryptoki provides a new utility called **`pkcshsm_mk_change`** to support the changing (rolling) of master keys of cryptographic coprocessors running in CCA or EP11 mode concurrently while applications using openCryptoki workload exploiting the CCA token or the EP11 token are running. All secure keys enciphered by old master keys need to be re-enciphered with the new master key. However applications using the mentioned tokens can continue to run while the master key change procedure is performed.
- Support for the CKA_DERIVE_TEMPLATE attribute as defined by PKCS #11 version 3.1 has been added to all token types. The CKA_DERIVE_TEMPLATE attribute of a base key contains a template that is applied to the derived key in addition to the user supplied derive template. Applications can use the CKA_DERIVE_TEMPLATE attribute on base keys to control the attributes of the keys that are to be derived from that base keys.
- The EP11 token offers two new mechanisms that can be used by crypto currency applications:
  - The CKM_IBM_BTC_DERIVE mechanism allows derivation of child keys from base ECC keys, generated from a selection of elliptic curves, using the methods described in *BIP-0032* and *SLIP-0010*.
  - The CKM_IBM_ECDSA_OTHER mechanism is a multi-variant signature mechanism for algorithms used by crypto currency applications. You can use this mechanism to generate Schnorr signatures, that is, ECC-based signatures that are not produced using the ECDSA or the EDDSA digital signature algorithms.

© Copyright IBM Corp. 2021, 2023

- The openCryptoki **pkcsslotd** daemon is hardened, especially against privilege escalation attacks. It does not run as root user anymore. It can still be started as `root` user, but will change its UID to a special, but unprivileged `pkcsslotd` service user shortly after startup.
- For PKCS #11 3.0, openCryptoki supports the new `C_SessionCancel()` function used to terminate active session-based operations.
- New mechanisms are provided by the CCA token, the ICA token, the EP11 token, and the Soft token to enable AES XTS protected-key cryptographic support for openCryptoki (encryption, decryption and key generation). The AES XTS mechanisms and key type are new for PKCS #11 version 3.0.

  Also, a new feature of the **p11sak** utility now supports the generation and management of AES-XTS keys.

  Furthermore, for the mentioned tokens you can now import clear AES XTS keys to secure AES XTS keys, for example, using the `C_CreateObject()` function of openCryptoki.
- Background information about the use of protected keys in general and especially for AES XTS for the CCA and EP11 tokens is provided in the new topics "How and why to exploit protected keys" on page 85 and "How to enable AES XTS support for CCA and EP11 tokens" on page 87.
- The **p11sak** tool now supports x.509 public key certificates as token objects. This is needed for compatibility with the use case where certificates are stored in a PKCS #11 token.
- The openCryptoki tools **pkcsconf** and **p11sak** now support the output of the *unified resource identifier* (URI) as defined by the *RFC 7512: The PKCS #11 URI Scheme* for PKCS #11 objects in PKCS #11 tokens.
- As for EP11 tokens, you can now create an optional CCA token configuration file where you can specify options to support protected key mode and to request a master key consistency check for the assigned APQNs.
- For EP11 tokens, openCryptoki offers support of quantum-safe algorithms for Dilithium Round 2 and 3 variants and Kyber Round 2. All new features are available with IBM z16 systems on a CEX8P Crypto Express EP11 coprocessor, including the latest EP11 host library and firmware code.
- Since version 3.19, openCryptoki supports several functions to perform two cryptographic operations simultaneously within a session. These functions are provided to avoid unnecessarily passing data back and forth to and from a token.
- There are attacks on RSA operations for which the timing of a computation may deliver a hint for well- or malformed input and thus, whether an attack may have a chance to be successful. Starting with version 3.21, openCryptoki itself cares for hiding the computation times, however, PKCS #11 applications must make sure to handle certain RSA decryption errors in a constant-time manner itself. In addition, the ICA token and the Soft token apply RSA message blinding to messages on all operations using the RSA private key (decryption and signature creation).
- The **p11-kit** utility is enhanced so that it supports IBM-specific mechanisms and attributes. The **p11-kit** utility can especially be used to provide remote PKCS #11 API access to openCryptoki tokens through an RPC-like communication protocol.

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

- A new information unit describing IBM-specific mechanisms and attributes has been added (see Chapter 20, "IBM-specific mechanisms," on page 139).
- Conceptional information on the handling of objects and attributes in PKCS #11 has been added.
- openCryptoki code samples have been transferred from *libica Programmer's Reference*, SC34-2602, and *Exploiting Enterprise PKCS #11 using openCryptoki*, SC34-2713, into this current edition.
- All openCryptoki-related information from *libica Programmer's Reference*, SC34-2602, has been integrated into this current edition.
- Comprehensive extensions have been applied to the documentation of all tokens in Part 4, "Token specifications," on page 79.

- A new topic to assist you in solving problems while working with openCryptoki has been included (see Chapter 23, “Trouble shooting,” on page 179).
- The **p11sak** offers a new option to obtain a listing of keys in a long output format.

Part 1. Common features of openCryptoki

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

The following topics explain the purpose of openCryptoki within the PKCS #11 standard and introduce the concept of the openCryptoki framework. In addition, you find information on how to customize and configure openCryptoki according to your requirements.

- Chapter 1, “Introducing PKCS #11 and openCryptoki,” on page 3
- Chapter 2, “Architecture and components of openCryptoki,” on page 5
- Chapter 4, “Installing openCryptoki ,” on page 19
- Chapter 5, “Adjusting the openCryptoki configuration file,” on page 21
- Chapter 6, “Supporting cryptographic policies for openCryptoki,” on page 25
- Chapter 7, “openCryptoki environment variables,” on page 35

© Copyright IBM Corp. 2021, 2023

Chapter 1. Introducing PKCS #11 and openCryptoki

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

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

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

What is PKCS #11?

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

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

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

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

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

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

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

`PKCS #11 Cryptographic Token Interface Standard`

© Copyright IBM Corp. 2021, 2023

What is openCryptoki?

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

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

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

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

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

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

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

Chapter 2. Architecture and components of openCryptoki

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

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

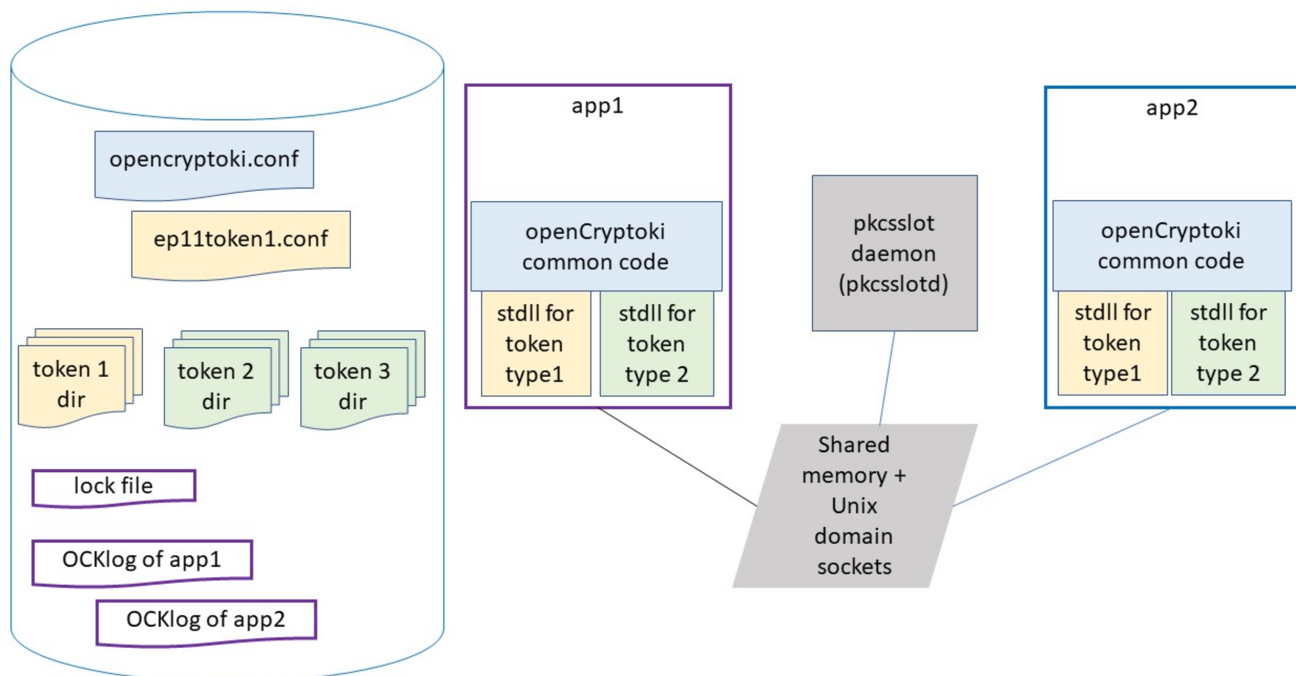


Figure 1. openCryptoki architecture

information between applications to help ensure conformance with the PKCS #11 specification.

Figure 1 on page 5 shows the architecture of openCryptoki:

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 5, "Adjusting the openCryptoki configuration file," on page 21). They can define and exploit multiple tokens of any token type, each with a different token name. EP11 tokens and CCA tokens can be configured using a token-specific configuration file for each token instance. Samples of such configuration files are shown in Figure 19 on page 113 and in Figure 16 on page 93. An example of how to define multiple EP11 tokens in the overall openCryptoki configuration file is shown in Figure 14 on page 82.

© Copyright IBM Corp. 2021, 2023

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.

For information on the location and content of the single token directories read Part 4, “Token specifications,” on page 79.

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 17, “Soft token,” on page 125) 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 22). 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 for a soft token  */

    return CKR_OK;
 }
```

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

Slot manager

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

Starting with openCryptoki 3.21, the **pkcsslotd** daemon does no longer run as `root` user. Though you can still start the daemon as `root` user, it changes its user ID to the `pkcsslotd` user shortly after startup, thus dropping any root privileges. This protects your environment against privilege escalation attacks. Applications must still be running under a user that is a member in the `pkcs11` group (see “Access control and groups” on page 12).

Note: The `pkcsslotd` user as well as the `pkcs11` group are configurable via package configuration, but they default to `pkcsslotd` and `pkcs11`. That way distributions can choose to use a different user and group names if desired.

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

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

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

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

```
/run/opencryptoki/pkcsslotd.socket
/run/opencryptoki/pkcsslotd.admin.socket
```

The `/run/opencryptoki` directory also contains the PID file (process identification file) that the *pkcsslotd* daemon creates.

The `/run/opencryptoki` directory is owned by the *pkcsslotd* user and *pkcs11* group, but only the *pkcsslotd* user is allowed to write `(drwx--x---)`. That way, only the *pkcsslotd* user (or *root* user) is able to start the **pkcsslotd** daemon.

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.

Starting the slot manager

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

Use the following command to start the slot manager daemon. The slot manager then reads out the configuration information and sets up the tokens.

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

For a permanent solution, specify:

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

To start the `pkcsslotd` daemon under the `pkcsslotd` user manually, use the following command:

```
runuser -u pkcsslotd pkcsslotd
```

This command can only be used as root user. Using the command **su pkcsslotd pkcsslotd** does not work, because the pkcsslotd user does not allow a login.

Main API

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

Roles and sessions

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

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

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

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

The PKCS #11 API provides session functions like:

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

Functions and mechanisms

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

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

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

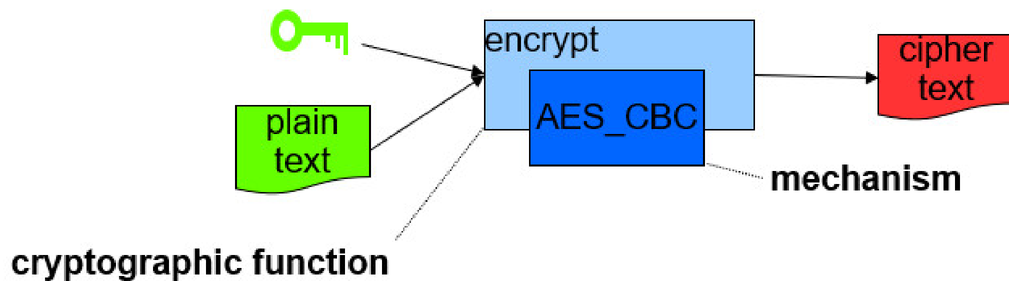


Figure 2. Functions and mechanisms

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

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

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

Slot token dynamic link libraries (STDLLs)

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

Shared memory

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

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

Objects and keys

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

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

An object comprises a set of attributes, each of which has precisely one given value. For an example, have a look at Step 6 of the "Sample openCryptoki program" on page 165, where function `C_GenerateKeyPair()` uses the mechanism CKM_RSA_PKCS_KEY_PAIR_GEN to create an RSA private and public key pair. The attributes are assigned to the public and private keys to be generated with two different templates of type array of CK_ATTRIBUTE (one applicable for the private key and one applicable to the public key), which are input to the function call.

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

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

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

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

Note: Token objects are stored in a token-specific object store, the token directory (see also Chapter 18, "Directory content for CCA, ICA, EP11, and Soft tokens," on page 131).

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

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

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

PKCS #11 defines multiple object classes, for example:

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

Each object class has its own set of attributes, where these attributes define an instance of an object from this class. There is one common attribute called CKA_CLASS for all object classes. This attribute defines the type (or class) of an object. For more information about objects, read "How to create and modify objects" on page 162 and "How to apply attributes to objects" on page 163).

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

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

Access control and groups

To properly configure the system, and to be authorized for performing the openCryptoki processes, for example, running the slot manager daemon **pkcsslotd**, the *pkcs11* group must be defined in file `/etc/group` of the system. Every user of openCryptoki (including the **pkcsslotd** user running the slot manager, and including the users configuring openCryptoki and its tokens) must be members 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 to application programmers. This level must be enabled at build time of the openCryptoki library via option `--enable-debug` in the **configure** script.

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

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

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

Lock files

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

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

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

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

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

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

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

Use and purpose of openCryptoki features

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

Figure 3 on page 14 shows the process flow within the Linux on IBM Z and IBM LinuxONE crypto stack. For example, an application sends an encryption request to the crypto adapter. Through various interfaces, such a request is propagated from the application layer down to the target crypto adapter. On its way down, the request passes through the involved layers: the standard openCryptoki interfaces, the adequate IBM Z crypto libraries, and the operating system kernel. The `zcrypt` device driver finally sends the request to the appropriate cryptographic coprocessor. The resulting request output is sent back to the application just the other way round through the layer interfaces.

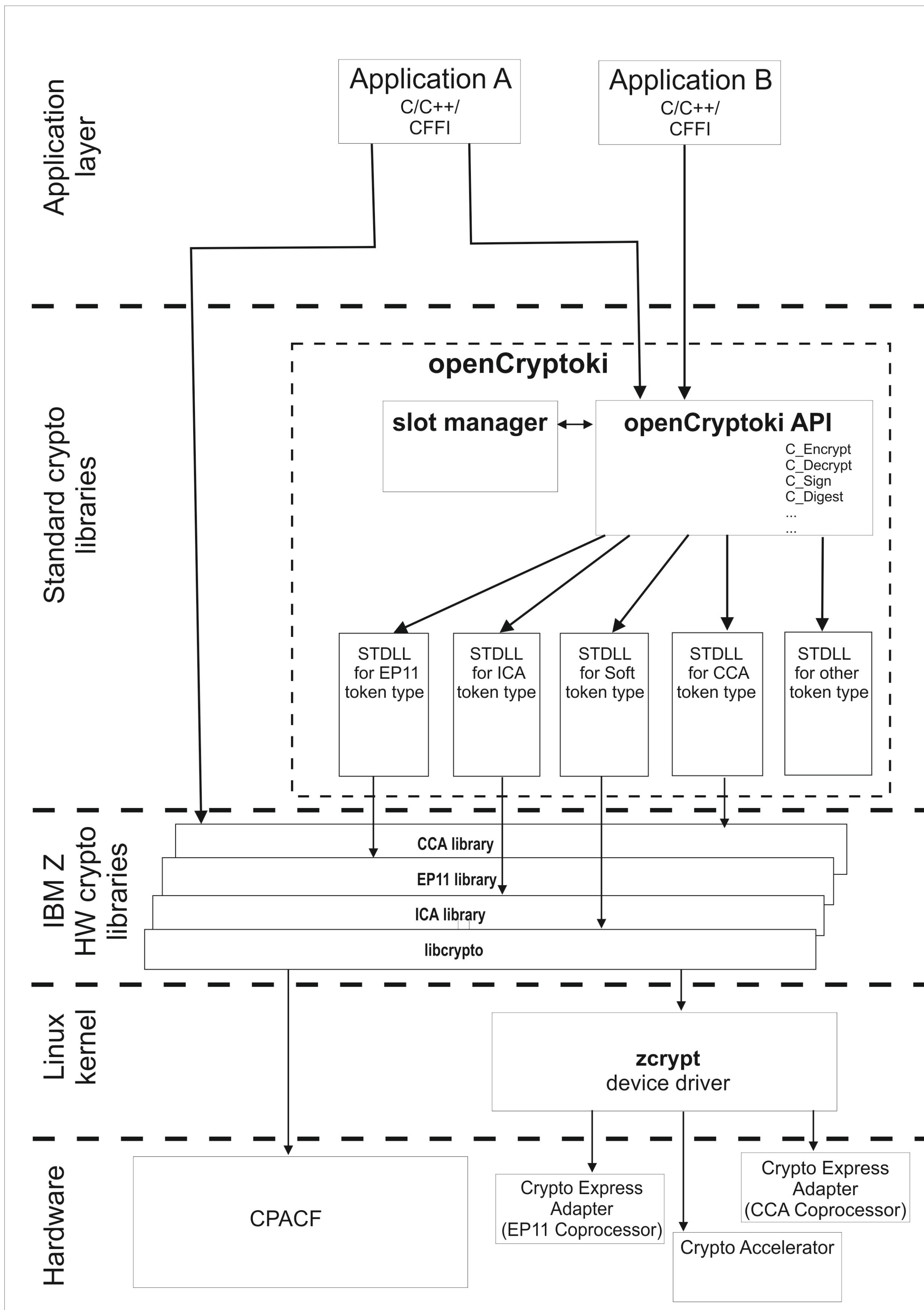


Figure 3. Linux on IBM Z and IBM LinuxONE crypto stack

Part 2. Preparing openCryptoki

The first topic of this part presents a fastpath to the actions required for setting up a running openCryptoki environment. Each step within this fastpath points to a subsequent topic or to other documentation containing a detailed description of the action.

© Copyright IBM Corp. 2021, 2023

Chapter 3. Fastpath to openCryptoki

Read this section for an overview of the steps and a fastpath to the most important actions required for preparing openCryptoki to be used by an application. Each step contains a reference to a detailed description in most cases within this document, or to external documentation.

1. Install openCryptoki and observe the post-installation checks: see Chapter 4, "Installing openCryptoki ," on page 19 and "Post-installation checks" on page 20.
2. Start the **pkcsslotd** slot manager – unless this step is done by the installation: see "Starting the slot manager" on page 8.
3. If you want to use CCA or EP11 tokens you may want to adjust their respective configuration files: see "Defining a CCA token configuration file " on page 91 and "Defining an EP11 token configuration file" on page 107.
4. Before you use a CCA or EP11 token, you must install the CCA or EP11 host package and set the CCA master keys or the EP11 wrapping key in the according domains of the Crypto Express adapter: see *Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide* or *Exploiting Enterprise PKCS #11 using openCryptoki*.
5. Prepare the token you want to use by performing the following sub-steps:

   a. Initialize the token.

   b. Change the SO PIN of the token (required in order to change the default SO PIN applied by openCryptoki during token initialization).

   c. Set the User PIN of the token.

   You can perform all of these sub-steps using the **pkcsconf** utility: see Chapter 8, "Managing tokens - pkcsconf utility," on page 39.
6. Add the user(s) of processes that use openCryptoki to the`pkcs11` group: see "Access control and groups" on page 12.
7. **Optional:** Apply global policies to restrict the usage of unwanted mechanisms and keys: see Chapter 6, "Supporting cryptographic policies for openCryptoki," on page 25.

Steps "1" on page 17 through "6" on page 17 are the minimal steps needed to prepare openCryptoki for using the most basic functions. The purpose of this publication is to provides you with all background information to understand how openCryptoki works and to document all options available with openCryptoki.

© Copyright IBM Corp. 2021, 2023

Chapter 4. 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 IBM Z distributions. This package might be split into several packages by the distributions, thus allowing to install individual tokens separately.

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

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

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

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

Notes:

- You can update an installed version of openCryptoki with a package of a newer version. Depending on the tokens to be used, further libraries need to be installed and cryptographic adapters must be enabled. If you had built openCryptoki from the source before, you must remove any previous installation of openCryptoki (`make uninstall`), before you can install a distribution package for a new openCryptoki version.
- Some tokens need a token-specific library to be installed on the system as a prerequisite for usage. These are mentioned for each token in Part 4, “Token specifications,” on page 79.

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`.

© Copyright IBM Corp. 2021, 2023

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.

Post-installation checks

After a successful installation, perform the following checks:

- Check the default global openCryptoki configuration file shipped with the package (`/etc/opencryptoki/opencryptoki.conf`). Delete all slot entries for tokens that you do not use. See Chapter 5, “Adjusting the openCryptoki configuration file,” on page 21.
- If you plan to use one or more CCA tokens or EP11 tokens, check the shipped default configuration files `ccatok.conf` and `ep11tok.conf`. Adapt them as required for your environment. Read “Defining a CCA token configuration file ” on page 91 and “Defining an EP11 token configuration file” on page 107.

Chapter 5. 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 21 lists the available libraries that may be in place after you successfully installed openCryptoki. It may vary for different distributions and is dependent from the installed packages.

Also, Linux on IBM Z and IBM 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 host library software are installed and a Crypto Express EP11 coprocessor is detected. For more information, read Exploiting Enterprise PKCS #11 using openCryptoki.

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

```
$ pkcsconf -t
```

For a sample output, see Figure 7 on page 41. Table 2 on page 21 shows available openCryptoki libraries:

Table 2. openCryptoki libraries

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

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

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

© Copyright IBM Corp. 2021, 2023

The /etc/opencryptoki/opencryptoki.conf file must exist and it must contain an entry for each instance of a token to make these instances available (see the provided sample configuration from Figure 4 on page 22).

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

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

```
version opencryptoki-3.22

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

disable-event-support                    # not part of the default config file
statistics (on,implicit,internal)        # not part of the default config file

slot 0
{
stdll = libpkcs11_tpm.so
tokversion = 3.12
}

slot 1
{
stdll = libpkcs11_ica.so
tokversion = 3.12
}

slot 2
{
stdll = libpkcs11_cca.so
confname = ccatok.conf
tokversion = 3.12
}

slot 3
{
stdll = libpkcs11_sw.so
tokversion = 3.12
}

slot 4
{
stdll = libpkcs11_ep11.so
confname = ep11tok.conf
tokversion = 3.12
}
```

Figure 4. Default opencryptoki.conf

Note:

- The standard path for slot token dynamic link libraries (STDLLs) is: /usr/lib64/opencryptoki/stdll/ see Table 2 on page 21). These paths may be distribution-specific.
- The standard path for the token-specific configuration file (in our example, ep11tok.conf) is /etc/opencryptoki/.

- You can specify multiple tokens of any type in different slots. Each token must be specified with a unique token name in each slot. The tokens are located in different file paths in the token directory, but use the same STDLL. For example:

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

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

Event notification support

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

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

Event notification support is enabled by default. You can disable this support by specifying keyword `disable-event-support` in the openCryptoki configuration file as shown in Figure 4 on page 22.

Collecting statistics

You can enable or disable the collection of statistics about mechanism usage in the openCryptoki configuration file at `/etc/opencryptoki/opencryptoki.conf`. Use the following option (see Figure 4 on page 22):

```
statistics (off|on[,implicit][,internal])
```

By default, statistics collection is enabled. A value of `(off)` disables all statistics collection. A value of `(on)` enables collection of mechanism usage. The collected statistics can be displayed using the **pkcsstats** utility (see Chapter 11, “Displaying usage statistics - pkcsstats utility,” on page 67).

In addition to enabling statistics collection for mechanisms used by PKCS #11 applications, you can specify to also enable collection of implicit mechanism usage, where additional mechanisms are specified in mechanism parameters. For example, RSA-PSS or RSA-OAEP allow to specify a hash mechanism and a mask generation function (MGF) in the mechanism parameter. ECDH allows to specify a key derivation function (KDF) in the mechanism parameter. For this purpose, use the option `statistics (on,implicit)`. By default only explicit mechanism usage statistics from PKCS #11 applications are collected.

You can additionally enable statistics collection of mechanisms internally used by openCryptoki by specifying `(on,internal)`. This option additionally collects usage statistics for cryptographic operations used internally for PIN handling and encryption of private token objects in the data store. You can also combine implicit and internal statistics collection: `(on,implicit,internal)`.

Collecting statistics can be a preparation step for enabling a policy (see Chapter 6, “Supporting cryptographic policies for openCryptoki,” on page 25). With statistics you can see which mechanism or key strengths would be rejected by a policy, even before the policy is made active. Look for all mechanisms and key strengths that a planned policy will not allow and make sure that the corresponding usage counters are zero.

Note: Implicit or internal mechanism usage can not be distinguished from explicit mechanism usage of PKCS #11 applications in the statistics displayed by the **pkcsstats** utility.

Chapter 6. Supporting cryptographic policies for openCryptoki

For openCryptoki, you can apply global policies to restrict the usage of unwanted mechanisms and keys. The policy is guided by the notion of cryptographic strength. You can specify a minimal strength, allowed mechanisms, and a way to derive the strength for a given key. openCryptoki then blocks all keys that are not strong enough and mechanisms that are not allowed. The policy is set globally for all applications using openCryptoki.

Applying a cryptographic policy to openCryptoki applications is based on two configuration files. You just need to adapt the strength configuration file `strength.conf` that is preinstalled in `/etc/opencryptoki`, and create the policy configuration file `policy.conf` into this openCryptoki folder as shown:

```
/etc/opencryptoki/strength.conf
/etc/opencryptoki/policy.conf
```

The strength configuration is mandatory. If you install openCryptoki from the source package, a default strength configuration file based on NIST recommendations is installed if no strength configuration exists. Only the root user can modify this file. A valid strength configuration file is prerequisite for activating a policy defined in the policy configuration file. However, a policy configuration is optional.

The strength configuration file is also used for collecting statistics. The key strength as defined by that file determines under which strength a key is counted.

Strength configuration file

The strength configuration file provides the definitions for the cryptographic strengths known to openCryptoki.

For every used key, openCryptoki computes a strength value depending on the key type, and compares it to the strengths found in the applicable strength configuration file to decide whether to accept or refuse the key. The rules how key strengths are calculated are explained in "Rules for key strength calculation" on page 27.

The content starts with a version specification of the file. The version specification must be the first non-empty and non-comment line and must look like:

```
version strength-<n>
```

where <n> specifies the version number. For example, the supported version at the time of writing is

```
version strength-0
```

Before and after this line, you can have as many empty lines or comment lines that start with a hash (#) sign.

A strength definition line within this file is an indexed structure of the form:

```
strength <num> { <content> }
```

where

<num>
: is one of 112, 128, 192, or 256, representing the corresponding strength.

<content>
: is a sequence of keyword/value pairs, with valid keywords as described hereafter.

Comments are allowed before and after every keyword/value pair as well as before and after every braces ({ or }).

Valid keywords defining the key constraint policy are:

MOD_EXP
: describes the minimal size in bits of the modulus used by RSA, DSA, and DH algorithms.

ECC
: for an ECDSA/EdDSA or ECDH/EdDH algorithm it describes the minimal size in bits of the prime defining the Galois field over which the elliptic curve is defined.

SYMMETRIC
: covers the minimal size of all symmetric keys. This includes GENERIC_SECRET, AES, and various HMAC keys.

Valid keywords defining the output length constraints are:

digest
: provides the minimal length that a digest must have to be in this strength class.

signature
: provides the minimal length that a signature or a message authentication code must have to be in this strength class.

If either of the **`MOD_EXP`**, **`ECC`**, or **`SYMMETRIC`** attributes are missing inside a strength definition, no key that would be compared against this attribute will have the corresponding strength. If either of the **`digest`** or **`signature`** attributes are missing, the output constraint represented by these attributes defaults to 0, thus effectively allowing all output sizes.

Example of a strength.conf configuration file

Have a look at a sample strength configuration file.

strength.conf file with NIST recommendations

Figure 5 on page 27 provides a sample for a strength configuration file that is specifying NIST recommendations on key and algorithm strength. The indentation is optional but helpful to simplify the maintenance of the file. openCryptoki applies this configuration during installation from the source package, if a strength configuration file does not yet exist in the target folder.

```
# openCryptoki strength example corresponding to NIST recommendations
# See https://www.keylength.com/en/4/

version strength-0
strength 112 {
         MOD_EXP = 2048
         ECC = 224
         SYMMETRIC = 112
         digest = 224
         signature = 112
}
strength 128 {
         MOD_EXP = 3072
         ECC = 256
         SYMMETRIC = 128
         digest = 256
         signature = 128
}
strength 192 {
         MOD_EXP = 7680
         ECC = 384
         SYMMETRIC = 192
         digest = 384
         signature = 192
}
strength 256 {
         MOD_EXP = 15360
         ECC = 512
         SYMMETRIC = 256
         digest = 512
         signature = 256
}
```

Figure 5. Sample of a strength.conf configuration file

Rules for key strength calculation

openCryptoki retrieves certain attributes of a key depending on the key type. The found value is compared against the definitions in the strength configuration file and thus the resulting strength is found. The rules in the policy configuration file finally decide whether the key is accepted or refused by the openCryptoki application. Therefore, you need to know the calculation rules to understand why a key may be refused, so that you can change your strength definition accordingly.

The strength of a key is calculated during key or key pair generation, key derivation, and key loading, by comparing key attributes (for example, its modular exponent for RSA keys) with the values defined in the strength configuration. This key strength is held in memory together with the key object. The policies in the policy configuration file decide whether a key with the calculated strength is allowed for the application.

Also the expected signature size when using this key is calculated and held in memory with the key. This size is used to check if the key is allowed in a sign or verify operation by the policy.

Additionally, a flag used by policies to mark EC keys as usable is held in memory.

Shortly spoken, the key strength computation is based on the key attributes, especially its key type. The value of the highest matching strength definition from the strength configuration file is used as strength.

RSA
: For keys of type CKK_RSA, the length of CKA_MODULUS multiplied by 8 is compared to the setting of the MOD_EXP property of the configurations.

DH and DSA
: For keys of type CKK_DH, CKK_DSA, or CKK_X9_42_DH, the length of CKK_PRIME multiplied by 8 is compared against the MOD_EXP property of the configurations.

EC (including Edwards and Montgomery)
: Based on the curve type specified in CKK_EC_PARAMS, the size of the elliptic curve is determined. This size is compared against the ECC property of the configurations.

DES
: The base strength of DES keys is fixed. Keys of type CKK_DES2 have 80 bits and keys of type CKK_DES3 have 112 bits of base strength. The base value is then compared against the SYMMETRIC property of the configurations.

AES and GENERIC_SECRET
: For keys of type CKK_AES, CKK_AES_XTS, or CKK_GENERIC_SECRET, a base strength is computed by multiplying CKA_VALUE_LEN with 8, for CKK_AES_XTS keys: CKA_VALUE_LEN * 8/2. The base value is then compared against the SYMMETRIC property of the configurations.

Post-quantum algorithms (Dilithium and Kyber)
: The strength of dilithium and kyber keys is always set to 256.

Note: All strength assignments for mechanisms are subject to change in accordance with new insights in cryptography research.

Similarly to the strength determination based on attributes, for key types CKK_RSA, CKK_DSA and CKK_ECC, a signature size is computed that would be achieved with this key. This signature size is also held in memory within the key object. This size is used to check if the key is allowed in a sign or verify operation by the policy.

Policy configuration file

If a policy configuration file is present in the openCryptoki folder, it can define certain cryptographic rules for your applications.

The policy definition file is located in `/etc/opencryptoki/policy.conf`. It requires a valid strength configuration file ("Strength configuration file" on page 25). If, however, only a strength definition is available, openCryptoki just computes the strength of keys used for accounting statistics (**`pkcsstats`**), but does not restrict any operation.

An existing policy definition file defines the cryptographic rules comprising the following aspects:

- specifying the minimum required strength of keys,
- allowing certain mechanisms,
- allowing certain EC curves, for example, NIST-approved curves,
- allowing certain mask generation functions (MGFs),
- allowing certain key derivation functions (KDFs),
- allowing certain pseudo random functions (PRFs) for use with PBKDF2.

The policy definition consists of various parameters specifying, for example, the desired minimal cryptographic strength, the allowed mechanisms, or the allowed elliptic curves. For the syntax of the policy configuration file view the sample in Figure 6 on page 31.

strength
: specifies the minimal cryptographic strength supported by this policy. Currently, openCryptoki supports five different strengths: 0, 112, 128, 192, and 256. The value must be an integer and is rounded up to the next supported strength (with 256 as limit). A minimal strength of 0 means to allow all key strengths (no restriction).

allowedmechs
: specifies a list of mechanisms that should be allowed. This parameter is optional. If present, only the values specified in this list are allowed. An empty list does not allow any mechanism. If you do not specify this configuration option, all mechanisms are allowed if their parameters satisfy the required strength.

allowedcurves
: specifies which elliptic curves are allowed. You can select from the curves listed in "Reference for valid values" on page 29. If this configuration is specified, it must list by name all curves that should be allowed. The empty list forbids all curves. If you do not specify this configuration option, all curves that satisfy the strength requirement are allowed.

allowedmgfs
: specifies a list of allowed MGFs. This configuration is optional. If not specified, all MGFs are allowed. Otherwise, only MGFs listed in "Reference for valid values" on page 29 are allowed.

allowedkdfs
: specifies a list of allowed KDFs. This configuration is optional. If not specified, all KDFs are allowed. Otherwise, only KDFs listed in "Reference for valid values" on page 29 are allowed.

allowedprfs
: specifies the allowed PRFs that can be used in PBKD operations. At the time of writing, only CKP_PKCS5_PBKD2_HMAC_SHA256 and CKP_PKCS5_PBKD2_HMAC_SHA512 are supported. This configuration is optional. If not specified, all PRFs are allowed. Otherwise only the specified PRFs are allowed.

Reference for valid values

This section provides a quick reference for valid curves or functions that you can specify with the parameters in the policy configuration file.

Valid ECC curves

- BRAINPOOL_P160R1
- BRAINPOOL_P160T1
- BRAINPOOL_P192R1
- BRAINPOOL_P192T1
- BRAINPOOL_P224R1
- BRAINPOOL_P224T1
- BRAINPOOL_P256R1
- BRAINPOOL_P256T1
- BRAINPOOL_P320R1
- BRAINPOOL_P320T1
- BRAINPOOL_P384R1
- BRAINPOOL_P384T1
- BRAINPOOL_P512R1
- BRAINPOOL_P512T1
- PRIME192V1
- SECP224R1
- PRIME256V1
- SECP384R1
- SECP521R1
- SECP256K1
- CURVE25519
- CURVE448
- ED25519
- ED448

Valid mask generation functions (MGFs)

- CKG_MGF1_SHA1
- CKG_MGF1_SHA224
- CKG_MGF1_SHA256
- CKG_MGF1_SHA384

- CKG_MGF1_SHA512
- CKG_IBM_MGF1_SHA3_224
- CKG_IBM_MGF1_SHA3_256
- CKG_IBM_MGF1_SHA3_384
- CKG_IBM_MGF1_SHA3_512

Valid key derivation functions (KDFs)

- CKD_NULL
- CKD_SHA1_KDF
- CKD_SHA1_KDF_ASN1
- CKD_SHA1_KDF_CONCATENATE
- CKD_SHA224_KDF
- CKD_SHA256_KDF
- CKD_SHA384_KDF
- CKD_SHA512_KDF
- CKD_IBM_HYBRID_SHA1_KDF
- CKD_IBM_HYBRID_SHA224_KDF
- CKD_IBM_HYBRID_SHA256_KDF
- CKD_IBM_HYBRID_SHA384_KDF
- CKD_IBM_HYBRID_SHA512_KDF

Example of a general policy configuration file

A policy configuration file starts with a version specification. The currently supported version specification is `policy-0`. If you want to only allow the DES algorithm family without specifying a strength and allow only the Brainpool curves, specify the following:

```
version policy-0
strength = 0
allowedmechs (
               CKM_DES_KEY_GEN,
               CKM_DES_CBC,
               CKM_DES_CBC_PAD,
               CKM_DES_CFB64,
               CKM_DES_CFB8,
               CKM_DES_ECB,
               CKM_DES_OFB64,
               CKM_DES2_KEY_GEN,
               CKM_DES3_KEY_GEN,
               CKM_DES3_CBC,
               CKM_DES3_CBC_PAD,
               CKM_DES3_CMAC,
               CKM_DES3_CMAC_GENERAL,
               CKM_DES3_ECB,
               CKM_DES3_MAC,
               CKM_DES3_MAC_GENERAL
             )

allowedcurves (
                BRAINPOOL_P160R1,
                BRAINPOOL_P160T1,
                BRAINPOOL_P192R1,
                BRAINPOOL_P192T1,
                BRAINPOOL_P224R1,
                BRAINPOOL_P224T1,
                BRAINPOOL_P256R1,
                BRAINPOOL_P256T1,
                BRAINPOOL_P320R1,
                BRAINPOOL_P320T1,
                BRAINPOOL_P384R1,
                BRAINPOOL_P384T1,
                BRAINPOOL_P512R1,
                BRAINPOOL_P512T1
)
```

Figure 6. Example of a policy.conf configuration file

Whether or not an elliptic curve is allowed is stored alongside the strength in every elliptic curve key object.

Examples of policy configuration files customized to token directories

In addition to affecting cryptographic processing of openCryptoki applications, an existing policy also applies to all cryptographic operations internally performed by openCryptoki to encrypt token data within the various token directories. Token directories exist in two versions: a legacy version and a FIPS compliant version. In both cases, all internal operations needed by openCryptoki within the token directory must be allowed by the policy. For the legacy format, TDES-CBC, AES-CBC, SHA1, and MD5 must be allowed mechanisms. Also, double-length TDES keys must be usable (which corresponds to a value of 80 for the strength configuration).

View an example policy file allowing all **legacy token directory** operations. This example comprises the minimal set of mechanisms required for openCryptoki internal legacy token format cryptographic operations.

