

# CKMULTISLOT

An alternate PKCS#11 library  
for Thales HSM devices.

Reference



Intrinsic is a registered trademark of Intrinsic Srl, a holding company located at Viale Mentana, 29 – 43121 Parma Italy. Intrinsic is a Symbolic company.

Thales is a registered trademark of Thales, a holding company located at 45, rue de Villiers - 92200 Neuilly-sur-Seine Cedex - France.

All other trademarks, service marks, registered trademarks, or registered service marks are the property of their respective owners.

Intrinsic assumes no responsibility for any inaccuracies in this document. Intrinsic reserves the right to change, modify, transfer, or otherwise revise this publication without notice.



## CKMULTISLOT REFERENCE MANUAL

### Introduction

CKMULTISLOT is an alternate PKCS#11 library working with Thales (nCipher line) HSM devices. It's specifically designed to support million of RSA keys by using encrypted blobs stored into an external filesystem or database.

The original PKCS#11 implementation by Thales is indeed suitable for a large number of applications, however when a company needs to handle million of keys, each one protected with a different PIN, a number of problems arise which prevent the system to work properly.

Careful resources allocation and external database storage are the two main factors on which the CKMULTISLOT hinges to allow the management of a scalable number of slots/keys which aren't possible with the original PKCS#11 implementation.

Since CKMULTISLOT is a software library meant to work with a driving application, it is assumed that the reader is a programmer, familiar with C/C++ compilers, cryptography, hardware security modules and the PKCS#11 interface in particular.

Documentation of the PKCS#11 standard API is available on the RSA web site.

### PKCS#11 models

PKCS#11 is an industry standard API designed by RSA (the Security Division of EMC) for operating cryptographic token devices. The standard is quite open and describes an abstract model that implements a wide number of cryptographic services like (encryption, signatures, MACs, digests, etc.). It supports both symmetric and asymmetric schemes along with some other more complex operations like - for example - managing crypto material of an SSL session.

Vendors are not required their device to support all the algorithms (with PKCS#11 calls mechanisms) included in the standard. The API is specifically designed to be flexible and offers a rich interface to allow the application to query for the supported functionalities. If the application requests an operation that a particular physical device is unable to support, proper error codes are returned to the caller informing that such a function is not supported.

In practice every device supports only a subset of all mechanisms depending on the hardware capabilities and on the manufacturer goals.

For example an HSM might work with only one RSA keypair which uses as a root key for a CA. Other devices might offer a richer portfolio of algorithms, including symmetric cryptography and elliptic curves signatures. Both have their own PKCS#11 implementation which allow an external application to interface and operate the HSM.

Fig.1 shows the abstract token supported by a the (generic) PKCS#11 standard. The cryptodevice is partitioned into n slots, each one can have a token plugged in or be empty. In the picture the device has 8 slots of which only #0, #3 and #6 have a token inserted.

Tokens are actually logical containers holding keys (or objects in general) and capable of making computations.

The usual behavior is that keys never leave the token; when a cryptographic operation is needed with a particular key, the application accesses that token by using the proper slot index. If the token is plugged into the slot, the operation requested with a particular key is performed and only the result are sent back to the caller.

If needed the token can be protected with a PIN, so the user have to log-in to the token before being able to access the private part of the token.

Since the model is abstract the number of slots/tokens as long as what “token is plugged into the slot” actually means depends on the characteristics of the physical device.

More over, a single physical device can have multiple layouts depending on how the PKCS#11 is configured, and in our case also from which PKCS#11 API is used.

