# CryptoServer

PKCS#11 R2

Developer Guide

utimaco®

## Imprint

|  |  |
|---|---|
| Copyright 2018 | Utimaco IS GmbH<br>Germanusstr. 4<br>D-52080 Aachen<br>Germany |
| Phone | +49 (0)241 / 1696-200 |
| Fax | +49 (0)241 / 1696-199 |
| Internet | http://hsm.utimaco.com |
| E-mail | hsm@utimaco.com |
| Document Version | 1.2.13 |
| Date | 2018-06-29 |
| Status | Final |
| Document No. | 2012-0007 |
| All Rights reserved | No part of this documentation may be reproduced in any form (printing, photocopy or according to any other process) without the written approval of Utimaco IS GmbH or be processed, reproduced or distributed using electronic systems.<br>Utimaco IS GmbH reserves the right to modify or amend the documentation at any time without prior notice. Utimaco IS GmbH assumes no liability for typographical errors and damages incurred due to them.<br>All trademarks and registered trademarks are the property of their respective owners. |

# Table of Contents

# 1 Introduction

Thank you for purchasing our CryptoServer security system. We hope you are satisfied with our product. Please do not hesitate to contact us if you have any complaints or comments.

## 1.1 About this Document

This document describes the cryptographic token interface PKCS#11 Release 2, as provided by Utimaco's hardware security module CryptoServer with SecurityServer version 3.00 or higher.

### 1.1.1 Target Audience for This Manual

This guide is intended to assist software developers by creating their own PKCS#11 application with the CryptoServer PKCS#11 library.

### 1.1.2 Contents of This Manual

After the introduction this manual is divided up as follows:

*Chapter 2* provides an overview about the CryptoServer's PKCS#11 interface.

*Chapter 3* contains the necessary prerequisites for the usage of CryptoServer's PKCS#11 interface.

*Chapter 4* describes the configuration options of the CryptoServer PKCS#11 library within the configuration file `cs_pkcs11_R2.cfg.`

*Chapter 5* explains the operating modes of the CryptoServer PKCS#11 library.

*Chapter 6* briefly explains the difference between an internal and external key storage for PKCS#11 objects (keys).

*Chapter 7* provides information about the header files and libraries required for the development of a PKCS#11 application.

*Chapter 8* contains details about Utimaco's PKCS#11 implementation for the CryptoServer.

*Chapter 9* describes the authentication concept and user management of the CryptoServer PKCS#11 R2 library.

*Chapter 10* contains an overview of the functions that can be executed by the users with the role Key Manager (KM) or Key User (KU).

*Chapter 11* contains details about the vendor defined PKCS#11 extensions.

*Chapter 12* is an overview of the mechanisms currently supported by the CryptoServer PKCS#11 library - both defined by the PKCS#11 standard and vendor defined - and the functions supporting these mechanisms.

*Chapter 13* provides information about important restrictions applying to a CryptoServer operating in FIPS mode and lists the mechanisms supported by the CryptoServer in FIPS mode.

### 1.1.3 Document Conventions

We use the following conventions in this manual:

| *Convention* | *Usage* | *Example* |
|---|---|---|
| **Bold** | Items of the Graphical User Interface (GUI), e.g., menu options | Press the **OK** button. |
| `Monospaced` | File names, folder and directory names, commands, file outputs, programming code samples | You will find the file `example.conf` in the `/exmp/demo/` directory. |
| *Italic* | References and important terms | See Chapter 3, "Sample Chapter" in the *CryptoServer LAN/CryptoServer CryptoServer Command-line Administration Tool -csadm -Manual for System Administrators or* [CSADMIN]. |

Table 1: Document conventions

We have used icons to highlight the most important notes and information.

*Here you find important safety information that should be followed.*

*Here you find additional notes or supplementary information.*

## 1.2 Recommended Reading

We highly recommend to read also the document *Learning PKCS#11 in Half a Day* (`PKCS11_HandsOn.pdf`) provided on the SecurityServer product CD in the same folder as this manual: `\Documentation\Crypto_APIs\PKCS11_R2`. There you can learn how to develop PKCS#11 applications.

# 2 The PKCS#11 R2 Interface - Overview

PKCS#11 is a general purpose Public Key Cryptography Standard originally developed by RSA Security [PKCS11] and currently maintained by the OASIS PKCS11 Technical Committee. It defines an interface between an application and a cryptographic device.

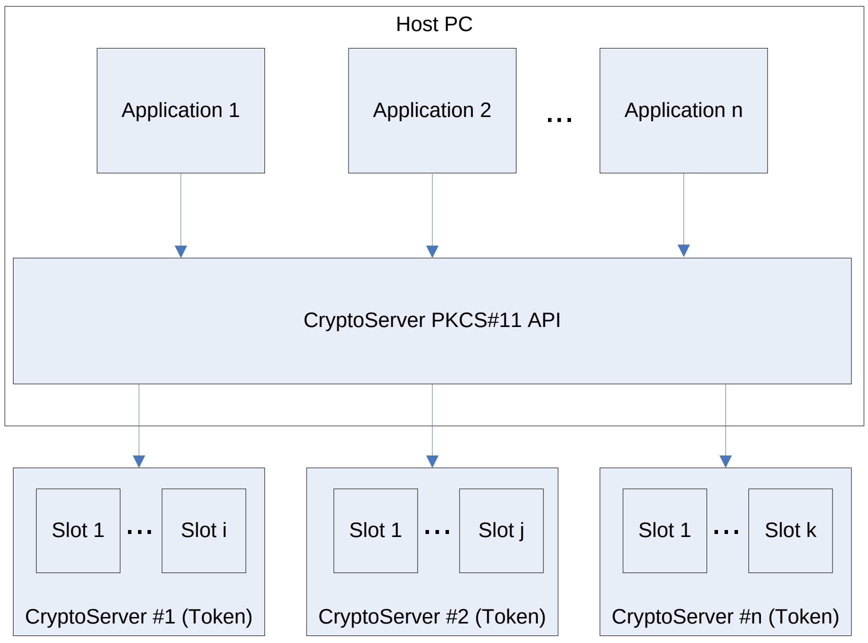
The CryptoServer provides a PKCS#11 interface. To use this interface a dedicated firmware package including the CXI firmware module must be loaded into the CryptoServer. Additionally, it is required that the PKCS#11 application is linked against the CryptoServer PKCS#11 library or it must be able to load the specific shared library (`DLL/so`). With this concept no specific drivers are needed for the application to access the CryptoServer directly.

The CryptoServer PKCS#11 library supports more than one CryptoServer device for each application.

The CryptoServer can handle up to 256 PKCS#11 sessions or applications per CryptoServer in parallel.

The CryptoServer PKCS#11 library provides an interface between the host application and one or more CryptoServer. It communicates directly with the configured CryptoServer or cluster of CryptoServer. Each CryptoServer can be a local PCI/PCIe plug-in card or a network appliance CryptoServer LAN.

Several applications on one host PC can access the CryptoServer PKCS#11 library in parallel. The CryptoServer PKCS#11 library provides also a mechanism to configure several CryptoServer. They can be accessed with the same library. Each PKCS#11 slot from every configured CryptoServer can be addressed directly. The CryptoServer PKCS#11 library maps all existing slots to a unique slot ID.



# 3 Requirements

To be able to use the CryptoServer PKCS#11 interface make sure that the following prerequisites are fulfilled.

- One or more CryptoServer have been installed – local (CryptoServer PCIe card) or remote (CryptoServer LAN).
- If a CryptoServer PCIe card is used - the CryptoServer driver has been installed as described in the corresponding CryptoServer Operating Manual.
- A valid Master Backup Key has been imported into the CryptoServer.
- A firmware package which fulfills the minimum requirements has been loaded into the CryptoServer (see Chapter 3.1, "Required Firmware Package").
- The application using the CryptoServer PKCS#11 interface can load the CryptoServer PKCS#11 shared library, or it is linked against a static version of this library.
- The configuration file `cs_pkcs11_R2.cfg` is configured correctly (see Chapter 4, "Configuration") and the CryptoServer PKCS#11 library is able to find and access it (see Chapter 3.2, "Location of the Configuration File cs_pkcs11_R2.cfg").