```
version policy-0
strength = 0
allowedmechs (
               CKM_DES3_KEY_GEN,
               CKM_DES3_CBC,
               CKM_AES_KEY_GEN,
               CKM_AES_CBC,
               CKM_SHA1,
               CKM_MD5
             )
```

The FIPS compliant format needs AES key wrapping, AES-GCM, and PKCS5_PBKD2 as allowed mechanisms and the CKP_PKCS5_PBKD2_HMAC_SHA512 pseudo random function allowed. Also, 256 bit AES keys must be usable according to the strength and policy definitions. View an example policy file allowing all FIPS compliant operations required for openCryptoki internal FIPS compliant token format cryptographic operations.

```
version policy-0
strength = 0
allowedmechs ( CKM_AES_KEY_GEN, CKM_AES_KEY_WRAP, CKM_AES_GCM,
               CKM_PKCS5_PBKD2 )
allowedprfs  ( CKP_PKCS5_PBKD2_HMAC_SHA512 )
```

The ICSF token has a different token directory. It needs AES CBC, SHA1, MD5 mechanisms, and PKCS5_PBKD2 with the CKP_PKCS5_PBKD2_HMAC_SHA256 pseudo random function allowed:

```
version policy-0
strength = 0
allowedmechs ( CKM_AES_KEY_GEN, CKM_AES_CBC,
               CKM_SHA1, CKM_MD5, CKM_PKCS5_PBKD2 )
allowedprfs ( CKP_PKCS5_PBKD2_HMAC_SHA256 )
```

Note: For all formats you can also remove the **`allowedmechs`** option completely. But in case you want to restrict mechanisms, the list must include all the mentioned mechanisms. Similarly, you can remove the **`allowedprfs`** option, since this allows all pseudo random functions.

If the token directory operations are not supported by the policy, the token does not load but produces error messages in the system log.

Processing configuration files

Strength and policy configuration files are loaded with `C_Initialize()` and translated into internal data structures. Configuration files must be owned by the `root` user and group `pkcs11`. Only the `root` user can modify these files. They must have mode bits 0640 (octal: owner read-write, group read, other nothing), and describe a valid configuration.

Additional configuration options or duplicated options (for example, specifying a strength multiple times in the policy) are invalid configurations. A valid strength configuration file is required by openCryptoki. Otherwise, openCryptoki returns CKR_GENERAL_ERROR while processing the `C_Initialize()` function and produces an error message in `syslog`. If a valid policy configuration file is found, openCryptoki enforces the defined policy. If no policy configuration is found, openCryptoki does not restrict any key or operation, but still computes the strength for all keys.

Processing of EP11 tokens: An EP11 token also checks the signing and verification key used for the mechanism CKM_IBM_ATTRIBUTEBOUND_WRAP (that is, attribute-bound wrapping and unwrapping).

Processing of ICSF tokens: An ICSF token processing the CKM_TLS_KEY_AND_MAC_DERIVE mechanism might generate up to four keys. All keys are checked for proper strength.

Providing information

For an existing policy configuration, the output of function `C_GetMechanismList()` only returns mechanisms that do apply to the defined policy. Consequently, function `C_GetMechanismInfo()` returns information only for allowed mechanisms and issues a CKR_MECHANISM_INVALID message for mechanisms that are not allowed by the policy. The minimum and maximum key size of a mechanism returned by `C_GetMechanismInfo()` is also adjusted to key sizes allowed by the policy.

Checking operations

openCryptoki performs checks to the initialization functions of the various encrypt, decrypt, digest, sign, and verify functions. Furthermore, it also checks wrap, unwrap, derive and key(-pair) generation functions. Checking depends on the attempted operation. If the policy forbids an operation either because the

mechanism is not allowed, or because one of the involved keys is too weak, or because the input or output is too small, then CKR_FUNCTION_FAILED is returned and the function aborts. Furthermore, the ulDeviceError field of the session info is set to CKR_POLICY_VIOLATION to aid in debugging the problem. Also, various trace messages identify the policy violation.

Key loading
Keys are loaded either implicitly when the token initializes or when the user logs into the token. Furthermore, keys are loaded during `C_FindObjects()`. In these cases, only sufficiently strong keys can be loaded. The strength of the key is determined based on the attributes in the key object.

Key creation
During key creation with `C_CreateObject()`, the user provides a template for the key including the key type and the attributes used to derive the strength. This is then used to classify the key and, if the classification is too low or the elliptic curve is not supported, key creation is aborted.

Key or key pair generation
During key or key pair generation, the strength of the key or key pair to be generated is computed. If it is too small or the elliptic curve is not supported, generation is aborted. Furthermore, the key or key pair generation mechanism must be allowed by the profile.

Key derivation and unwrapping
During key derivation and unwrapping, both the input key and the output key(s) are checked for appropriate strength and allowed elliptic curve. If either check fails, derivation or unwrapping is aborted. Furthermore, the derivation and unwrapping mechanisms must be allowed.

Encryption and decryption
Keys are checked only during the initialization functions of these operations. For updates and finalization no further checks are needed.

Signing and verifying
For signatures, the key must be allowed, and the size of the signature must be at least as big as the signature property of the active strength configuration. If either the key is not allowed or the signature is not large enough, the operation is aborted.

Digests
The digest output length in bits is compared to the digest property of the active strength configuration. If it is smaller, the operation is aborted.

Chapter 7. 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 library libopencryptoki.so. You should never change this variable. Maybe useful for debugging information.
`OCK_EP11_LIBRARY`	Specifies the path to the EP11 host library. Only of interest, if multiple EP11 libraries are installed. Normally, you should never change this variable. Maybe useful for debugging information.
`OCK_EP11_TOKEN_DIR`	The standard path for the token-specific EP11 token configuration file is `/etc/opencryptoki/`. You can change this path by using this variable.
`PKCS_APP_STORE`	Specifies the path to the main directory. Never change this variable. Maybe useful for debugging information.
`PKCS11_SHMEM_FILE`	Used for pointing to the SHM-Key Files. Only of interest, if you run multiple slot manager daemons (**pkcsslotd**).
`PKCS11_USER_PIN`	Provides a possibility to transfer the ***User*** PIN to the **p11sak** utility. It is required for running the openCryptoki tests.
`OPENCRYPTOKI_TRACE_LEVEL`	Enables logging support. If the environment variable is not set, logging is disabled by default.

© Copyright IBM Corp. 2021, 2023

Part 3. 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 8, “Managing tokens - pkcsconf utility,” on page 39
- Chapter 9, “Managing token keys - p11sak utility,” on page 43
- Chapter 10, “Migrating to FIPS compliance - pkcstok_migrate utility,” on page 65
- Chapter 11, “Displaying usage statistics - pkcsstats utility,” on page 67
- Chapter 12, “Managing a concurrent master key change - pkcshsm_mk_change utility,” on page 71

Chapter 8. Managing tokens - pkcsconf utility

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

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

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

The available options have the following meanings:

-i
: display PKCS #11 info

-t
: display token info

-s
: display slot info

-m
: display mechanism list

-l
: display slot description

-I
: initialize token

-u
: initialize user PIN

-p
: set the user PIN

-P
: set the SO PIN

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

-c
: specify the token slot ID

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

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

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

The **pkcsconf** functions for obtaining PKCS #11 information (**pkcsconf -i**), token information (**pkcsconf -t**), and slot information (**pkcsconf -s**) also display the current information in form of a PKCS #11 Unified Resource Identifier (URI).

Examples:

```
$ pkcsconf -s
Slot #0 Info
        Description: Linux
        Manufacturer: IBM
        Flags: 0x1 (TOKEN_PRESENT)
        Hardware Version: 0.0
        Firmware Version: 0.0
        URI: pkcs11:slot-id=0;slot-description=Linux;slot-manufacturer=IBM

$ pkcsconf -t
Token #0 Info:
        Label: softtok
        Manufacturer: IBM
        Model: Soft
        Serial Number:
        Flags: 0x44D (RNG|LOGIN_REQUIRED|USER_PIN_INITIALIZED|CLOCK_ON_TOKEN|TOKEN_INITIALIZED)
        Sessions: 0/[effectively infinite]
        R/W Sessions: [information unavailable]/[effectively infinite]
        PIN Length: 4-8
        Public Memory: [information unavailable]/[information unavailable]
        Private Memory: [information unavailable]/[information unavailable]
        Hardware Version: 0.0
        Firmware Version: 0.0
        Time: 2022042908395700
        URI: pkcs11:manufacturer=IBM;model=Soft;token=softtok
```

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

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 9, PKCS #11 defines a ***security officer*** (SO) and a (standard) ***User*** for each token instance. openCryptoki requires that for both a log-in PIN is defined as part of the token initialization.

The following command provides some useful slot information:

```
# pkcsconf -s

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

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

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

Figure 7. Token information with ***pkcsconf -t***

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

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

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

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

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

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

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

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

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

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

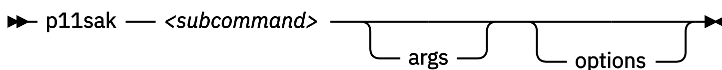
Chapter 9. Managing token keys - p11sak utility

Use the **p11sak** tool to manage token keys and certificates in an openCryptoki token repository with their PKCS #11 attributes. You can generate, import, export, change, copy, and remove symmetric and asymmetric keys in an openCryptoki token repository. With this tool, you can also import, export, copy and list certificates.

The general invocation scheme of the command line tool is:

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

p11sak syntax - General invocation scheme



where

subcommand

- **generate-key**: "Generating keys in the openCryptoki repository" on page 43
- **list-key**: "Listing keys in the repository" on page 47
- **remove-key**: "Removing keys" on page 50
- **set-key-attr**: "Setting attributes of keys" on page 51
- **copy-key**: "Copying keys in the repository" on page 53
- **import-key**: "Importing keys from a binary file or a PEM file" on page 54
- **export-key**: "Exporting keys to a binary file or a PEM file" on page 54
- **extract-pubkey**: "Extracting the public key from a private key" on page 55
- **list-cert**: "Listing certificates in the repository" on page 57
- **remove-cert**: "Removing certificates from the repository" on page 58
- **set-cert-attr**: "Setting or updating attributes of certificates" on page 58
- **copy-cert**: "Copying certificates in the repository" on page 59
- **import-cert**: "Importing certificates from a binary file or a PEM file" on page 59
- **export-cert**: "Exporting certificates to a binary file or PEM file" on page 60
- **extract-cert-pubkey**: "Extracting public keys from certificates" on page 60

Also, read the information in section "Command help" on page 61 to learn how to use the tool efficiently.

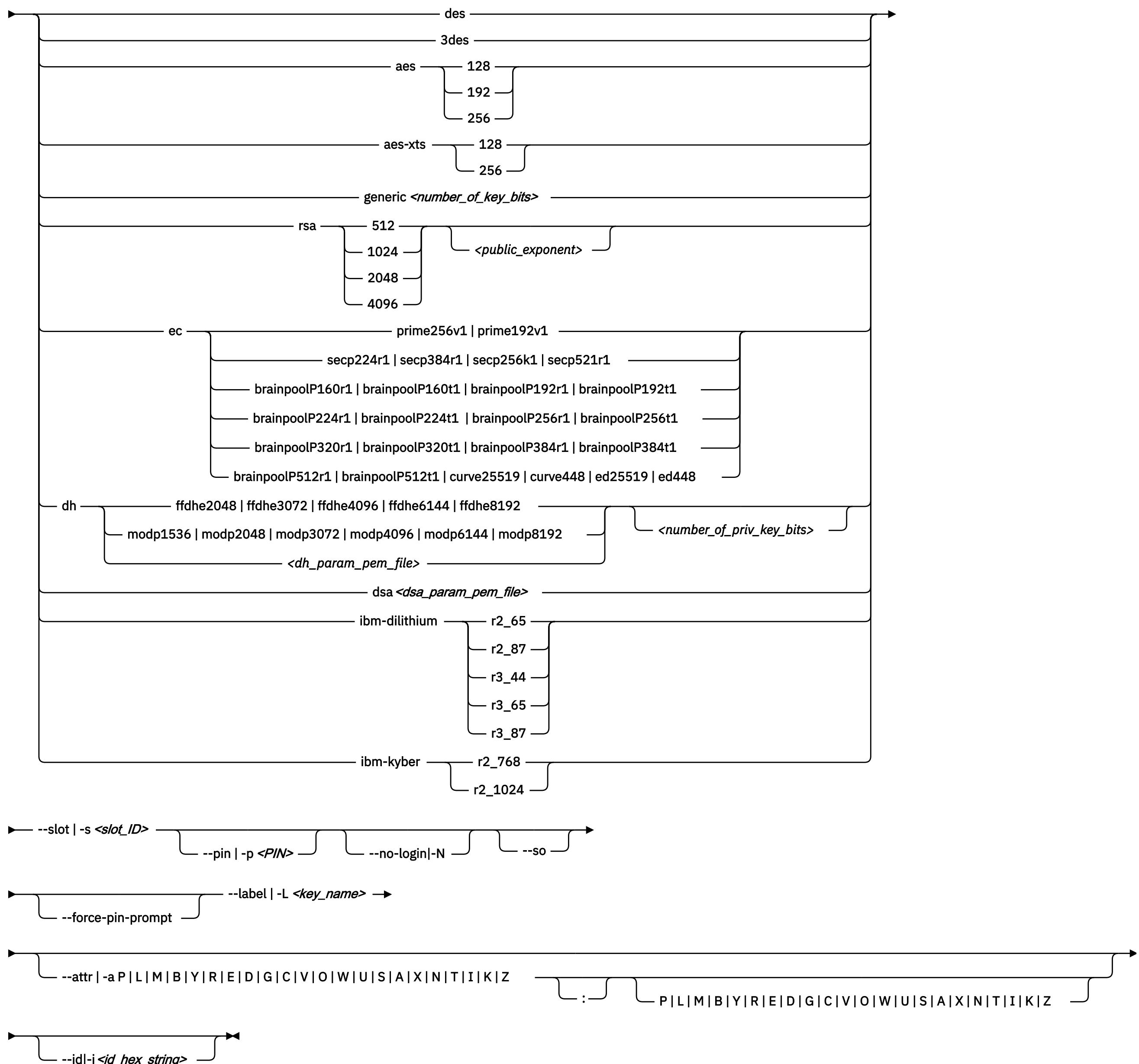
Generating keys in the openCryptoki repository

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

- symmetric keys (AES, AES-XTS, 3DES, DES, generic secret) with PKCS #11 attributes
- asymmetric keys (RSA, DH, DSA, EC, IBM Kyber, IBM Dilithium) with PKCS #11 attributes.

p11sak syntax - Generating keys

p11sak — generate−key | gen−key | gen



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.

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

generic

specifies a generic secret key to be generated (CKK_GENERIC_SECRET), which you can use for example, for HMAC. You must specify a key length in bits.

rsa

specifies an asymmetric RSA key pair (private and public) to be generated. You must specify a key length in bits: 512, 1024, 2048, or 4096. Optionally, you can specify a public exponent (parameter **`<public_exponent>`**) for an RSA key pair generation. The default is 65537.

ec

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

dh

specifies a Diffie-Hellman key pair (private and public) to be generated. You must specify a group name or the name of a PEM file (**`<dh-param-pem-file>`**) containing the DH-parameter for the key pair. Valid group names as shown in the syntax diagram are:

```
ffdhe2048|ffdhe3072|ffdhe4096|ffdhe6144|ffdhe8192
modp1536|modp2048|modp3072|modp4096|modp6144|modp8192
```

Optionally specify the size of the private key in bits in parameter **`<number_of_priv_key_bits>`**.

dsa

specifies a DSA key pair (private and public) to be generated. In parameter **`<dsa_param_pem_file>`** you must specify the name of a PEM file containing the DSA-parameter for the key pair.

ibm-dilithium

specifies an IBM Dilithium key pair (private and public) to be generated. You must specify a version for the IBM Dilithium key pair. Select one supported version as shown in the syntax diagram.

ibm-kyber

specifies an IBM Kyber key pair (private and public) to be generated. You must specify a version for the IBM Kyber key pair. Select one supported version as shown in the syntax diagram.

--slot|-s

specifies the slot ID of an openCryptoki token within the repository.

--pin|-p

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

--no-login|-N

Do not login to the session. This means that only public token objects (CKA_PRIVATE=FALSE) can be accessed.

--so

Login as SO (security officer). Option `--pin|-p` must specify the SO pin, or if the `--pin|-p` option is not specified, environment variable PKCS11_SO_PIN is used. If PKCS11_SO_PIN is not set, then you are prompted for the SO PIN. The security officer (SO) can only access public token objects (CKA_PRIVATE=FALSE).

--force-pin-prompt

enforces a PIN prompt regardless whether a PIN has been specified elsewhere.

--label|-L

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

For asymmetric keys, set individual labels for public and private key, separated by a colon: `pub_label:priv_label`, for example, `rsa1_public:rsa1_private`. You can use the same label for public and private keys by specifying the equal sign ('=') for the private key label part, for example, `rsa1:=`. If only one label is specified for an asymmetric key, the label is automatically extended by `:pub` and `:prv` for the public and private keys respectively, for example, `--label rsa1` is extended to `rsa1:pub` and `rsa1:prv`.

--attr|-a

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

For asymmetric keys set individual key attributes for public and private key separated by a colon: `public_attributes:private_attributes`, for example, `MLD:MrS`.

P
CKA_PRIVATE

L
CKA_LOCAL (read only)

M
CKA_MODIFIABLE

B
CKA_COPYABLE

Y
CKA_DESTROYABLE

R
CKA_DERIVE

E
CKA_ENCRYPT

D
CKA_DECRYPT

G
CKA_SIGN

C
CKA_SIGN_RECOVER

V
CKA_VERIFY

O
CKA_VERIFY_RECOVER

W
CKA_WRAP

U
CKA_UNWRAP

S
CKA_SENSITIVE

A
CKA_ALWAYS_SENSITIVE (read only)

X
CKA_EXTRACTABLE

N
CKA_NEVER_EXTRACTABLE (read only)

T
CKA_TRUSTED (can be set to TRUE by the security officer (SO) only)

I
CKA_WRAP_WITH_TRUSTED

K
CKA_IBM_PROTKEY_EXTRACTABLE (IBM specific, not all tokens support this)

Z
CKA_IBM_PROTKEY_NEVER_EXTRACTABLE (IBM specific, not all tokens support this, read only)

An uppercase letter sets the corresponding attribute to CK_TRUE, a lower case letter to CK_FALSE. If an attribute is not set explicitly, its default value is used. Not all attributes may be accepted for all key types. Attribute CKA_TOKEN is always set to CK_TRUE.

--id|-i
specifies a hex string (not prefixed with 0x) of any number of bytes. For asymmetric keys the same ID is set for both, the public and the private key.

Example for generating an RSA private and public key pair:

```
p11sak generate-key rsa 1024 --slot 1 --pin 11111111 –label test
```

The resulting key pair of this command is shown in Figure 8 on page 49. If you do not specify attributes using the `–attr` option, the resulting key will obtain the default attributes as shown. Also, if you would specify the label using `--label test:=` , you get just `test` as the key names, without `prv` or pub appended.

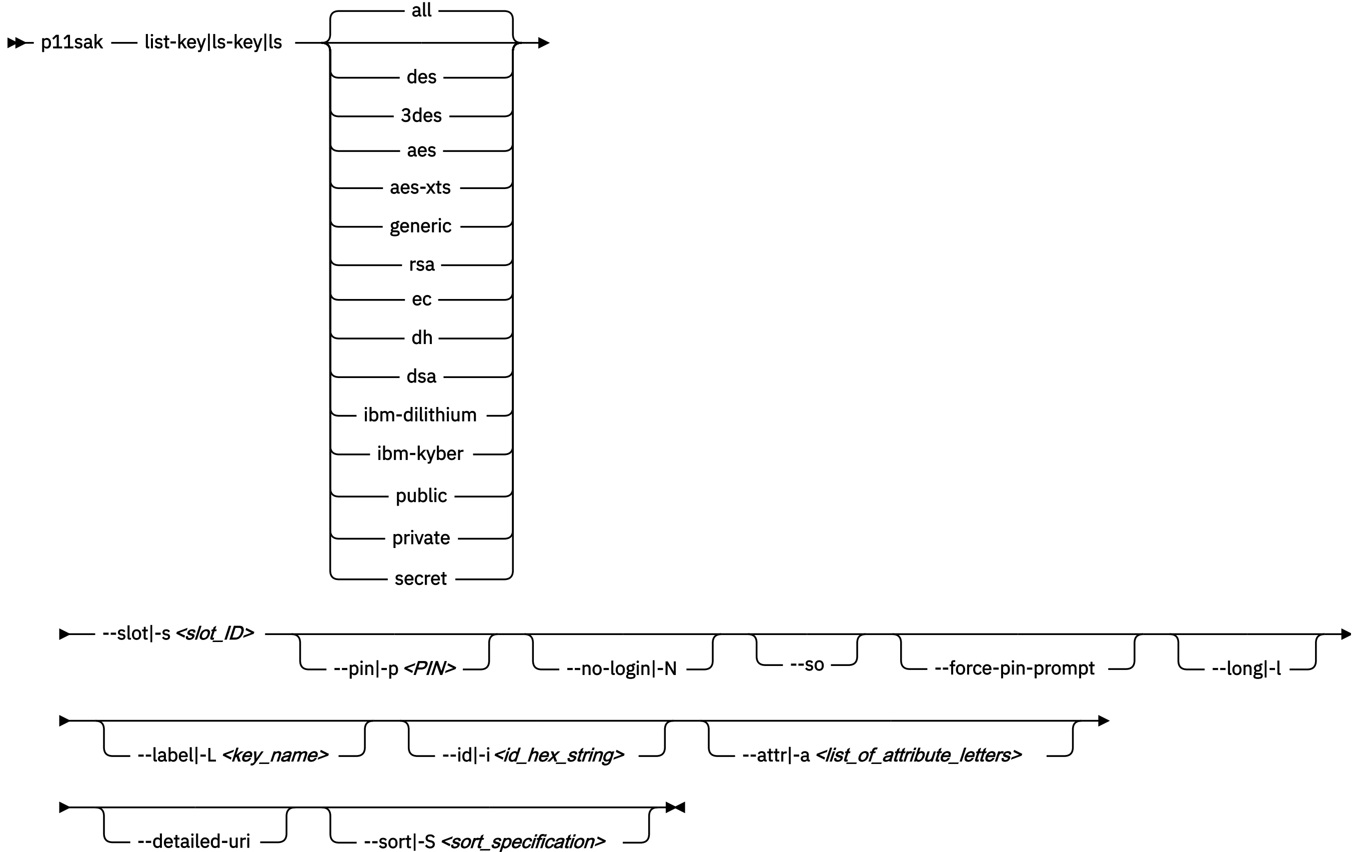
Listing keys in the repository

The tool supports the listing of:

- symmetric keys (AES, AES-XTS, 3DES, DES, generic secret) with PKCS #11 attributes,
- asymmetric keys (RSA, DH, DSA, EC, IBM Kyber, IBM Dilithium),
- public, private and secret keys,
- all keys of any type.

With the options in the **`list-key`** subcommand, you can filter the keys that you want to list. They have the same meaning as described in section “Generating keys in the openCryptoki repository” on page 43, with additional options to filter for public, private and secure keys. The `--long` or `-l` parameter produces the long output format. If omitted, you obtain a short output format.

p11sak syntax - Listing keys



The parameters and options for the **p11sak list-key** subcommand are the same as for the **generate-key** subcommand (“p11sak syntax - Generating keys” on page 44). Differences in the use and meaning are described hereafter.

In addition to the **p11sak generate-key** subcommand, there are the following key types which you can use to filter for listing keys:

secret
: lists all symmetric keys (for example, AES and DES) that are created with the CKO_SECRET attribute.

public
: lists the public key part of a public/private asymmetric key pair that is created with the CKO_PUBLIC_KEY attribute.

private
: lists the private key part of a public/private asymmetric key pair (for example, AES and DES) that is created with the CKO_PRIVATE_KEY attribute.

all
: lists all keys no matter of which type.

--attr|-a
: Attributes whose letters are not specified in parameter **list_of_attribute_letters** are not used to filter the keys. The format of the list of attributes is shown in “p11sak syntax - Generating keys” on page 44.

--sort|-S <sort_specification>
: Sort the keys by label, key type, object class, and key size. Specify a **<sort_specification>** parameter of up to four criteria, each represented by its corresponding letter, separated by a comma:

l
: label

k
: key type

c
: object class

s
: key size

The sort order (a = ascending (default), d = descending)) can be appended to the criteria designators by a colon. Example: `l:a,k:d` will sort by label in ascending order and then by key type in descending order.

--label|-L
: You can use wildcards (* and ?) in the label specification. To specify a wildcard character that should not be treated as a wildcard, it must be escaped using a backslash (* or \?). Also, a backslash character that should not be treated as an escape character must be escaped (\\).

--detailed-uri
: includes all information into the Unified Resource Identifier (URI). This detailed URI is only printed if the option --long has been specified. If this parameter is not set, the URI will not contain information about the library (CK_INFO) and the slot (CK_SLOT_INFO).

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

```
# p11sak list-key rsa --slot 1 --pin 11111111

| P L M B Y R E D G C V O W U S A X N T I K Z |              KEY TYPE | LABEL
|--------------------------------------------+-----------------------+--------------------
| 0 1 1 1 1 - 1 - - - 1 1 1 - - - - - 0 - - - |       public RSA 1024 | "test:pub"
| 0 1 1 1 1 - - 1 1 1 - - - 1 0 0 1 0 - 0 0 - |      private RSA 1024 | "test:prv"

2 key(s) displayed
```

Figure 8. Listing RSA key pairs

How to get a listing output in long format

Request a long output format by specifying option `--long` or `-l` with the **`list-key`** command. You optionally can provide an output configuration file called `p11sak_defined_attrs.conf` to specify attributes to be printed in addition to the ones defined by the PKCS #11 standard. You can use the environment variable `P11SAK_DEFAULT_CONF_FILE` to set the full path name for this file. If this variable is not set, the system looks for this configuration file in your user directory. If no configuration file is found there, the default output configuration file `/etc/opencryptoki/p11sak_defined_attrs.conf` is used.

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

```
attribute {
   name = CKA_IBM_RESTRICTABLE
   id = 0x80010001
   type = CK_BBOOL
}
attribute {
   name = CKA_IBM_ATTRBOUND
   id = 0x80010004
   type = CK_BBOOL
}
attribute {
   name = CKA_IBM_USE_AS_DATA
   id = 0x80010008
   type = CK_BBOOL
}
```

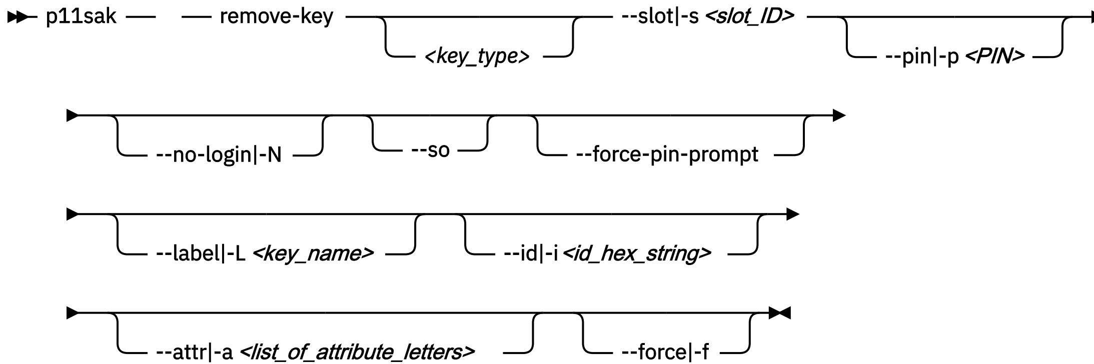
Example for a listing output in long format:

```
# p11sak list-key rsa --slot 1 --pin 11111111 --detailed-uri --long
Please enter user PIN:

Label: "test:pub"
        URI: pkcs11:library-description=openCryptoki;library-manufacturer=IBM;
             library-version=3.20;slot-id=1;slot-description=Linux;slot-manufacturer=IBM;
             manufacturer=IBM;model=ICA;token=ica;object=test:pub;type=public
        Key: public RSA 1024
        Attributes:
             CKA_TOKEN: CK_TRUE
             CKA_PRIVATE: CK_FALSE
             CKA_LOCAL: CK_TRUE
             ...
             CKA_ALLOWED_MECHANISMS: [no restriction]
             CKA_SUBJECT: [no value]
             CKA_WRAP_TEMPLATE: 0 attributes
             CKA_PUBLIC_KEY_INFO: len=162 value:
                30 81 9F 30 0D 06 09 2A 86 48 86 F7 0D 01 01 01
                ...
             CKA_MODULUS: len=128 value:
                 C6 F3 7A E4 19 09 C5 6B C3 5F 21 1E 50 3A 0D 45
                 ...
             CKA_MODULUS_BITS: 1024 (0x400)
             CKA_PUBLIC_EXPONENT: len=3 value:
                 01 00 01
             CKA_IBM_PROTKEY_EXTRACTABLE: CK_FALSE
Label: "test:prv"
    URI: pkcs11:library-description=openCryptoki;library-
manufacturer=IBM;library-version=3.20;slot-id=1;slot-description=Linux;slot-
manufacturer=IBM;manufacturer=IBM;model=ICA;token=ica;object=test:prv;type=private
    Key: private RSA 1024
    Attributes:
         CKA_TOKEN: CK_TRUE
         CKA_PRIVATE: CK_FALSE
         CKA_LOCAL: CK_TRUE
         ...
         CKA_ALLOWED_MECHANISMS: [no restriction]
         CKA_SUBJECT: [no value]
         CKA_UNWRAP_TEMPLATE: 0 attributes
         CKA_PUBLIC_KEY_INFO: len=162 value:
             30 81 9F 30 0D 06 09 2A 86 48 86 F7 0D 01 01 01
             ...
         CKA_DERIVE_TEMPLATE: 0 attributes
         CKA_MODULUS: len=128 value:
             C6 F3 7A E4 19 09 C5 6B C3 5F 21 1E 50 3A 0D 45
             ...
         CKA_PUBLIC_EXPONENT: len=3 value:
             01 00 01
         CKA_PRIVATE_EXPONENT: [no value]
         CKA_PRIME_1: len=64 value:
             ED 9D EC 54 69 F2 C6 BE F7 04 4C 70 46 3B 64 EB
             ...
         CKA_PRIME_2: len=64 value:
             D6 57 C4 CB F4 25 FE F3 E8 6C AD 12 89 29 19 0E
             ...
         CKA_EXPONENT_1: len=64 value:
             8F 42 1F 31 E5 8E 91 74 A0 C8 DE AC F2 2A EC F5
             ...
         CKA_EXPONENT_2: len=64 value:
             68 12 93 A6 67 F4 6E F7 64 FA 27 8A E1 78 48 07
             ...
         CKA_COEFFICIENT: len=64 value:
             91 00 D0 24 BD 15 7A 27 57 C8 81 A4 F2 C3 62 F5
             ...
         CKA_IBM_PROTKEY_EXTRACTABLE: CK_FALSE

2 key(s) displayed
```

Removing keys

Use the **`p11sak remove-key`** subcommand to delete keys from the openCryptoki token repository.

p11sak syntax - Removing keys



The parameters and options for the **p11sak remove-key** subcommand that are not listed here, are the same as for the **generate-key** subcommand. Differences in the use and meaning are described hereafter.