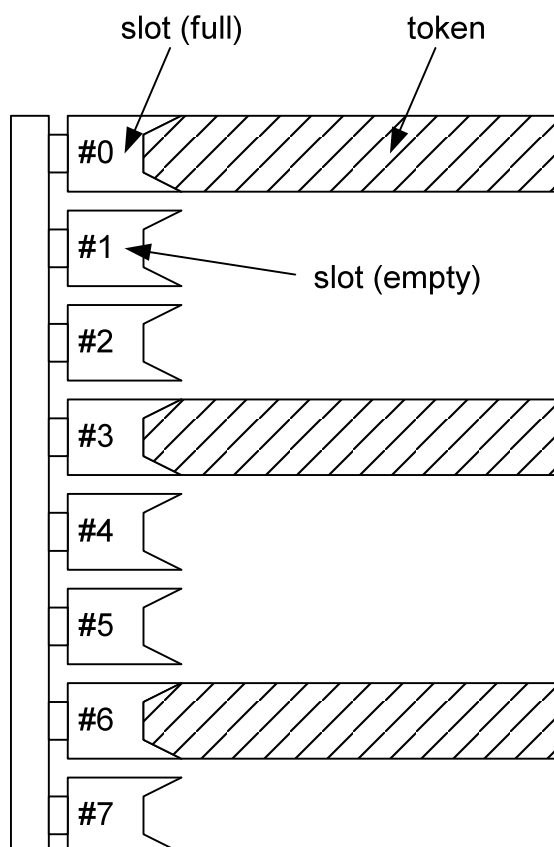


fig.1

It is important to notice that by using an abstract model, the API isn't bound to a particular physical device and so applications can query and use the interface without even knowing which hardware is really attached to the system. Clearly a well written application is expected query the library and check if the needed functionalities are supported.

This greatly helps developers because the hardware becomes transparent to the application and they can concentrate on the functionalities rather than compatibility.

In general however pkcs#11 offers a set of functionalities that only partially overlaps with those offered by a certain physical device.

As said before there might be mechanisms and algorithms which are included into pkcs#11 standard, but are not available on the hardware. On the other hand, complex devices like Thales' nShields and netHSMes can perform a lot more operations than those supported by pkcs#11.

The intersection of these two sets is generally further narrowed considering that a specific implementation of PKCS#11 might support only some of the functions that are both included in the standard and available on the device.

The result is that the device can perform through pkcs#11 only a subset of the intersection of the two sets (see fig.2).

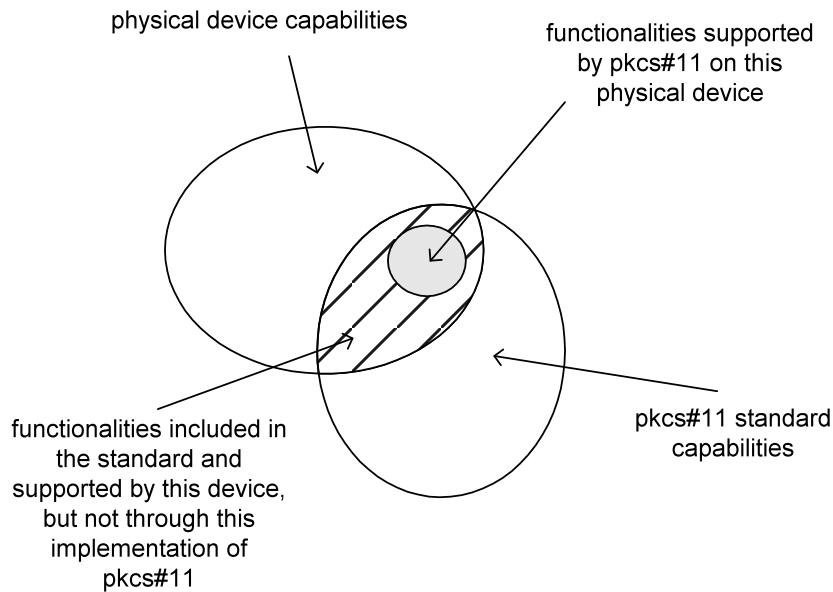


fig.2

More generally a vista of the physical token is mapped onto the logical crypto device and this can cause issues if there isn't a straight and obvious way to perform such mapping. For example Thales devices use an abstract framework called security world which allows – among other things - the creation of keys protected by either the module, an operator card set and a softcard. In this scenario there does not exist a simple way to partition the HSM into logical slots/tokens and have its keys distributed within them. Manufacturers have to make choices when they implement a PKCS#11 vista of the token.

### Thales' PKCS#11 vista of the Security World

Thales's original pkcs#11 library allows two different vistas of the security world, depending on how the environment variable CKNFAST\_LOADSHARING is set. Notice that by being an conditioned by an environment variable, this mapping is done per process/shell, not per client or even globally to the security world to which many HSMes might belong. In other words, each application can have its own mapping, which is done by the local instance of pkcs#11 library, to which the application is statically or dynamically linked. The hardserver (the daemon routing commands to the devices) has nothing to do with such mapping.