For general administration of the CryptoServer you can use one of the CryptoServer's administration tools: CAT (Java graphical user interface) or csadm (command line tool). See [CSMSADM] or [CSADMIN] for help. Additionally, Utimaco provides dedicated tools for the administration of the PKCS#11 interface on the Security Server product CD resp. CryptoServer SDK product CD in the folder `…\Utimaco\CryptoServer\Administration`: p11CAT – a Java graphical user interface, and a command line tool p11tool2.

## 3.1 Required Firmware Package

All firmware modules are contained in the CryptoServer firmware package `SecurityServer-<Type>-<Version>.mpkg`, where `<Type>` describes the type of the CryptoServer-Series (`CS-Series`, `CSe-Series, Se2-Series` or `Se-Series`) and `<Version>` describes the version of the package. To load the corresponding firmware package into the CryptoServer, see [CSMSADM] for details.

A special firmware package is required to ensure that the PKCS#11 interface works properly: Please use the firmware package provided on the SecurityServer product CD version 3.00 or higher.

## 3.2 Location of the Configuration File cs_pkcs11_R2.cfg

After creation of the configuration file, the CryptoServer PKCS#11 library should be able to locate and load it. There are several possibilities to tell the CryptoServer PKCS#11 library where the configuration file is located.

- Set the `CS_PKCS11_R2_CFG` environment variable to the correct path and location of the configuration file.

  Example for a Windows system:

```
#> set CS_PKCS11_R2_CFG="C:\My Documents\utimaco\cs_pkcs11_R2.cfg"
```

  Example for a Linux system:

```
#> CS_PKCS11_R2_CFG=~/.utimaco/cs_pkcs11_R2.cfg
#> export CS_PKCS11_R2_CFG
```

- Place the configuration file in the current working directory (useful for development).
- Place the configuration file in the same directory where the application is located (only Windows and Linux).
- The configuration file in a system specific directory (see below).

The CryptoServer PKCS#11 library looks for the `cs_pkcs11_R2.cfg` configuration file in the following order:

### On Windows operating systems:

1. First it checks if the `CS_PKCS11_R2_CFG` environment variable is set and if it contains the name and location of the configuration file.

*The `CS_PKCS11_R2_CFG` environment variable is set, by default, to the correct name and location of the configuration file during the installation of the product CD:*
*For the SecurityServer product CD:*
*- As of version 4.20: C:\ProgramData\cs_pkcs11_R2.cfg.*
*- Earlier than version 4.20: C:\Program Files\Utimaco\CryptoServer\Lib\cs_pkcs11_R2.cfg.*
*For the CryptoServer SDK product CD:*
*C:\Utimaco\CryptoServer\Lib\cs_pkcs11_R2.cfg*

2. If not, it checks if the configuration file is located in the user's home directory (`%USERPROFILE%`).
3. If not, it checks if the configuration file is located in the current working directory.
4. If not, it checks if the configuration file is located in the same directory where the application is located.

5. If not, it checks if the configuration file is located somewhere in `%PATH%`.
6. If not, it checks if the configuration file is located in the WINDOWS directory (e.g. `c:\WINDOWS`).

### On Linux operating systems:

1. First it will be checked if the `CS_PKCS11_R2_CFG` environment variable is set, and if it contains the name and location of the configuration file.
2. If not, it will be checked if the configuration file is located in the user's home directory (`~/.utimaco/cs_pkcs11_R2.cfg`).
3. If not it will be checked if the configuration file is located in the current working directory.
4. If not, it will be checked if the file is located in the same directory where the executable is located.
5. If not it, the following path names are checked for the configuration file in the following order:

```
/usr/local/etc/utimaco
/usr/local/etc
/etc/utimaco
/etc
```

### Search order under other Unix systems:

1. First it checks if the `CS_PKCS11_R2_CFG` environment variable is set and if it contains the name and location of the configuration file.
2. If not, it checks if the configuration file is located in the user's home directory (`~/.utimaco/cs_pkcs11_R2.cfg`).
3. If not, it checks whether the configuration file is located in the current working directory.
4. If not, the following path names are checked for the configuration file in the given order:

```
/usr/local/etc/utimaco
/usr/local/etc
/etc/utimaco
/etc
```

# 4 Configuration

The configuration of the CryptoServer PKCS#11 library is done within the `cs_pkcs11_R2.cfg` configuration file. For details about the location of the configuration file, see Chapter 3.2, "Location of the Configuration File cs_pkcs11_R2.cfg". This file can contain several sections:

- `[Global]-` section for general configuration
- `[CryptoServer]` - section for each CryptoServer device that should be addressed by the application
- `[Slot]` – (optional) section for every slot that is in use.

The following table gives an overview of all parameter that can be configured in the `cs_pkcs11_R2.cfg` configuration file.

| *Parameter* | *Description* |
|---|---|
| `Logging` | Specifies the log level 0, 1, 2, 3 or 4 (see chapter 4.2 for details). |
| `Logpath` | Specifies the path where the logfile shall be created. In case of /tmp directory e.g., where the sticky bit is set, file deletion and therefore log rotation is not possible. The logfile might grow above the limit given by the “Logsize” parameter. |
| `Logsize` | Defines the maximum size of the logfile. If the maximum is reached,a log rotation is performed overwriting a previously backed up logfile. Can be defined as value in bytes or as formatted text. E.g. value of ‘1000’ means logsize is 1000 bytes whereas value of ‘1000kb’ means 1000 kilobytes. Allowed formats are 'kb', 'mb' and 'gb'. |
| `KeysExternal` | Specifies the default behavior for object creation and generation. If “true”, new created or generated objects will be stored in an external key storage, i.e., a key database outside the CryptoServer. |
| `KeyStore` | Specifies the path to the external key storage (e.g. `C:/utimaco/P11.sdb`). This parameter must be set if `KeysExternal = true`. |
| `SlotMultiSession` | Specifies session connection behavior. If “true”, every session establishes its own connection to the CryptoServer. |
| `SlotCount` | Maximum number of slots that can be used. Per default the CryptoServer has 10 configured slots available. To avoid that the application scans all configured slots the maximum number can be reduced with the configuration item. |

| *Parameter* | *Description* |
|---|---|
| `KeepLeadZeros` | Defines if leading zeros of a decryption operation will be kept (`true` or `false`) |
| `FallbackInterval` | Configures load balancing mode (`FallbackInterval = 0`, default) or failover mode (`FallbackInterval > 0`) |
| `KeepAlive` | Keep sessions alive and prevent them from expiring after 15 minutes idle time (`true` or `false`) |
| `Device` | Device address to connect a CryptoServer device (see chapter 4.1) |
| `ConnectionTimeout` | Specifies the maximum time in milliseconds to wait before the connection establishment is aborted if the device is not responding. |
| `CommandTimeout` | Specifies the maximum time in milliseconds to wait for the answer from CryptoServer after sending a command. |
| `SlotNumber` | Number of the slot to be configured |
| `<username>` | Defines the authentication mechanism of the user with name <username> (see Chapter 11.2, “Authentication via Configuration File”) |
| `ExtendedLoginSO` | Activates extended login for the SO |
| `ExtendedLoginUSER` | Activates extended login for the USER |
| `CustomMechanisms` | List of official PKCS#11 mechanisms which should be customized |

Table 2: Configuration parameter in the cs_pkcs11_R2.cfg configuration file

Some parameters can be defined in more than one section. For example, the parameter `KeysExternal` is evaluated in the `[Global]` section, the `[CryptoServer]` section and the `[Slot]` section. Thereby the parameter in the section with the highest priority is evaluated first. If the parameter is not set, the parameter in the next lower section is evaluated. If the parameter is not set somewhere, the default value is used. The evaluation starts with the `Slot` section (highest priority) and ends with the default values (lowest priority):