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

Use the **<key_type>** parameter to select the type or types of keys to be deleted. Possible values for the **<key_type>** argument are the following (as explained in "p11sak syntax - Generating keys" on page 44 and "Listing keys in the repository" on page 47):

```
des|3des|generic|aes|aes-xts|rsa|dh|dsa|ec|ibm-dilithium|ibm-kyber|public|private|secret|all
```

If **<key_type>** is omitted, then all key types are selected for deleting.

--slot|-s
: specifies the slot ID of an openCryptoki token within the repository from which you want to delete the matching keys.

--id|-i
: specifies a hex string (not prefixed with 0x) of any number of bytes. For asymmetric keys the same ID is set for both, the public and the private key.

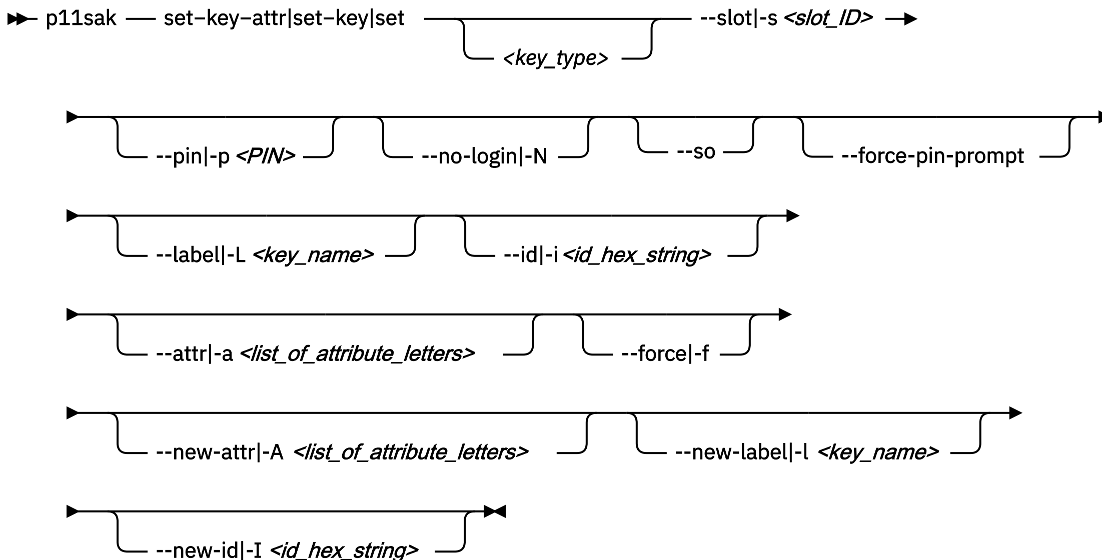
--attr|-a
: Attributes whose letters are not specified in this list are not used to filter the keys.

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

Setting attributes of keys

Use the **p11sak set-key-attr** subcommand and the optional `key_type` argument to set or update boolean attributes of symmetric or asymmetric keys. Public, private, secret, or all keys can also be selected for updating, irrespective of the key type.

p11sak syntax - Setting attributes of keys



Most of the options are the same as used for and explained in “p11sak syntax - Generating keys” on page 44 .

Use the **<key_type>** parameter to select the type or types of keys to be updated. Possible values for the **<key_type>** parameter are the following (as explained in “p11sak syntax - Generating keys” on page 44 and “Listing keys in the repository” on page 47):

```
des|3des|generic|aes|aes-xts|rsa|dh|dsa|ec|ibm-dilithium|ibm-kyber|public|private|secret|all
```

If **<key_type>** is omitted, then all key types are selected for updating.

Specify the `--label|-L <key_name>`, the `--id|-i <id_hex_string>`, or the `--attr|-a <list_of_attribute_letters>` options to filter the keys to be updated. The **`<id_hex_string>`** must be specified as hex string (not prefixed with 0x) of any number of bytes. You can use wildcards (* and ?) in the **<key_name>** specification.

Use the `--new-attr|-A <list_of_attribute_letters>` option to specify the boolean attributes of the key you want to update. Keys can be filtered by all attributes, setting is possible for all except L `| A | N | Z`. An uppercase letter sets the corresponding attribute to CK_TRUE, a lower case letter to CK_FALSE. If an attribute is not set explicitly, its value is not changed. Not all attributes may be allowed to be changed for all key types, or to all values.

The boolean attributes to set for the filtered key or keys:

```
P | M | B | Y | R | E | D | G | C | V | O | W | U | S | X | T | I | K
p | m | b | y | r | e | d | g | c | v | o | w | u | s | x | t | i | k
```

Use the `--new-label|-l <key_name>` option to specify the new label, or use the `--new-id|-I <id_hex_string>` option to specify the new ID to be set for the key.

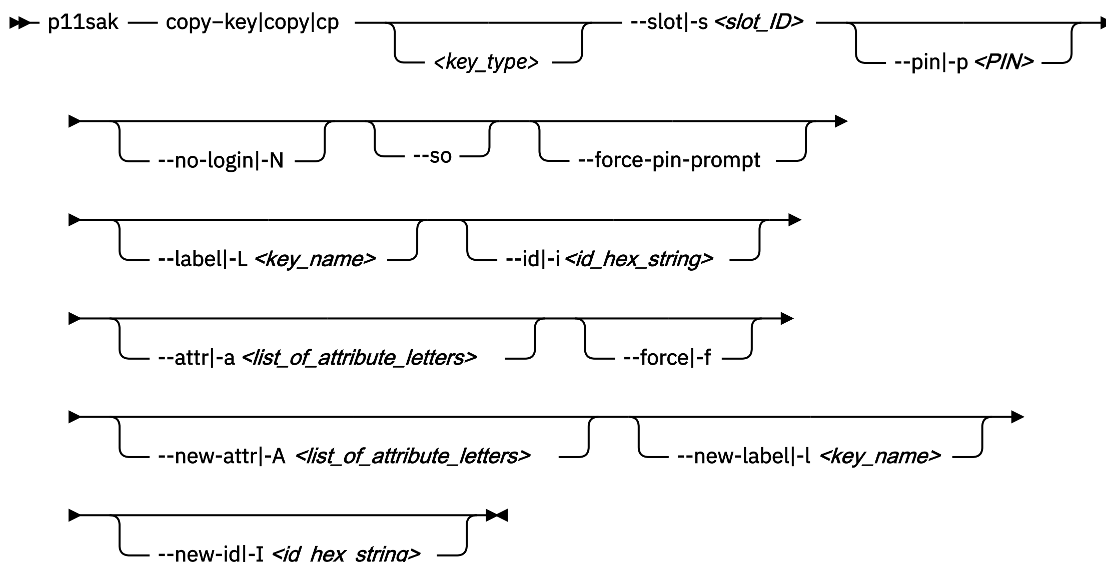
At least one of the `--new-attr|-A <list_of_attribute_letters>`, `--new-label|-l <key_name>`, or `--new-id|-I <id_hex_string>` options must be specified.

You are prompted to confirm the updating of the key or keys. To suppress the prompt, use the `--force|-f` option.

Copying keys in the repository

Use the **p11sak copy-key** subcommand to copy symmetric or asymmetric keys. You can change attributes, the label, or the ID of the copied key or keys.

p11sak syntax - Copying keys in the repository



The parameters are the same as for "Setting attributes of keys" on page 51.

Use the **<key_type>** parameter to select the type or types of keys to be copied. Possible values for the **<key_type>** argument are the following (as explained in "p11sak syntax - Generating keys" on page 44 and "Listing keys in the repository" on page 47):

```
des|3des|generic|aes|aes-xts|rsa|dh|dsa|ec|ibm-dilithium|ibm-kyber|public|private|secret|all
```

If **<key_type>** is omitted, then all key types are selected for copying.

Besides filtering the keys to be copied by the key types, use the `--attr|-a <list_of_attribute_letters>` options to additionally filter the keys with matching key types by their attributes.

Use the `--new-attr|-A <list_of_attribute_letters>` option to specify the boolean attributes to be set for the copied key(s). Attributes that are not specified are not set. Restrictions on attribute values may apply.

Keys can be filtered by all attributes, setting is possible for all except `L | A | N | Z`. An uppercase letter sets the corresponding attribute to CK_TRUE, a lower case letter to CK_FALSE. If an attribute is not set explicitly, its value is not changed. Not all attributes may be allowed to be changed for all key types, or changed to all values.

The boolean attributes to set for the key or keys to be copied:

```
P | M | B | Y | R | E | D | G | C | V | O | W | U | S | X | T | I | K
p | m | b | y | r | e | d | g | c | v | o | w | u | s | x | t | i | k
```

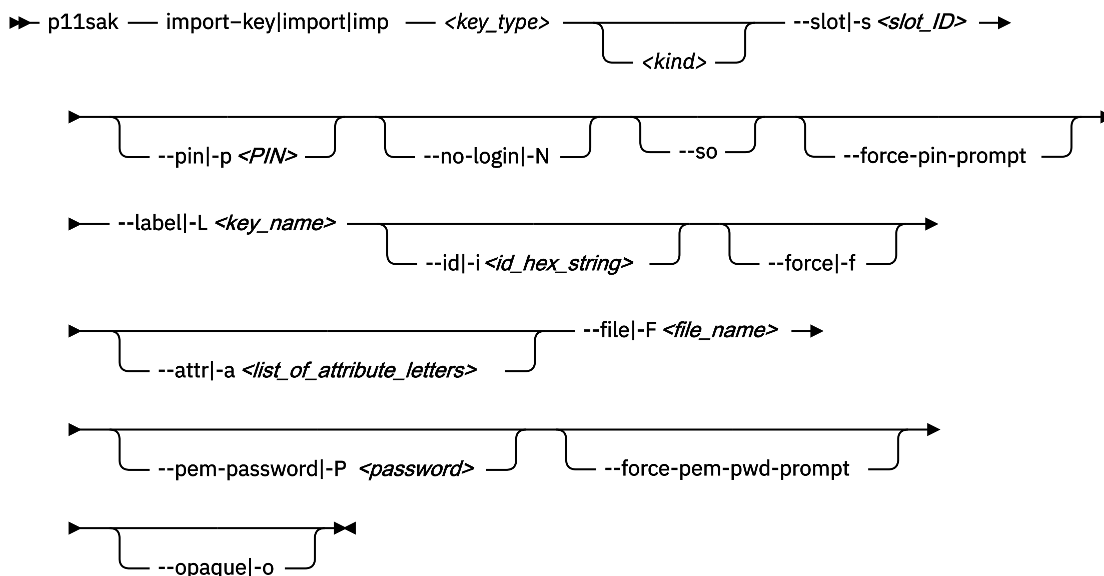
Use the `--new-label|-l <key_name>` option to specify the new label to be stet for the copied key or keys (optional).

Use the `--new-id|-I <id_hex_string>` option to specify the new ID to be set for the copied key or keys (optional).

Importing keys from a binary file or a PEM file

Use the **p11sak import-key** subcommand together with the <key_type> argument to import symmetric or asymmetric keys from a file.

p11sak syntax - Importing keys from a binary file or a PEM file



The options and parameters that are specific to the **import-key** subcommand have the following meaning:

<kind>

When importing an asymmetric key, the <kind> argument is required and specifies to either import a private or public key.

<key_type>

Possible values for the <key_type> argument are:

```
des|3des|generic|aes|aes-xts|rsa|dh|dsa|ec|ibm-dilithium|ibm-kyber|public|private|secret|all
```

--pem-password|-P

--opaque|-o

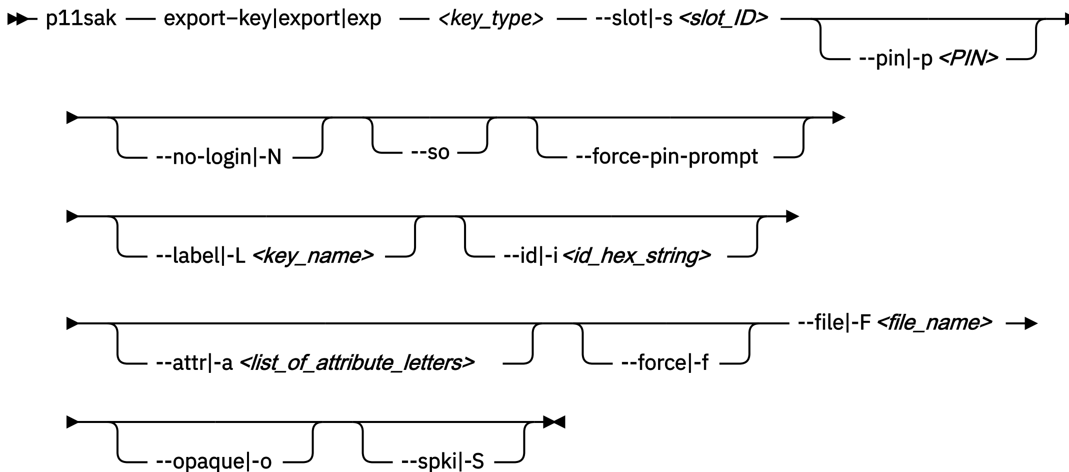
Specify this option to import the opaque secure key blobs of the key. Not all tokens support this.

Other options have a similar purpose as in previously described options. The --label|-L <key_name> option sets the CKA_LABEL attribute of the key and the option --attr|-a <list_of_attribute_letters> can be used to set the boolean attributes of the key as described in “Generating keys in the openCryptoki repository” on page 43. Use the --id|-i <id_hex_string> option to set the value of the CKA_ID attribute of the key.

Exporting keys to a binary file or a PEM file

Use the **p11sak export-key|export|exp** subcommand and the optional <key_type> parameter to export symmetric and asymmetric keys to a file. Public, private, secret, or all keys can also be selected for export, irrespective of the key type.

p11sak syntax -Exporting keys to a binary file or a PEM file



Possible values for the <key_type> argument are:

```
des|3des|generic|aes|aes-xts|rsa|dh|dsa|ec|ibm-dilithium|ibm-kyber|public|private|secret|all
```

If <key_type> is omitted, then all key types are selected for export.

Specify one or more of the --label|-L <key_name>, the --id|-i <id_hex_string>, or the --attr|-a <list_of_attribute_letters> options to filter the list of keys for which to change the attributes. You can use wildcards (* and ?) in the <key_name> specification. The --id|-i option must be specified as hex string (not prefixed with 0x) of any number of bytes.

The --file|-F <file_name> option specifies the file name of the file to which the keys to be exported are written to. For symmetric keys, this is a binary file where the key material in clear is written to. For asymmetric keys, this is an OpenSSL PEM file where the public or private keys are written to. If multiple asymmetric keys match the filter, the keys are appended to the PEM file specified with the --file|-F <file_name> option. If multiple symmetric keys or a mixture of asymmetric and symmetric keys match the filter, then you are prompted to confirm to overwrite the previously created file, unless the [--force|-f] option is specified.

Specify the --opaque|-o option to export the opaque secure key blobs of the key. Not all tokens support this.

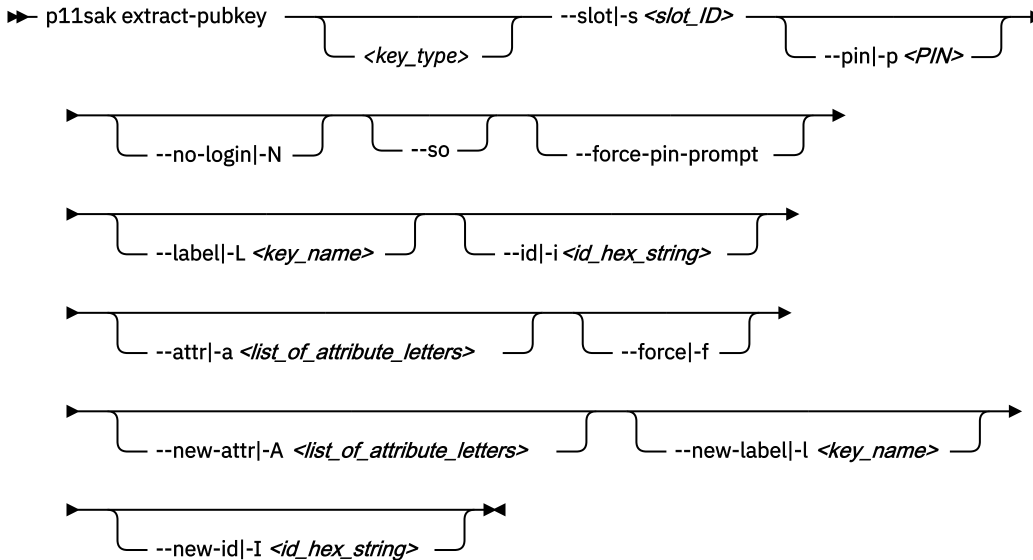
Specify the --spki|-S option to export the Subject Public Key Info (SPKI) from the CKA_PUBLIC_KEY_INFO attribute of an asymmetric private key instead of its private key material. This option can only be used with private keys.

Note: Not all keys can be exported, because its attribute setting may forbid to reveal the values of certain attributes. To allow exporting of a secret (CKO_SECRET_KEY) or private (CKO_PRIVATE_KEY) key, attribute CKA_SENSITIVE must be CK_FALSE and attribute CKA_EXTRACTABLE must be CK_TRUE. Secret or private keys that contain an opaque secure key blob (attribute CKA_IBM_OPAQUE) can also not be exported in clear, even if the attributes would allow it. For such keys only the opaque secure key blob can be exported by using the--opaque|-o option.

Extracting the public key from a private key

Use the **p11sak extract-pubkey** subcommand to extract the public key from a private key. Use the label, the ID and existing attributes to filter the private key from which you want to extract the public key.

p11sak syntax - Extracting the public key from a private key



where the following arguments and options have a specific use or meaning for this subcommand:

<key_type>
Optional. The type of the private key from which you want to extract the public key. If no key type is specified, all private key types are selected. Possible values for the **`<key_type>`** parameter are:

```
rsa|dh|dsa|ec|ibm-dilithium|ibm-kyber|private|all
```

--attr|-a
Optional. Filter the key by its boolean attribute values:

```
P | L | M | B | Y | R | E | D | G | C | V | O | W | U | S | A | X | N | T | I | K | Z
```

Specify a set of these letters without any blanks in between. The meaning for each of the attribute letters is explained in “Generating keys in the openCryptoki repository” on page 43. Attributes that are not specified are not used to filter the keys.

--new-attr|-A
Optional. The boolean attributes to set for the extracted public key:

```
P | M | B | Y | R | E | D | G | C | V | O | W | U | S | X | T | I | K
p | m | b | y | r | e | d | g | c | v | o | w | u | s | x | t | i | k
```

Specify a set of these letters without any blanks in between. The meaning for each of the attribute letters is explained in “Generating keys in the openCryptoki repository” on page 43. Restrictions on attribute values may apply.

Note: Keys can be filtered by all attributes, setting is possible for all except `L | A | N | Z`.

An uppercase letter sets the corresponding attribute to CK_TRUE, a lower case letter to CK_FALSE. If an attribute is not set explicitly, its value is not changed. Not all attributes may be allowed to be changed for all key types, or to all values.

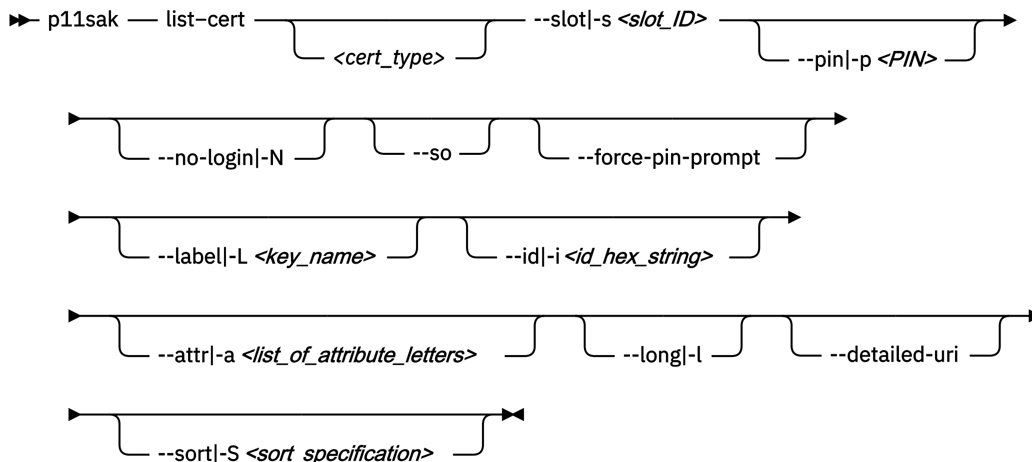
--new-label|-l
The new label to set for the extracted public key (optional). If no new label is specified, the new label is derived from the private key label by appending _pubkey.

--new-id|-I
Optional. The new ID to be set for the extracted public key.

Listing certificates in the repository

Use the **p11sak list-cert** subcommand to show existing certificates that match the specified parameters. As for keys, there is a short and a long output. Matching certificates can be sorted by label and common name.

p11sak syntax - Listing certificates



where the following arguments and options have a specific use or meaning for this subcommand:

<cert_type>
The type of the certificate to be listed. If no certificate type is specified, certificate type `x509` is used.

--attr|-a
Filter the certificate by its boolean attribute values: P M B Y T (optional). Specify a set of these letters without any blanks in between. Attributes that are not specified are not used to filter the certificates. For the meaning of these attributes, see "Generating keys in the openCryptoki repository" on page 43.

--sort|-S <sort_specification>
Sort the certificates by label or subject common name (CN), or both. Specify a sort specification of up to two fields, each represented by its corresponding letter, separated by comma:

l
label

n
subject common name

The sort order (default: a = ascending, d = descending) can be appended to the criteria designators by a colon. Example: `l:a,n:d` sorts by label in ascending order and then by common name in descending order.

All other options are explained in "p11sak syntax - Generating keys" on page 44 or in "Listing keys in the repository" on page 47.

Example of a short output

```
# p11sak list-cert x509 --slot 4 --pin 11111111

| P M B Y T | CERT TYPE | SUBJECT-CN            | LABEL
|-----------+-----------+-----------------------+------------------------
| 1 1 1 1 - | X.509     | /C=DE/ST=BB/L=BB/O=... | mycert2
| 0 1 1 1 - | X.509     | system1               | mycert1
| 0 1 1 1 - | X.509     | user1                 | dsacert1
```

To obtain a long output, specify a command similar to the following:

```
# p11sak list-cert x509 --slot 4 --pin 11111111 --long
```

Removing certificates from the repository

Use the **p11sak remove-cert** subcommand to delete certificates in the repository.

p11sak syntax - Removing certificates

```
p11sak remove–cert [<cert_type>] --slot|-s <slot_ID> [--pin|-p <PIN>]
  [--no-login|-N] [--so] [--force-pin-prompt]
  [--label|-L <key_name>] [--id|-i <id_hex_string>]
  [--attr|-a <list_of_attribute_letters>] [--force|-f]
```

The arguments and options have the same meaning as described in "p11sak syntax - Generating keys" on page 44 or "Listing certificates in the repository" on page 57.

Setting or updating attributes of certificates

Use the **p11sak set-cert-attr** subcommand to set or update boolean attributes of certificates. Optionally, a new label, a new ID, and new attributes can be specified for the token object.

p11sak syntax - Setting or updating attributes of certificates

```
p11sak set-cert-attr <cert_type> --slot|-s <slot_ID> [--pin|-p <PIN>]
  [--no-login|-N] [--so] [--force-pin-prompt]
  [--label|-L <key_name>] [--id|-i <id_hex_string>]
  [--attr|-a <list_of_attribute_letters>] [--force|-f]
  [--new-attr|-A <list_of_attribute_letters>] [--new-label|-l <key_name>]
  [--new-id|-I <id_hex_string>]
```

where the following argument is specific for this subcommand:

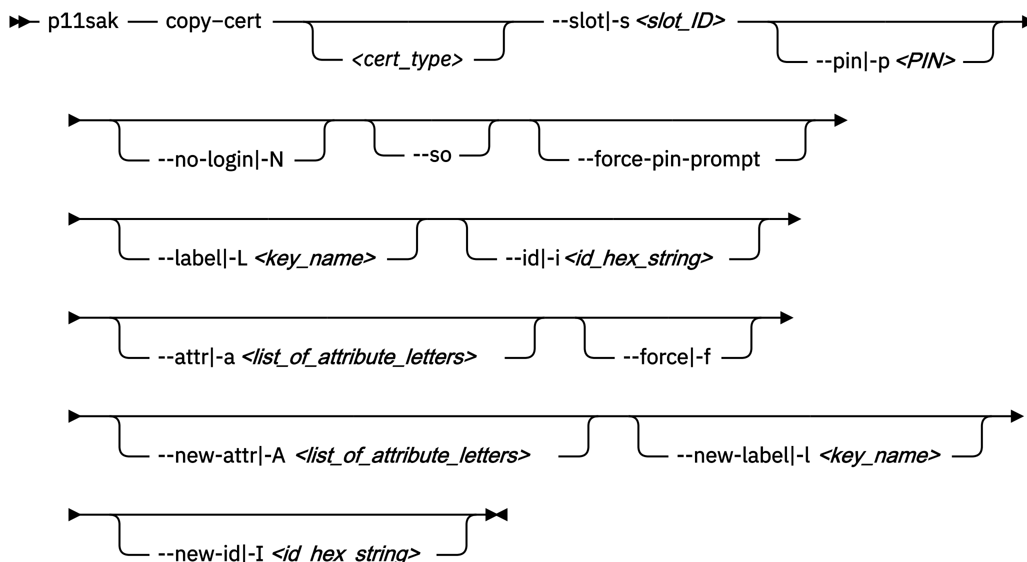
<cert_type>
: The type of the certificate. Must be set to x509.

The other arguments and options have the same meaning as described in "Setting attributes of keys" on page 51 or "Listing certificates in the repository" on page 57.

Copying certificates in the repository

Use the **`p11sak copy-cert`** subcommand to copy certificates in the repository.

p11sak syntax - Copying certificates in the repository



The following arguments and options have a specific use or meaning for this subcommand:

<cert_type>
: The type of the certificate to be listed. If no certificate type is specified, certificate type `x509` is used.

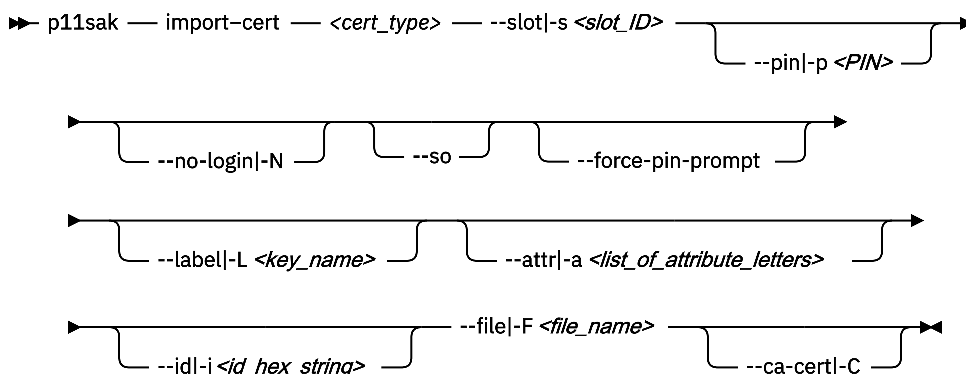
--attr|-a
: Filter the certificate by its boolean attribute values: P M B Y T (optional). Specify a set of these letters without any blanks in between. Attributes that are not specified are not used to filter the certificates. For the meaning of these attributes, see "Generating keys in the openCryptoki repository" on page 43.

All other options and parameters are the same as for "Generating keys in the openCryptoki repository" on page 43 or for "Setting attributes of keys" on page 51.

Importing certificates from a binary file or a PEM file

Use the **`p11sak import-cert`** subcommand to import x.509 public certificates into the openCryptoki token repository. Importing is supported in two formats: Base64 (PEM) form and binary (DER-encoded) form. The format is detected from the given input and does not need to be specified as input parameter.

p11sak syntax - Importing certificates from a binary file or a PEM file



where the following arguments and options are specific for this subcommand:

<cert_type>
: The type of the certificate. Must be set to x509.

--attr|-a
: The boolean attributes to set for the certificate: P M B Y (optional). Specify a set of these letters without any blanks in between. For the meaning of these attributes, see "Generating keys in the openCryptoki repository" on page 43.

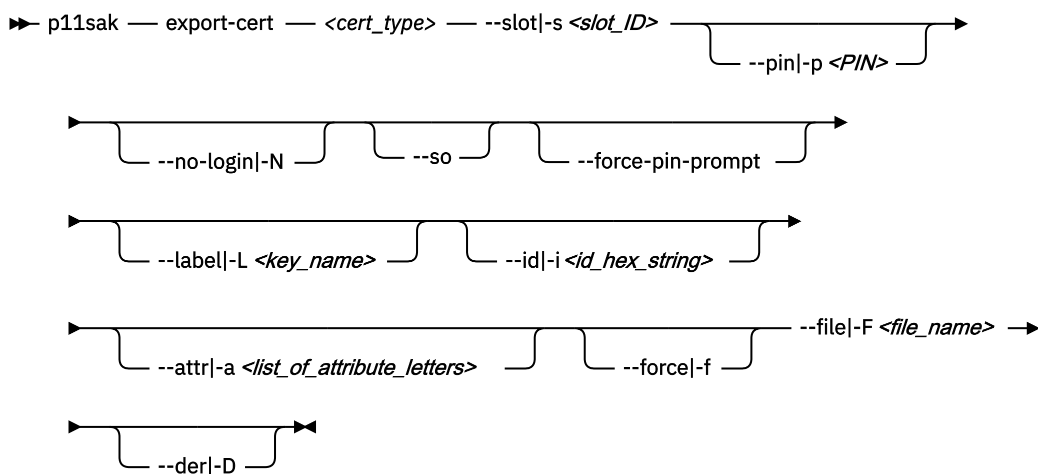
--file|-F
: The name of the file that contains the certificate to be imported. Supported input formats are PEM and binary (DER-encoded). The format is automatically detected.

--ca-cert|-C
: The certificate is a Certificate Authority (CA) certificate.

Exporting certificates to a binary file or PEM file

Use the **p11sak export-cert** subcommand to export certificates. Exporting of public key certificates is supported in two formats: Base64 (PEM) and binary (DER encoding). The default is PEM. Exporting multiple certificates into the same output file is only possible in PEM format.

p11sak syntax -Export certificates to a binary file or a PEM file



where the following arguments and parameters are specific for this subcommand:

<cert_type>
: The type of the certificate. Must be set to x509.

--attr|-a
: Filter the certificate by its boolean attribute values: P M B Y T (optional). Specify a set of these letters without any blanks in between. Attributes that are not specified are not used to filter the certificates. For the meaning of these attributes, see "Generating keys in the openCryptoki repository" on page 43.

--der|-D
: The certificate is written to the file in binary (DER-encoded) form. Default is Base64 (PEM).

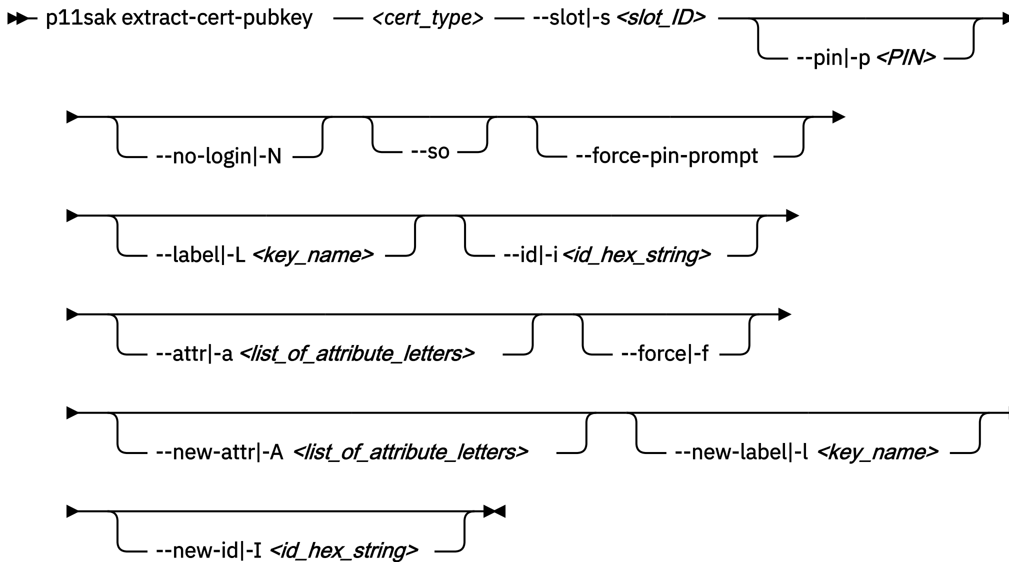
The other options and parameters are the same as for "Exporting keys to a binary file or a PEM file" on page 54.

Extracting public keys from certificates

Use the **p11sak extract-cert-pubkey** subcommand to extract the public key of a given certificate into a new token object. Supported public key types are RSA, EC, and DSA. If no new label is specified as

input parameter, the labels for extracted public keys are derived from the certificate labels by appending *_pubkey*.

p11sak syntax - Extract the public keys from certificates



where the following argument is specific for this subcommand:

<cert_type>
: The type of the certificate. Must be set to `x509`.

The other parameters are the same as for "Extracting the public key from a private key" on page 55.