### Load sharing disabled

When loadsharing is disabled, the library checks for every physical device attached to the local hardserver and creates a logical token whose slot number depends on the number of devices detected. For simplicity, assuming n devices are found and each device has only one card reader attached the layout is the following:

Slot 0	Module 1
Slot 1	Card reader of module 1
Slot 2	Module 2
Slot 3	Card reader of module 2



...	...
Slot 2n-2	Module n
Slot 2n-1	Card reader of module n

Slots with odd number are related to card readers and can be used with OCS with quorum 1/N.

Slots with even number are essentially all equivalent (all have the same keys) except that each one refers to a different hardware module. As the title suggests, loadsharing is disabled and the application by explicitly choosing a slot number also decides which module to use. If a particular module fails, the slot associated becomes unusable, so no automatic failover is available. It's up to the application to choose and use another slot.

This mapping has however the advantage that all modules and card readers are independent so if physical devices are far away one another, an application aware of this can help the user by operating the card reader closest to him.

## Load sharing enabled

If loadsharing is enabled, the library enumerates all operator cardset and softcards regardless of the number of modules attached. The slot list of the logical token are build in the following manner:

Slot 0	Loadshared accelerator
Slot 1	OCS#1
Slot 2	OCS#2
Slot 3	Softcard#1
Slot 4	Softcard#2
...	...

The order on which OCSes and Softcards are listed is alphabetical respect to the token hash, which essentially means that it's a random shuffling. Each time a new OCS or Softcard is created, its position in the list should be assumed to be random with the only exception all OCSes precede all Softcards. Applications should therefore find the proper slot to work on by checking the slot label and not by assuming a particular index.

Softcards can be used with original pkcs#11 implementation only by enabling loadsharing; once it's done, the functionality is pretty straight: the pkcs#11 login functionality requires the softcard passphrase to succeed.

OCSes on the contrary become a little more difficult to manage because the system expects an operator card to be present in each reader and as before, only a quorum of 1/N is supported.

Load balancing and failover of the resources can be managed transparently by having the library (and not the application) choosing to which module send the requests, since slots and modules are decoupled.

Using multiple keys in this scenario seems easy. Just create the number of softcards needed and generate a key in each softcard. Unfortunately the system won't scale too well for large number of softcards. Due to the inner workings of the HSM and the security world, for best performance the library loads all the keyblobs into host memory and fetches them into the device once the user logs into a slot. This normally has a light impact on the application and boosts performance. However when the number of softcards (and therefore keys) is in the order of ten thousands, the memory consumption on the host becomes an issue. Practical problems tend to arise around 50000 keys, where the library kicks the host of out memory by consuming more than 2Gb.



Some optimization are possible (for example using 64-bit system might allow us to go beyond the 4Gb barrier), but it's clear that it won't scale much higher.

Also, it takes a lot of time for the library to start, when upon initialization it loads everything up into the host.

For a really scalable solution, a customer pkcs#11 library is needed.

### **CKMULTISLOT library**

The CKMULTISLOT library has been designed after this goal: allowing millions of keys to be protected by a Thales HSM and used by authorized applications. To accomplish this a different vista on the security world has been developed.

First, the library won't use the kmdata/local folder to store the key material but uses a different folder organized in a more suitable tree structure or even a database for a huge number of keys.

The number of slots does not depend on OCS or modules number, but is stored in a configuration file. A support application is delivered along the library which takes care of handling all the extra pkcs#11 processes.

The typical usage scenario is the following:

- Administrator sets the initial parameters such as the number of slots, the storage type (local filesystem or external database), by editing a configuration file.
- Administrator uses the supplied tool to initialize the storage which involves storage allocation and possibly keypair generation.
- The pkcs#11 aware application is then run and slots/token/keypairs are ready to be used.

Everytime the application issues a C\_OpenSession on a slot, the library fetches proper encrypted data from the external storage (filesystem or database) to the host. After a successful C\_Login, data is decrypted inside the HSM and keys become available to the logged-in session.

The library behaves differently respect to the pkcs#11 since sessions do not share the log-in status, contrary to what standard says. This has been done to ease the writing of applications which typically spawn many threads, each one running independently a signature/encryption session.



Intrinsic  
Viale Mentana, 29  
I-43121 Parma  
T. 0521-708866  
F. 0521-776190  
[info@intrinsic.it](mailto:info@intrinsic.it)