- `[Slot]`
- `[CryptoServer]`
- `[Global]`
- Default

The following table shows the default values for all parameters and the section where they can be defined.

| *Parameter* | *Allowed Sections* | *Default Value* |
|---|---|---|

| *Parameter* | *Allowed Sections* | *Default Value* |
|---|---|---|
| `Logging` | [Global] | 0 (NONE) |
| `Logpath` | [Global] | No default value (no logfile is created per default). |
| `Logsize` | [Global] | 1 000 000 |
| `KeysExternal` | [Global], [CryptoServer], [Slot] | false |
| `KeyStore` | [Global] | No default path (no external key storage is created per default) |
| `SlotMultiSession` | [Global], [CryptoServer], [Slot] | true |
| `SlotCount` | [Global], [CryptoServer] | 10 |
| `KeepLeadZeros` | [Global] | false |
| `FallbackInterval` | [Global] | 0 |
| `KeepAlive` | [Global], [CryptoServer], [Slot] | false |
| `Device` | [CryptoServer] | No default value (value must be defined) |
| `ConnectionTimeout` | [Global], [CryptoServer] | 5000 |
| `CommandTimeout` | [Global], [CryptoServer] | 60000 |
| `SlotNumber` | [Slot] | No default value (device address must be defined) |
| `<username>` | [Slot] | No default value |
| `ExtendedLoginSO` | [Slot] | No default value |
| `ExtendedLoginUSER` | [Slot] | No default value |
| `CustomMechanisms` | [Global] | No default value |

Table 3: Default settings for the parameters in the cs_pkcs11_R2.cfg configuration file

## 4.1 The Parameter Device

There are several possibilities to address the CryptoServer with the `Device` configuration parameter.

Here are some examples:

| *Address* | *Description* |
|---|---|
| `/dev/cs2.n`<br>where n = {0, 1, 2, ..., 7} | Local CryptoServer No. n+1 on a UNIX system.<br>The maximum number of eight CryptoServer PCIe cards can be changed in the source of the Linux driver. |
| `PCI:n`<br>where n = {0, 1, 2, ..., 31} | Local CryptoServer No. n+1 on a Windows system |
| `TCP:288@194.168.4.107` | IP address and port number of a CryptoServer LAN<br>In commands, always use IP addresses without leading zeros although they are shown in the CryptoServer LAN display, e.g., `194.168.004.107`. |
| `TCP:194.168.4.107` | IP address of a CryptoServer LAN (default: port=288)<br>In commands, always use IP addresses without leading zeros although they are shown in the CryptoServer LAN display, e.g., `194.168.004.107`. |
| `194.168.4.107` | IP address of a CryptoServer LAN (default: protocol=TCP, port=288)<br>In commands, always use IP addresses without leading zeros although they are shown in the CryptoServer LAN display, e.g., `194.168.004.107`. |
| `TCP:288@cslan01` | Host name and port number of a CryptoServer LAN (using DNS request to resolve host name) |
| `TCP:cslan01` | Host name of a CryptoServer LAN (using DNS request to resolve host name, default: port=288) |
| `cslan01` | Host name of a CryptoServer LAN (using DNS request to resolve host name, default: protocol=TCP, port=288) |
| `TCP:3001@127.0.0.1` or<br>`TCP:3001@localhost` | Protocol, IP address and port number of the local CryptoServer simulator for Windows/Linux (SDK). The simulator can be used for test and evaluation purposes; see [CSMSADM] for further details. |
| `3001@127.0.0.1` or<br>`3001@localhost` | IP address and port number of the local CryptoServer Simulator for Windows/Linux (SDK) with the default protocol TCP. The simulator can be used for test and evaluation purposes; see [CSMSADM] for further details. |

Table 4: Examples for setting the Device parameter

Example of the configuration file `cs_pkcs11_R2.cfg`:

```
[Global]
```

```
Logging = 3
Logpath = c:/tmp
Logsize = 10mb

KeysExternal = true
KeyStore = c:/global/P11.sdb

KeepLeadZeros = true
KeepAlive = true

ConnectionTimeout = 7000
CommandTimeout = 70000

# CryptoServer is a CryptoServer LAN
[CryptoServer]
Device    = 192.168.4.137
CommandTimeout = 80000
SlotCount = 3
SlotMultiSession = true

# first slot
[Slot]
SlotNumber = 0
KeysExternal = true
```

## 4.2 Logging

The logging interface of the CryptoServer PKCS#11 library shall be used only when problems have occurred. Depending on the configuration, information like returned error codes, called functions, etc. are logged. By default, the log level (parameter `Logging`) is set to `NONE`.

| *Name* | *Level* | *Description* |
|---|---|---|
| `NONE` | 0 | No logging output will be produced (default). |
| `ERROR` | 1 | Log errors of the CryptoServer PKCS#11 library and CryptoServer firmware modules |
| `WARNING` | 2 | Log errors and warnings of the CryptoServer PKCS#11 library and CryptoServer modules |
| `INFO` | 3 | Log errors and warnings of the CryptoServer PKCS#11 library and CryptoServer firmware modules. Additionally, information of the CryptoServer PKCS#11 library will be logged. |

| | | |
|---|---|---|
| TRACE | 4 | Log errors, warnings and information of the CryptoServer PKCS#11 library and CryptoServer firmware modules. Additionally, trace output like function calls will be logged. |

Table 5: Logging levels

*It is not recommended to raise the logging level higher than* ***`WARNING`*** *on production systems. It slows down the application and it writes permanently into the logfile.*

# 5 Operating Modes

This chapter describes the operating modes of the CryptoServer PKCS#11 library.

The CryptoServer PKCS#11 library can be used in either load balancing or failover mode.

## Preconditions

Before you start configuring and using the CryptoServer PKCS#11 library in either mode make sure that the following preconditions are fulfilled:

*We recommend to read Chapter "Clustering for Load Balancing and Failover" in [CSMSADM] before you start configuring and using the CryptoServer PKCS#11 library in either load balancing or failover mode.*

- You have initialized the same PKCS#11 slot on every CryptoServer that belongs to the CryptoServer cluster.
- You have created the same user Security Officer (SO) on every CryptoServer that belongs to the CryptoServer cluster.
- You have created the same user User on every CryptoServer that belongs to the CryptoServer cluster.

*Make sure that the preconditions mentioned in Chapter "Clustering for Load Balancing and Failover" in [CSMSADM] (provided on the delivered product CD in folder …\Documentation\Administration Guides) are also fulfilled.*

## 5.1 Load Balancing Mode

In load balancing mode multiple CryptoServer devices are linked together to one logical device also known as a cluster. This is done by setting a list of all (physical) CryptoServer devices as `Device` parameter in the `cs_pkcs11_R2.cfg` configuration file, embraced in "{}" brackets, and keeping the default setting `FallbackInterval = 0` unchanged. In this case the devices can be simultaneously used to distribute the connection processing across the clustered devices.

A `cs_pkcs11_R2.cfg` configuration file in load balancing mode could look for example as follows:

```
[Global]
```

```
Logging = 3
Logpath = c:/tmp
Logsize = 10mb

KeysExternal = true
KeyStore = c:/global/P11.sdb

# Configures load balancing mode (== 0) or failover mode (> 0)
FallbackInterval = 0

SlotCount = 5

# logical CryptoServer device consisting of three CryptoServer LAN
# devices
[CryptoServer]
Device    = { 192.168.0.136 192.168.0.137 192.168.0.138}
ConnectionTimeout = 70000
```

The resulting slot IDs of the CryptoServer in the example above are:

| *CryptoServer Device (logical)* | *Slot ID (deviceID slot ID)* |
|---|---|
| `192.168.0.136 192.168.0.137 192.168.0.138` | 0x 0000 0000 |
| | 0x 0000 0001 |
| | 0x 0000 0002 |
| | 0x 0000 0003 |
| | 0x 0000 0004 |

Table 6: Example for Slot IDs in a Load Balancing mode

To access, e.g. the second slot on the logical device use the slot ID 0x00000001. The CryptoServer PKCS#11 library then decides which CryptoServer is used for this connection request, and establishes the connection to the CryptoServer with the least connections.