Command help

Request general help with the following command:

```
# p11sak -h
```

to receive the following information:

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

COMMANDS:
    generate-key                       Generate a key.
    list-key                           List keys in the repository.
    remove-key                         Delete keys in the repository.
    set-key-attr                       Set attributes of keys in the repository.
    copy-key                           Copy keys in the repository.
    import-key                         Import a key from a binary file or PEM file.
    export-key                         Export keys to a binary file or PEM file.
    extract-pubkey                     Extract the public key from private keys in
                                       the repository.
    list-cert                          List certificates in the repository.
    remove-cert                        Delete certificates in the repository.
    set-cert-attr                      Set attributes of certificates in the
                                       repository.
    copy-cert                          Copy certificates in the repository.
    import-cert                        Import a certificate from a binary file or
                                       PEM file.
    export-cert                        Export certificates to a binary file or PEM
                                       file.
    extract-cert-pubkey                Extract the public key from certificates in
                                       the repository.


COMMON OPTIONS
    -h, --help                         Print this help, then exit.
    -v, --version                      Print version information, then exit.

For more information use 'p11sak COMMAND --help'.
```

Request help for a subcommand:

```
# p11sak copy-key -h
# p11sak export-key -h
# p11sak extract-pubkey -h
# p11sak generate-key -h  (see Figure 9 on page 63 and Figure 10 on page 64)
# p11sak import-key -h
# p11sak list-key -h
# p11sak remove-key -h
# p11sak set-key-attr -h

# p11sak copy-cert -h
# p11sak export-cert -h
# p11sak extract-cert-pubkey -h
# p11sak import-cert -h
# p11sak list-cert -h
# p11sak remove-cert -h
# p11sak set-cert-attr -h
```

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

```
# p11sak generate-key -h

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

ARGS:
    KEYTYPE                        The type of the key. One of the following:
        des
        3des
        generic KEYBITS

            KEYBITS                Size of the generic key in bits.
            ...
        rsa KEYBITS [PUBL-EXP]

            KEYBITS                Size of the RSA key in bits:
                512
                ...
            PUBL-EXP               The public exponent for RSA (optional).

        dh GROUP [PRIV-BITS]

            GROUP                  The Diffie-Hellman FFC group name or the
                                   name of a DH parameters PEM file:
                ffdhe2048
                ...
                modp1536
                ...
                DH-PARAM-PEM-FILE
            PRIV-BITS              Size of the DH private key in bits
                                   (optional).

        dsa DSA-PARAM-PEM-FILE     The name of a DSA parameters PEM file.

        ec CURVE

            CURVE                  The curve name. One of the following:
                prime256v1
                ...
                ed448

        ibm-dilithium VERSION

            VERSION                The version of the IBM Dilithium key pair:
                r2_65
                ...
        ibm-kyber VERSION

            VERSION                The version of the IBM Kyber key pair:
                r2_768
                ...

OPTIONS:
    -s, --slot SLOT                The PKCS#11 slot ID.
    -p, --pin USER-PIN             The PKCS#11 user pin. If this option is not
                                   specified, ...
        --force-pin-prompt         Enforce user PIN prompt, even if environment
                                   variable PKCS11_USER_PIN is set, ...
    -L, --label LABEL              The label of the key to be generated.
                                   ...
    -a, --attr ATTRS               The boolean attributes to set for the key:
                                   P L M B Y R E D G C V O W U S A X N T I   ...
    -i, --id ID                    The ID of the key to be generated.   ...
    -h, --help                     Print this help, then exit.
    -v, --version                  Print version information, then exit.

ATTRIBUTES:
    'P':   CKA_PRIVATE
    'M':   CKA_MODIFIABLE
    ...

    An uppercase letter sets the corresponding attribute to CK_TRUE, a lower case letter to
CK_FALSE.
    If an attribute is not set explicitly, its default value is used.   ...
```

Figure 9. Request help for generate-key

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

Figure 10. Request help for generating AES keys

```
# p11sak generate-key aes -h

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

ARGS:
    KEYTYPE                          The type of the key. One of the following:
        aes KEYBITS

            KEYBITS                  Size of the AES key in bits:
                128
                192
                256

OPTIONS:
    -s, --slot SLOT                  The PKCS#11 slot ID.
    -p, --pin USER-PIN               The PKCS#11 user pin. If this option is not
                                     specified, ...
        --force-pin-prompt           Enforce user PIN prompt, even if environment
                                     variable PKCS11_USER_PIN is set, ...
    -L, --label LABEL                The label of the key to be generated. For
                                     asymmetric keys set individual labels for
                                     public and private key, ...
    -a, --attr ATTRS                 The boolean attributes to set for the key:
                                     P L M B Y R E D G C V O W U S A X N T I
                                     (optional). ...
    -i, --id ID                      The ID of the key to be generated. ...
    -h, --help                       Print this help, then exit.
    -v, --version                    Print version information, then exit.

ATTRIBUTES:
    'P':   CKA_PRIVATE
    'M':   CKA_MODIFIABLE
    'B':   CKA_COPYABLE
 ...

    An uppercase letter sets the corresponding attribute to CK_TRUE, a lower
    case letter to CK_FALSE.
    If an attribute is not set explicitly, its default value is used.
    Not all attributes may be accepted for all key types.
    Attribute CKA_TOKEN is always set to CK_TRUE.
```

Chapter 10. Migrating to FIPS compliance - pkcstok_migrate utility

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

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

Parameters

```
# pkcstok_migrate -h

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

Options:

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

Functionality

The utility:

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

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

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

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

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

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

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

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

© Copyright IBM Corp. 2021, 2023

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


service pkcsslotd start
```

The output may look similar to the following:

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

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

Chapter 11. Displaying usage statistics - pkcsstats utility

openCryptoki provides a command line program **pkcsstats** to display usage statistics of mechanisms per slot IDs, either on the basis of individual users or accumulated for all users, and broken down to available key-sizes.

Description

The **pkcsstats** utility displays mechanism usage statistics for openCryptoki. Usage statistics are collected per user. For each user, mechanism usage is counted per configured slot and mechanism. For each mechanism, one counter exists for each cryptographic strength of the key used with the mechanism. The available strengths are defined in the strength configuration file `/etc/opencryptoki/strength.conf` (see "Strength configuration file" on page 25). A strength of zero is used to count those mechanisms that do not use a key, or where the key strength is less than 112 bits.

Note: The strength does not specify the cryptographic strength of the mechanism, but the cryptographic strength of the key used with the mechanism (if any). For example, usage of mechanism CKM_SHA256 is reported under strength 0, because no key is used with this mechanism. However, usage of mechanism CKM_AES_CBC is reported under strength 128, 192, or 256, dependent on the cryptographic size of the used AES key.

Statistics collection is enabled by default. You can disable and configure the collection in the openCryptoki configuration file `/etc/opencryptoki/opencryptoki.conf` using option `statistics (off|on[,implicit][,internal])`. For example, you can collect only mechanisms that are explicitly called by a PKCS #11 application or also collect those mechanisms that are called implicitly by another mechanism. So before using the **pkcsstats** utility, be sure to understand the information provided in "Collecting statistics" on page 23.

The usage of a mechanism is counted once when the cryptographic operation is successfully initialized, that is, during `C_DigestInit()`, `C_EncryptInit()`, `C_DecryptInit()`, `C_SignInit()`, `C_SignRecoverInit()`, and `C_VerifyInit()`. Multi-part operations involving the update functions like `C_DigestUpdate()`, `C_EncryptUpdate()`, `C_DecryptUpdate()`, `C_SignUpdate()`, and `C_VerifyUpdate()`, are not counted additionally. Other operations such as key generation, key derivation, key wrapping and unwrapping are counted during the respective functions like `C_GenerateKey()`, `C_GenerateKeyPair()`, `C_DeriveKey()`, `C_UnwrapKey()`.

Statistics are collected in a POSIX shared memory segment per user. This shared memory segment contains all counters for all configured slots, mechanisms, and key-strengths. The shared memory segments are named var.lib.opencryptoki_stats_<uid>, where <uid> is the numeric user ID of the user the statistics belong to. The shared memory segments are automatically created for a user on the first attempt to collect statistics. All users can only display their own statistics, and only the root user can display all users' statistics.

Invocation and usage

Get an overview about available options using the **pkcsstats -h** command:

```
# pkcsstats -h
Usage: pkcsstats [OPTIONS]

Display mechanism usage statistics for openCryptoki.

OPTIONS:
-U, --user USERID show the statistics from one user. (root user only)
-S, --summary show the accumulated statistics from all users. (root user only)
-A, --all show the statistic tables from all users. (root user only)
-a, --all-mechs show all mechanisms, also those with all zero counters.
-s, --slot SLOTID show the statistics from one slot only.
-r, --reset set the own statistics to zero.
```

© Copyright IBM Corp. 2021, 2023

```
 -R, --reset-all reset the statistics from all users. (root user only)
 -d, --delete delete your own statistics.
 -D, --delete-all delete the statistics from all users. (root user only)
 -j, --json output the statistics in JSON format.
 -h, --help display help information.
```

where

-U, --user <user_id>
: specifies the user ID of the user who wants to display, reset, or delete statistics. If this option is omitted, the statistics of the current user are displayed, reset, or deleted. Only the root user can display, reset, or delete statistics of other users.

-S, --summary
: shows the accumulated statistics from all users. Only the *root* user can display the accumulated statistics from other users.

–A, ––all
: shows the statistics from all users. Only the *root* user can display statistics from all users.

–a, ––all–mechs
: shows the statistics for all mechanisms, also those with all–zero counters. If this option is omitted, only those mechanisms are displayed where at least one counter is non–zero.

–s, ––slot <slot_id>
: specifies the slot ID for which to display statistics. If this option is omitted, the statistics for all configured slots are displayed.

–r, ––reset
: resets the statistics counters for the current user, or for the user specified with the `––user` option. Only the *root* user can reset the statistics from other users.

–R, ––reset–all
: resets the statistics counters for all users. Only the *root* user can reset the statistics from other users.

–d, ––delete
: deletes the shared memory segment containing the statistics counters for the current user, or for the user specified with the `––user` option. Only the root user can delete the statistics from other users.

–D, ––delete–all
: deletes the shared memory segment containing the statistics counters for all users. Only the root user can delete the statistics from other users.

–j, ––json
: shows the statistics in JSON format to obtain the values in a machine readable format for automatic processing.

–h, ––help
: displays help text and exits.

Example (default table format):

To obtain the statistics for the *root* user in slot 2, where a CCA token is installed, enter the shown command:

```
# pkcsstats -s 2 -U root
User: root
Slot: 2 (label: 'cca' model: 'CCA')
-------------------------------+--------------------------------------------------------------------------
mechanism                      | strength 0     strength 112    strength 128   strength 192   strength 256
                               | or no key
-------------------------------+--------------------------------------------------------------------------
CKM_AES_CBC                    |          0                0              16             16             16
CKM_AES_CBC_PAD                |          0                0               8              8              8
CKM_AES_ECB                    |          0                0              18             16             16
CKM_AES_KEY_GEN                |          0                0              22             20             20
CKM_RSA_PKCS_KEY_PAIR_GEN      |         24                0               0              0              0
-------------------------------+--------------------------------------------------------------------------
```

Figure 11. pkcsstats output in table format

Example (JSON format):

The example from Figure 11 on page 68 requested in JSON format looks like shown in Figure 12 on page 69:

```
[root@system1 opencryptoki]# pkcsstats -s 2 -U root --json
"host": {
                "nodename": "a1251960.domain.com",
                "sysname": "system1",
                "release": "6.0.7-301.fc37.s390x",
                "machine": "s390x",
                "date": "2023-06-02T11:14:25Z"
        },
        "users": [
                {
                        "user": "root",
                        "slots": [
                                {
                                        "slot": 2,
                                        "token-present": true,
                                        "label": "cca",
                                        "model": "CCA",
                                        "mechanisms": [
                                                {
                                                        "mechanism": "CKM_AES_CBC",
                                                        "strength-0": 0,
                                                        "strength-112": 0,
                                                        "strength-128": 16,
                                                        "strength-192": 16,
                                                        "strength-256": 16
                                                },
                                                {
                                                        "mechanism": "CKM_AES_CBC_PAD",
                                                        "strength-0": 0,
                                                        "strength-112": 0,
                                                        "strength-128": 8,
                                                        "strength-192": 8,
                                                        "strength-256": 8
                                                },
                                                {
                                                        "mechanism": "CKM_AES_ECB",
                                                        "strength-0": 0,
                                                        "strength-112": 0,
                                                        "strength-128": 18,
                                                        "strength-192": 16,
                                                        "strength-256": 16
                                                },
                                                {
                                                        "mechanism": "CKM_AES_KEY_GEN",
                                                        "strength-0": 0,
                                                        "strength-112": 0,
                                                        "strength-128": 22,
                                                        "strength-192": 20,
                                                        "strength-256": 20
                                                },
                                                {
                                                        "mechanism": "CKM_RSA_PKCS_KEY_PAIR_GEN",
                                                        "strength-0": 24,
                                                        "strength-112": 0,
                                                        "strength-128": 0,
                                                        "strength-192": 0,
                                                        "strength-256": 0
                                                }
                                        ]
                                }
                        ]
                }
        ]
}
```

Figure 12. pkcsstats output in JSON format

For further information you can read the **pkcsstats** man page.

Chapter 12. Managing a concurrent master key change - pkcshsm_mk_change utility

For CCA tokens and EP11 tokens, openCryptoki provides a command line program **pkcshsm_mk_change** to manage the concurrent re-enciphering of secure keys for a HSM master key change while applications using openCryptoki workload are running.

Description

The **pkcshsm_mk_change** utility initiates and maintains master key changes for openCryptoki. It manages and controls the re-enciphering of secure keys for a concurrent HSM master key change. It maintains the state of each master key change operation on files stored in the file system (that is, in directory `/var/lib/opencryptoki/HSM_MK_CHANGE/`). Multiple master key change operations can be ongoing concurrently to running workloads. Concurrency between multiple master key change operations themselves is only possible if these changes do not apply to the same APQNs and the same master key types simultaneously. Such a concurrency is only possible on the same APQN for different master key types or on different APQNs.

Notes:

- A master key is called wrapping key in EP11. In the **pkcshsm_mk_change** utility, the term *master key* is generally used and applies to both and therefore, this term might denote an EP11 wrapping key.

Coordination of a concurrent master key rotation

Your security policies may require that in certain time intervals a new master key must be generated, for example, using a Trusted Key Entry workstation. This new master key must be loaded onto the cryptographic coprocessors . Changing master keys needs to be coordinated between the HSM security officer and an openCryptoki administrator and possibly also between further applications. All secure keys enciphered by the master key need to be re-enciphered with the new master key as part of the master key change process. However, when performing the master key change process with the help of **pkcshsm_mk_change**, applications can continue to run while the master key change procedure is performed.

Two parties are involved in a master key change on the HSM. The openCryptoki administrator uses the **pkcshsm_mk_change** tool to initiate and control a master key change operation, coordinated with the HSM security officer who performs the master key changes on the HSMs via the TKE (Trusted Key Entry) workstation. During the ongoing master key change, applications using openCryptoki can continue to run without being affected, besides possibly a slight performance degradation while master key change actions are performed.

A concurrent master key change works as follows:

1. The HSM security officer loads the new master key(s) using the TKE into the NEW register of the set of APQNs logically belonging together, that is, APQNs configured with the same master key.
2. The HSM security officer notifies the openCryptoki administrator that a new master key has been loaded for all the APQNs. Preferably, the HSM security officer also communicates the master key verification pattern of the new master key to the openCryptoki administrator.
3. The openCryptoki administrator uses the **pkcshsm_mk_change** tool to initiate a master key change for openCryptoki, specifying the set of APQNs that are to be changed, and the verification patterns of the new master keys to be set (per master key type). The tool communicates with the openCryptoki runtime used by the running applications and performs and controls the re-encipherment of the key objects with the new master key.

© Copyright IBM Corp. 2021, 2023

4. When the **pkcshsm_mk_change** tool has completed its re-encipherment processing, the openCryptoki administrator notifies the HSM security officer that openCryptoki is prepared to have the new master keys being activated.
5. The HSM security officer coordinates with other (non-openCryptoki) applications and once all users of the APQNs are prepared, he or she activates the new master keys on the APQNs.
6. The HSM security officer notifies the openCryptoki administrator as soon as for all APQNs the new master key has been activated.
7. The openCryptoki administrator uses the **pkcshsm_mk_change** tool to finalize the master key change for openCryptoki. The tool communicates with the openCryptoki runtime used by the running applications and performs and controls the finalization of the re-encipherment of the key objects with the new master key.
8. When the **pkcshsm_mk_change** utility has completed its finalizing processing, the master key change operation is complete.

The time between the moment when the new master key has been loaded on all APQNs, and the moment when the new master keys are activated can even last several days, due to the time required for coordination with other (non-openCryptoki) applications or users of the APQNs.

You can restart the Linux system where openCryptoki runs while a master key change is ongoing, provided that the re-encipherment and finalization steps (step "3" on page 71 and step "7" on page 72) are not interrupted.

You can cancel an ongoing master key change operation, as long as for none of the APQNs the new master key has been activated, that is up to step "5" on page 72.

A backup of the old secure keys is kept in attribute CKA_IBM_OPAQUE_OLD of the key objects. In case something goes wrong, you can restore the old secure keys from that attribute. For this purpose, you must implement a PKCS #11 application accessing all the key objects through regular PKCS #11 API calls to restore the secure keys by moving the secure keys from CKA_IBM_OPAQUE_OLD to CKA_IBM_OPAQUE.

Considerations for a master key rotation on CCA cryptographic coprocessors

A CCA cryptographic adapter has four different master key types:

SYM
: A Triple-DES master key, used to encipher DES and Triple-DES secure keys.

ASYM:
: A Triple-DES master key, used to encipher older RSA secure keys.

AES
: An AES master key, used to encipher AES secure keys.

APKA
: An AES master key, used to encipher ECC secure keys, and newer RSA secure keys (those with section X'30' and X'31'). This master key type is also used for QSA key types in CCA, wich are currently not yet supported by openCryptoki.

All four master keys can be changed together, or individually. Consequently, though this is not typical for a security environment, one can initiate multiple concurrent master key change operations, separated for the different CCA master key types, possibly for the same APQNs. An openCryptoki CCA token uses only the SYM, AES and APKA master keys. The ASYM master key is not used, because only newer RSA secure key types are used.

A CCA cryptographic adapter has three registers per master key type:

NEW
: New master keys are loaded into this register. Master keys in the NEW register can only be used to re-encipher secure keys.

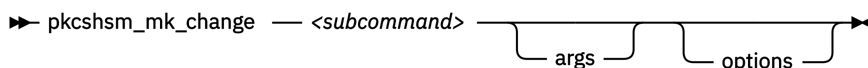
CURRENT
: The current master key to perform cryptographic operations with secure keys.

OLD
: The previous master key. Secure keys enciphered with the master key in the OLD register can still be used to perform cryptographic operations.

Invocation and usage

The general invocation scheme of the command line tool is:

pkcshsm_mk_change syntax - General invocation scheme



where

subcommand
: may accept one of the following values:

- **`reencipher`** (see "Initiate a master key change plus re-encryption for openCryptoki" on page 73)
- **`list`** (see "List master key change operations" on page 75)
- **`finalize`** (see "Finalize a master key change for openCryptoki" on page 75)
- **`cancel`** (see "Cancel a master key change for openCryptoki" on page 76)

For further information you can read the **`pkcshsm_mk_change`** man page.

Initiate a master key change plus re-encryption for openCryptoki

Use the **`pkcshsm_mk_change reencipher`** subcommand to initiate a master key change for the specified APQNs and the specified master key (wrapping key) types together with the verification patterns of the new master keys to be set. This subcommand also re-enciphers all session and token key objects of the affected tokens. It stores information persistently, returns and prints the ID of the initiated master key change operation. You must use this subcommand at the point in time when all APQNs have the new master key loaded but not yet set (that is, not yet activated). This subcommand tells the openCryptoki runtime within all applications using openCryptoki to re-encipher their key objects. It waits until all applications' openCryptoki runtime instances have completed to re-encipher their key objects. For each master key type that is changed, the verification pattern of the new, to be set master key must be specified.

A cryptographic adapter in CCA coprocessor mode can have four different types of master keys: SYM, ASYM, AES, and APKA. The CCA token of openCryptoki only uses SYM, AES, and APKA. Each master key type can be changed individually, or together with others. You can use the TKE or the **`panel.exe`** tool to query the master key verification patterns issuing the following command:

```
panel.exe --mk-query --mktype=<type> --mkregister=NEW
```

For master key types SYM and ASYM, use the hex string under [RND], for types AES and APKA use the hex string under [VER]. For AES and APKA you can also find the master key verification patterns in `sysfs`:

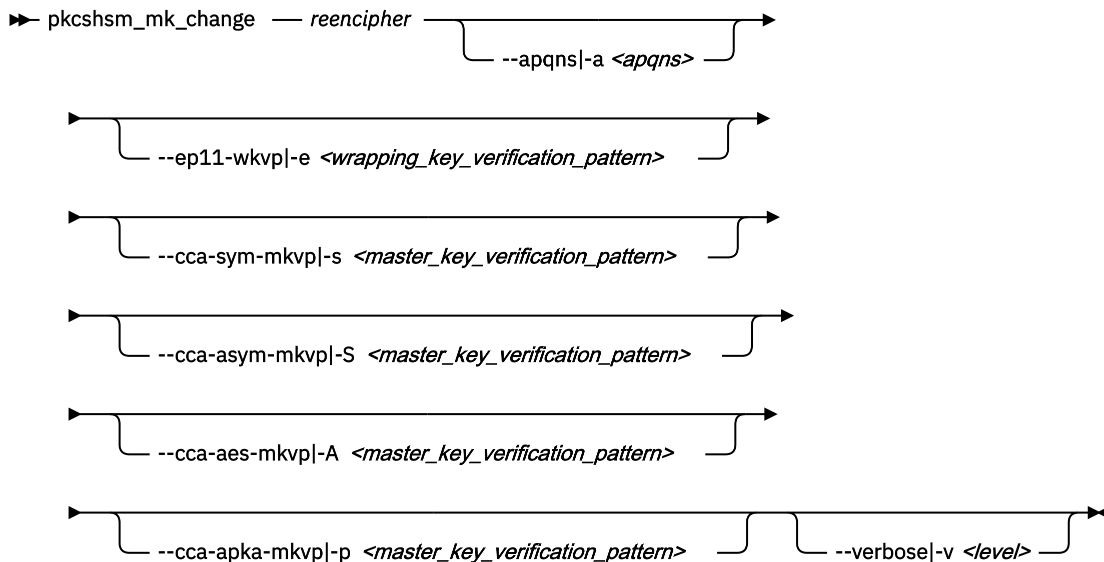
```
cat /sys/bus/ap/devices/<card>.<domain>/mkvps
```

A cryptographic coprocessor in EP11 coprocessor mode has only one master key, called the EP11 wrapping key (WK).

The **`pkcshsm_mk_change reencipher`** subcommand queries all available slots and determines if the token in the slot is affected by the master key change, based on the list of APQNs and master key types. For each affected slot, it prompts for the USER PIN.

On successful completion, the ID of the master key change operation is displayed. This ID must be specified when finalizing or canceling the operation with the **finalize** or **cancel** subcommand.

pkcshsm_mk_change syntax - Initiate a master key change



The options for all subcommands are explained in "Options for the pkcshsm_mk_change utility" on page 76.

Example: Initialization of an EP11 wrapping key change:

The following example shows how to initiate a master key change for all EP11 tokens that are allowed to access the specified APQNs. This master key change will then start concurrently to all running applications that exploit the applicable EP11 tokens. A prerequisite of this action is that a new EP11 wrapping key (or master key) has been loaded into the NEW register using the TKE. You need to know the USER PINs of all affected slots.

```
# pkcshsm_mk_change reencipher --apqns 06.0033,0e.0033,1c.0033 --ep11-wkvp
ef92899ebe57291ec8e8716d850ee2f7

WARNING: The following slots have no token present:
  Slot 0
ATTENTION: If you start a concurrent master key change operation while not
all expected tokens are present, the key objects of those tokens may be lost,
if the token would be affected by the master key change.
Continue [y/N]? y
The following tokens are affected by this master key change:
  Slot 4: Label: ep11
Enter the USER PIN for slot 4:
Re-enciphering, please wait...
Completed.

Master key change operation 'uNA5QP' created.

Once the new master keys have been set/activated:
 - If you specified EXPECTED_MKVPS in your token configuration file(s),
   you must now replace the old MKVPs with the new MKVPs.
 - Run 'pkcshsm_mk_change finalize --id uNA5QP' when the new master
   keys have been set/activated.
```

The ID of this master key change operation is **uNA5QP** and can or must be re-used in option `--id|-i` in subsequent subcommands.

Example: Initialization of a CCA master key change for various master key types:

```
# pkcshsm_mk_change reencipher --apqns 08.0033,09.0033,1b.0033 --cca-sym-mkvp 130E2053F18E5F4C
                                 --cca-asym-mkvp F8AE2ACF8BFC57F0
                                 --cca-aes-mkvp 7D10D17BC8A409C4
                                 --cca-apka-mkvp 82A5E2CD5030D5EC

WARNING: The following slots have no token present:
...
The following tokens are affected by this master key change:
  Slot 2: Label: cca
...
Master key change operation 'FYZ4WC' created.
...
```

The ID of this master key change operation is **FYZ4WC** and can or must be re-used in option `-i|--id` in subsequent subcommands.

For further information you can read the **pkcshsm_mk_change** man page.

List master key change operations

Use the **pkcshsm_mk_change list** subcommand to list currently active master key change operations. If you do not specify an operation ID in option `--id|-i` (described in "Initiate a master key change plus re-encryption for openCryptoki" on page 73), all currently active master key change operations are displayed, otherwise only the specified one is displayed.

pkcshsm_mk_change syntax - List master key change operations

```
►► pkcshsm_mk_change ── list ──┬──────────────────────────┬──┬──────────────────────┬──►◄
                               └─ --id|-i <operation_id> ─┘  └─ --verbose|-v <level> ─┘
```

Example of listing a master key change operation:

To display information about the current master key change process as initialized with the command shown in "Initiate a master key change plus re-encryption for openCryptoki" on page 73, enter the following command:

```
# pkcshsm_mk_change list -i uNA5QP

Operation:       uNA5QP
    State:       Key objects have been re-enciphered,
                 new master key(s) can now be set/activated
    APQNs:
        06.0033
        0E.0033
        1C.0033
    New master key verification patterns:
        Type:    EP11
        MKVP:    EF92899EBE57291EC8E8716D850EE2F7
    Affected slots:
        Slot: 4 Label: ep11
            Current master key verification patterns:
                Type:    EP11
                MKVP:    8B991263E3A8F4E4BE0D5EC8F0A4DF9E
```

The optional **-i** parameter specifies the operation ID of the current master key change process. For a detailed description, see "Initiate a master key change plus re-encryption for openCryptoki" on page 73.

Now you need to activate the new EP11 master key using the TKE. Then you can finalize the master key change process as described in "Finalize a master key change for openCryptoki" on page 75.

Finalize a master key change for openCryptoki

Use the **pkcshsm_mk_change finalize** subcommand to finalize a master key change operation when the new master key has been activated on the APQNs. You must specify the ID of the master key change operation to be finalized.

pkcshsm_mk_change syntax - Finalize a master key change

```
►► pkcshsm_mk_change — finalize — --id|-i <operation_id> ─┬──────────────────────────┬─►◄
                                                         └─ --verbose|-v <level> ─┘
```

Example of finalizing a master key change:

To finalize the current master key change process as initialized with the command shown in "Initiate a master key change plus re-encryption for openCryptoki" on page 73, enter the following command.

```
# pkcshsm_mk_change finalize --id uNA5QP
Enter the USER PIN for slot 4:
Finalizing, please wait...

Master key change operation 'uNA5QP' successfully finalized.
```

Cancel a master key change for openCryptoki

Use the **pkcshsm_mk_change cancel** subcommand to cancel a master key change operation. You must specify the ID of the master key change operation to be canceled. You can only cancel a master key change operation as long as for none of the APQNs the new master key has been set (activated).

pkcshsm_mk_change syntax - Cancel a master key change

```
►► pkcshsm_mk_change — cancel — --id|-i <operation_id> ─┬──────────────────────────┬─►◄
                                                       └─ --verbose|-v <level> ─┘
```

Example of canceling a master key change:

To cancel the current master key change, as initialized with the command shown in "Initiate a master key change plus re-encryption for openCryptoki" on page 73, enter the following command:

```
# pkcshsm_mk_change cancel --id uNA5QP
Enter the USER PIN for slot x:
     /* ( ==> prompted for every effected token .....) */
Canceling, please wait...

Master key change operation 'uNA5QP' successfully canceled.
```

Options for the `pkcshsm_mk_change` utility

The following list describes the meanings of all options available for all subcommands:

--apqns|-a
: This option specifies a comma separated list of APQNs for which a master key change is to be performed. Each APQN must be specified in the form `card.domain`, denoting the cryptographic adapter and the domain on the adapter card, for example, `0a.0024`, where both numbers are specified with hexadecimal format, as displayed by the **lszcrypt** command.

--ep11-wkvp|-e
: In parameter **<wrapping_key_verification_pattern>**, this option specifies the EP11 wrapping key verification pattern (WKVP) of the new EP11 wrapping key (master key) to be set as a 16 bytes hexadecimal string.

You can use the TKE or the **ep11info** tool to query the current wrapping key verification pattern (WKVP) of an EP11 APQN:

```
ep11info -m <adapter> -d <domain> -D
```

You can also find the wrapping key verification pattern for EP11 APQNs in `sysfs`:

```
cat /sys/bus/ap/devices/<card>.<domain>/mkvps
```

For more information about the **ep11info** tool, see *Obtaining information about an EP11 environment with the ep11info tool.*

--cca-sym-mkvp|-s
In parameter **<master_key_verification_pattern>**, this option specifies the CCA master key verification pattern (MKVP) of the new CCA SYM master key to be set as an eight bytes hex string. You can use the TKE or the **panel.exe** tool to query the master key verification patterns. Use the hex string under [RND].

```
panel.exe --mk-query --mktype=SYM --mkregister=NEW
```

--cca-asym-mkvp|-S
In parameter **<master_key_verification_pattern>**, this option specifies the CCA master key verification pattern (MKVP) of the new CCA ASYM master key to be set as an eight bytes hex string. You can use the TKE or the **panel.exe** tool to query the master key verification patterns. Use the hex string under [RND].

```
panel.exe --mk-query --mktype=ASYM --mkregister=NEW
```

--cca-aes-mkvp|-A
In parameter **<master_key_verification_pattern>**, this option specifies the CCA master key verification pattern (MKVP) of the new CCA AES master key to be set as an eight bytes hex string. You can use the TKE or the **panel.exe** tool to query the master key verification patterns. Use the hex string under [VER].

```
panel.exe --mk-query --mktype=AES --mkregister=NEW
```

You can also find the AES master key verification patterns in `sysfs` as previously described.

--cca-apka-mkvp|-p
In parameter **<master_key_verification_pattern>**, this option specifies the CCA master key verification pattern (MKVP) of the new CCA APKA master key to be set as an eight bytes hex string. You can use the TKE or the **panel.exe** tool to query the master key verification patterns. Use the hex string under [VER].

```
panel.exe --mk-query --mktype=APKA --mkregister=NEW
```

You can also find the APKA master key verification patterns in `sysfs` as previously described.

--id|-i
This option specifies the ID of the master key change operation in parameter **<operation_id>**, for example, in the **pkcshsm_mk_change finalize** subcommand. You cannot specify this option for the **reencipher** subcommand, however this ID is produced and displayed as a result of a successful completion of the **reencipher** subcommand. The use of this option is only required for the **finalize** and **cancel** subcommands, and is optional for the **list** subcommand.

--verbose|-v
Specifies the verbose level (optional): none (default), error, warn, info, devel, or debug.

--help|-h

Displays help text and exits. For example you can request help for the complete tool or for the subcommands separately:

```
pkcshsm_mk_change -h
pkcshsm_mk_change initialize -h
pkcshsm_mk_change finalize -h
pkcshsm_mk_change cancel -h
pkcshsm_mk_change list -h
```

Part 4. Token specifications

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

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

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

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

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

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

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

The output depends on the supported Crypto Express coprocessors together with the openCryptoki version. The beginning of the output list may look as shown in Figure 13 on page 79. 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 13. 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)
…
```

The following information about openCryptoki token types is provided:

- Chapter 13, “Common token information,” on page 81
- Chapter 14, “CCA token,” on page 91
- Chapter 15, “ICA token,” on page 101
- Chapter 16, “EP11 token,” on page 107

© Copyright IBM Corp. 2021, 2023

- Chapter 17, “Soft token,” on page 125
- Chapter 18, “Directory content for CCA, ICA, EP11, and Soft tokens,” on page 131
- Chapter 19, “ICSF token,” on page 135

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

Chapter 13. Common token information

Read information that applies to all openCryptoki token types.

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

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

Adding tokens to openCryptoki

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

Note: For CCA tokens and EP11 tokens, you can configure each token instance differently using a token-specific configuration file (CCA token configuration file or EP11 token configuration file, specified with the `confname` option, see "Defining a CCA token configuration file " on page 91 or "Defining an EP11 token configuration file" on page 107).

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

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

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

```
/var/lib/opencryptoki/ep11token01/
/var/lib/opencryptoki/ep11token02/
/var/lib/opencryptoki/ccatoken01/
/var/lib/opencryptoki/ccatoken02/
```

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

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

The default openCryptoki configuration file `opencryptoki.conf` (see Figure 4 on page 22) provides a slot entry for a CCA token, preconfigured to slot #2 and a slot entry for an EP11 token, preconfigured to slot #4. Each slot entry must set the `stdll` attribute to the appropriate slot token dynamic link library (STDLLs), for example, `libpkcs11_ep11.so` for an EP11 token.

You can check the entries in the default `opencryptoki.conf` file to find out whether you can use it as is.

For each configured EP11 token, you must create a specific token configuration file. For a CCA token, such a configuration file is optional. A default configuration file is shipped with openCryptoki for both token types. The default EP11 token configuration file is only valid if all APQNs assigned to the token are configured with the same master key. For the CCA token, the CCA adapter selected as default adapter must be configured with the desired master keys.

An EP11 token configuration file, for example, defines the target adapters and target adapter domains to which the EP11 token sends its cryptographic requests.

© Copyright IBM Corp. 2021, 2023

In turn, each slot entry in the global openCryptoki configuration file for CCA and EP11 tokens must specify the name of the used token-specific configuration file (if applicable for a CCA token). For this purpose, for both token types, use the `confname` attribute with the unique name of the respective CCA token configuration file or EP11 token configuration file as value.

The example from Figure 14 on page 82 configures two EP11 tokens in slots 4 and 5 in the openCryptoki configuration file. It defines the names of the specific token configuration files to be `ep11tok01.conf` and `ep11tok02.conf`. Per default, these files are searched in the directory where openCryptoki searches its global configuration file. Figure 19 on page 113 shows an example of an EP11 token configuration file and Figure 16 on page 93 shows an example of a CCA 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
     tokname = ep11token02
  }
```

Figure 14. Multiple EP11 token instances

How to recognize tokens

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

The screen from Figure 15 on page 83 presents excerpts of a **`pkcsconf -t`** command output. The slot number is associated with the shown token number. The ICA token is plugged into slot 1. Therefore, you see information about the ICA token with the label `IBM ICA PKCS #11` in section **`Token #1 Info`**. Accordingly, in Figure 15 on page 83, you see a CCA token in slot 2, a Soft token in slot 3, and an EP11 token in slot 4.

```
$ pkcsconf -t

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

        Sessions: 0/[effectively infinite]
        R/W Sessions: [information unavailable]/[effectively infinite]
        PIN Length: 4-8
        Public Memory: [information unavailable]/[information unavailable]
        Private Memory: [information unavailable]/[information unavailable]
        Hardware Version: 0.0
        Firmware Version: 0.0
        Time: 2021081811215500
Token #2 Info:
        Label: ccatok
        Manufacturer: IBM
        Model: CCA
        Serial Number:
        Flags: 0x880045 (RNG|LOGIN_REQUIRED|CLOCK_ON_TOKEN|USER_PIN_TO_BE_CHANGED|

SO_PIN_TO_BE_CHANGED)

        Sessions: 0/[effectively infinite]
        R/W Sessions: [information unavailable]/[effectively infinite]
        PIN Length: 4-8
        ...
Token #3 Info:
        Label: softtok
        ...
Token #4 Info:
        Label: ep11tok
        Manufacturer: IBM
        Model: EP11
        Serial Number: 93AABC7X69330380
        Flags: 0x80004D (RNG|LOGIN_REQUIRED|USER_PIN_INITIALIZED|CLOCK_ON_TOKEN|

SO_PIN_TO_BE_CHANGED)

        ...
        Hardware Version: 7.28
        Firmware Version: 3.1
        Time: 2021081811215500
```

Figure 15. Token information

The most important information is as follows:

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

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

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

ECC header file

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

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

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

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

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

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

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

Read the following topics:

- “ECC curves supported by the CCA token” on page 96
- “ECC curves supported by the ICA token” on page 105
- “ECC curves supported by the EP11 token” on page 117.

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

OID file for post-quantum algorithms

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

```
#ifndef _PQC_OIDS_H_
#define _PQC_OIDS_H_

/*
 * OIDs and their DER encoding for the post-quantum crypto algorithms
 * supported by OpenCryptoki:
 */

/* Dilithium Round 2 high-security (SHAKE-256): 1.3.6.1.4.1.2.267.1.6.5 */
#define OCK_DILITHIUM_R2_65        { 0x06, 0x0B, 0x2B, 0x06, 0x01, 0x04, \
                                     0x01, 0x02, 0x82, 0x0B, 0x01, 0x06, 0x05 }

/* Dilithium Round 2 for outbound authentication: 1.3.6.1.4.1.2.267.1.8.7 */
#define OCK_DILITHIUM_R2_87        { 0x06, 0x0B, 0x2B, 0x06, 0x01, 0x04, \
                                     0x01, 0x02, 0x82, 0x0B, 0x01, 0x08, 0x07 }

/* Dilithium Round 3 weak (SHAKE-256): 1.3.6.1.4.1.2.267.7.4.4 */
#define OCK_DILITHIUM_R3_44        { 0x06, 0x0B, 0x2B, 0x06, 0x01, 0x04, \
                                     0x01, 0x02, 0x82, 0x0B, 0x07, 0x04, 0x04 }

…
…

/* Kyber Round 2 768 (SHAKE-128): 1.3.6.1.4.1.2.267.5.3.3 */
#define OCK_KYBER_R2_768           { 0x06, 0x0B, 0x2B, 0x06, 0x01, 0x04, \
                                     0x01, 0x02, 0x82, 0x0B, 0x05, 0x03, 0x03 }

/* Kyber Round 2 1024 (SHAKE-128): 1.3.6.1.4.1.2.267.5.4.4 */
#define OCK_KYBER_R2_1024          { 0x06, 0x0B, 0x2B, 0x06, 0x01, 0x04, \
                                     0x01, 0x02, 0x82, 0x0B, 0x05, 0x04, 0x04 }

#endif                          // _PQC_OIDS_H_
```

Also read the following topic:

- Chapter 20, "IBM-specific mechanisms," on page 139

How and why to exploit protected keys

For the CCA token and the EP11 token you can optionally improve the performance of your cryptographic processing of certain algorithms by enabling the use of protected keys for these tokens. For the use of the AES XTS cipher mode, exploitation of protected keys is mandatory for these tokens.

The keys processed by the openCryptoki tokens fall into one of the following categories:

clear key
: A clear key is an effective key in plain text. That is, the bit pattern of a clear key is the one that is used by the cipher algorithm. Thus, the clear key is the same as the effective key. The term clear key is commonly used in the context of IBM Z cryptographic hardware. Therefore, whoever knows the clear key can perform cryptographic operations (like encrypt or decrypt) using that clear key.

secure key
: A secure key is a clear or effective key encrypted by an HSM master key. Secure keys can be used in cryptographic functions performed by the HSM. Thus, each cryptographic function on a secure key requires I/O operations to the HSM. A secure key is only usable in the HSM in which it was generated, or in an HSM that is configured with the same master key as the HSM that was used to generate the secure key.

protected key
: Protected keys are keys encrypted by the machine-generated IBM firmware wrapping key of an LPAR, a virtual server, or a z/VM® or KVM guest. A protected key is created by IBM Z firmware and can only be used by the instance of the operating system that created that key. Also, protected keys are volatile, because they are only valid as long as the LPAR, the virtual server, or the z/VM or KVM guest that generated the key is running. The operating system has no access to the effective key within a protected key, and protected keys are useless on any other system. Protected keys are used for high performance symmetric AES cryptographic functions processed by the CPACF. The CPACF also provides protected key support for DES/TDES and ECDSA and EdDSA signing functions. As the CPACF is implemented as a feature of an IBM Z CPU, it performs cryptographic operations at CPU speed.

IBM Z cryptographic hardware can use clear keys, secure keys or protected keys. The difference between these key types is not the value of the effective key which is the plain text key that is used to encrypt the data. However, the difference is the strength of protection that is applied to the effective key. And, of course, resulting from the different strength of protection, there is also a difference in the performance. There is no difference in the cryptographic algorithms used by the hardware, and the resulting cipher text is the same for all types of key if the underlying effective key is the same.

When working within the openCryptoki environment, the type of key that you use is determined by the token type. That is, in openCryptoki there are clear key tokens and secure key tokens:

- ICA tokens and Soft tokens are clear key tokens.
- CCA tokens and EP11 tokens are secure key tokens. Secure keys can only be processed on IBM Z cryptographic coprocessors configured as HSM with a valid master key. However, openCryptoki on IBM Z provides an option for these token types to transparently derive a high performance protected key from a strongly shielded secure key, which provides advantages for certain use cases.

Protected keys versus secure keys

The differences between protected and secure keys are the following:

- In contrast to cryptographic operations performed with protected keys on the CPACF, operations with secure keys are performed on IBM Z cryptographic coprocessors.
- Protected keys offer performance advantages, but are not quite as secure than secure keys. On the other hand, secure keys work on tamper-proof hardware security modules like the IBM Z cryptographic adapters, and therefore require input/output operations.

- With CCA tokens and EP11 tokens you can explicitly set a configuration option PKEY_MODE to allow the derivation of a protected key from a secure key if adequate and required. This derivation then works transparently during the affected cryptographic operations.

  Thus, you can optionally enable CCA and EP11 tokens to exploit the performance optimization provided by protected keys in contrast to secure keys. If you enable option PKEY_MODE, then all applicable cryptographic operations are performed with derived or existing protected keys on the CPACF. If no protected key is allowed or cannot currently be derived, then the affected cryptographic operations are performed with a secure key on the cryptographic coprocessor.

Protected keys in read-only and read-write sessions

In PKCS #11, an application either opens a read-only or a read-write session.

- In both session types, applications can create, read, modify, and delete **session objects**. Session objects have the attribute CKA_TOKEN=FALSE.
- In read-write sessions, applications can additionally create, read, modify, and delete **token objects**. Token objects have the attribute CKA_TOKEN=TRUE.
- Read-only sessions can also read **token objects**.

Token objects are persistent and are saved on disk in the token repository. **Session objects are non-persistent** and are destroyed when the session ends.

If you create a key object with CCA or EP11 tokens where the PKEY_MODE option is enabled, then the protected key is not derived from the secure key immediately, but only at the first use of the key object (valid for both session or token objects).

- With a key object that is a persistent token object (CKA_TOKEN=TRUE), a derived protected key is bound to the key object as attribute CKA_IBM_OPAQUE_PKEY and thus is saved in the token repository as part of the key object (see also "Miscellaneous attributes" on page 154). This is, however, only possible in a read-write session, because only in a read-write session, an application can create, modify, and delete persistent token objects.
- With a non-persistent key object (CKA_TOKEN=FALSE), a derived protected key is also bound to the key object as attribute CKA_IBM_OPAQUE_PKEY, but is not saved in the token repository and only exists within memory as long as the creating session exists.

With all this background knowledge, the implemented processing and its consequences for the use of protected keys may be understandable:

If a key object is used for a cryptographic operation, the CCA or EP11 tokens check whether a valid protected key is available (valid means: the current firmware master key can still unwrap the protected key). If this is the case, the protected key is used for improved performance.

If no (or no valid) protected key is available, then the CCA or EP11 tokens try to derive a new protected key. Derivation is of course only performed if the prerequisites of an enabled PKEY_MODE are fulfilled.

- If the derivation succeeds, the new protected key is used.
- If a protected key cannot be derived, then the current cryptographic operation is performed with the secure key on the cryptographic coprocessor, which works slower than protected key processing on the CPACF.

  Note: However, if you use the AES XTS cipher mode, then the current cryptographic operation fails, because for CCA and EP11 tokens there is no AES XTS processing possible on the cryptographic coprocessors.

Conditions for a protected key derivation

A protected key can be successfully derived from a key object if several conditions are fulfilled. In addition, a derivation can only be performed in session types where the PKCS #11 standard allows object modification. For a detailed information you can read about PKCS #11 session types.

- The hardware supports protected keys: CPACF is required.

- PKEY_MODE is enabled.
- The key object has boolean attribute CKA_IBM_PROTKEY_EXTRACTABLE=TRUE.
- For persistent token key objects (CKA_TOKEN=TRUE):
  - If they have attribute CKA_PRIVATE=TRUE, then derivation is only allowed in R/W User sessions.
  - If they have attribute CKA_PRIVATE=FALSE, then derivation is allowed in R/W Public sessions, R/W User sessions, or R/W SO sessions.
- For non-persistent session key objects (CKA_TOKEN=FALSE):
  - If they have attribute CKA_PRIVATE=TRUE, then derivation is only allowed in R/W or R/O User sessions.
  - If they have attribute CKA_PRIVATE=FALSE, then derivation is always allowed.

An example where protected key derivation does not work: A persistent key token object that has been created in a R/W session may be used for cryptographic operations for the first time in another session. If this session is for example, a read-only session, it is not possible to derive a protected key, because the key object would need to be updated. For AES XTS keys, such a scenario would lead to an error.

Prerequisites

Note: The documented prerequisites (including a functioning configuration of the `PKEY_MODE` option in the respective configuration files) are required for protected key processing. For the AES XTS cipher mode, a functioning `PKEY_MODE` setting is mandatory. For secure key types other than AES XTS, secure key processing on cryptographic coprocessors is used if PKY_MODE is not enabled or other prerequisites are not fulfilled. All detailed information is provided in "Defining a CCA token configuration file " on page 91 and "Defining an EP11 token configuration file" on page 107.

There are the following hardware and software prerequisites for enabling protected key support in CCA tokens and EP11 tokens:

Hardware prerequisites common for CCA and EP11 tokens

Using the protected key support is possible on IBM z15™ or later processors with a CEX7C coprocessor in CCA mode for CCA tokens or on CEX7P EP11 cryptographic coprocessors for EP11 tokens.

Software prerequisites - CCA token:

- the CCA host library 7.0 or later,
- the Linux kernel 5.10 or later.

Software prerequisites - EP11 tokens:

- the EP11 host library version 3.0 or later.

Configurations in the `ccatok.conf` and `ep11tok.conf` configuration files

- CCA and EP11 tokens must have protected keys enabled using the `PKEY_MODE` option set as ENABLED (for CCA tokens), ENABLE4NONEXTR (for EP11 tokens), or DEFAULT (valid for both).
- If `PKEY_MODE` is set to DISABLED, no protected keys can be used and AES XTS mechanisms are not supported.

How to enable AES XTS support for CCA and EP11 tokens

AES XTS support is provided for CCA tokens, EP11 tokens, ICA tokens and Soft tokens. With AES XTS support, you can exploit the AES XTS block cipher mode together with a key that is composed from two concatenated AES keys.

For ICA tokens and Soft tokens, AES XTS support is provided with clear keys. These tokens can exploit AES XTS support without any preparation.

Exploiting the AES XTS support with CCA tokens and EP11 tokens is only possible with the use of protected keys on the CP Assist for Cryptographic Functions (CPACF) feature of IBM Z systems.

Protected keys are derived from secure keys and therefore requires a functioning configuration of the PKEY_MODE option in the applicable configuration files. Therefore, some background knowledge about the transformation of secure keys into protect keys and the use of protected keys in openCryptoki sessions is required. Read "How and why to exploit protected keys" on page 85 for more information.

Generating and importing AES XTS keys

All openCryptoki token types documented in this publication support the generation of AES XTS keys either with PKCS #11 function `C_GenerateKey()` or with the **p11sak** utility (see Chapter 9, "Managing token keys - p11sak utility," on page 43).

Furthermore, with the `C_CreateObject()` function, you can import a key for all token types into openCryptoki.

An AES XTS key import for CCA tokens offers two options:

Importing clear keys for AES XTS (works for CCA and EP11 tokens): Double length AES XTS clear keys, consisting of a pair of two keys are internally converted into a pair of secure keys using a CCA function. These two secure keys are then concatenated to each other and stored in the CKA_IBM_OPAQUE attribute.

Importing CCA secure keys for AES XTS: In case of a secure key blob import for AES XTS, the CKA_IBM_OPAQUE attribute contains two secure key blobs concatenated to each other. openCryptoki checks if the properties of each key of the key pair match the CCA AESDATA secure key type in the CKA_IBM_OPAQUE attribute. It sets CKA_SENSITIVE to TRUE and clears the CKA_VALUE attribute.

See also "Usage notes for CCA library functions" on page 97).

Restrictions for CCA tokens and EP11 tokens

- With CCA and EP11 tokens, AES XTS keys cannot be used for wrapping and unwrapping other keys using the CKM_AES_XTS mechanism, in contrast to the specifications in the *AES XTS* section of PKCS #11 Specification Version 3.1.
- The CCA and EP11 tokens also do not support wrapping and unwrapping of AES XTS keys.
- With CCA and EP11 tokens, AES XTS keys cannot be used for deriving other keys, nor can AES XTS keys be derived from other keys.
- The EP11 token does not support function `C_IBM_ReencryptSingle()` with AES XTS keys.

PKCS #11 Baseline Provider support

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

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

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

Dual-function cryptographic functions support

Since version 3.19, openCryptoki supports several functions to perform two cryptographic operations simultaneously within a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and from a token.

The following dual-function operations are provided:

Table 4. Dual-function cryptographic functions

Function	Purpose
C_DigestEncryptUpdate()	continues multiple-part digest and encryption operations, processing another data part. Digest and encryption operations must both be active (they must have been initialized with `C_DigestInit()` and `C_EncryptInit()`, respectively). This function may be called any number of times in succession, and may be interspersed with `C_DigestUpdate()` and `C_EncryptUpdate()` calls.
C_DecryptDigestUpdate()	continues a multiple-part combined decryption and digest operation, processing another data part. Decryption and digesting operations must both be active (they must have been initialized with `C_DecryptInit()` and `C_DigestInit()`, respectively). This function may be called any number of times in succession, and may be interspersed with `C_DecryptUpdate()` and `C_DigestUpdate()` calls.
C_SignEncryptUpdate()	continues a multiple-part combined signature and encryption operation, processing another data part. Signature and encryption operations must both be active (they must have been initialized with `C_SignInit()` and `C_EncryptInit()`, respectively). This function may be called any number of times in succession, and may be interspersed with `C_SignUpdate()` and `C_EncryptUpdate()` calls
C_DecryptVerifyUpdate()	continues a multiple-part combined decryption and verification operation, processing another data part. Decryption and signature operations must both be active (they must have been initialized with `C_DecryptInit()` and `C_VerifyInit()`, respectively). This function may be called any number of times in succession, and may be interspersed with `C_DecryptUpdate()` and `C_VerifyUpdate()` calls.

You can find detailed information about dual-function cryptographic functions, illustrated with code samples, in PKCS #11 Specification Version 3.1.

Supported features of PKCS #11 3.0 and 3.1

The contained sections describe enhancements of openCryptoki that support new versions of PKCS #11, starting with version 3.0.

AES XTS support

The AES XTS mechanisms and the AES XTS key type are new for PKCS #11 version 3.0. The AES XTS support of PKCS #11 3.0 is described in detail in "How to enable AES XTS support for CCA and EP11 tokens" on page 87.

`C_SessionCancel()`

All openCryptoki token types support the `C_SessionCancel()` function which you can use to terminate active operations on a session. Operations are usually initialized by the `C_XyzInit()` function for a specific operation, and are terminated by the successful completion of either the single block function `C_Xyz()`, or by the multi block final function C_XyzFinal() (see also "Functions and mechanisms" on page 9). In case a regular termination is not possible or wanted, you can terminate the operation also by using the `C_SessionCancel()` function with the corresponding operation flag. All operation flags are documented in

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

After a successful `C_SessionCancel()` invocation for an operation, the operation is no longer active, and a new operation can be initialized on that session, if wanted. The `C_SessionCancel()` function is new with PKCS #11 version 3.0, and thus is contained in function list `CK_FUNCTION_LIST_3_0` which must be explicitly obtained via `C_GetInterface()` with interface name PKCS 11 and version 3.0. The function list returned by the regular `C_GetFunctionList()` function does not contain the `C_SessionCancel()` function.

CKA_DERIVE_TEMPLATE

The CKA_DERIVE_TEMPLATE attribute of a base key contains a template that is applied to the derived key in addition to the user supplied derive template. Applications can use the CKA_DERIVE_TEMPLATE attribute on base keys to control the attributes of the keys that are to be derived from that base keys.

A private or secret key that is used as base key to derive other keys from can contain attribute CKA_DERIVE_TEMPLATE, which contains an array of attributes that are to be applied to the derived key. The number of array elements is determined by the `ulValueLen` component of the attribute divided by the size of CK_ATTRIBUTE.

Attribute CKA_DERIVE_TEMPLATE is allowed for private and secret keys only. It defaults to an empty array. If the attributes specified by the CKA_DERIVE_TEMPLATE conflict with those attributes that are explicitly specified in the derive template with the `C_DeriveKey()` function, then CKR_TEMPLATE_INCONSISTENT is returned.

The CKA_DERIVE_TEMPLATE attribute as such is not stored within the base keys's secure key blobs of EP11 or CCA, but only in the openCryptoki key object.

For more information, read PKCS #11 Specification Version 3.1.

Chapter 14. CCA token

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

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

Prerequisites for exploiting a CCA token:

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 14).

Additionally, a running CCA token requires certain types of master keys to be set on the applicable cryptographic adapters:

- Up to openCryptoki 3.15: AES, SYM, and ASYM master keys are required.
- Starting with openCryptoki 3.15: AES, SYM, and APKA master keys are required.

To query the master key verification pattern of available keys for any master-key register in the current domain, use the `panel.exe` utility and issue a command similar to the following:

```
panel.exe --mk-query --mktype=SYM --mkregister=CURRENT
```

where **--mktype** can be one of [ASYM|SYM|AES|APKA] and **--mkregister** must be CURRENT to query the information from the currently active master key.

Or you can use the `ivp.e` utility. This is an easy-to-use utility which you can invoke without any arguments. It is used to verify an installation, and among others, provides information about current master keys for all available CEX*C features on the system.

For AES and APKA master keys, you can also find the master key verification patterns in `sysfs` using the following command:

```
$ cat /sys/bus/ap/devices/<card>.<domain>/mkvps
```

For information on how to install the CCA library and on how to use the `panel.exe` and `ivp.e` utilities, read *Secure Key Solution with the Common Cryptographic Architecture Application Programmer's Guide*.

Note: The CCA token directory must not be located in a directory that is either an NFS or a CIFS file system, but must be located in a file system that supports the **flock()** function which manages file locks.

Defining a CCA token configuration file

You can define an optional configuration file for a CCA token.

Other than for the EP11 token, there is no default CCA token configuration file name, and also no environment variable to specify the directory containing the configuration file. Instead, you can specify the CCA token configuration file name (and optionally its absolute path) in the global openCryptoki configuration file called `opencryptoki.conf` using the keyword `confname`. If no path is specified, then a specified CCA token configuration file must exist within the openCryptoki directory `/etc/opencryptoki/`.

© Copyright IBM Corp. 2021, 2023

Example for a CCA token configuration file specification in `opencryptoki.conf`:

```
...
slot 2
{
   stdll = libpkcs11_cca.so
   confname = /etc/opencryptoki/ccatok1.conf
}
...
```

You can specify the following options in the CCA token configuration file:

EXPECTED_MKVPS

This option defines the expected master key verification patterns for the three CCA master key types: SYM, AES, and APKA. Master key type ASYM is not used by the CCA token. At token initialization, the CCA token checks whether the expected MKVP value is consitent on all assigned APQNs. The set of APQNs is determined by the CCA host library, based on the specifications of the CCA environment variables CSU_DEFAULT_ADAPTER and CSU_DEFAULT_DOMAIN.

Syntax:

```
EXPECTED_MKVPS
{
   SYM = "<8-bytes-hex-string>"
   AES = "<8-bytes-hex-string>"
   APKA = "<8-bytes-hex-string>"
}
```

For function calls to the key generation functions `C_GenerateKey()`, `C_GenerateKeyPair()`, `C_UnwrapKey()`, `C_DeriveKey()`, or `C_CreateObject()`, the CCA token also performs a master key verification check to verify if the master key is generated with the correct and expected master key verification pattern (MKVP). That means, that for an existing `EXPECTED_MKVPS` specification, the MKVP of the generated key must match the specified value, or, if no `EXPECTED_MKVPS` value is specified, it must be the MKVP encountered at token initialization time.

In case of a mismatch, token initialization or key generation fails and a syslog message is issued and a flag is set in the token to reject all subsequent cryptographic operations with CKR_DEVICE_ERROR.

PKEY_MODE

Prerequisite for exploiting the `PKEY_MODE` option is the Linux kernel 5.10 or later. You can specify this option to define that for CCA secure keys of type AES, AES XTC, or EC, an additional pertaining protected key shall be created, and this protected key shall be used by CPACF whenever possible because of performance reasons (see "How and why to exploit protected keys" on page 85).

- Supported AES key lengths are 128, 192, and 256 bits.
- EC key support is limited to the curves supported both by CPACF with MSA9 and by the CCA token. Therefore only the following three curves are supported:
  - prime256 with OID 1.2.840.10045.3.1.7
  - secp384 with OID 1.3.132.0.34
  - secp521 with OID 1.3.132.0.35
- The following mechanisms are supported for protected keys:
  - CKM_AES_ECB
  - CKM_AES_CBC
  - CKM_AES_CBC_PAD
  - CKM_AES_XTS
  - CKM_ECDSA
  - CKM_ECDSA_SHA1
  - CKM_ECDSA_SHA224

- CKM_ECDSA_SHA256
  - CKM_ECDSA_SHA384
  - CKM_ECDSA_SHA512
- The creation of protected keys requires that the native CCA key object (blob) has been created with the CCA keyword XPRTCPAC which is equivalent with the boolean attribute CKA_IBM_PROTKEY_EXTRACTABLE = TRUE. This keyword has been introduced with CCA 7.0 and CEX7C. Therefore, using the protected key support is possible on IBM z15 or later processors with a CEX7C coprocessor in CCA mode.
- In CCA a key can be both CKA_EXTRACTABLE and CKA_IBM_ PROTKEY_EXTRACTABLE. Therefore, all new keys may get attribute CKA_IBM_PROTKEY_EXTRACTABLE = TRUE if the application did not explicitly specify CKA_IBM_PROTKEY_EXTRACTABLE = FALSE. If you also work with EP11 tokens, you might notice that this is in contrast to EP11.
- CCA protected keys are added to key objects via the IBM-specific key attribute CKA_IBM_OPAQUE_PKEY at first use of the key.
- Protected keys are created only from symmetric and private keys, not from public keys.
- For information how to use protected keys for AES XTS support, read “How to enable AES XTS support for CCA and EP11 tokens” on page 87.

Set the PKEY_MODE option to one of the following modes:

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

DEFAULT
This option specifies to apply the default and works like follows:

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

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

ENABLED
Enable protected key support for all keys. If the application did not specify CKA_IBM_ PROTKEY_EXTRACTABLE = FALSE in its template, new keys of any type get CKA_IBM_PROTKEY_EXTRACTABLE = TRUE and a protected key is automatically created at first use of the key.

```
version cca-0
#
# CCA token configuration
#
EXPECTED_MKVPS
{
   SYM = "130E2053F18E5F4C"
   AES = "7D10D17BC8A409C4"
   APKA = "82A5E2CD5030D5EC"
}

PKEY_MODE = ENABLED
```

Figure 16. Sample of a CCA token configuration file

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 5 on page 94 lists all mechanisms that are supported by the CCA token in the specified slot.

Table 5. 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_SHA256_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_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	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA512_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_DES_CBC	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES_CBC_PAD	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES3_CBC	24-24 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES3_CBC_PAD	24-24 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_AES_KEY_GEN	16-32 bytes	GENERATE	before 3.16
CKM_AES_ECB	16-32 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_AES_CBC	16-32 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_AES_CBC_PAD	16-32 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_AES_XTS[1)]	32 - 64 bytes	ENCRYPT, DECRYPT	3.22

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_AES_XTS_KEY_GEN[1)]	32 - 64 bytes	GENERATE	3.22
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
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

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
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
Note: 1) only applicable with protected key (see "How and why to exploit protected keys" on page 85).			

Table 5. PKCS #11 mechanisms supported by the CCA token (continued)

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

ECC curves supported by the CCA token

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

Table 6 on page 96 shows the curves that the CCA token supports for elliptic curve cryptography.

Table 6. Curves supported by the CCA token for elliptic curve cryptography (ECC)

Curve	Purpose
brainpoolP160r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP192r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP224r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP256r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP320r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP384r1	• 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
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

Table 6. Curves supported by the CCA token for elliptic curve cryptography (ECC) (continued)

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

Usage notes for CCA library functions

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

- Plain CCA key objects, that is, CCA secure key objects generated by the CCA token internally via the CCA library `libcsulcca.so`, which can also be imagined as a blob (binary large object), can be extracted from sensitive openCryptoki key objects for the CCA token by accessing the value of the CKA_IBM_OPAQUE attribute.

  A CCA key blob created by a native CCA application via the CCA library `libcsulcca.so` can be imported into sensitive openCryptoki key objects for the CCA token by assigning the content of the CCA key blob to the CKA_IBM_OPAQUE attribute. This is valid for the `C_CreateObject()` function. This import is supported for key types (called key tokens in CCA):

  – CCA DES key token
  – CCA DES3 key token
  – CCA AESDATA key token. For AES XTS, two such AESDATA key tokens are concatenated.
  – CCA internal RSA private key token (RSA-AESM and RSA-AESC)
  – CCA RSA public key token (RSA-AESM and RSA-AESC)
  – CCA HMAC key token
  – CCA internal EC private key token
  – CCA EC public key token

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

  Clear keys can also be imported into sensitive openCryptoki key objects for the CCA token by assigning the clear key 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. The CCA token also supports the `C_CreateObject()` function for AES, DES, DES3, and generic secret keys, as well as plain HMAC and EC keys with different curves.
- The default CKA_SENSITIVE setting for generating a key is CK_FALSE although the openCryptoki CCA token handles only secure keys, which correspond to sensitive keys in PKCS #11.

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

Migrate to a new CCA master key - pkcscca utility

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

Before you begin

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

About this task

There may be situations when CCA master keys must be changed. With openCryptoki, you can choose between a concurrent master key change described in Chapter 12, “Managing a concurrent master key change - pkcshsm_mk_change utility,” on page 71 or an offline master key change where you need to stop all openCryptoki applications that access the CCA token. Use the tool described in this topic to perform this offline scenario.

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>
```

   Specify the following parameters:

   -s
   slot number for the CCA token

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

-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.

Migrate to a new RSA format - pkcscca utility

Up to openCryptoki version 3.14, RSA keys were created using the RSA-CRT key token format (private key section X'08'). RSA-CRT keys are encrypted with the CCA ASYM master key, and can not be used for certain mechanisms, for example, RSA-PSS or RSA-OAEP. Starting with openCryptoki version 3.15.0, RSA keys are created using the RSA-AESC key token format (private key section X'31').

To convert old RSA keys (RSA-CRT) to the new format (RSA-AESC) of a CCA token in a selected slot, issue a command similar to the following:

```
pkcscca -m oldrsakeys -s <slotid>
```

RSA public keys created with openCryptoki up to version 3.16.0 also may contain a full CCA secure key token, including the private key section (which is encrypted by the CCA master key). The `oldrsakeys` migration option extracts the public key sections only from RSA public key tokens containing a full RSA CCA secure key token.

Chapter 15. 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 14).

Note: The ICA token directory must not be located in a directory that is either an NFS or a CIFS file system, but must be located in a file system that supports the **flock()** function which manages file locks.

PKCS #11 mechanisms supported by the ICA token

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

The command output depends on the libica version, whether libica is running in FIPS mode, and on the processor generation. The output of the **pkcsconf -m -c <slot>** command corresponds to the list shown in Table 7 on page 101 which presents all mechanisms supported by the ICA token on an IBM z15 machine.

Table 7. Supported mechanism list for the ICA token

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

© Copyright IBM Corp. 2021, 2023

Table 7. Supported mechanism list for the ICA token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_SHA224_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA224_RSA_PKCS_PSS	512-4096 bits	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_224	n/a	HW, DIGEST	before 3.16
CKM_SHA512_224_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_224_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_256	n/a	HW, DIGEST	before 3.16
CKM_SHA512_256_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_256_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_AES_XTS	32 -64 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	3.20
CKM_AES_XTS_KEY_GEN	32 -64 bytes	HW, GENERATE	3.20
CKM_DES_KEY_GEN	8-8 bytes	HW, GENERATE	before 3.16
CKM_DES_ECB	8-8 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES_CBC	8-8 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES_CBC_PAD	8-8 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_KEY_GEN	24-24 bytes	HW, GENERATE	before 3.16
CKM_DES3_ECB	24-24 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_CBC	24-24 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_MAC	24-24 bytes	SIGN, VERIFY	before 3.16
CKM_DES3_MAC_GENERAL	24-24 bytes	SIGN, VERIFY	before 3.16
CKM_DES3_CBC_PAD	24-24 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_DES3_CMAC_GENERAL	16-24 bytes	SIGN, VERIFY	before 3.16
CKM_DES3_CMAC	16-24 bytes	SIGN, VERIFY	before 3.16
CKM_DES_OFB64	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES_CFB64	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_DES_CFB8	8-8 bytes	ENCRYPT, DECRYPT	before 3.16
CKM_MD5_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_MD5_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA_1	n/a	HW, DIGEST	before 3.16

Table 7. Supported mechanism list for the ICA token (continued)

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_SHA_1_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA_1_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA256	n/a	HW, DIGEST	before 3.16
CKM_SHA256_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA256_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA224	n/a	HW, DIGEST	before 3.16
CKM_SHA224_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA224_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA384	n/a	HW, DIGEST	before 3.16
CKM_SHA384_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA384_HMAC_GENERAL	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512	n/a	HW, DIGEST	before 3.16
CKM_SHA512_HMAC	n/a	HW, SIGN, VERIFY	before 3.16
CKM_SHA512_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	HW, GENERATE	before 3.16
CKM_ECDSA_KEY_PAIR_GEN	160-521 bits	HW, GENERATE_KEY_PAIR, EC_F_P, EC_OID	before 3.16
CKM_ECDSA	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA1	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA224	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA256	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA384	160-521 bits	HW, SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA512	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDH1_DERIVE	160-521 bits	HW, DERIVE, EC_F_P, EC_F_P, EC_OID	before 3.16
CKM_AES_KEY_GEN	16-32 bytes	HW, GENERATE	before 3.16
CKM_AES_ECB	16-32 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CBC	16-32 bytes	HW, ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_MAC	16-32 bytes	SIGN, VERIFY	before 3.16

Table 7. Supported mechanism list for the ICA token (continued)

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

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

For a description of mechanisms with a name pattern of `CKM_IBM_...` refer to Chapter 20, "IBM-specific mechanisms," on page 139.

Usage notes for the ICA library functions

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

- As of openCryptoki version 3.6, the `C_SeedRandom()` function of the ICA token always returns CKR_RANDOM_SEED_NOT_SUPPORTED.
- If your system is running in FIPS mode, libica also runs in FIPS mode and only provides a FIPS-compliant subset of algorithms and key lengths. For example, all Brainpool curves listed in Table 8 on page 105 are not supported in FIPS mode.

ECC curves supported by the ICA token

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

Table 8 on page 105 shows the maximum number of curves that the ICA token can support if all prerequisites are fulfilled at their best conditions. The following dependencies exist:

- Which openCryptoki version (and thus which libica version) is used? Refer to the applicable libica documentation for information about supported curves.
- Which cryptographic coprocessors are available?
- Is the MSA9 component of IBM z15 or later available?
- Is libica or OpenSSL running in FIPS mode? When FIPS mode is active for libica, OpenSSL is also set into FIPS mode. However, note that the case where libica does not run in FIPS mode, but OpenSSL does, may cause errors when software fallbacks are used. If, for example, an elliptic curve is supported by hardware in libica, but not by OpenSSL, because OpenSSL runs in FIPS mode, this software fallback fails.