For example, if the requesting application has one open connection to each of the devices `192.168.4.136` and `192.168.4.138`, the CryptoServer PKCS#11 library will open the next connection to the device `192.168.4.137`.

*In load balancing mode, always use external keys. If want to use internal keys, contact the support department of Utimaco IS GmbH first to clarify the steps to be performed..*

## 5.2 Failover Mode

In failover mode multiple devices are linked together to one logical device also known as a cluster. This is done by setting a list of all (physical) CryptoServer devices as `Device` parameter, embraced in "{}" brackets and setting the `FallbackInterval` parameter to the time interval [s] after which a re-connection attempt to the Primary CryptoServer should be started.

A `cs_pkcs11_R2.cfg` configuration file in failover mode could look for example as follows:

```
[Global]
Logging = 3
Logpath = c:/tmp
Logsize = 10mb

KeysExternal = true
KeyStore = c:/global/P11.sdb

# Configures load balancing mode ( == 0 ) or failover mode ( > 0 )
FallbackInterval = 3600

SlotCount = 3

# logical CryptoServer device consisting of two CryptoServer LAN devices
# 192.168.4.136 and 192.168.4.137
[CryptoServer]
Device    = { 192.168.4.136 192.168.0.137 }
ConnectionTimeout = 70000
```

The resulting slot IDs of the CryptoServer for the example above are:

| *CryptoServer Device (logical)* | *Slot ID (deviceID slot ID)* |
|---|---|
| `192.168.4.136 192.168.0.137` | 0x 0000 0000 |
| | 0x 0000 0001 |
| | 0x 0000 0002 |

Table 7: Example for Slot IDs in Failover mode

To access, e.g. the second slot on the logical device, use the slot ID 0x00000001. In this mode, the CryptoServer PKCS#11 library decides which CryptoServer is used. If the CryptoServer PKCS#11 library detects an error on the current device, it switches to the "next" device in the cluster.

For example, the device `192.168.4.136` has an error, the CryptoServer PKCS#11 library automatically switches to the device `192.168.4.137`.

*Failover mode shall not be used for functions manipulating users or internal keys like `C_InitToken`, `C_InitPIN`, `C_SetPIN`, `C_CreateObject`, `C_CopyObject`, `C_DestroyObject`, `C_SetAttributeValue`, `C_UnwrapKey`, `C_DeriveKey`. Therefore, if you are operating a CryptoServer failover cluster, do not use p11tool2 or PKCS#11 CryptoServer Administration Tool (P11CAT) for key management.*

## 5.3 Initialization of Slot and User PIN in Failover/Load Balancing Mode

The functions `C_InitToken`, `C_InitPIN` and `C_SetPIN` create respectively modify the user accounts for PKCS#11 Security Officer and PKCS#11 User in a CryptoServer. When executing any of these commands on a logical device, the command will only be executed on a single of the physical devices, no matter whether the logical device represents a cluster for failover or a cluster for load balancing. The PKCS#11 user accounts will not be created or modified on any other than this single physical device, causing subsequent PKCS#11 connections to fail if they are not “accidentally” opened on the physical device on which the PKCS#11 user accounts have actually been created resp. modified.

Therefore, for initialization of the slot and the user PIN, the logical CryptoServer device (consisting of N physical devices) shall be disintegrated and replaced by N single devices in the PKCS#11 configuration file.

For the example above we thus have to create a new cluster configuration file:

```
[Global]
Logging = 3
Logpath = c:/tmp
Logsize = 10mb

KeysExternal = true
KeyStore = c:/global/P11.sdb

SlotCount = 3

# first CryptoServer LAN device of the failover cluster
[CryptoServer]
Device    = 192.168.0.136
ConnectionTimeout = 70000

# second CryptoServer LAN device of the failover cluster
[CryptoServer]
Device    = 192.168.0.137
ConnectionTimeout = 70000
```

Now we initialize the devices separately by executing `C_InitToken` on the slot 0x00000000, 0x00000001 and 0x00000002 for the first device (192.168.0.136) and 0x00010000, 0x00010001 and 0x00010002 for the second device (192.168.0.137):

```
CK_UTF8CHAR_PTR pPinSlot0 = "123456";
CK_UTF8CHAR_PTR pPinSlot1 = "1234567";
CK_UTF8CHAR_PTR pPinSlot2 = "12345678";
CK_ULONG    ulPinLenSlot0 = strlen(pPinSlot0);
CK_ULONG    ulPinLenSlot1 = strlen(pPinSlot1);
CK_ULONG    ulPinLenSlot2 = strlen(pPinSlot2);

CK_UTF8CHAR_PTR pLabel = "Testlabel";

C_InitToken(0x00000000, pPinSlot0, ulPinLenSlot0,  pLabel);
C_InitToken(0x00000001, pPinSlot1, ulPinLenSlot1,  pLabel);
C_InitToken(0x00000002, pPinSlot2, ulPinLenSlot2,  pLabel);

C_InitToken(0x00010000, pPinSlot0, ulPinLenSlot0,  pLabel);
C_InitToken(0x00010001, pPinSlot1, ulPinLenSlot1,  pLabel);
C_InitToken(0x00010002, pPinSlot2, ulPinLenSlot2,  pLabel);

C_InitPIN has to be executed device by device and slot by slot in a similar way:

CK_UTF8CHAR_PTR pPin0 = "654321";
CK_UTF8CHAR_PTR pPin1 = "7654321";
CK_UTF8CHAR_PTR pPin2 = "87654321";
CK_ULONG    ulPinLen0 = strlen(pPin0);
CK_ULONG    ulPinLen1 = strlen(pPin1);
CK_ULONG    ulPinLen2 = strlen(pPin2);

C_InitPIN(hSessionSlot00000000, pPin0, ulPinLen0);
C_InitPIN(hSessionSlot00000001, pPin1, ulPinLen1);
C_InitPIN(hSessionSlot00000002, pPin2, ulPinLen2);

C_InitPIN(hSessionSlot00010000, pPin0, ulPinLen0);
C_InitPIN(hSessionSlot00010001, pPin1, ulPinLen1);
C_InitPIN(hSessionSlot00010002, pPin2, ulPinLen2);
```

Slots and PINs are initialized now and can be used by replacing the cluster configuration file by the failover configuration file.

*The PINs of SO and USER for a certain slot must be the same on every failover device. Example: If the PIN of the SO for slot 0 on device PCI:0 is “123456” than the PIN of SO for slot 0 on device 192.168.0.137 must also be “123456”.*

# 6 Internal and External Key Storage

PKCS#11 keys (objects) can be stored in two locations:

- Internal: Keys are stored in the CryptoServer hardware security module.
- External: Keys are stored in an external database.

The default behavior for the generation or creation of PKCS#11 objects can be controlled by the configuration parameter `KeysExternal` configuration value.

# 7 Development of a PKCS#11 Application

The PKCS#11 interface is a pure C-interface. A detailed specification of function prototypes, cryptographic mechanisms, etc. is described in the PKCS#11 specifications [PKCS11BS] and [PKCS11ICMS].

For the development of a PKCS#11 application the following header files are needed:

- `cryptoki.h`
- `pkcs11.h`
- `pkcs11f.h`
- `pkcs11t.h`
- `pkcs-11v2-20a3.h`
- `pkcs11t_cs.h`

These files can be downloaded from [PKCS11] except for the last one, which is delivered with the CryptoServer and contains CryptoServer specific definitions.

To develop a PKCS#11 application with the CryptoServer PKCS#11 library the following preconditions must be fulfilled:

- The application shall include the header file `cryptoki.h`.
- The configuration file `cs_pkcs11_R2.cfg` must contain the needed information to access the CryptoServer (see chapter 4).
- The SecurityServer firmware package (version 3.00 or higher) must have been loaded.
- All additional requirements of chapter 3 must be fulfilled.