Table 8. 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
brainpoolP384r1 **(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

Curve	Purpose
secp521r1 **(1), (2)**	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE

Table 8. Curves supported by the ICA token for elliptic curve cryptography (ECC) (continued)

Notes:

(1) supported via Crypto Express 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 16. 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 14, “CCA token,” on page 91.

As a prerequisite for an operational EP11 token, the EP11 host library (also called EP11 host library in other documentations) must be installed (see Figure 3 on page 14).

For EP11 tokens, you can introduce one or multiple tokens into the openCryptoki framework (see “Adding tokens to openCryptoki” on page 81) and configure them differently. For information on how to install the EP11 host library, refer to Exploiting Enterprise PKCS #11 using openCryptoki. You can download the EP11 host library from:

```
IBM PCIe Cryptographic Coprocessors
```

Note: The EP11 token directory must not be located in a directory that is either an NFS or a CIFS file system, but must be located in a file system that supports the **flock()** function which manages file locks.

Defining an EP11 token 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 81, these names are defined as `ep11tok01.conf` and `ep11tok02.conf`. If the environment variable `OCK_EP11_TOKEN_DIR` is set, then the EP11 token looks for the configuration file or files in the directory specified with this variable. If `OCK_EP11_TOKEN_DIR` is not set, then the EP11 token configuration files are searched in the global openCryptoki directory, for example: `/etc/opencryptoki/ep11tok.conf`.

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

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

then your EP11 token configuration file appears here:

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

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

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

The following is a list of available options for an EP11 token configuration file. A sample of such a file is shown in Figure 19 on page 113.

APQN_ALLOWLIST

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

You start this list of adapter/domain pairs starting with a line containing the keyword `APQN_ALLOWLIST`. Next follows the list which can specify up to 512 adapter/domain pairs, denoted

© Copyright IBM Corp. 2021, 2023

by decimal numbers in the range 0 - 255. Each pair designates an adapter (first number) and a domain (second number) accessible to the EP11 token. Close the list using the keyword END.

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

Notes:

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

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

There are two ways to specify the cryptographic adapter:

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

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

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

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

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

  ```
  APQN_ANY
  ```

In the example from Figure 19 on page 113, adapter 0 with domains 0 and 1, and adapter 2 with domain 84 are specified as target for requests from the EP11 token. In Figure 17 on page 109, these adapter/domain pairs are shown in hexadecimal notation as APQNs (00,0000), (00,0001), and (02,0054).

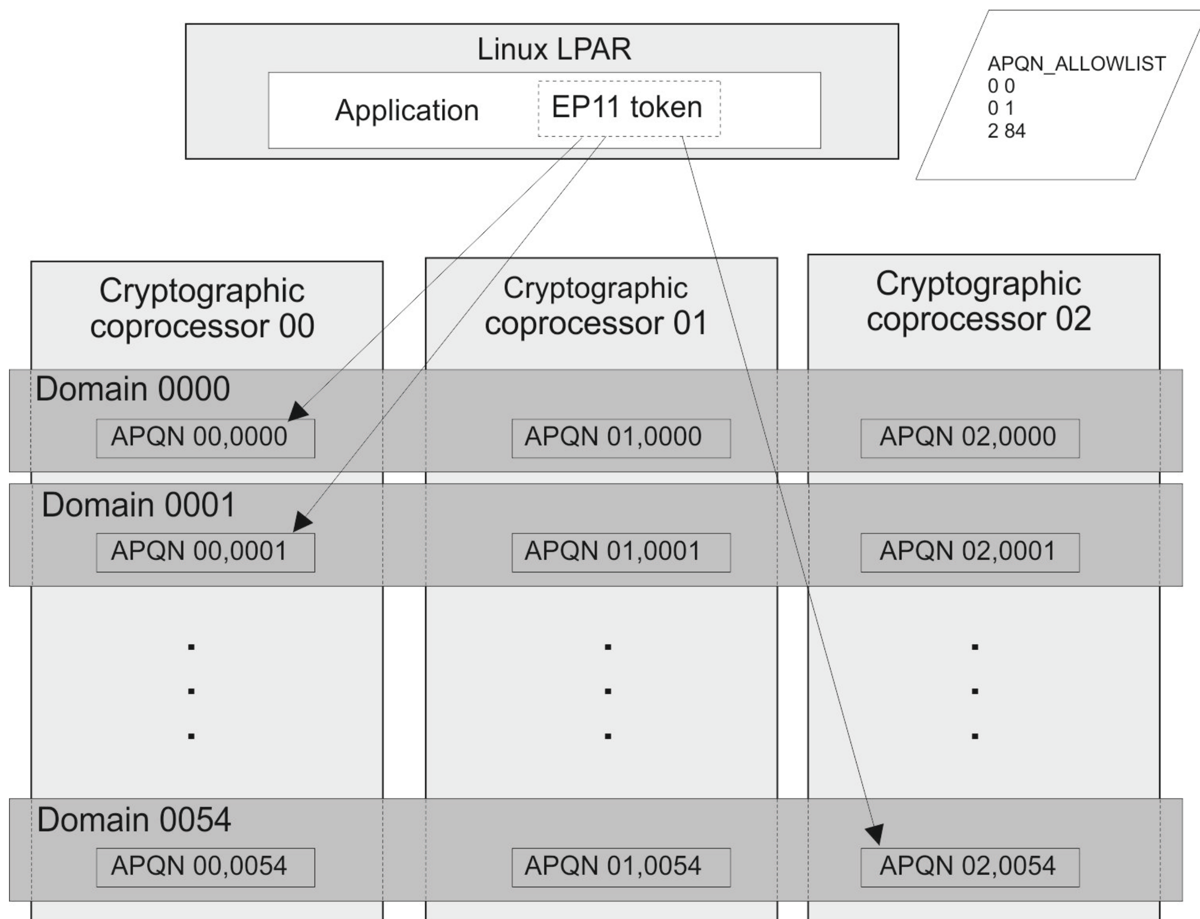


Figure 17. Cryptographic configuration for an LPAR

CPFILTER

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

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

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

# sign with HMAC or CMAC
XCP_CPB_SIGN_SYMM: CKM_SHA256_HMAC, CKM_SHA256_HMAC_GENERAL, CKM_SHA224_HMAC,
...

# verify with HMAC or CMAC
XCP_CPB_SIGVERIFY_SYMM: CKM_SHA256_HMAC, CKM_SHA256_HMAC_GENERAL, CKM_SHA224_HMAC,
...

# sign with private keys
XCP_CPB_SIGN_ASYMM: CKM_RSA_PKCS, CKM_RSA_PKCS_PSS, CKM_SHA1_RSA_X9_31, CKM_SHA1_RSA_PKCS,
...

# encrypt with symmetric keys
XCP_CPB_ENCRYPT_SYMM: CKM_AES_ECB, CKM_AES_CBC, CKM_AES_CBC_PAD, CKM_DES3_ECB,
...

# decrypt with symmetric keys
XCP_CPB_DECRYPT_SYMM: CKM_AES_ECB, CKM_AES_CBC, CKM_AES_CBC_PAD, CKM_DES3_ECB, CKM_DES3_CBC,
...

# key export with symmetric keys
XCP_CPB_WRAP_SYMM: CKM_AES_CBC, CKM_AES_CBC_PAD, CKM_DES3_CBC, CKM_DES3_CBC_PAD
...

# key import with symmetric keys
XCP_CPB_UNWRAP_SYMM: CKM_AES_CBC, CKM_AES_CBC_PAD, CKM_DES3_CBC, CKM_DES3_CBC_PAD

# generate asymmetric keypairs
XCP_CPB_KEYGEN_ASYMM: CKM_RSA_PKCS_KEY_PAIR_GEN, CKM_RSA_X9_31_KEY_PAIR_GEN,

# generate or derive symmetric keys
XCP_CPB_KEYGEN_SYMM: CKM_AES_KEY_GEN, CKM_DES2_KEY_GEN, CKM_DES3_KEY_GEN,

# RSA private-key or key-encrypt use
XCP_CPB_ALG_RSA: CKM_RSA_PKCS, CKM_RSA_PKCS_KEY_PAIR_GEN, CKM_RSA_X9_31_KEY_PAIR_GEN,
...

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

# EC private-key use
XCP_CPB_ALG_EC: CKM_EC_KEY_PAIR_GEN, CKM_ECDH1_DERIVE, CKM_ECDSA, CKM_ECDSA_SHA224,
...

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

# allow key derivation (symmetric+EC/DH)
XCP_CPB_DERIVE: CKM_SHA1_KEY_DERIVATION, CKM_SHA256_KEY_DERIVATION,
...

# enable support of curve25519, c448 and related algorithms incl. EdDSA (ed25519 and ed448)
XCP_CPB_ALG_EC_25519: CKM_IBM_EC_X25519, CKM_IBM_ED25519_SHA512, CKM_IBM_EC_X448,
...

#enable support of Dilithium
XCP_CPB_ALG_PQC: CKM_IBM_DILITHIUM, CKM_IBM_KYBER
...
```

Figure 18. Excerpt of a sample ACP-filter configuration file

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

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

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

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

DEFAULT

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

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

FORCE_SENSITIVE

Specify this option to force that the default for CKA_SENSITIVE is CK_TRUE for secret keys. For more information, see "Usage notes for the EP11 host library functions" on page 123.

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 host library as long as there is no corresponding multi-part operation.

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

CKR_OBJECT_HANDLE_INVALID
CKR_ATTRIBUTE_VALUE_INVALID
CKR_KEY_HANDLE_INVALID
CKR_KEY_SIZE_RANGE
CKR_KEY_TYPE_INCONSISTENT
CKR_MECHANISM_INVALID
CKR_MECHANISM_PARAM_INVALID

PKEY_MODE

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

Secure keys for which you want a pertaining protected key being produced and used, must have the CKA_IBM_PROTKEY_EXTRACTABLE = TRUE attribute specified in their template when being created. Secure keys with this attribute set to FALSE are not eligible for protected key support. See also "How and why to exploit protected keys" on page 85.

Keys created before introducing the protected key option are not usable for protected key support, because they do not have the CKA_IBM_PROTKEY_EXTRACTABLE = TRUE attribute. Only keys created after activating the protected key option with the ENABLE4NONEXTR mode are eligible for getting a protected key, depending on their key type (only AES and EC) and attributes. See also "Miscellaneous attributes" on page 154.

Keys created before introducing the protected key option do not have a CKA_IBM_PROTKEY_EXTRACTABLE attribute and are considered to be not usable for protected key support.

Notes:

- Protected keys are created only from symmetric and private keys, not from public keys. They are created by a transparent IBM-specific mechanism. This mechanism in turn reads certain attributes of the processed key object to determine the generation of the protected key. The new protected key is bound to the original secure key object by the attribute CKA_IBM_OPAQUE_PKEY (see also "Miscellaneous attributes" on page 154.
- The PKEY_MODE option is only supported on IBM z15 processors or later, with EP11 cryptographic coprocessors starting with CEX7P. It requires the EP11 host library 3.0 or later.
- Currently, an EP11 secure key (AES or EC) cannot be both CKA_EXTRACTABLE = TRUE and CKA_IBM_PROTKEY_EXTRACTABLE = TRUE. Therefore, an application that wants to use the PKEY_MODE support, must explicitly specify CKA_EXTRACTABLE=FALSE for all keys, which do not require to be extractable for other reasons, for example, because (according to the PKCS#11 standard,) keys to be wrapped must be extractable. In addition, the EP11 host library 3.0 or later does not allow attribute-bound keys to be transformed into protected keys.

Set the PKEY_MODE option to one of the following modes:

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

DEFAULT
This option specifies to apply the default and works like follows:

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

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

ENABLE4NONEXTR
Enable protected key support for non-extractable keys. If the application did not specify CKA_IBM_PROTKEY_EXTRACTABLE = FALSE in its template for key generation, new keys of any type with CKA_EXTRACTABLE = FALSE (non-extractable keys) get CKA_IBM_PROTKEY_EXTRACTABLE = TRUE and a protected key is automatically created at first use of the key.

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

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

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

VHSM_MODE

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

EXPECTED_WKVP

Use this option to ensure that all APQNs assigned to an EP11 token - dependent on the APQN configuration (APQN_ANY or APQN_ALLOWLIST) - are configured with the same wrapping key. You specify the expected wrapping key verification pattern (WKVP) as hex string. With this option, the assigned APQNs are checked to all match the specified expected WKVP during token initialization, or if no `EXPECTED_WKVP` is specified, all WKVPs must match on all assigned APQNs anyway.

For functions calls to the key generation functions `C_GenerateKey()`, `C_GenerateKeyPair()`, `C_UnwrapKey()`, `C_DeriveKey()`, or `C_CreateObject()`, the EP11 token also performs a wrapping key verification pattern check to verify if the wrapping key is generated with the correct and expected wrapping key verification pattern (WKVP). That means, that for an existing `EXPECTED_WKVP` specification, the WKVP of the generated key must match the specified value, or, if no `EXPECTED_WKVP` value is specified, it must be the WKVP encountered at token initialization time.

In case of a mismatch, token initialization or key generation fails and a syslog message is issued and a flag is set in the token to reject all subsequent cryptographic operations with CKR_DEVICE_ERROR.

The EP11 verification pattern is 16 bytes in length, although sometimes 32 bytes are reported. Nevertheless, only the first 16 bytes are compared.

```
#
# EP11 token configuration
#
APQN_ALLOWLIST
  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
EXPECTED_WKVP "303344b12b8258840fa11852a4ecc6d5"
```

Figure 19. Sample of an EP11 token configuration file

PKCS #11 mechanisms supported by the EP11 token

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

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

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

On an Crypto Express EP11 coprocessor (CEX*P) which is configured to support all applicable PKCS #11 mechanisms from the current openCryptoki version, the EP11 token can exploit the mechanisms listed by the **pkcsconf -m -c <slot>** command output. This output corresponds to the list shown in Table 9 on page 114. 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 9. PKCS #11 mechanisms supported by the EP11 token

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
CKM_RSA_PKCS_OAEP	1024-4096 bits	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_KEY_PAIR_GEN	1024-4096 bits	GENERATE_KEY_PAIR	before 3.16
CKM_RSA_X9_31_KEY_PAIR_GEN	1024-4096 bits	GENERATE_KEY_PAIR	before 3.16
CKM_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_X9_31	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA256_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA256_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA224_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA224_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA384_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA512_RSA_PKCS	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA512_RSA_PKCS_PSS	1024-4096 bits	SIGN, VERIFY	before 3.16
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_AES_XTS[1)]	32 - 64 bytes	ENCRYPT, DECRYPT	3.20
CKM_AES_XTS_KEY_GEN[1)]	32 - 64 bytes	GENERATE	3.20
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

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
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 3.21
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 3.21
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 3.21
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 3.21
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 3.21
CKM_SHA512_HMAC	256-256 bytes	SIGN, VERIFY	before 3.16
CKM_SHA512_256	n/a	DIGEST	before 3.16
CKM_SHA512_256_HMAC	128-256 bytes	SIGN, VERIFY	before 3.16
CKM_SHA512_224	n/a	DIGEST	before 3.16
CKM_SHA512_224_HMAC	112-256 bytes	SIGN, VERIFY	before 3.16
CKM_ECDSA_KEY_PAIR_GEN	192-521 bits	GENERATE_KEY_PAIR, EC_F_P, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDSA	192-521 bits	SIGN, VERIFY, EC_F_P, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA1	192-521 bits	SIGN, VERIFY, EC_F_P, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDH1_DERIVE	192-521 bits	DERIVE, EC_F_P, EC_UNCOMPRESS	before 3.16
CKM_DSA_PARAMETER_GEN	1024-3072 bits	GENERATE	before 3.16
CKM_DSA_KEY_PAIR_GEN	1024-3072 bits	GENERATE_KEY_PAIR	before 3.16
CKM_DSA	1024-3072 bits	SIGN, VERIFY	before 3.16
CKM_DSA_SHA1	1024-3072 bits	SIGN, VERIFY	before 3.16

Table 9. PKCS #11 mechanisms supported by the EP11 token (continued)

Table 9. PKCS #11 mechanisms supported by the EP11 token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
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 3.21
CKM_IBM_DILITHIUM	256-256 bytes	SIGN, VERIFY, GENERATE_KEY_PAIR	before 3.16
CKM_IBM_KYBER	204-396 bytes	ENCRYPT, DECRYPT, GENERATE, DERIVE	3.21
CKM_RSA_X9_31	1024-4096 bits	SIGN, VERIFY	before 3.16
CKM_PBE_SHA1_DES3_EDE_CBC	24-24 bytes	GENERATE	before 3.16
CKM_IBM_SHA3_224	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_256	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_384	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_512	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_224_HMAC	112-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_256_HMAC	128-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_384_HMAC	192-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_512_HMAC	256-256 bytes	SIGN, VERIFY	before 3.16
CKM_ECDSA_SHA224	192-521 bits	SIGN, VERIFY, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA256	192-521 bits	SIGN, VERIFY, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA384	192-521 bits	SIGN, VERIFY, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_ECDSA_SHA512	192-521 bits	SIGN, VERIFY, EC_F_P, EC_OID, EC_UNCOMPRESS	before 3.16
CKM_IBM_EC_C25519	256-256 bytes	DERIVE, EC_F_P, EC_UNCOMPRESS	before 3.16
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

Table 9. PKCS #11 mechanisms supported by the EP11 token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_IBM_ED448_SHA3	448-448 bytes	SIGN, VERIFY, EC_F_P, EC_UNCOMPRESS	before 3.16
CKM_IBM_CMAC	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_AES_CMAC	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_DES3_CMAC	16-24 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_ATTRIBUTEBOUND_WRAP	0-4096 bits	WRAP, UNWRAP	3.16
Note: 1) only applicable with protected key (see “How and why to exploit protected keys” on page 85).			

For a description of mechanisms with a name pattern of `CKM_IBM_...` refer to Chapter 20, “IBM-specific mechanisms,” on page 139.

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 113.

Table 10. 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

Table 10. Curves supported by the EP11 token for elliptic curve cryptography (ECC) (continued)

Curve	Purpose
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
brainpoolP384r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP384t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP512r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
brainpoolP512t1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
prime192v1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
prime256v1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
secp224r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
secp256k1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
secp384r1	• for Sign/Verify operations with CKM_ECDSA and CKM_ECDSA_SHAnnn • for ECDH with CKM_ECDH1_DERIVE
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	

Curve	Purpose
ed448	Sign/Verify with CKM_IBM_ED448_SHA3
ed25519	Sign/Verify with CKM_IBM_ED25519_SHA512

Table 10. Curves supported by the EP11 token for elliptic curve cryptography (ECC) (continued)

The EP11 host 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
```

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.

With openCryptoki, you can choose between a concurrent master key change described in Chapter 12, “Managing a concurrent master key change - pkcshsm_mk_change utility,” on page 71 or an offline master key change where you need to stop all openCryptoki applications that access the EP11 token. Use the tool described in this topic to perform this offline scenario.

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 120 unless step "4" on page 120 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 120 of the master key migration process, the new master key must be submitted and committed via the TKE interface. That means the *new EP11 master key* must be in the state `Full Committed`. The current MK is in the state `Valid`. Now both (current and new) *EP11 master keys* are available and accessible. The utility can now decrypt all relevant key objects within the token and re-encrypt all these key objects with the new master key.

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

Invocation:

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

The following parameters are mandatory:

-slot
- slot number for the EP11 token

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

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

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

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

Usage: You are prompted for your user PIN.

Examples:

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

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

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

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

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

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

Managing EP11 sessions - pkcsep11_session utility

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

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

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

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

pkcsep11_session syntax

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

```
# pkcsep11_session -h

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

pkcsep11_session sub-command usage examples

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

  ```
  pkcsep11_session show
  ```

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

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

Enter the USER PIN:
List of EP11 sessions:

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

2 EP11-Sessions displayed
```

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

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

```
pkcsep11_session show -pid 1234
```

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

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

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

```
pkcsep11_session logout
```

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

```
pkcsep11_session logout -pid 1234
```

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

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

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

```
pkcsep11_session logout -force
```

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

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

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

  See also "Defining an EP11 token configuration file" on page 107 or read topic *Controlling access to cryptographic objects* in *Exploiting Enterprise PKCS #11 using openCryptoki*.

```
# pkcsep11_session vhsmpin -slot 4

Using slot #4...

Enter the USER PIN:
```

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

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

```
pkcsep11_session status
```

See the following sample output:

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

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

Usage notes for the EP11 host library functions

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

- The EP11 host library implements the *secure key concept* as explained in the introduction of Chapter 14, "CCA token," on page 91.

  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 20 on page 123. 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_ALLOWLIST
  5 2
  6 2
END
```

Figure 20. 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 11 on page 123 and Table 12 on page 124 show the EP11 token-specific attributes and their default values for private and secure keys.

Table 11. Private key (CKO_PRIVATE_KEY) default attributes of the EP11 token

Private key attributes	value
CKA_SENSITIVE	CK_TRUE
CKA_EXTRACTABLE	CK_TRUE

Table 12. 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 host library.
- Pure EP11 key objects can be extracted from sensitive openCryptoki key objects for the CCA token by accessing the value of the CKA_IBM_OPAQUE attribute value.

Restriction to extended evaluations

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

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

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

Read the information about filter mechanisms in

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

for information on how to manage the access to PKCS #11 mechanisms using ACPs.
The available mechanisms and their attributes are then reflected by the openCryptoki functions `C_GetMechanismList` and `C_GetMechanismInfo`. However, there is one restriction on RSA mechanisms that cannot be reflected in the result of `C_GetMechanismInfo`: The `CKA_PUBLIC_EXPONENT` must have a value of at least 17.

Chapter 17. 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 14).

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

Note: The Soft token directory must not be located in a directory that is either an NFS or a CIFS file system, but must be located in a file system that supports the **`flock()`** function which manages file locks.

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 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 13 on page 125 displays all mechanisms that are supported by the Soft token. Each mechanism provides its supported key size and some further properties such as hardware support and mechanism information flags. These flags provide information about the PKCS #11 functions that may use the mechanism. In some cases, the flags also provide further attributes that describe the supported variants of the mechanism. Typical functions are for example, *encrypt*, *decrypt*, *wrap key*, *unwrap key*, *sign*, or *verify*.

The **`pkcsconf -m -c <slot>`** command output corresponds to the list shown in Table 13 on page 125.

Table 13. 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

Table 13. PKCS #11 mechanisms supported by the Soft token (continued)

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	512-4096 bits	ENCRYPT, DECRYPT, SIGN, SIGN RECOVER, VERIFY, VERIFY RECOVER, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA224_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA256_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA384_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA512_RSA_PKCS_PSS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA224_RSA_PKCS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA256_RSA_PKCS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA384_RSA_PKCS	512-4096 bits	SIGN, VERIFY	3.16
CKM_SHA512_RSA_PKCS	512-4096 bits	SIGN, VERIFY	3.16
CKM_RSA_X_509	512-4096 bits	ENCRYPT, DECRYPT, SIGN, SIGN_RECOVER, VERIFY, VERIFY_RECOVER, WRAP, UNWRAP	before 3.16
CKM_RSA_PKCS_OAEP	512-4096 bits	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_MD5_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_SHA1_RSA_PKCS	512-4096 bits	SIGN, VERIFY	before 3.16
CKM_DH_PKCS_DERIVE	512-8192 bits	DERIVE	before 3.16
CKM_DH_PKCS_KEY_PAIR_GEN	512-8192 bits	GENERATE KEY PAIR	before 3.16
CKM_AES_XTS	32 -64 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.20
CKM_AES_XTS_KEY_GEN	32 -64 bytes	GENERATE	3.20
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

Table 13. PKCS #11 mechanisms supported by the Soft token (continued)

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
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
CKM_SHA384_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512	n/a	DIGEST	before 3.16
CKM_SHA512_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_224	n/a	DIGEST	before 3.16
CKM_SHA512_224_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_224_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_256	n/a	DIGEST	before 3.16
CKM_SHA512_256_HMAC	n/a	SIGN, VERIFY	before 3.16
CKM_SHA512_256_HMAC_GENERAL	n/a	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_224	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_256	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_384	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_512	n/a	DIGEST	before 3.16
CKM_IBM_SHA3_224_HMAC	112-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_256_HMAC	128-256 bytes	SIGN, VERIFY	before 3.16
CKM_IBM_SHA3_384_HMAC	n/a	SIGN, VERIFY	before 3.16

Mechanism	Key sizes in bits or bytes	Properties	Support with OC version
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_DES3_MAC	16-24 bytes	HW, SIGN, VERIFY	before 3.16
CKM_DES3_MAC_GENERAL	16-24 bytes	HW, SIGN, VERIFY	before 3.16
CKM_AES_MAC	16-24 bytes	HW, SIGN, VERIFY	before 3.16
CKM_AES_MAC_GENERAL	16-24 bytes	HW, SIGN, VERIFY	before 3.16
CKM_SSL3_MD5_MAC	384-384 bits	SIGN, VERIFY	before 3.16
CKM_SSL3_SHA1_MAC	384-384 bits	SIGN, VERIFY	before 3.16
CKM_AES_CTR	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_OFB	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_CFB8	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_CFB128	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_GCM	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_DES_OFB64	24-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_DES_CFB8	24-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_DES_CFB64	24-24 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	3.17
CKM_AES_KEY_GEN	16-32 bytes	GENERATE	before 3.16
CKM_AES_ECB	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CBC	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CBC_PAD	16-32 bytes	ENCRYPT, DECRYPT, WRAP, UNWRAP	before 3.16
CKM_AES_CMAC	16-32 bytes	SIGN, VERIFY	before 3.16

Table 13. PKCS #11 mechanisms supported by the Soft token (continued)

Table 13. PKCS #11 mechanisms supported by the Soft token (continued)			
Mechanism	**Key sizes in bits or bytes**	**Properties**	**Support with OC version**
CKM_AES_CMAC_GENERAL	16-32 bytes	SIGN, VERIFY	before 3.16
CKM_GENERIC_SECRET_KEY_GEN	80-2048 bits	GENERATE	before 3.16
CKM_ECDSA_KEY_PAIR_GEN	160-521 bits	GENERATE_KEY_PAIR, EC_F_P, EC_OID	before 3.16
CKM_ECDSA	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA1	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA224	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA256	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA384	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDSA_SHA512	160-521 bits	SIGN, VERIFY, EC_F_P, EC_OID	before 3.16
CKM_ECDH1_DERIVE	160-521 bits	DERIVE, EC_F_P, EC_OID	before 3.16

For a description of mechanisms with a name pattern of `CKM_IBM_...` refer to Chapter 20, “IBM-specific mechanisms,” on page 139.

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

Chapter 18. Directory content for CCA, ICA, EP11, and Soft tokens

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

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

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

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

The directory containing the token directories, in our example `/var/lib/opencryptoki`, must not be an NFS nor a CIFS file system, but must be a file system that supports the **`flock()`** function which manages file locks.

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 21. 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 10, “Migrating to FIPS compliance - pkcstok_migrate utility,” on page 65.

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

© Copyright IBM Corp. 2021, 2023

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

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

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

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

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

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

The old `NVTOK.DAT` format (still valid)

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

The new `NVTOK.DAT` format

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

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

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

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

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

Changes from old to new data store format

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

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

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

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

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

The new format is:

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

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

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

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

```
struct OBJECT_PRIV {
    u32 total_len;
    //- auth -------        <--+
```

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

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

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

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

Table 14.

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 19. 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.

© Copyright IBM Corp. 2021, 2023

Part 5. IBM-specific mechanisms and features for openCryptoki

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

The following topics are presented:

- Chapter 20, “IBM-specific mechanisms,” on page 139
- Chapter 21, “Re-encrypting data with a mechanism,” on page 157

© Copyright IBM Corp. 2021, 2023

Chapter 20. IBM-specific mechanisms

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

Table 15. IBM-specific mechanisms

Mechanism	Reference
CKM_IBM_DILITHIUM	“CKM_IBM_DILITHIUM” on page 139
CKM_IBM_KYBER	“CKM_IBM_KYBER” on page 143
CKM_IBM_SHA3_224	“CKM_IBM_SHA3_nnn” on page 146
CKM_IBM_SHA3_256	“CKM_IBM_SHA3_nnn” on page 146
CKM_IBM_SHA3_384	“CKM_IBM_SHA3_nnn” on page 146
CKM_IBM_SHA3_512	“CKM_IBM_SHA3_nnn” on page 146
CKM_IBM_SHA3_224_HMAC	“CKM_IBM_SHA3_nnn_HMAC” on page 147
CKM_IBM_SHA3_256_HMAC	“CKM_IBM_SHA3_nnn_HMAC” on page 147
CKM_IBM_SHA3_384_HMAC	“CKM_IBM_SHA3_nnn_HMAC” on page 147
CKM_IBM_SHA3_512_HMAC	“CKM_IBM_SHA3_nnn_HMAC” on page 147
CKM_IBM_CMAC	“CKM_IBM_CMAC” on page 148
CKM_IBM_EC_C448 (synonym: CKM_IBM_EC_X448)	“Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA” on page 148
CKM_IBM_EC_C25519 (synonym: CKM_IBM_EC_X25519)	“Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA” on page 148
CKM_IBM_ED448_SHA3	“Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA” on page 148
CKM_IBM_ED25519_SHA512 (synonym: CKM_IBM_EDDSA_SHA512)	“Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA” on page 148
CKM_IBM_ATTRIBUTEBOUND_WRAP	“CKM_IBM_ATTRIBUTEBOUND_WRAP” on page 149
CKM_IBM_BTC_DERIVE	“CKM_IBM_BTC_DERIVE” on page 152
CKM_IBM_ECDSA_OTHER	“CKM_IBM_ECDSA_OTHER” on page 153

CKM_IBM_DILITHIUM

Availability:

The CKM_IBM_DILITHIUM mechanism is available with the EP11 token.

Description:

Dilithium is a post-quantum signature scheme. That is, it can generate signatures that resist attacks from quantum computers, but also from classical computers. *Dilithium* is one of the candidate algorithms submitted to the NIST post-quantum cryptography project.

© Copyright IBM Corp. 2021, 2023

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

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

- CKM_AES_CBC_PAD
- CKM_DES3_CBC_PAD

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

```
65    XCP_CPB_ALG_PQC           enable support for post quantum cryptographic algorithms
```

Prerequisites:

Using the CKM_IBM_DILITHIUM mechanism requires the following prerequisites:

- For the IBM Dilithium key form CK_IBM_DILITHIUM_KEYFORM_ROUND2_65, the EP11 host library 3.0 or later is required. For all other IBM Dilithium keys available with openCryptoki 3.20 or later, the EP11 host library 4.0 or later is required.
- A Crypto Express7S feature or later, configured as Crypto Express EP11 coprocessor (CEX7P) with firmware level 7.15 or later
- Not all firmware versions of EP11 cryptographic coprocessors support all Round 2 and Round 3 variants. Thus, a specific variant can only be used when all APQNs configured for the EP11 token support the desired variant. The EP11 token checks if all configured APQNs support the desired variant, and only then it allows to generate, import, or use an IBM Dilithium key with the desired variant.

Key type:

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

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

Key form:

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

- **Public key**

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

- **Private key**

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

The currently supported key forms are:

Table 16. Supported IBM Dilithium key forms		
OID	**Key form and value**	**Meaning**
1.3.6.1.4.1.2.267.1.6.5	CK_IBM_DILITHIUM_KEYFORM_ROUND2_65 = 1 (Alias: IBM_DILITHIUM_KEYFORM_ROUND2)	Round 2 dilithium-high
1.3.6.1.4.1.2.267.1.8.7	CK_IBM_DILITHIUM_KEYFORM_ROUND2_87 = 2	Round 2 dilithium-87
1.3.6.1.4.1.2.267.7.4.4	CK_IBM_DILITHIUM_KEYFORM_ROUND3_44 = 3	Round 3 dilithium-r3-weak
1.3.6.1.4.1.2.267.7.6.5	CK_IBM_DILITHIUM_KEYFORM_ROUND3_65 = 4	Round 3 dilithium-r3-rec
1.3.6.1.4.1.2.267.7.8.7	CK_IBM_DILITHIUM_KEYFORM_ROUND3_87 = 5	Round 3 dilithium-r3-vhigh

Attributes:

Table 17. Attributes for IBM Dilithium key		
Attribute	**Data type**	**Meaning**
CKA_IBM_DILITHIUM_KEYFORM (CKA_VENDOR_DEFINED + 0xd0001)	CK_ULONG	The Dilithium key form, currently Round 2 and Round 3. Can be one of the following: • CK_IBM_DILITHIUM_KEYFORM_ROUND2_65 = 1 • CK_IBM_DILITHIUM_KEYFORM_ROUND2_87 = 2 • CK_IBM_DILITHIUM_KEYFORM_ROUND3_44 = 3 • CK_IBM_DILITHIUM_KEYFORM_ROUND3_65 = 4 • CK_IBM_DILITHIUM_KEYFORM_ROUND3_87 = 5
CKA_IBM_DILITHIUM_MODE (CKA_VENDOR_DEFINED + 0x10)	CK_BYTE	Specifies the OID of the Dilithium variant used as byte array.
CKA_IBM_DILITHIUM_RHO (CKA_VENDOR_DEFINED + 0xd0002)	CK_BYTE	The private rho key component
CKA_IBM_DILITHIUM_SEED (CKA_VENDOR_DEFINED + 0xd0003)	CK_BYTE	The private seed key component
CKA_IBM_DILITHIUM_TR (CKA_VENDOR_DEFINED + 0xd0004)	CK_BYTE	The private tr key component
CKA_IBM_DILITHIUM_S1 (CKA_VENDOR_DEFINED + 0xd0005)	CK_BYTE	The private s1 key component
CKA_IBM_DILITHIUM_S2 (CKA_VENDOR_DEFINED + 0xd0006)	CK_BYTE	The private s2 key component
CKA_IBM_DILITHIUM_T0 (CKA_VENDOR_DEFINED + 0xd0007)	CK_BYTE	The private t0 key component

Table 17. Attributes for IBM Dilithium key (continued)		
Attribute	**Data type**	**Meaning**
CKA_IBM_DILITHIUM_T1 (CKA_VENDOR_DEFINED + 0xd0008)	CK_BYTE	The public t1 key component
CKA_VALUE	CK_BYTE	For a public Dilithium key, this attribute contains the BER encoded SPKI of the public key. For a private Dilithium key, it contains the PKCS#8 encoded private key value in case of clear key import, otherwise it is empty.

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

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

When generating Dilithium keys, an application can specify the CKA_IBM_DILITHIUM_KEYFORM or CKA_IBM_DILITHIUM_MODE attribute and thus select the variant to be used. For backward compatibility, if neither the CKA_IBM_DILITHIUM_KEYFORM nor CKA_IBM_DILITHIUM_MODE attribute is specified, it defaults to IBM_DILITHIUM_KEYFORM_ROUND2_65.

Key generation and unwrap operations supply the public Dilithium key attributes CKA_IBM_DILITHIUM_RHO and CKA_IBM_DILITHIUM_T1, as well as attributes CKA_IBM_DILITHIUM_KEYFORM and CKA_IBM_DILITHIUM_MODE to both, the public and the private key.

The SPKI is supplied in attribute CKA_VALUE to the public key.

When importing Dilithium keys from clear key values, you have two options:

- Specify the CKA_IBM_DILITHIUM_KEYFORM or CKA_IBM_DILITHIUM_MODE attribute together with the other public or private Dilithium key attributes. In this case, openCryptoki supplies the SPKI in the CKA_VALUE attribute to the public key during import.
- Specify the CKA_VALUE attribute to contain the SPKI or PKCS#8 encoding of the public or private key to import. In this case, CKA_IBM_DILITHIUM_KEYFORM and CKA_IBM_DILITHIUM_MODE must not be specified, since the OID is contained within the SPKI and PKCS#8 encoded key material.

Attributes CKA_IBM_DILITHIUM_KEYFORM and CKA_IBM_DILITHIUM_MODE are supplied to both, the public key and the private key during import.

The different Dilithium variants have different cryptographic strengths associated. The strength calculation used for openCryptoki policies and statistics take the used variant into consideration.

Functions for IBM Dilithium key creation and unwrapping:

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

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

CK_MECHANISM mech;
CK_SESSION_HANDLE session;
CK_OBJECT_HANDLE publ_key = CK_INVALID_HANDLE, priv_key = CK_INVALID_HANDLE;
...
mech.mechanism = CKM_IBM_DILITHIUM;
```

```
mech.ulParameterLen = 0;
mech.pParameter = NULL;
CK_RV rv;

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

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

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

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

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

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

/* Generate wrapping key */
...

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

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

Restrictions for using IBM Dilithium keys:

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

CKM_IBM_KYBER

Availability:

The CKM_IBM_KYBER mechanism is available with the EP11 token.

Description:

You can use this mechanism to generate quantum-safe Kyber key pairs as well as to perform encryption and decryption operations using Kyber keys. In addition, this mechanism offers methods to encapsulate or decapsulate (wrap or unwrap) keys (key encapsulation methods, KEM) which you can use to create shared secrets between to communicating parties.

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

```
65    XCP_CPB_ALG_PQC           enable support for post quantum cryptographic algorithms
```

Prerequisites:

Using the CKM_IBM_KYBER mechanism has the following prerequisites:

- The EP11 host library 4.0 or later is required.
- A Crypto Express8S feature, configured as Crypto Express EP11 coprocessor (CEX8P) with firmware level 8.9 or later is required.
- Not all firmware versions of EP11 cryptographic coprocessors support all Round 2 and Round 3 variants. Thus, a specific variant can only be used when all APQNs configured for the EP11 token support the desired variant. The EP11 token checks if all configured APQNs support the desired variant, and only then it allows to generate, import, or use an IBM Dilithium key with the desired variant.

Key type:

The IBM-specific key type for an IBM Kyber key object is:

```
#define CKK_IBM_PQC_KYBER       CKK_VENDOR_DEFINED + 0x00010024
#define CKK_IBM_KYBER           CKK_IBM_PQC_KYBER
```

Key form:

Kyber keys consist of a private key component sk and public key component pk, as defined by the ANS1 encodings:

```
KyberPublicKey ::= BIT STRING {
    SEQUENCE {
        pk BIT STRING -- public key
    }
}

KyberPrivateKey := SEQUENCE {
    version INTEGER,      -- v0; reserved 0
    sk      BIT STRING    -- secret key material
            [0] IMPLICIT OPTIONAL {
                pk||rs BIT STRING
                 -- public key (pk) concatenated with 2x32 bytes rs (rs = 64 times 0x30)
            }
    }
```

The currently supported key forms are:

Table 18. Supported IBM Kyber key forms

OID	Key form and value	Meaning
1.3.6.1.4.1.2.267.5.3.3	CK_IBM_KYBER_KEYFORM_ROUND2_768 = 1	Round 2 Kyber-r2rec
1.3.6.1.4.1.2.267.5.4.4	CK_IBM_KYBER_KEYFORM_ROUND2_1024 = 2	Round 2 Kyber-r2high

Attributes:

According to the key components, the following Kyber key component attributes are defined:

Table 19. Attributes for an IBM Kyber key

Attribute	Data type	Meaning
CKA_IBM_KYBER_KEYFORM (CKA_VENDOR_DEFINED + 0x000d0009)	CK_ULONG	The Kyber key form, currently Round 2. Can be one of the following: • (CK_IBM_KYBER_KEYFORM_ROUND2_768 =1 • CK_IBM_KYBER_KEYFORM_ROUND2_1024 = 2
CKA_IBM_KYBER_MODE (CKA_VENDOR_DEFINED + 0x0000000E)	CK_BYTE	Specifies the OID of the Kyber variant used as byte array.
CKA_IBM_KYBER_PK (CKA_VENDOR_DEFINED + 0x000d000A)	CK_BYTE	public Kyber key component
CKA_IBM_KYBER_SK (CKA_VENDOR_DEFINED + 0x000d000B)	CK_BYTE	private Kyber key component

When generating Kyber keys, an application can specify the CKA_IBM_KYBER_KEYFORM or CKA_IBM_KYBER_MODE attribute and thus select the variant to be used. If neither the CKA_IBM_KYBER_KEYFORM nor CKA_IBM_KYBER_MODE attribute is specified, it defaults to CK_IBM_KYBER_KEYFORM_ROUND2_HIGH.

Key generation and unwrap operations supply the public Kyber key attribute CKA_IBM_KYBER_PK, as well as attributes CKA_IBM_KYBER_KEYFORM and CKA_IBM_KYBER_MODE to both, the public and the private key.

The SPKI is supplied in attribute CKA_VALUE to the public key.

When importing Kyber keys from clear key values, an application must either specify the CKA_IBM_KYBER_KEYFORM attribute together with the other public or private Kyber key attribute(s), or the application must specify CKA_VALUE containing the SPKI or PKCS#8 encoding of the public or private key to import. In this case, CKA_IBM_KYBER_MODE must not be specified, since the OID is contained within the SPKI and PKCS#8 encoded key material.

Import supplies the SPKI in CKA_VALUE to the public key, if the public key is imported from the individual Kyber key component attributes. Attributes CKA_IBM_KYBER_KEYFORM and CKA_IBM_KYBER_MODE are supplied to both, the public key and the private key during import.

Kyber offers both encryption/decryption and encapsulation/decapsulation mechanisms.

Encapsulation or decapsulation mechanisms, also summarized by the term *key encapsulation mechanisms (KEM)* can be used for creating shared secrets between two parties. Respective functionality is implemented using the `C_DeriveKey()` interface for key encapsulation and decapsulation.

The base key argument to `C_DeriveKey()` is either a public or a private Kyber key, respectively. Encapsulation generates a new secret and encrypts it using Kyber encryption. Both, the cipher text used for exchanging the generated secret as well as a handle of a new key are output from the encapsulation operation.

Kyber KEM functions

Use the following functions and parameters for Kyber KEM processing.

```
#define CKD_IBM_HYBRID_NULL         CKD_VENDOR_DEFINED + 0x00000001UL
#define CKD_IBM_HYBRID_SHA1_KDF     CKD_VENDOR_DEFINED + 0x00000002UL
#define CKD_IBM_HYBRID_SHA224_KDF   CKD_VENDOR_DEFINED + 0x00000003UL
#define CKD_IBM_HYBRID_SHA256_KDF   CKD_VENDOR_DEFINED + 0x00000004UL
#define CKD_IBM_HYBRID_SHA384_KDF   CKD_VENDOR_DEFINED + 0x00000005UL
#define CKD_IBM_HYBRID_SHA512_KDF   CKD_VENDOR_DEFINED + 0x00000006UL
```

Structure for Kyber KEM parameters:

```
typedef struct CK_IBM_KYBER_PARAMS {
    CK_ULONG ulVersion;
    CK_IBM_KYBER_KEM_MODE mode;
    CK_IBM_KYBER_KDF_TYPE kdf;
    CK_BBOOL bPrepend;
    CK_BYTE *pCipher;
    CK_ULONG ulCipherLen;
    CK_BYTE *pSharedData;
    CK_ULONG ulSharedDataLen;
    CK_OBJECT_HANDLE hSecret;
} CK_IBM_KYBER_PARAMS;
```

The mechanism parameter CK_IBM_KYBER_PARAMS is used with the `C_DeriveKey()` function to provide Kyber-KEM parameters. For encapsulation with a public key, field mode is set to CK_IBM_KYBER_KEM_ENCAPSULATE. Fields `pCipher` and `ulCipherLen` are output in this case, and must specify a buffer large enough to hold the returned cipher text. A symmetric key with a key type according to attribute CKA_KEY_TYPE in the derive template is returned, together with the cipher text in field `pCipher`. Field `ulCipherLen` is updated to hold the size of the returned cipher text. If the supplied cipher text buffer is too small to hold the returned cipher text, CKR_BUFFER_TOO_SMALL is returned.

For decapsulation with a private key, field mode is set to CK_IBM_KYBER_KEM_DECAPSULATE. Fields `pCipher` and `ulCipherLen` are input in this case and must specify the cipher text that was generated by the encapsulation step. A symmetric key with a key type according to attribute CKA_KEY_TYPE in the derive template is returned.

Hybrid key generation or derivation can be performed to concatenate multiple secrets into a generic secret key.

Initialize a hybrid secret using either an ECDH key derivation, or a Kyber KEM using CKD_IBM_HYBRID_NULL as key derivation function (KDF) in field `kdf` of CK_IBM_KYBER_PARAMS or CK_ECDH1_DERIVE_PARAMS.

It can then be concatenated with other secrets using Kyber KEM by specifying the hybrid secret in field `hSecret` together with a CKD_IBM_HYBRID_SHAxxx key derivation function in field `kdf`, and set field `bPrepend` to CK_TRUE. Optionally you can specify additional shared data in fields `pSharedData` and `ulSharedDataLen` for the ANSI 9.63 key derivation.

For key encapsulation, the derive template for the resulting key must have CKA_EXTRACTABLE = CK_TRUE. All Kyber keys used in the derivation must have CKA_DERIVE = CK_TRUE. Hybrid secrets used in hybrid key derivation must have CKA_IBM_USE_AS_DATA = CK_TRUE.

The different Kyber variants have different cryptographic strength associated. The strength calculation used for openCryptoki policies and statistics must be adjusted to additionally take the used variant or keyform into consideration.

CKM_IBM_SHA3_nnn

Availability:

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

- CKM_IBM_SHA3_224
- CKM_IBM_SHA3_256
- CKM_IBM_SHA3_384
- CKM_IBM_SHA3_512

Description:

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

Mechanism	Flags and functions
CKM_IBM_SHA3_224	The CKF_DIGEST flag is set to true for this mechanism. Therefore, the function `C_DigestInit()` accepts the CKM_IBM_SHA3_224 mechanism.
CKM_IBM_SHA3_256	Same as for mechanism CKM_IBM_SHA3_224.
CKM_IBM_SHA3_384	Same as for mechanism CKM_IBM_SHA3_224.
CKM_IBM_SHA3_512	Same as for mechanism CKM_IBM_SHA3_224.

CKM_IBM_SHA3_nnn_HMAC

Availability:

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

- CKM_IBM_SHA3_224_HMAC
- CKM_IBM_SHA3_256_HMAC
- CKM_IBM_SHA3_384_HMAC
- CKM_IBM_SHA3_512_HMAC

Description:

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

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

Mechanism	Flags and functions
CKM_IBM_SHA3_224_HMAC	The CKF_SIGN and CKF_VERIFY flags are set to true for this mechanism. Therefore, the functions `C_SignInit()` and `C_VerifyInit()` accept the CKM_IBM_CMAC mechanism. Possible processing sequences are: • for single part sign or verify operations: – C_SignInit() followed by C_Sign() – C_VerifyInit() followed by C_Verify() • for multi-part sign or verify operations: – `C_SignInit()` followed by multiple `C_SignUpdate()`, then followed by `C_SignFinal()` – `C_VerifyInit()` followed by multiple `C_VerifyUpdate()`, then followed by `C_VerifyFinal()`
CKM_IBM_SHA3_256_HMAC	Same as for mechanism CKM_IBM_SHA3_224_HMAC.
CKM_IBM_SHA3_384_HMAC	Same as for mechanism CKM_IBM_SHA3_224_HMAC.
CKM_IBM_SHA3_512_HMAC	Same as for mechanism CKM_IBM_SHA3_224_HMAC.

CKM_IBM_CMAC

Availability:

The CKM_IBM_CMAC mechanism is available with the EP11 token.

Description:

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

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

Flags and functions:

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

Mechanisms for Edwards Curves 25519 and 448 for ECDH and EdDSA

Availability:

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

- CKM_IBM_EC_C448 (synonym: CKM_IBM_EC_X448)
- CKM_IBM_EC_C25519 (synonym: CKM_IBM_EC_X25519)
- CKM_IBM_ED448_SHA3
- CKM_IBM_ED25519_SHA512 (synonym: CKM_IBM_EDDSA_SHA512)

Description:

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

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

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

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

Mechanism	Flags and functions
CKM_IBM_EC_C448	• The CKF_DERIVE flag specifies that the mechanism can be used with `C_DeriveKey()`. • The CKF_EC_F_P flag specifies that the mechanism can be used with EC domain parameters over the finite field F_p. • The CKF_EC_UNCOMPRESS flag specifies that the mechanism can be used with elliptic curve point uncompressed. • This mechanism can also use the CK_ECDH1_DERIVE_PARAMS structure.
CKM_IBM_EC_C25519	Same as for mechanism CKM_IBM_EC_C448.
CKM_IBM_ED448_SHA3	The CKF_SIGN and CKF_VERIFY flags are set to true for this mechanism. Therefore, the functions `C_SignInit()` and `C_VerifyInit()` accept the CKM_IBM_ED448_SHA3 mechanism. • Possible processing sequences are the same as described with mechanism CKM_IBM_SHA3_224_HMAC. • The CKF_EC_F_P flag specifies that the mechanism can be used with EC domain parameters over the finite field F_p. • The CKF_EC_UNCOMPRESS flag specifies that the mechanism can be used with elliptic curve point uncompressed. • This mechanism also must use the CK_ECDH1_DERIVE_PARAMS structure.
CKM_IBM_ED25519_SHA512	Same as for mechanism CKM_IBM_ED448_SHA3.

CKM_IBM_ATTRIBUTEBOUND_WRAP

Availability:

The CKM_IBM_ATTRIBUTEBOUND_WRAP mechanisms is available with the EP11 token.

Description:

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

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

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

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

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

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

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

- CKA_EXTRACTABLE
- CKA_NEVER_EXTRACTABLE
- CKA_MODIFIABLE
- CKA_SIGN
- CKA_SIGN_RECOVER
- CKA_DECRYPT
- CKA_ENCRYPT
- CKA_DERIVE
- CKA_UNWRAP
- CKA_WRAP
- CKA_VERIFY
- CKA_VERIFY_RECOVER
- CKA_LOCAL
- CKA_WRAP_WITH_TRUSTED
- CKA_TRUSTED
- CKA_IBM_NEVER_MODIFIABLE (unsupported by openCryptoki)
- CKA_IBM_RESTRICTABLE (unsupported by openCryptoki)
- CKA_IBM_ATTRBOUND
- CKA_IBM_USE_AS_DATA
- CKA_KEY_TYPE
- CKA_VALUE_LEN

Following key types are supported as attribute-bound keys:

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

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

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

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

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

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

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

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

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

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

How to use the CKM_IBM_ATTRIBUTEBOUND_WRAP mechanism:

The mechanism requires a parameter of type CK_IBM_ATTRIBUTEBOUND_WRAP.

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

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

The CKM_IBM_ATTRIBUTEBOUND_WRAP mechanism has the following restrictions:

- All keys involved (target, wrapping/unwrapping, signature/verification) must be attribute-bound keys (CKA_IBM_ATTRBOUND = CK_TRUE). Otherwise:
  - For the target key on `C_WrapKey()`, CKR_KEY_NOT_WRAPPABLE is returned.
  - For the wrapping and unwrapping keys and the signature and verification keys, CKR_KEY_FUNCTION_NOT_PERMITTED is returned.

- On `C_WrapKey()`, the signing private key must be capable of signing (CKA_SIGN = CK_TRUE). Otherwise CKR_KEY_FUNCTION_NOT_PERMITTED is returned.
- On `C_UnwrapKey()`, the verification public key must be capable of verifying (CKA_VERIFY = CK_TRUE). Otherwise CKR_KEY_FUNCTION_NOT_PERMITTED is returned.

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

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

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

Flags and functions:

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

CKM_IBM_BTC_DERIVE

Availability:

The CKM_IBM_BTC_DERIVE mechanism is available with the EP11 token. On the TKE workstation, you must enable bitcoin derivation by setting the following access control point 66 on the used cryptographic coprocessors:

```
66  XCP_CPB_BTC   enable bitcoin (BTC) derivation with the CKM_IBM_BTC_DERIVE mechanism
```

Description:

You can use the CKM_IBM_BTC_DERIVE mechanism to derive a number of child keys from parent ECC keys generated from certain elliptic curves. The derived keys can be applied for encryption used by bitcoin. This mechanism in turn uses the methods described in *BIP-0032* and *SLIP-0010*.

With method *BIP-0032* you can derive keys from SECP256K1 curve EC keys. With method *SLIP-0010* you can derive EC keys from secp256k1, prime256v1, or ED25519 curves.

Derivation is furthermore restricted to keys bearing the attributes CKA_IBM_USE_AS_DATA and CKA_DERIVE. For *SLIP-0010* key derivation, EC keys on different curves can be derived. Thus, the CKA_EC_PARAMS attribute needs to be present in the key attribute templates of both base and child key. The only exception is *SLIP-0010* master key derivation for which the key type is determined solely from the attributes of the child key.

Each parent key can derive 2^{31} normal and 2^{31} hardened child keys given by a child key index.

In order to prevent the derived keys from depending solely on the parent key itself, both private and public keys are at first extended with an entropy of 256 bits (32 bytes). This extension, called the chain code, is identical for corresponding private and public keys.

The following derivation types are supported:

- Private parent key derives private child keys
- Public parent key derives public child keys
- Private parent key derives public child keys
- Master key derivation

The output of this mechanism contains the derived key, public or private, and the corresponding chain code.

For master key derivation, a generic secret key (CKK_GENERIC_SECRET) must be specified as base key. The mechanism parameter called CK_IBM_BTC_DERIVE_PARAMS (see Figure 22 on page 153) must specify a buffer of 32 bytes size for the returned chain code, but the chain code length field within this parameter must be zero, because no chain code is input to the operation.

For child key derivation, an EC private or public key (CKK_EC) must be specified as base key. The CK_IBM_BTC_DERIVE_PARAMS mechanism parameter must specify a buffer of 32 bytes size containing the chain code corresponding to the base key. The chain code of the derived child key is also returned in this buffer. Thus you can use the derived child key again as a base key. The chain code length field must be 32 bytes.

For *SLIP-0010* key derivation with ED25519 curves, only hardened key generation from private parent key to private or public child key is supported.

The following definitions are provided in header file `pkcs11types.h` which is in turn included in file `pkcs11.h` (see also “Sample openCryptoki program” on page 165).

```
#define CKM_IBM_BTC_DERIVE   CKM_VENDOR_DEFINED + 0x70001

typedef struct CK_IBM_BTC_DERIVE_PARAMS {
    CK_ULONG type;
    CK_ULONG childKeyIndex;
    CK_BYTE_PTR pChainCode;
    CK_ULONG ulChainCodeLen;
    CK_ULONG version;
} CK_IBM_BTC_DERIVE_PARAMS;

typedef CK_IBM_BTC_DERIVE_PARAMS   CK_PTR CK_IBM_BTC_DERIVE_PARAMS_PTR;

#define CK_IBM_BIP0032_HARDENED 0x80000000 // key index flag

#define CK_IBM_BIP0032_PRV2PRV 1
#define CK_IBM_BIP0032_PRV2PUB 2
#define CK_IBM_BIP0032_PUB2PUB 3
#define CK_IBM_BIP0032_MASTERK 4
#define CK_IBM_SLIP0010_PRV2PRV 5
#define CK_IBM_SLIP0010_PRV2PUB 6
#define CK_IBM_SLIP0010_PUB2PUB 7
#define CK_IBM_SLIP0010_MASTERK 8

#define CK_IBM_BTC_CHAINCODE_LENGTH 32

#define CK_IBM_BTC_DERIVE_PARAMS_VERSION_1 1
```

Figure 22. Definitions for the CKM_IBM_BTC_DERIVE mechanism

CKM_IBM_ECDSA_OTHER

Availability:

The CKM_IBM_ECDSA_OTHER mechanism is available with the EP11 token.

Description:

You can use this mechanism to generate Schnorr signatures, that is, EC-based signatures that are not produced using the ECDSA or the EDDSA digital signature algorithms.

On the TKE workstation, you must enable non-ECDSA and non-EdDSA elliptic curve signature algorithms, by setting the following access control point 67 on the used cryptographic coprocessors:

```
67  XCP_CPB_ECDSA_OTHER   enable non-ECDSA and non-EdDSA elliptic
                          curve signature algorithms with the CKM_IBM_ECDSA_OTHER mechanism
```

The CKM_IBM_ECDSA_OTHER mechanism is a multi-variant signature mechanism for algorithms used by crypto-currency applications which differ from ECDSA or EDDSA. Several of these elliptic curve signature algorithms are generally variations of Schnorr signatures. The CKM_IBM_ECDSA_OTHER mechanism uses a mechanism parameter CK_IBM_ECDSA_OTHER_PARAMS that specifies the sub-mechanism to perform:

- **CKM_IBM_ECSDSA_RAND**: Randomized Schnorr signatures (BSI TR03111 ECSDSA, no pre-hashing, SHA-256 only)
- **CKM_IBM_ECSDSA_COMPR_MULTI**: Randomized Schnorr signatures (BSI TR03111 ECSDSA 2012, internally using compressed key format, and including signing party's public key, no pre-hashing, SHA-256 only)

Only 256-bit elliptic curves are supported, in particular secp256k1, prime256v1, brainpoolP256r1, and brainpoolP256t1 curves.

The following definitions are provided in header file `pkcs11types.h` which is in turn included in file `pkcs11.h` (see also "Sample openCryptoki program" on page 165).

```
#define CKM_IBM_ECDSA_OTHER            CKM_VENDOR_DEFINED + 0x00010031

typedef struct CK_IBM_ECDSA_OTHER_PARAMS {
    CK_MECHANISM_TYPE submechanism;
} CK_IBM_ECDSA_OTHER_PARAMS;

typedef CK_IBM_ECDSA_OTHER_PARAMS CK_PTR CK_IBM_ECDSA_OTHER_PARAMS_PTR;

/* CKM_IBM_ECDSA_OTHER sub-mechanisms */
#define CKM_IBM_ECSDSA_RAND                3
#define CKM_IBM_ECSDSA_COMPR_MULTI         5
```

Figure 23. Definitions for the CKM_IBM_ECDSA_OTHER mechanism

Miscellaneous attributes

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

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

CKA_IBM_OPAQUE
Use this attribute for importing and exporting plain CCA key objects into or from sensitive openCryptoki key objects. See also "Usage notes for CCA library functions" on page 97.

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

CKA_IBM_PROTKEY_EXTRACTABLE
This key attribute is internally set to CK_TRUE, if for CCA tokens or for EP11 tokens the configuration option PKEY_MODE is enabled and the key has CKA_EXTRACTABLE set to FALSE. This makes the key eligible for being transformed into a protected key for better performance, if applicable (see also "Defining an EP11 token configuration file" on page 107).

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

CKA_IBM_STD_COMPLIANCE1

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

CKA_IBM_USE_AS_DATA

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

Chapter 21. Re-encrypting data with a mechanism

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

The `C_IBM_ReencryptSingle` function has the following signature:

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

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

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

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

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

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

© Copyright IBM Corp. 2021, 2023

Part 6. Programming basics and user scenarios

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

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

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

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

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

For developers of new openCryptoki applications, Chapter 22, “Programming with openCryptoki,” on page 161 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 24, “Configuring a remote PKCS #11 service with openCryptoki,” on page 183 provides a user scenario showing how to set up a Soft token on a server for use from an application on a remote client.

© Copyright IBM Corp. 2021, 2023

Chapter 22. 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_`

Handling RSA decrypt operation errors

There are attacks on RSA operations for which the timing of a computation may deliver a hint for well- or malformed input and thus, whether an attack may have a chance to be successful. RSA operations that may be the target of such attacks are for example, wrong RSA-OAEP padding or malformed RSA PKCS#1 v1.5 input messages. Starting with version 3.21, openCryptoki itself cares for hiding the computation times, however, PKCS #11 applications that might be subject to RSA timing attacks (for example, a SSL/TLS software stack) must make sure to handle certain RSA decryption errors in a constant-time manner itself.

openCryptoki can only make internal differences in computation times transparent, but can not influence the code in the cryptographic coprocessors, in the respective host libraries, nor in the calling applications. Therefore, it is the PKCS #11 application's responsibility to further handle such an error situation in a constant-time manner to not be vulnerable to timing attacks. That is, the PKCS #11 application should not expose a timing difference when an RSA decryption operation fails (for example. due to wrong padding), compared to a successful operation.

For example, to mitigate the Bleichenbacher attack on SSL/TLS RSA key exchange, the application should return a random pre-computed RSA secret, if the RSA decryption using PKCS#1 v1.5 padding fails due to padding errors. The random secret must be generated in any case, prior to performing the decrypt operation.

To protect PKCS #11 applications using the ICA token and the Soft token from attacks against RSA operations, RSA message blinding is applied to messages on all operations using the RSA private key (decryption and signature creation).

© Copyright IBM Corp. 2021, 2023

How to create and modify objects

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

An application can use the following functions for creating objects:

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

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

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

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

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

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

CK_KEY_TYPE keyType = CKK_RSA;

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

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

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

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

How to apply attributes to objects

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

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

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

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

Structure of an openCryptoki application

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

Before you begin

Consider two things:

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

Procedure

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

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

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

```
CK_RV C_Initialize(CK_VOID_PTR pInitArgs);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CK_RV rv;

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

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

```
#define PLAINTEXT_BUF_SZ 200
#define CIPHERTEXT_BUF_SZ 256

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

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

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


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

  /* Get last little encrypted bit */

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

}
```

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

```

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

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

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

```

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

Sample openCryptoki program

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

Note: This sample program does not include the operation to encrypt data with the previously generated RSA private key as described in step "7" on page 164 of "Structure of an openCryptoki application" on page 163.

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

```
 */
/* Step 1 */
#include <opencryptoki/pkcs11.h>
#include <string.h>
#include <stdlib.h>
#include <dlfcn.h>

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

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

    if (argc != 4) goto err;

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

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

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

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

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

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

/* Step 5 */
    rv = fn->C_Login(session, CKU_USER, (CK_UTF8CHAR *)argv[3], strlen(argv[3]));
    if (rv != CKR_OK) goto err;

 /* Step 6  (Step 7 not coded in this example */
    {
        CK_MECHANISM mechanism = {CKM_RSA_PKCS_KEY_PAIR_GEN, NULL, 0};
        CK_BYTE e[] = {0x01, 0x00, 0x01};
        CK_ULONG modbits = 4096;
        CK_BYTE subject[] = "RSA4096 Test";
        CK_BYTE id[] = {1};
        CK_BBOOL true_ = CK_TRUE;

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

        CK_OBJECT_HANDLE publ, priv;

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

```
    }

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

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

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

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

openCryptoki code samples (C)

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

There are two ways to access the library:

- **Dynamic linking and loading:** Load the shared `libopencryptoki.so` library using dynamic library calls (dlopen).
- **Static linking:** Link the static `libopencryptoki.so` library to your application during build time.

Dynamic library call

View a openCryptoki code sample for a dynamic library call.

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

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

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

CK_RV init(char *libPath){

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

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

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

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

Statically linked library

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

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

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

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

Base procedures

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

Main program

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

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

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

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

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

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

C_Initialize

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

C_Finalize

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

Session and log-in procedures

View some openCryptoki code samples for opening and closing sessions and for log-in.

C_OpenSession:

```
/*
 * opensession
 */

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

C_CloseSession:

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

C_Login:

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

C_Logout:

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

Object handling procedures

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

C_CreateObject:

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

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

C_FindObjects:

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

Cryptographic operations

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

C_Encrypt (AES):

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

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

  return CKR_OK;
}
```

C_Decrypt (AES):

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

C_GenerateKeyPair (RSA):

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

C_Encrypt (RSA):

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

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

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

C_Decrypt (RSA):

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

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

Chapter 23. Trouble shooting

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

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

1. "Walk through a check list" on page 179
2. "Checking the syslog messages" on page 180
3. "Apply tracing" on page 182
4. "Final check" on page 182

In addition, you may find helpful information in the following topic:

- "Re-initialize a token" on page 182

Walk through a check list

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

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

   Every user of openCryptoki must be a member of this pkcs11 group. For more information, read "Access control and groups" on page 12.

2. Is the slot manager daemon **`pkcsslotd`** up and running?

   The **`pkcsslotd`** daemon must be active in the system to coordinate token accesses from multiple processes. For more information, read "Slot manager" on page 7.

3. Is the host library of the token installed that you want to exploit?

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

4. Are the required cryptographic coprocessors available and online?

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

```
# lszcrypt

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

© Copyright IBM Corp. 2021, 2023

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

   A list of all available tokens is required before you can use openCryptoki. This list is provided by the global configuration file called `opencryptoki.conf`. For more information, read Chapter 5, "Adjusting the openCryptoki configuration file," on page 21.

Checking the syslog messages

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

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

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

chmod(<file>): <strerror>

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

Could not open <lockfilename>

<dlerror>

Directory(<dir>) missing: <strerror>

DL_Load: dlopen() failed for [<dll_location>]; dlerror = <dlerror>

ep11_load_host_lib: Error loading shared library <file>' [<strerror>]
ep11_load_host_lib: Error loading shared library 'libep11.so[.3|.2|.1]' [<strerror>]
ep11_login_handler: Error: VHSM-Pin blob of adapter <apqn> is not equal to other adapters
                    for same session
ep11_login_handler: Error: Pin blob of adapter <apqn> is not equal to other adapters
                    for same session
ep11_resolve_lib_sym: Error: <dlerror>

ep11tok_load_libica: Error loading shared library '<file>' [<strerror>]
ep11tok_load_libica: Failed to initialize the target lock
ep11tok_load_libica: Error: EP 11 library initialization failed
ep11tok_load_libica: Failed to get the EP11 library version rc=x
ep11tok_load_libica: Failed to get the target info rc=x
ep11tok_load_libica: Error: CKR_IBM_WK_NOT_INITIALIZED occurred, no master key set ?
ep11tok_load_libica: Error: CKR_FUNCTION_CANCELED occurred, control point 13
                     (generate or derive symmetric keys including DSA parameters) disabled ?
ep11tok_load_libica: Warning: Could not get mk_vp, protected key support not available.
ep11tok_login_session: Error: A VHSM-PIN is required for VHSM_MODE.
ep11tok_handle_apqn_event: Failed to get the target info rc=x


fchmod(<file>): <strerror>
fchown(<file>): <strerror>
fchown(<file>,-1,pkcs11) failed: <strerror>. Tracing is disabled.

getgrnam() failed: <strerror>
getgrnam(pkcs11) failed: <strerror>. Tracing is disabled.
getgrnam(): <strerror>
getpwuid(): <strerror>
```

```
init_socket_data: read error on daemon socket, errno=x
init_socket_data: read returned with eof but we still expect x bytes from daemon

Invalid strength configuration in policy!

lock directory path too long
lock file path too long

mkdir(<file>): <strerror>

OPENCRYPTOKI_TRACE_LEVEL '<string>' is invalid. Tracing disabled.

open(<file>) failed: <strerror>. Tracing disabled.
open(<file>): <strerror>

Parsing policy configuration failed!

POLICY: Could not retrieve "pkcs11" group!
POLICY: Could not stat configuration file <file>: <strerror>
POLICY: Configuration file <file> should be owned by "root"
POLICY: Configuration file <file> should have group "pkcs11"!
POLICY: Configuration file <file> has wrong permissions!
POLICY: Unknown curve "<curve>" in line <line>
POLICY: allowedmechs has wrong type!
POLICY: allowedcurves has wrong type!
POLICY: allowedmgfs has wrong type!
POLICY: allowedkdfs has wrong type!
POLICY: allowedprfs has wrong type!
POLICY: Failed to open <file>: <strerror>
POLICY: Could not allocate policy private data!
POLICY: Strength definition <file> failed to parse!
POLICY: Failed to open <file>: <strerror>
POLICY: Policy definition <file> failed to parse!

POLICY VIOLATION: CKM_AES_KEY_GEN needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_AES_KEY_WRAP needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_AES_GCM needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_PKCS5_PBKD2 needed by Token-Store for slot <slot>
POLICY VIOLATION: CKP_PKCS5_PBKD2_HMAC_SHA512 needed by Token-Store for slot <slot>
POLICY VIOLATION: Token-Store encryption method not allowed for slot <slot>!
POLICY VIOLATION: Token-Store requires SHA1 for slot <slot>!
POLICY VIOLATION: Token-Store requires MD5 for slot <slot>!
POLICY VIOLATION: CKM_DES3_KEY_GEN needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_AES_KEY_GEN needed by Token-Store for slot <slot>
POLICY VIOLATION: Unknown Token-Store encryption method for slot <slot>!
POLICY VIOLATION: CKM_AES_KEY_GEN needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_AES_CBC needed by Token-Store for slot <slot>
POLICY VIOLATION: CKM_PKCS5_PBKD2 needed by Token-Store for slot <slot>
POLICY VIOLATION: CKP_PKCS5_PBKD2_HMAC_SHA256 needed by Token-Store for slot <slot>
POLICY VIOLATION: Token-Store encryption key too weak for slot <slot>!

read_adapter_config_file: Error: EP 11 config file ''<file>' not found
read_adapter_config_file: Error: EP 11 config file '<file>' is too large
read_adapter_config_file: Error: Expected APQN_ALLOWLIST, APQN_ANY, LOGLEVEL, FORCE_SENSITIVE,
                          CPFILTER, STRICT_MODE, VHSM_MODE, OPTIMIZE_SINGLE_PART_OPERATIONS,
                          PKEY_MODE, DIGEST_LIBICA, or USE_PRANDOM keyword, found '<token>
                          in config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>',
                          expected 'END' or adapter number
read_adapter_config_file: Error: Expected valid adapter number, found '<token>'
                          in config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>,
                          expected domain number (2nd number)
read_adapter_config_file: Error: Expected valid domain number (2nd number), found '<token>'
                          in config file <file>
read_adapter_config_file: Error: Too many APQNs in config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>',
                          expected LOGLEVEL value
read_adapter_config_file: Error: Invalid LOGLEVEL value '<token>' in config file <file>
read_adapter_config_file: Warning: LOGLEVEL setting is not supported any more. Use opencryptoki
                          logging/tracing facilities instead.
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>',
                          expected CP-Filter file name
read_adapter_config_file: Error: CP-Filter config file name '<file>' is too long in
                          config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>,
                          expected libica path, 'DEFAULT', or 'OFF'
read_adapter_config_file: Error: libica path '<token>' is too long in config file '<file>'
read_adapter_config_file: Error: Unexpected end of file found in config file '<file>',
                          expected pkey_mode 0 .. 3
read_adapter_config_file: Error: unsupported pkey mode '<token> in config file '<file>'
```

```
read_adapter_config_file: Error: At least one APQN mode needs to be present in config file
                          '<file>': APQN_ALLOWLIST or APQN_ANY
read_adapter_config_file: Error: Only one APQN mode can be present in config file '<file>':
                          APQN_ALLOWLIST or APQN_ANY
read_adapter_config_file: Error: At least one APQN needs to be defined in config file '<file>'
read_cp_filter_config_file: Warning: EP 11 CP-filter config file '<file>' does not exist,
                            no filtering will be used
read_cp_filter_config_file: Error: Expected valid control point name or number,
                            found '<ftoken>' in CP-filter config file '<file>'
read_cp_filter_config_file: Error: Expected valid mechanism name or number, found '<token>'
                            in CP-filter config file '<file>'
read_cp_filter_config_file: Error: Out of memory while parsing CP-filter config file '<file>'

SHM segment has wrong gid/mode combination (expected: x/0x; got: x/0x)

start_event_thread: pthread_create failed, errno=x

token_specific_init: Error loading library: 'libcsulcca.so' [<dlerror>]

Trace level x is out of range. Tracing disabled.

Token object <file> appears corrupted (ignoring it)

Tspi_Key_GetPubKey failed: rc=x

Username(<name>) too long

Warning: CCA symmetric master key is not yet loaded
Warning: CCA asymmetric master key is not yet loaded
Warning: Your TPM is not configured to allow reading the public SRK by anyone but the owner.
         Use tpm_restrictsrk -a to allow reading the public SRK
Warning: Adapter <apqn1> has different control points than adapter <apqn2>, using minimum
Warning: Adapter <apqn1> has a different number of control points than adapter <apqn2>,
         using maximum
Warning: Adapter <apqn1> has a different API versionversion than the previous CEXxP adapters: x
Warning: Adapter <apqn1> has a different firmware version than the previous CEXxP adapters: x.x
```

Apply tracing

If you successfully checked all issues as described in the previous sections, you may start to exploit the openCryptoki tracing capabilities. If a log level > 0 is activated by setting the environment variable OPENCRYPTOKI_TRACE_LEVEL, then log entries are written to file `/var/log/opencryptoki/trace.<process_id>`. Application programmers may apply the higher tracing log levels 4 and 5.