## 7.1 Libraries

For development the following libraries exist:

- For Microsoft Windows operating systems: Dynamic Link Library (DLL) `cs_pkcs11_R2.dll`. The library is built with Microsoft Visual Studio components.
- For Linux, and other UNIX systems: shared library `libcs_pkcs11_R2.so` and static library `libcs_pkcs11_R2_m.a`. Both are built with the GNU Compiler Collections.

The libraries contain everything that is needed to communicate between the PKCS#11 application and CryptoServer.

*The libraries for Windows are packed using 1 byte alignment. All other libraries (Linux, etc.) are compiled with alignment to the processor specific word boundaries.*

# 8 Runtime

This chapter describes details about the Utimaco's PKCS#11 implementation for the CryptoServer.

## 8.1 Initialization

When the PKCS#11 function `C_Initialize()` is called inside an PKCS#11 application the configuration file `cs_pkcs11_R2.cfg` will be parsed. In error case `C_Initialize()`returns standard PKCS#11 error code. The logging mechanism can be used to determine which error occurred.

*If the **`Global`** section is part of the problem, the logfile may not be created. Fix the **`Global`** section first and continue.*

The command `C_GetSlotList()` returns a list of all available slots. If more than one CryptoServer is configured, the slots are already mapped to the specific schema described in Chapter *5*, “Operating Modes”.

## 8.2 Limited Data Length

The data length transferred between the CryptoServer PKCS#11 library and the CryptoServer is limited to a maximum of 250 kByte (256000 byte). This affects all functions that can handle large data sizes e.g. `C_Digest(), C_Crypt(),` etc.

The maximum command size includes also a small command header that is prepended to each data block send to the CryptoServer. The size of the command header varies depending on the executing function. Therefore, the maximum size of the input data is always a little bit smaller than this limit (about 25 byte or more).

If the application should handle data blocks that exceeds this limit it should be considered to split the data into smaller pieces and use always the triple set of functions (`C_xxx_init`, `C_xxx_update`, `C_xxx_final`) to avoid any problems.

### 8.2.1 Key Wrapping with AES GCM/CCM

The mechanisms `CKM_AES_GCM` and `CKM_AES_CCM` require, among other things, the mechanism parameter `ulAADLen`, which provides the length of the additional authentication data (AAD). In case of `C_Wrap` and `C_Unwrap`, it is limited to 64 kByte – 1 (0xFFFF) because no auto-chunking of AAD is supported for these functions.

### 8.2.2 Initialization Vector Length for AES GCM

For AES GCM, an initialization vector must be provided as mechanism parameter `pIv`. The length of this initialization vector must also be set using the mechanism parameter `ulIvLen`. A length of 12 byte is recommended by the NIST.

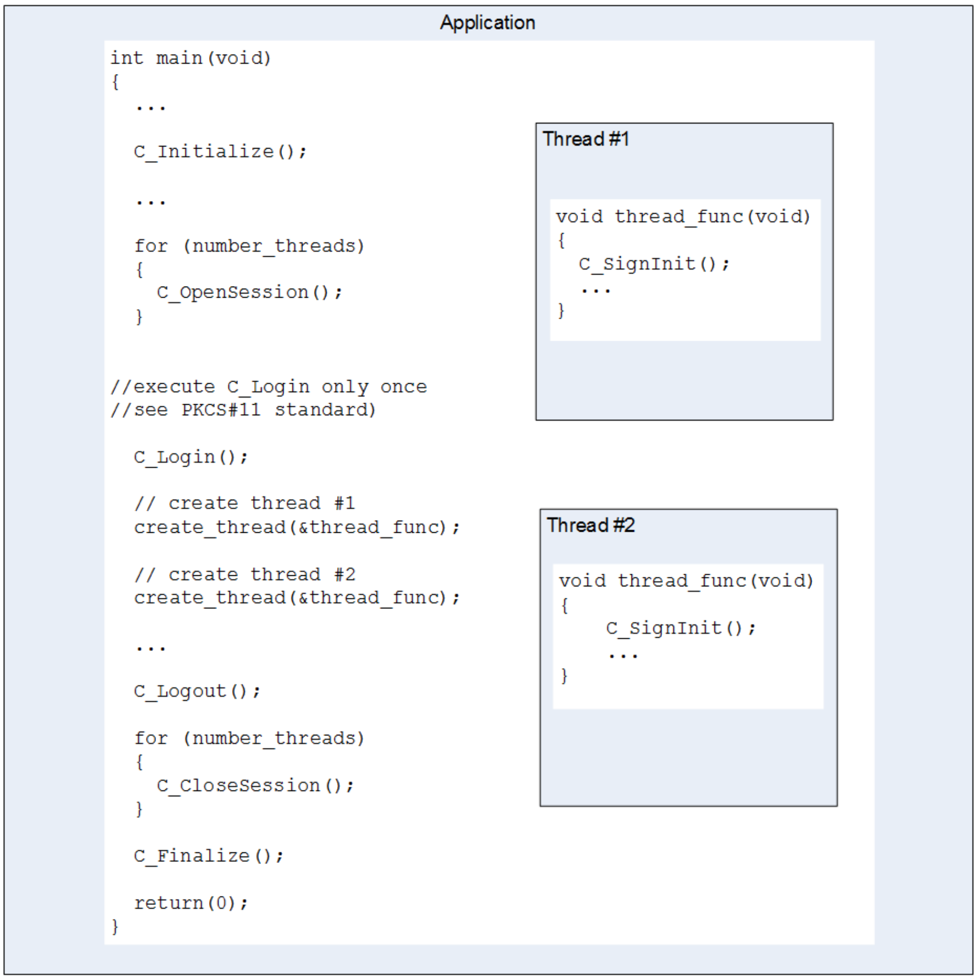
### 8.2.3 Data Length for Key Wrapping with AES GCM/CCM

For AES CCM, the explicit input of data length is usually required using the mechanism parameter `ulDataLen`. But in case of the functions `C_Wrap` and `C_Unwrap`, the correct size of the data, which is the key size plus overhead, is automatically provided by the CXI module. Therefore, this mechanism parameter is ignored.

## 8.3 Multithreading

If the program accesses the CryptoServer PKCS#11 library from a multithreaded application where several threads are simultaneously calling PKCS#11 functions, then the following approach should be used.

The main thread should call the `C_Initialize()` function and create all sessions by executing the `C_OpenSession()` function. A single login (`C_Login`) should be performed for all open sessions. The session handles must be provided to the threads so that each thread can perform its operations.



```
int main(void)
{
  ...

  C_Initialize();

  ...

  for (number_threads)
  {
    C_OpenSession();
  }


//execute C_Login only once
//see PKCS#11 standard)

  C_Login();

  // create thread #1
  create_thread(&thread_func);

  // create thread #2
  create_thread(&thread_func);

  ...

  C_Logout();

  for (number_threads)
  {
    C_CloseSession();
  }

  C_Finalize();

  return(0);
}
```

```
void thread_func(void)
{
  C_SignInit();
  ...
}
```

```
void thread_func(void)
{
    C_SignInit();
    ...
}
```

For example, if one session is shared with two threads, the following problem occurs: The first thread performs `C_SignInit()` and then performs several `C_SignUpdate()` steps. If the second thread performs also a `C_SignInit()` and `C_SignUpdate()` at the same time the result is unspecified. The CryptoServer cannot distinguish between these two threads because it knows only the session and handles both threads as if they were only one thread.

# 9 Authentication Concept

In this chapter the authentication concept and user management of the CryptoServer PKCS#11 R2 library is described.

For standard PKCS#11 usage it is sufficient to use the standard authentication concept as provided by the CryptoServer PKCS#11 library. This combines the standard PKCS#11 concept of Security Officer and normal User role with the CryptoServer's secure user and authentication concept. See Chapter 9.1 "Standard Authentication Concept" for details.