For detailed information, read "Logging and tracing in openCryptoki" on page 12.

Final check

Finally, if your openCryptoki environment is successfully set up, you can check your settings using the **pkcsconf** utility as described in Chapter 8, "Managing tokens - pkcsconf utility," on page 39.

Re-initialize a token

In case you need to clean a token and initialize it freshly (for example, because you forgot the SO PIN, or the SO PIN got locked), perform the following actions:

1. Remove all the files in the token directory (for example, `/var/lib/opencrptoki/<token>/`), that is, MK_SO, MK_USER, NVTOK.DAT, as well as all files inside TOK_OBJ (but do not remove the TOK_OBJ directory itself).
2. Remove the shared memory segment under `/dev/shm` for that token using the command:

```
rm /dev/shm/var.lib.opencryptoki.<token>
```

3. Freshly initialize the token using the command **pkcsconf -I**.
4. Set the SO and USER PINs.

Attention: You will loose all token objects, and you need to setup the PINs freshly.

Chapter 24. Configuring a remote PKCS #11 service with openCryptoki

A user scenario shows how to set up a Soft token on a server for use from an application on a remote client.

The user scenario presented in this topic describes how you can set up a remote token on an IBM z15 system with an installed Ubuntu 21.04 Linux environment. This token is accessed and exploited from an application running on an x86 client with an installed Red Hat Enterprise Linux 7.9 environment. The Ubuntu 21.04 setup is selected, because at the time of writing, this distribution shipped all required packages and package versions. An analogous setup is possible with subsequent distributions.

Information about the required set up on the server and client side is presented in the contained subtopics:

- “Server side setup” on page 183
- “Client side setup” on page 185

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 (see “Support of IBM-specific mechanisms - p11-kit” on page 187).

   This tool provides a way to load and enumerate PKCS #11 modules and also provides a standard configuration setup for installing PKCS #11 modules in such a way that they are discoverable.

   To install the **p11-kit** package and the **p11tool**, enter the following command:

   ```
   # apt install p11-kit p11-kit-modules gnutls-bins
   ```

2. Create and edit an `opencryptoki.module` configuration file in the shown filepath: `/etc/pkcs11/modules/opencryptoki.module`

   Enter the following line into this configuration file:

   ```
   module: /lib64/opencryptoki/libopencryptoki.so
   ```

3. To list the available PKCS #11 modules, enter the following command:

   ```
   # p11-kit list-modules
   ```

   You will see an output similar to the following:

© Copyright IBM Corp. 2021, 2023

```
p11-kit-trust: p11-kit-trust.so
    library-description: PKCS#11 Kit Trust Module
    library-manufacturer: PKCS#11 Kit
    library-version: 0.23
    token: System Trust
        manufacturer: PKCS#11 Kit
        model: p11-kit-trust
        serial-number: 1
        hardware-version: 0.23
        flags:
               write-protected
               token-initialized
    opencryptoki: /lib64/opencryptoki/libopencryptoki.so
    library-description: openCryptoki
    library-manufacturer: IBM
    library-version: 3.16
    token: soft
        manufacturer: IBM
        model: Soft
        serial-number:
        flags:
               rng
               login-required
               user-pin-initialized
               clock-on-token
               token-initialized
```

4. To list the available tokens using the **p11tool** utility, enter the following command:

```
# p11tool --list-tokens
```

You will see an output similar to the following:

```
Token 0:
        URL: pkcs11:model=p11-kit-
trust;manufacturer=PKCS%2311%20Kit;serial=1;token=System%20Trust
        Label: System Trust
        Type: Trust module
        Flags: uPIN uninitialized
        Manufacturer: PKCS#11 Kit
        Model: p11-kit-trust
        Serial: 1
        Module: p11-kit-trust.so

Token 1:
        URL: pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft
        Label: soft
        Type: Generic token
        Flags: RNG, Requires login
        Manufacturer: IBM
        Model: Soft
        Serial:
        Module: opencryptoki: /lib64/opencryptoki/libopencryptoki.so
```

As you can see in the example, the Soft token is available now as **Token 1**. With the shown URL, you can access this token.

5. To start the **p11-kit** server to allow remote clients to access the token, enter the following command:

```
# p11-kit server --provider /lib64/opencryptoki/libopencryptoki.so
"pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft"
```

As output, you will see the following generated commands:

```
P11_KIT_SERVER_ADDRESS=unix:path=/run/user/0/p11-kit/pkcs11-1296159; export
P11_KIT_SERVER_ADDRESS;
P11_KIT_SERVER_PID=1296160; export P11_KIT_SERVER_PID;
```

6. To set and export the following two environment variables, copy and paste the commands from the output from step 5 and enter them into a command line:

```
# P11_KIT_SERVER_ADDRESS=unix:path=/run/user/0/p11-kit/pkcs11-1296159; export
P11_KIT_SERVER_ADDRESS;
# P11_KIT_SERVER_PID=1296160; export P11_KIT_SERVER_PID;
```

Results

You can now continue to set up the client as described in "Client side setup" on page 185.

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 (see "Support of IBM-specific mechanisms - p11-kit" on page 187), enter the following command:

```
$ sudo yum install p11-kit
```

2. To query the user run-time path, enter the following command:

```
$ systemd-path user-runtime
```

You will see an output similar to the following:

```
/run/user/1000
```

3. To forward the local UNIX socket to the remote socket, enter the following commands, using the information from step 2 and then log in as a root user into the remote server:

```
$ mkdir /run/user/1000/p11-kit/
$ ssh -L /run/user/1000/p11-kit/pkcs11-1296159:/run/user/0/p11-kit/pkcs11-1296159
root@<remote_server_name>
```

4. To export the **`p11-kit`** server address environment variable, enter the following command:

```
$ P11_KIT_SERVER_ADDRESS=unix:path=/run/user/1000/p11-kit/pkcs11-1296159; export
P11_KIT_SERVER_ADDRESS;
```

5. As the Red Hat Enterprise Linux 7.9 distribution does not package the `p11-kit-client.so` file, you need to build it from the source. Therefore, clone the shown GitHub repository. To achieve this, enter the following command sequence:

```
$ git clone https://github.com/p11-glue/p11-kit.git
$ cd p11-kit
$ git checkout 0.23.10
$ ./autogen.sh
$ ./configure
$ make
```

6. To view a list of available tokens, use the **p11tool**:

```
$ p11tool --provider /<path>/p11-kit-client.so --list-tokens
```

   You will see an output similar to the following, showing that the Soft token is remotely available.

```
Token 0:
        URL: pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft
        Label: soft
        Type: Generic token
        Manufacturer: IBM
        Model: Soft
        Serial:
```

7. To view a list of available mechanisms of the Soft token, use the **p11tool** utility:

```
$ p11tool --provider /<path>/p11-kit-client.so --list-mechanisms
"pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft"
```

   You will see an output list similar to the following (see also “PKCS #11 mechanisms supported by the Soft token” on page 125):

```
[0x0000] CKM_RSA_PKCS_KEY_PAIR_GEN
[0x0120] CKM_DES_KEY_GEN
[0x0131] CKM_DES3_KEY_GEN
[0x0001] CKM_RSA_PKCS
[0x0006] CKM_SHA1_RSA_PKCS
[0x0040] CKM_SHA256_RSA_PKCS
[0x0041] CKM_SHA384_RSA_PKCS
[0x0042] CKM_SHA512_RSA_PKCS
[0x000d] CKM_RSA_PKCS_PSS
[0x0003] CKM_RSA_X_509
[0x0009] CKM_RSA_PKCS_OAEP
[0x0005] CKM_MD5_RSA_PKCS
[0x0006] CKM_SHA1_RSA_PKCS
[0x0020] CKM_DH_PKCS_KEY_PAIR_GEN
[0x0121] CKM_DES_ECB
[0x0132] CKM_DES3_ECB
[0x0134] CKM_DES3_MAC
[0x0220] CKM_SHA_1
[0x0221] CKM_SHA_1_HMAC
[0x0250] CKM_SHA256
[0x0251] CKM_SHA256_HMAC
[0x0260] CKM_SHA384
[0x0261] CKM_SHA384_HMAC
[0x0270] CKM_SHA512
[0x0271] CKM_SHA512_HMAC
[0x0210] CKM_MD5
[0x0211] CKM_MD5_HMAC
[0x0370] CKM_SSL3_PRE_MASTER_KEY_GEN
[0x0380] CKM_SSL3_MD5_MAC
[0x0381] CKM_SSL3_SHA1_MAC
[0x1080] CKM_AES_KEY_GEN
[0x1081] CKM_AES_ECB
[0x1083] CKM_AES_MAC
[0x0350] CKM_GENERIC_SECRET_KEY_GEN
[0x1040] CKM_ECDSA_KEY_PAIR_GEN
[0x1041] CKM_ECDSA
[0x1042] CKM_ECDSA_SHA1
```

8. Use the **p11tool** utility to issue the following command to generate an RSA private and public key pair of a length of 2048 bits:

```
$ p11tool --provider /<path>/p11-kit-client.so --generate-rsa --bits 2048 --login
"pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft"
```

You will see an output similar to the following;

```
warning: no --outfile was specified and the generated public key will be printed on screen.
note: in some tokens it is impossible to obtain the public key in any other way after
generation.
warning: Label was not specified. Label: my-rsa-key Token 'soft' with URL
'pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft' requires user PIN
Enter PIN:
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAzwrYewbV0LybCcb9inQ4
1n/jReFtjrYGx2M4B373em+gMiaDlc+T8Y9yvofDoEwZkjN2OOkUPD2GFb8P88a5
jGF8M+FlkZe+E7XlcHvttFPlULHDpAIXK0UnZJrbAR1ncP8O9lKqhV3CdrXw8dwm
ovdG/FVCyaKv4IlGVj4OKwx5IL0L9JBoSluRRtPNqwSYrXKGEYUjfko+PXm7MVuu
DQv2Ckr6KDEnIsk8U7W9hOHWfjZ4OVKSpbqPlRmG5whWL/hYoGQ181IDXeMajH/1
KgQAI7ree8JS2R4/Os0fzR7+Rp6AvpE4BQ6rXZOkO/7EQLbiCSq930TWsE9IEbMT
xQIDAQAB
-----END PUBLIC KEY-----
```

9. Issue the following command to list all available objects in the token:

```
$ p11tool --provider /<path>/p11-kit-client.so --list-all --login
"pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft"
```

You are prompted for your user PIN:

```

Token 'soft' with URL 'pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft' requires user PIN
 Enter PIN: <USER PIN>
 [...]

Object 6:
         URL:
pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft;id=%8a%b8%84%b3%f0%60%1c%32%2e%19%6e%f1%5
5%7f%30%e3%bf%6c%f3%82;object=my-rsa-key;type=private
         Type: Private key
         Label: my-rsa-key
         Flags: CKA_WRAP/UNWRAP; CKA_PRIVATE; CKA_SENSITIVE;
         ID: 8a:b8:84:b3:f0:60:1c:32:2e:19:6e:f1:55:7f:30:e3:bf:6c:f3:82

Object 7:
         URL:
pkcs11:model=Soft;manufacturer=IBM;serial=;token=soft;id=%8a%b8%84%b3%f0%60%1c%32%2e%19%6e%f1%5
5%7f%30%e3%bf%6c%f3%82;object=my-rsa-key;type=public
         Type: Public key
         Label: my-rsa-key
         Flags: CKA_WRAP/UNWRAP;
         ID: 8a:b8:84:b3:f0:60:1c:32:2e:19:6e:f1:55:7f:30:e3:bf:6c:f3:82
```

Results

On your client, you can now write cryptographic applications that exploit the mechanisms of the Soft token using the openCryptoki API (see also Chapter 22, “Programming with openCryptoki,” on page 161).

Support of IBM-specific mechanisms - p11-kit

You can use the **p11-kit** command line tool to perform operations on PKCS #11 tokens configured on the system. It can especially be used to provide remote PKCS #11 API access to openCryptoki tokens through an RPC-like communication protocol.

The **p11-kit** tool is enhanced so that it supports IBM-specific mechanisms and attributes used by openCryptoki. You should verify which mechanisms and attributes are exactly supported by the **p11-kit** version packaged by your distribution.

Using p11-kit to access an openCryptoki token from a remote system

On the client side, a **p11-kit** client library acts as a PKCS #11 interface for the application. It forwards all API calls to a **p11-kit** server over a network connection. On the server side, the API calls are then passed to another PKCS #11 interface library, that is, to openCryptoki's API library.

The communication between **p11-kit** client and **p11-kit** server is based on UNIX domain sockets, which are forwarded through an SSH tunnel from the client to the server. Because an SSH tunnel is used, the communication is encrypted and authenticated by means of regular SSH authentication.

The client side may run on a different architecture than the server, because the RPC protocol is endianess-save. It is for example possible to run the client application on an x86-Linux, connecting to an s390x-Linux running openCryptoki with the EP11 token.

The **p11-kit** tool needs to explicitly know all mechanisms and attributes in order to support them. For mechanisms, it needs to know if a mechanism uses a mechanism parameter, and if so, how to serialize the mechanism parameter. Some complex mechanism parameters require specific serialization, especially when the mechanism parameter structure contains pointers to other buffers.

Mechanisms that are not known by **p11-kit** on the client side are filtered out. The mechanism list retrieved by the application only contains those mechanisms that are supported by both sides.

For attributes, the **p11-kit** tool needs to know the data type of each attribute, that is, if the attribute contains a boolean value, an ULONG value, or a byte array (binary) value, in order to serialize it properly (that means,. endianess for ULONG attributes).

Support for the following mechanisms (including mechanism parameter serialization support as needed) is added:

- CKM_IBM_SHA3_224
- CKM_IBM_SHA3_256
- CKM_IBM_SHA3_384
- CKM_IBM_SHA3_512
- CKM_IBM_CMAC
- CKM_IBM_EC_X25519 (mechanism parameter: CK_ECDH1_DERIVE_PARAMS)
- CKM_IBM_ED25519_SHA512
- CKM_IBM_EC_X448 (mechanism parameter: CK_ECDH1_DERIVE_PARAMS)
- CKM_IBM_ED448_SHA3
- CKM_IBM_DILITHIUM
- CKM_IBM_SHA3_224_HMAC
- CKM_IBM_SHA3_256_HMAC
- CKM_IBM_SHA3_384_HMAC
- CKM_IBM_SHA3_512_HMAC
- CKM_IBM_ATTRIBUTEBOUND_WRAP (mechanism parameter: CK_IBM_ATTRIBUTEBOUND_WRAP_PARAMS)

Support for the following attributes is added:

- CKA_IBM_OPAQUE (binary)
- CKA_IBM_RESTRICTABLE (boolean)
- CKA_IBM_NEVER_MODIFIABLE (boolean)
- CKA_IBM_RETAINKEY (boolean)
- CKA_IBM_ATTRBOUND (boolean)
- CKA_IBM_KEYTYPE (ULONG)
- CKA_IBM_CV (binary)

- CKA_IBM_MACKEY (binary)
- CKA_IBM_USE_AS_DATA (boolean)
- CKA_IBM_STRUCT_PARAMS (binary)
- CKA_IBM_STD_COMPLIANCE1 (ULONG)
- CKA_IBM_PROTKEY_EXTRACTABLE (boolean)
- CKA_IBM_PROTKEY_NEVER_EXTRACTABLE (boolean)
- CKA_IBM_DILITHIUM_KEYFORM (ULONG)
- CKA_IBM_DILITHIUM_RHO (binary)
- CKA_IBM_DILITHIUM_SEED (binary, sensitive)

Support for the following key type is added:

- CKK_IBM_PQC_DILITHIUM

Besides the previously listed IBM-specific mechanisms, support for the following standard mechanisms is added, which require special mechanism parameter serialization support:

- CKM_ECDH1_DERIVE (Mechanism parameter: CK_ECDH1_DERIVE_PARAMS)
- CKM_SHA1_RSA_PKCS_PSS
- CKM_SHA224_RSA_PKCS_PSS
- CKM_SHA256_RSA_PKCS_PSS
- CKM_SHA384_RSA_PKCS_PSS
- CKM_SHA512_RSA_PKCS_PSS
- CKM_AES_CBC (mechanism parameter: 16 bytes IV)
- CKM_AES_CBC_PAD (mechanism parameter: 16 bytes IV)
- CKM_AES_OFB (mechanism parameter: 16 bytes IV)
- CKM_AES_CFB1 (mechanism parameter: 16 bytes IV)
- CKM_AES_CFB8 (mechanism parameter: 16 bytes IV)
- CKM_AES_CFB64 (mechanism parameter: 16 bytes IV)
- CKM_AES_CFB128 (mechanism parameter: 16 bytes IV)
- CKM_AES_CTS (mechanism parameter: 16 bytes IV)
- CKM_AES_CTR (mechanism parameter: CK_AES_CTR_PARAMS)
- CKM_AES_GCM (mechanism parameter: CK_GCM_PARAMS)
- CKM_DES_CBC (mechanism parameter: 8 bytes IV)
- CKM_DES_CBC_PADC (mechanism parameter: 8 bytes IV)
- CKM_DES3_CBCC (mechanism parameter: 8 bytes IV)
- CKM_DES3_CBC_PADC (mechanism parameter: 8 bytes IV)
- CKM_DES_CFB8C (mechanism parameter: 8 bytes IV)
- CKM_DES_CFB64C (mechanism parameter: 8 bytes IV)
- CKM_DES_OFB64C (mechanism parameter: 8 bytes IV)
- CKM_SHA_1_HMAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_SHA224_HMAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_SHA256_HMAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_SHA384_HMAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)

- CKM_SHA512_HMAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_SHA512_224_HMAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_SHA512_256_HMAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_AES_MAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_AES_CMAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_DES3_MAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_DES3_CMAC_GENERAL (mechanism parameter: CK_MAC_GENERAL_PARAMS, which is a CK_ULONG)
- CKM_DH_PKCS_DERIVE (mechanism parameter: public value of the other party)

The following features of openCryptoki are not supported by the p11-kit:

- Attribute-array attributes are not supported by p11-kit (CKA_WRAP_TEMPLATE, CKA_UNWRAP_TEMPLATE, CKA_DERIVE_TEMPLATE).
- PKCS #11 version 3.0 interfaces are not supported (`C_GetInterfaceList()`, `C_GetInterface()`).
- `C_IBM_ReencryptSingle()` is not supported because it is available only via a vendor specific interface obtainable from `C_GetInterface()`.

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 Specification Version 3.1`
- `PKCS #11 Profiles Version 3.1`
- `PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 3.0`

© Copyright IBM Corp. 2021, 2023

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 IBM Z and IBM 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.

The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Index

A

access control 12
access control point
  ACP 107
accessibility 193
ACP
  access control point 107, 152
  XCP_CPB_BTC 152
ACP-filter configuration file 107
AES XTS
  CCA token 87
  EP11 token 87
APQN_ALLOWLIST 107
attributes
  CKA_IBM_ATTRBOUND 154
  CKA_IBM_OPAQUE 154
  CKA_IBM_OPAQUE_PKEY 154
  CKA_IBM_PROTKEY_EXTRACTABLE 154
  CKA_IBM_PROTKEY_NEVER_EXTRACTABLE 154
  CKA_IBM_STD_COMPLIANCE1 154
  CKA_IBM_USE_AS_DATA 154
  openCryptoki 163
available libraries in openCryptoki 21

B

Baseline Provider 88
bit coin curve
  secp256k1 117
bitcoin derivation
  CKM_IBM_BTC_DERIVE 152

C

C API vii, 4
C_CloseSession 170
C_CreateObject 171
C_Decrypt (AES) 173
C_Decrypt (RSA) 173
C_Encrypt (AES) 173
C_Encrypt (RSA) 173
C_FindObjects 172
C_GenerateKeyPair (RSA) 173
C_GetMechanismInfo 124
C_GetMechanismList 124
C_IBM_ReencryptSingle 157
C_Initialize sample 170
C_login 171
C_Logout 171
C_OpenSession 170
C_SessionCancel 89
cancel subcommand
  pkcshsm_mk_change 76
CCA library
  usage notes 97
CCA library functions
CCA library functions *(continued)*
  restrictions 97
CCA master key migration 98
CCA token
  AES XTS 87
  directory content 131
  migration of RSA format 99
  supported ECC curves 96
  supported PKCS #11 mechanisms 94
CCA token configuration file 81, 82, 91
CCA tokenCCA token configuration file 91
check list
  trouble shooting) 179
CKA_DERIVE_TEMPLATE 89
CKA_IBM_ATTRBOUND 154
CKA_IBM_OPAQUE 154
CKA_IBM_OPAQUE_PKEY 154
CKA_IBM_PROTKEY_EXTRACTABLE 154
CKA_IBM_PROTKEY_NEVER_EXTRACTABLE 154
CKA_IBM_STD_COMPLIANCE1 154
CKA_IBM_USE_AS_DATA 154
CKM_IBM_ATTRIBUTEBOUND_WRAP 149
CKM_IBM_BTC_DERIVE
  bitcoin derivation 152
CKM_IBM_CMAC 148
CKM_IBM_DILITHIUM 139
CKM_IBM_ECDSA_OTHER
  Schnorr signatures 153
CKM_IBM_KYBER 143
CKM_IBM_SHA3_nnn 146
CKM_IBM_SHA3_nnn_HMAC 147
clear keys 85
client side setup 185
code sample
  base procedures 169
  cryptographic operations 173
  dynamic library calls 168
  object handling 171
  session and log-in procedures 170
  statically linked library 168
collecting statistics 23
command pkcsconf 21
common token information 81
components of openCryptoki 5
concurrent HSM master key change
  pkcshsm_mk_change 71
configuration file
  ep11tok01.conf 81
configuring
  CCA token 91
  EP11 token 81
  extended evaluations 124
  multiple EP11 tokens 81
configuring applications 183
configuring extended evaluations 124
CPFILTER 107
Crypto Express adapter 5

crypto stack with Linux on IBM Z and IBM LinuxONE 13
cryptographic operations 173
cryptographic policies 25, 28
cryptographic token 3
cryptography
    asymmetric 3
    public key 3
Cryptoki 3
Cryptoki API 3

D

DEB 19
Dilithium key mechanism 139
directory content
    CCA token 131
    EP11 token 131
    ICA token 131
    Soft token 131
disable-event-support 23
domain control point
    access control point 107
dual-function cryptographic functions 88
dynamic library call 168

E

ec_curves.h 84
ECC
    ec_curves.h 84
    header file 84
ECDH 148
EdDSA 148
Edwards Curves
    25519 148
    448 148
effective key 85
elliptic curve cryptography
    EP11 token 117
elliptic curves
    ec_curves.h 84
    header file 84
environment variables 35
EP11 host library
    restrictions 123
EP11 session
    definition 121
    managing 121
EP11 session tool 121
EP11 token
    AES XTS 87
    bit coin curve 117
    configuring 81
    directory content 131
    elliptic curve cryptography 117
    master key change, concurrent 71
    secp256k1 117
    supported PKCS #11 mechanisms 113
EP11 token configuration file
    APQN_ALLOWLIST 107
    CPFILTER 107
    OPTIMIZE_SINGLE_PART_OPERATIONS 107
    sample 107
EP11 token configuration file *(continued)*
    STRICT_MODE 107
    VHSM_MODE 107
ep11info tool 73
ep11tok01.conf 81
ep11tok01.conf configuration file 81
event notification 23
extended evaluations
    configuring 124

F

fastpath openCryptoki 17
features
    common for openCryptoki 1
finalize subcommand
    pkcshsm_mk_change 75
FIPS compliance
    pkcstok_migrate 65
function specific parameter 9

G

general structure
    openCryptoki application 163

H

hardware security module
    HSM 3
hardware security module (HSM) 123
header file
    ec_curves.h 84
    elliptic curves 84
HSM
    hardware security module 3

I

IBM-specific mechanisms and features
    common for openCryptoki 137, 139
    complete list 139
ICA token
    directory content 131
    restrictions 104
    status information 82
    supported ECC curves 105
    supported PKCS #11 mechanisms 101
ICSF token 135
initiate subcommand
    pkcshsm_mk_change 73
installing openCryptoki 19

K

keyword
    disable-event-support 23
    statistics 23
Kyber key mechanism 143

L

libopencryptoki.so 21
Linux on IBM Z and IBM LinuxONE crypto stack 13
list subcommand
    pkcshsm_mk_change 75
lock files 13
log-in PIN 40
logging with openCryptoki 12

M

managing EP11 session 121
managing token keys
    p11sak 43
managing tokens
    pkcsconf 39
master key (MK)
    migration process 119
master key (MK) migration tool
    installing, configuring, using 119
    pkcsep11_migrate 119
master key change
    concurrent 71
    concurrently with pkcshsm_mk_change 71
    EP11 token 71
master key migration
    pkcscca utility 98
mechanism parameter 9
migrating master keys 119
migrating to FIPS compliance
    pkcstok_migrate 65
migration
    CCA token master keys 98
    master key 98
migration of RSA format
    RSA-AESC 99
    RSA-CRT 99
migration tool
    for master (wrapping) keys 119
multiple CCA tokens
    configuring 91
multiple EP11 tokens
    configuring 81

N

NVTOK.DAT 131

O

object handling 171
objects
    openCryptoki
        create 162
        modify 162
OID file
    post-quantum algorithms 84
    pqc_oids.h 84
openCryptoki
    application programmers vii
    architecture 5
    attributes 163
openCryptoki *(continued)*
    base library 21
    C_Decrypt (AES) 173
    C_Decrypt (RSA) 173
    C_Encrypt (AES) 173
    C_Encrypt (RSA) 173
    C_GenerateKeyPair (RSA) 173
    CCA token 91
    coding samples 167
    common IBM-specific mechanisms and features 137
    common token information 81
    components 5
    configuration file 21
    definition) 4
    ECC curves for the ICA token 105
    ECC curves of the CCA token 96
    environment variables 35
    fastpath 17
    IBM-specific mechanisms and features 139
    ICA token 101
    installing
        from DEB 19
        from RPM 19
    installing from source package 19
    logging and tracing 12
    mechanisms for the CCA token 94
    mechanisms for the EP11 token 113
    mechanisms for the ICA token 101
    mechanisms for the Soft token 125
    objects create 162
    objects modify 162
    preparing 15
    programming 161
    programming basics 159
    roles 9
    sessions 9
    shared library (C API) 4
    slot 6
    SO PIN 40
    standard PIN 40
    status information 82
    terminology 161
    token libraries 21
    token-specific configurations 79
    trouble shooting 179
    user scenarios 159
    users vii
openCryptoki application
    general structure 163
    sample 165
    template 163
openCryptoki configuration file
    sample for opencryptoki.conf 21
openCryptoki features 1
openCryptoki lock files 13
openCryptoki main API
    PKCS11_API.so 8
openCryptoki token repository 43
openCryptoki tools 37
opencryptoki.conf
    openCryptoki configuration file 21
OPTIMIZE_SINGLE_PART_OPERATIONS 107

P

p11-kit
  support of IBM-specific mechanisms 187
p11sak tool 43
PIN 40
PKCS #11
  Baseline Provider 88
PKCS #11 3.0
  C_SessionCancel 89
  Supported features 89
PKCS #11 3.1
  CKA_DERIVE_TEMPLATE 89
PKCS #11 and openCryptoki 1
pkcs11 group 7, 12
PKCS11_API.so
  main API 8
pkcscca
  RSA format migration 99
  RSA-AESC 99
  RSA-CRT 99
pkcscca tool
  master key migration 98
pkcsconf 39, 40
pkcsconf -t 82
pkcsconf -t command 21
pkcsconf command 21
pkcsep11_migrate 119
pkcsep11_session 121
pkcsficsf tool 135
pkcshsm_mk_change
  cancel subcommand 76
  finalize subcommand 75
  initiate subcommand 73
  list subcommand 75
  reencipher subcommand 73
pkcsslotd
  slot manager daemon 7
  starting 8
pkcsslotd user 12
pkcsstats 67
pkcstok_migrate 65
policy configuration 25
policy configuration file
  policy.conf 28
  processing 32
policy.conf 28
POSIX shared memory segments 7
post-quantum algorithms
  pqc_oids.h 84
pqc_oids.h 84
preparing openCryptoki 15
processing policy configuration files 32
processing strength configuration files 32
programming basics 159
protected keys 85
Public-Key Cryptography Standards 3

R

reencipher subcommand
  pkcshsm_mk_change 73
referenced literature 191
restrictions
restrictions *(continued)*
  ICA token 104
restrictions of EP11 host library 123
roles 9
RPM 19
RSA decrypt operation errors 161
RSA format migration
  RSA-AESC 99
  RSA-CRT 99
RSA-AESC 99
RSA-CRT 99

S

sample code
  object handling 171
Schnorr signatures
  CKM_IBM_ECDSA_OTHER 153
secp256k1
  bit coin curve 117
secure key concept 97, 123
security officer (SO)
  log-in PIN 40
server side setup 183
session and log-in procedures 170
session functions 9
sessions 9
shared memory segments
  POSIX 7
  System V 7
slot
  management functions 6
slot entry 21
slot entry, defining 81
slot manager
  starting 8
slot manager daemon
  pkcsslotd 7
slot token dynamic link libraries
  STDLLs 5, 10
SO
  log-in PIN 40
Soft token
  directory content 131
  supported PKCS #11 mechanisms 125
source package 19
standard user (User)
  log-in PIN 40
starting
  pkcsslotd 8
  slot manager 8
statically linked library 168
statistics 23
statistics collection 23
status information 82
STDLL
  slot token dynamic link library 5, 10
strength configuration 25
strength configuration file
  processing 32
  strength.conf 25
strength.conf 25
STRICT_MODE 107
summary of changes

summary of changes *(continued)*
edition SC34-7730-01 x
edition SC34-7730-02 ix
version 3.17 x
versions 3.18 - 3.21 ix
support of IBM-specific mechanisms
p11-kit 187
System V shared memory segments 7

T

template
openCryptoki application 163
token
initializing 40
management functions 6
token descriptions 79
token information 82
token information, common for all tokens 81
token libraries 21
token recognizing 82
token status information 82
token-specific configurations 79
tools
common for openCryptoki 37
managing token keys 43
p11sak 43
pkcscca 98
pkcsconf 39
pkcsep11_migrate 119
pkcsep11_session 121
pkcsficsf 135
pkcshsm_mk_change 71
pkcsstats 67
pkcstok_migrate 65
usage statistics 67
tracing with openCryptoki 12
trouble shooting 179

U

Ubuntu 21.04 183
Unified Resource Identifier
URI 39
URI
Unified Resource Identifier 39
usage notes
ICA token 104
usage notes for CCA library functions 97
usage statistics
pkcsstats 67
User
log-in PIN 40
normal user 9
user scenario
client side setup 185
server side setup 183
user scenarios 159
utilities
pkcsep11_migrate 119
pkcsep11_session 121
pkcstok_migrate 65
utility
utility *(continued)*
managing tokens 39
p11sak 43
pkcscca 98
pkcsconf 39
pkcshsm_mk_change 71
pkcsstats 67

V

VHSM_MODE 107
VHSM-PIN 121

W

wrapping key 119

X

XCP_CPB_BTC access control point 152

SC34-7730-02