For non-standard requirements like special authentication mechanisms, authentication according to the two-person rule or a dedicated key manager role see Chapter 9.2 "Enhanced Authentication Concept".

## 9.1 Standard Authentication Concept

PKCS#11 recognizes two types of users: The Security Officer (SO) and the User (USER). These users are mapped to users on the CryptoServer: For example, the SO of slot number 0 is mapped to the CryptoServer user 'SO_0000'. When a PKCS#11 user is logged onto the CryptoServer PKCS#11 library, also the corresponding CryptoServer user is logged onto the CryptoServer.

Additionally, Utimaco's PKCS#11 R2 implementation introduces another user: the administrator. The administrator corresponds also to a CryptoServer user with a minimum permission of '20000000'. To create an SO using the `C_InitToken` command, the administrator must be logged in first. To initialize a slot, create an SO and a USER the following steps are required:

1. Login a CryptoServer user with a permission of at least 20000000. For instance, login as the CryptoServer default user ADMIN with the `C_Login` command.
2. Create the SO executing the `C_InitToken` command.
3. Logout the CryptoServer administrator with the `C_Logout` command.
4. Login the SO with the `C_Login` command.
5. Create the USER executing the `C_InitPIN` command.
6. Logout the SO with the `C_Logout` command.

### Example authentication:

```
//1.
CK_UTF8CHAR_PTR pPin = "ADMIN,C:\\tmp\\ADMIN.key";
CK_ULONG    ulPinLen = strlen(pPin);

err = C_Login(hSession, CKU_CS_GENERIC, pPin,  ulPinLen);
```

```
// 2.
CK_SLOT_ID slotID = 0;
CK_UTF8CHAR_PTR pPin = "123456";
CK_ULONG    ulPinLen = strlen(pPin);
CK_UTF8CHAR_PTR pLabel = "Testlabel";

err = C_InitToken(slotID, pPin, ulPinLen,  pLabel);

//3.
Err = C_Logout(hSession);

//4.
CK_UTF8CHAR_PTR pPin = "123456";
CK_ULONG    ulPinLen = strlen(pPin);

err = C_Login(hSession, CKU_SO, pPin,  ulPinLen);

//5.
CK_UTF8CHAR_PTR pPin = "654321";
CK_ULONG    ulPinLen = strlen(pPin);

err = C_InitPIN(hSession, pPin,  ulPinLen);

//6.
Err = C_Logout(hSession);
```

After the execution of the six steps, the command `csadm ListUser` shows the following output (example listing for slot 0):

```
Name       Permission  Mechanism    Attributes
ADMIN      22000000    RSA sign
SO_0000    00000200    HMAC passwd  A[CXI_GROUP=SLOT_0000]
USR_0000   00000002    HMAC passwd  A[CXI_GROUP=SLOT_0000]
```

## 9.2 Enhanced Authentication Concept

### 9.2.1 Create Users with Other Authentication Mechanisms

The default authentication mechanism used with PCKS#11 users (SO and USER) is HMAC password. To create users using other authentication mechanisms, information about the user credentials (password, path to keyfile or smartcard with signature key) has to be provided with the `pPin` parameter of the `C_InitToken` and the `C_InitPIN` functions in the following syntax with the prefix `CKU_VENDOR:`.

| *Authentication Mechanism* | *Syntax (pPin parameter of C_InitToken or C_InitPIN)* |
|---|---|
| RSA signature with keyfile | `CKU_VENDOR:RSASign={Hash}<filename>`<br>`CKU_VENDOR:RSASign={Hash}<filename>#<password>`<br>■ `<filename>` - Name of the file containing the RSA key for the user incl. path to the file<br>■ `<password>` - Password of the keyfile, if encrypted |
| RSA Signature with smartcard directly connected to the host | `CKU_VENDOR:RSASign={Hash}<key-specifier>`<br>`<key-specifier>` - Description of smartcard, PIN pad and interface (example: `:cs2:auto:USB0`). See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers. |
| RSA Signature with smartcard connected to the CryptoServer | `CKU_VENDOR:RSASC={Hash}<key-specifier>`<br>`<key-specifier>` - Description of smartcard, PIN pad and interface (example: `:cs2:auto:USB0`). See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers. |
| HMAC password | `CKU_VENDOR:HMACPwd={Hash}<password>`<br>`<password>` - Password of the user |
| ECDSA signature with keyfile | `CKU_VENDOR:ECDSA={Hash}<filename>`<br>`CKU_VENDOR:ECDSA={Hash}<filename>#<password>`<br>■ `<filename>` - Name of the file containing the RSA key of the user incl. path to the file<br>■ `<password>` - Password of keyfile, if encrypted |
| ECDSA signature with smartcard connected to the host | `CKU_VENDOR:ECDSA={Hash}<key-specifier>`<br>`<key-specifier>` - Description of smartcard, PIN pad and interface (example: `:cs2:auto:USB0`). See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers. |

Table 8: Authentication mechanisms and their syntax

*In failover operation mode, the users are created by  using the CryptoServer administration tools csadm or CAT.*

## Example (SO using RSA signature with smartcard directly connected to the host):

```
CK_SLOT_ID slotID = 0;
```

```
CK_UTF8CHAR_PTR pPin = "CKU_VENDOR:RSASign=:cs2:auto:USB0";
CK_ULONG    ulPinLen = strlen(pPin);
CK_UTF8CHAR_PTR pLabel = "Testlabel";

err = C_InitToken(slotID, pPin, ulPinLen,  pLabel);
```

#### Example (SO using RSA signature with keyfile):

```
CK_SLOT_ID slotID = 0;
CK_UTF8CHAR_PTR pPin = "CKU_VENDOR:RSASign=C:\\tmp\\RSA.key#1234";
CK_ULONG    ulPinLen = strlen(pPin);
CK_UTF8CHAR_PTR pLabel = "Testlabel";

err = C_InitToken(slotID, pPin, ulPinLen,  pLabel);
```

#### Example (SO using HMAC password):

```
CK_SLOT_ID slotID = 0;
CK_UTF8CHAR_PTR pPin = "CKU_VENDOR:HMACPwd=123456";
CK_ULONG    ulPinLen = strlen(pPin);
CK_UTF8CHAR_PTR pLabel = "Testlabel";

err = C_InitToken(slotID, pPin, ulPinLen,  pLabel);
```

#### Example (SO using ECDSA signature with smartcard directly connected to the host):

```
CK_SLOT_ID slotID = 0;
CK_UTF8CHAR_PTR pPin = "CKU_VENDOR:ECDSA=:cs2:auto:USB0";
CK_ULONG    ulPinLen = strlen(pPin);
CK_UTF8CHAR_PTR pLabel = "Testlabel";

err = C_InitToken(slotID, pPin, ulPinLen,  pLabel);
```

#### Example (SO using ECDSA signature with keyfile):

```
CK_SLOT_ID slotID = 0;
CK_UTF8CHAR_PTR pPin = "CKU_VENDOR:ECDSA=C:\\tmp\\ECDSA.key#4321";
CK_ULONG    ulPinLen = strlen(pPin);
CK_UTF8CHAR_PTR pLabel = "Testlabel";
err = C_InitToken(slotID, pPin, ulPinLen,  pLabel);
```

### 9.2.2 Login User with Other Authentication Mechanisms

The login of a user using other authentication mechanisms than the default one is similar to the creation of a user. The `userType` provided to the `C_Login` must be `CKU_CS_GENERIC`. The `pPin` parameter of `C_Login` provides the information about the user credentials (password, path to the keyfile or smartcard) must be provided in the following syntax:

| *Authentication Mechanism* | *Syntax (pPin parameter of C_Login)* |
|---|---|
| RSA signature with keyfile | `<username>,<filename>`<br>`<username>,<filename>#<password>`<br>`<filename>` - Name of the file containing the user's RSA key incl. path<br>`<password>` - Password of keyfile, if encrypted |
| RSA Signature with smartcard directly connected to the host<br>PIN is read in over the PIN pad | `<username>,<key-specifier>`<br>`<key-specifier>` - Description of smartcard, PIN pad and interface (example: `:cs2:auto:USB0`). See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers. |
| RSA Signature with smartcard connected to the CryptoServer<br>PIN is read in over the PIN pad. | `<username>`<br>`<key-specifier>` - Description of smartcard, PIN pad and interface (example: `:cs2:auto:USB0`). See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers. |
| HMAC password | `<username>,<password>`<br>`<password>` - Password of the user |
| ECDSA signature with keyfile | `<username>,<filename>`<br>`<username>,<filename>#<password>`<br>`<filename>` - Name of the file containing the user's ECDSA incl. path to the file<br>`<password>` - Password of keyfile, if encrypted |
| ECDSA signature with smartcard connected to the host<br>PIN is read in over the PIN pad | `<username>,<key-specifier>`<br>`<key-specifier>` - Description of smartcard, PIN pad and interface (example: `:cs2:auto:USB0`). See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers. |

Table 9: Authentication mechanisms and corresponding syntax

### Example (login of SO on slot 0) using HMAC password):

```
CK_UTF8CHAR_PTR pPin = "SO_0000,123456";
```

```
CK_ULONG    ulPinLen = strlen(pPin);

err = C_Login(hSession, CKU_CS_GENERIC, pPin,  ulPinLen);
```

#### Example (login of SO on slot 0) using RSA signature with keyfile):

```
CK_UTF8CHAR_PTR pPin = "SO_0000,C:\\tmp\\RSA.key#1234";
CK_ULONG    ulPinLen = strlen(pPin);

err = C_Login(hSession, CKU_CS_GENERIC, pPin,  ulPinLen);
```

#### Example (login of SO on slot 0) using RSA signature with smartcard directly connected to the host):

```
CK_UTF8CHAR_PTR pPin = "SO_0000,:cs2:auto:USB0";
CK_ULONG    ulPinLen = strlen(pPin);

err = C_Login(hSession, CKU_CS_GENERIC, pPin,  ulPinLen);
```

#### Example (login of USER on slot 0) using ECDSA signature with keyfile):

```
CK_UTF8CHAR_PTR pPin = "USR_0000,C:\\tmp\\ECDSA.key";
CK_ULONG    ulPinLen = strlen(pPin);

err = C_Login(hSession, CKU_CS_GENERIC, pPin,  ulPinLen);
```

#### Example (login of USER on slot 0) using ECDSA signature with smartcard directly connected to the host):

```
CK_UTF8CHAR_PTR pPin = "USR_0000,:cs2:auto:USB0";
CK_ULONG    ulPinLen = strlen(pPin);

err = C_Login(hSession, CKU_CS_GENERIC, pPin,  ulPinLen);
```

### 9.2.3 Change PIN for Other Authentication Mechanisms

To change the PIN of a user with authentication mechanism HMAC password (changing the PIN for other mechanisms is not possible) the `pOldPin` and `pNewPin` parameter of the `C_SetPin` function must be provided in the following syntax with the prefix `CKU_VENDOR:`.

| *Mechanism* | *Syntax (pPin parameter of C_SetPIN)* |
|---|---|
| HMAC password | `CKU_VENDOR:<password>`<br>`<password>` - Password of the user |

Table 10: Syntax for changing the user password

### Example (change PIN of USER using HMAC password):

```
CK_UTF8CHAR_PTR pOldPin = "CKU_VENDOR:123456";
CK_ULONG    ulOldLen = strlen(pOldPin);
CK_UTF8CHAR_PTR pNewPin = "CKU_VENDOR:654321";
CK_ULONG    ulNewLen = strlen(pNewPin);

err = C_SetPIN(hSession, pOldPin, ulOldLen , pNewPin, ulNewLen);
```

### Example (RSA keyfile):

```
CK_UTF8CHAR_PTR pOldPin = "CKU_VENDOR:12345678";
CK_ULONG    ulOldLen = strlen(pOldPin);
CK_UTF8CHAR_PTR pNewPin = "CKU_VENDOR:87654321";
CK_ULONG    ulNewLen = strlen(pNewPin);

err = C_SetPIN(hSession, pOldPin, ulOldLen , pNewPin, ulNewLen);
```

### 9.2.4 Authentication via Configuration File

Sometimes it may be necessary to use an authentication mechanism different from the default mechanism HMAC password but it is not possible to provide the mechanism information with the parameters via the CryptoServer PKCS#11 library as described in the previous sections. In this case, the information for the user whose authentication mechanism differs from the default one can be written to the configuration file `cs_pkcs11_R2.cfg` instead. The configuration item must specify the user name as its <key> parameter and the authentication mechanism details as <value> of the parameter given in quotation marks.

| *Authentication Mechanism* | *Syntax in configuration file ("<key> = <value>")* |
|---|---|
| RSA Signature with keyfile | `<username> = "RSASign={Hash}<filename>"`<br><br>`<filename>` - Name of the file containing the user's RSA key incl. path to the file |
| RSA Signature with smartcard connected to the host | `<username> = "RSASign={Hash}<key-specifier>"`<br><br>`<key-specifier>` - Description of smartcard, PIN pad and interface (example: `:cs2:auto:USB0`). See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers. |
| RSA Signature with smartcard directly connected to the CryptoServer | `<username> = "RSASC ={Hash}<key-specifier>"`<br><br>`<key-specifier>` - Description of smartcard, PIN pad and interface (example: `:cs2:auto:USB0`). See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers. |

| *Authentication Mechanism* | *Syntax in configuration file ("<key> = <value>")* |
|---|---|
| HMAC password | `<username> = "HMACPwd={Hash}"` |
| ECDSA Signature with keyfile | `<username> = "ECDSA ={Hash}<filename>"` <br> `<filename>` - Name of the file containing the user's ECDSA incl. path to the file |
| ECDSA Signature with smartcard connected to the host | `<username> = "ECDSA ={Hash}<key-specifier>"` <br> `<key-specifier>` - Description of smartcard, PIN pad and interface (example: `:cs2:auto:USB0`). See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers. |

Table 11: Syntax for defining user's authentication mechanism in cs_pkcs11_R2.cfg

### Example configuration file (default SO of slot 0 who uses authentication mechanism RSA signature with keyfile):

```
[Global]
Logging = 3
Logpath = c:/tmp
Logsize = 10mb

[CryptoServer]
Device    = PCI:0

[Slot]
SlotNumber = 0

# CryptoServer user ‘SO_0000’
# using RSASign mechanism (SO of slot 0)
# with the key located at ‘C:\tmp\RSA.key’
SO_0000 = "RSASign=C:\tmp\RSA.key"
```

### Example configuration file (non-default CryptoServer user who uses authentication mechanism RSA signature with smartcard connected to the host):

```
[Global]
Logging = 3
Logpath = c:/tmp
Logsize = 10mb

[CryptoServer]
Device    = PCI:0

[Slot]
SlotNumber = 2
```

```
#CryptoServer user ‘CSuser’
# using RSASign mechanism with smartcard authentication
# (in slot 2)
# with the PIN pad connected to an USB interface
CSuser = "RSASign=:cs2:auto:USB0"
```

See the chapters "Key Specifiers" and "Using a Local PIN Pad for a Remote CryptoServer" in [CSADMIN] for details about key specifiers.

*Quotation marks at the beginning and the end of the authentication mechanism value are mandatory.*

In case that the information about the authentication mechanism of a specific user is given in the configuration file as described above, the `pPin` parameter in functions `C_InitToken`, `C_InitPIN`, `C_Login` and `C_SetPin` have to be provided as follows:

| *Mechanism* | *C_InitToken, C_InitPIN* | *C_Login* | *C_SetPin* |
|---|---|---|---|
| RSA signature with keyfile | `NULL_PTR` or password of keyfile | `NULL_PTR` or password of keyfile | NOT ALLOWED |
| RSA Signature with smartcard connected to the host | `NULL_PTR` | `NULL_PTR` | NOT ALLOWED |
| RSA Signature with smartcard connected to the CryptoServer | `NULL_PTR` | `NULL_PTR` | NOT ALLOWED |
| HMAC password | password | password | password |
| ECDSA signature with keyfile | `NULL_PTR` or password of keyfile | `NULL_PTR` or password of keyfile | NOT ALLOWED |
| ECDSA signature with smartcard connected to the CryptoServer | `NULL_PTR` | `NULL_PTR` | NOT ALLOWED |

Table 12: Definition of pPin parameter if user's auth. mechanism is specified in cs_pkcs11_R2.cfg

### 9.2.5 Automatic Login of Administrator via Configuration File

Sometimes, it might not be possible to logon as an administrator (function `C_Login` for user with administrator rights) before the execution of the slot initialization (function `C_InitToken`). Therefore, a special user with administrator rights who is logged in automatically before the initialization can be configured via configuration file. The user's name is “AD”, and he uses the authentication mechanism HMAC with minimum permission ‘20000000’. First, the user must be created using the administration tools csadm or CAT.

After creating the special user, the output of the command `csadm ListUser` should for example look as follows:

```
Name       Permission  Mechanism    Attributes
ADMIN      22000000    RSA sign
AD         20000000    HMAC passwd
```

For the automatic login of the user AD to a slot, the user credentials must be written into the configuration file according to the following syntax:

```
AD = "HMACPwd=[{Hash}]#<password>"
```

#### Example configuration file (administrator user AD is automatically logged in when executing first `C_InitToken` on slot 0):

```
[Global]
Logging = 3
Logpath = c:/tmp
Logsize = 10mb

[CryptoServer]
Device    = PCI:0

[Slot]
SlotNumber = 0
# the administrator user ‘AD’ is logged in during C_InitToken
# process with the HMAC authentication mechanism
# and password ‘123456’
AD = "HMACPwd=#123456"
```

### 9.2.6 Authentication According to the Two-Person Rule

The CryptoServer PKCS#11 interface provides the possibility to realize an authentication concept according to the two-person rule.

If authentication according to the two-person rule is required (e.g. due to a security policy), specific users obeying to the rule have to be created manually with the CryptoServer administration tools, CAT or csadm.

For every user role (SO, USER or key manager KM (if optionally configured)) that must follow the two-person rule for authentication, minimum two users with permission 1 in the respective user group have to be created.

### Examples for user creation according to the two-person rule (slot 0):

- In order to implement the two-person rule for the SO in slot 0, create minimum two users `SO1_0000` and `SO2_0000`, with user permission 00000100 and attribute `CXI_GROUP=SLOT_0000`.
- In order to implement the two-person rule for the USER in slot 0, create minimum two users `USR1_0000` and `USR2_0000`, with user permission 00000001 and attribute `CXI_GROUP=SLOT_0000`.
- In order to implement the two-person rule for the KM in slot 0, create minimum two users `KM1_0000` and `KM2_0000`, with user permission 00000010 and attribute `CXI_GROUP=SLOT_0000`.

To login two users for a PKCS#11 command execution according to the two-person rule the `C_Login` function with user type `CKU_CS_GENERIC` has to be used twice (once for each user).

### Example login (two SOs with authentication mechanism HMAC for slot 0):

```
CK_UTF8CHAR_PTR pPinSO1 = "SO1_0000,123456";
CK_ULONG    ulPinLenSO1 = strlen(pPinSO1);
CK_UTF8CHAR_PTR pPinSO2 = "SO2_0000,654321";
CK_ULONG    ulPinLenSO2 = strlen(pPinSO2);

err = C_Login(hSession, CKU_CS_GENERIC, pPinSO1,  ulPinLenSO1);
err = C_Login(hSession, CKU_CS_GENERIC, pPinSO2,  ulPinLenSO2);
```

### 9.2.7 Extended Login

The extended login mechanism enables the execution of multiple login during a `C_Login` function call. It is often useful in situations where the CryptoServer PKCS#11 library is integrated into another software but a requirement demands a login using the two-person rule with authentication via a PIN pad.

First, two users for the two-person rule must be created like explained in chapter 9.2.6. To activate extended login for a slot, an entry shall be set in the corresponding slot section of the configuration file `cs_pkcs11_R2.cfg`:

- The configuration entry `ExtendedLoginSO` enables the extended login for the SO.
- The configuration entry `ExtendedLoginUSER` enables the extended login for the USER.

The value of the configuration entry is a list of login data which are formatted in the same way like the `pPin` parameter in chapter 9.2.2.

### Example configuration file (two-person configuration of SO with authentication via a PIN pad on slot 0):

```
[Global]
Logging = 3
Logpath = c:/tmp
Logsize = 10mb

[CryptoServer]
Device    = PCI:0

[Slot]
SlotNumber = 0
# after C_Login was called, the user SO1_0000 must authenticate via the
# PIN pad connected to an USB interface; then the user SO2_0000 must
# authenticate via the same PIN pad
ExtendedLoginSO = {
SO1_0000,:cs2:auto:USB0
SO2_0000,:cs2:auto:USB0
}
```

## 9.2.8 Key Manager and Key User Role

In PKCS#11 the USER has, by default, the permissions/tasks to manage keys (create, delete, import, export, etc.) and to use them in cryptographic operations. These tasks can also be split into two groups and assigned to different PKCS#11 users. This requires the CryptoServer PKCS#11 library to be configured accordingly so it can handle the two resulting user roles:

- Key manager (KM) with permission to manage cryptographic keys
- Key user (KU) with the permission to use cryptographic keys.

## 9.2.9 Create and Login the Key Manager

The KM must be created out of the scope of the CryptoServer PKCS#11 library with the CryptoServer's administration tools csadm or CAT. After creation the configuration value `CKA_CFG_AUTH_KEYM_MASK` must be set to the necessary permission. For step-by-step instructions on how to create a new user with the key manager role with CAT and P11CAT, please read Chapter "Creating a New User" in [CSMSADM], and Chapter "Changing the Global Configuration" in [CS_PKCS11CAT]. An example for creating a user in the key manager role with csadm is provided in Chapter "AddUser" of the [CSADMIN].

## Example (KM for slot 1):

Create a user KM_0001 with permission mask of 0000020 and attribute `CXI_GROUP=SLOT_0001`. This command requires authentication by a user with the user administrator role (min. permissions 20000000) or by the default CryptoServer user ADMIN.

1. Login as a user with the user administrator role, or as the ADMIN, or alternatively as the SO for the corresponding slot `SO_0001` by using the `C_Login` function.
   IMPORTANT: The last one requires the slot configuration object `CKA_CFG_ALLOW_SLOTS` be previously set to `CK_TRUE`.
2. Change the attribute `CKA_CFG_AUTH_KEYM_MASK` with function `C_SetAttributeValue` to the value `0x00000020.`
3. Login as the user `KM_0001` with the `C_Login` function and user type `CKU_CS_GENERIC`.
   The KM is logged in now and can perform key management functions.

# 10 Key Management Functions in PKCS#11

The following lists show an overview of functions which can be executed by the KM or KU. If a KM tries to execute functions from the KU (and vice versa) the error `CKR_USER_NOT_LOGGED_IN` occurs. For a detailed description of the functions listed below please see [PKCS11BS].

| *PKCS#11 Function* | *Description* | *Permitted user* | |
|---|---|---|---|
| | | *KM* | *KU* |
| `C_CreateObject` | Creates a new object | ✓ | ✗ |
| `C_CopyObject` | Creates a copy of an object | ✓ | ✗ |
| `C_DestroyObject` | Destroys an object | ✓ | ✗ |
| `C_SetAttributeValue` | Modifies an attribute value of an object | ✓ | ✗ |
| `C_GenerateKey` | Generates a secret key | ✓ | ✗ |
| `C_GenerateKeyPair` | Generates a public-key/private-key pair | ✓ | ✗ |
| `C_WrapKey` | Wraps (encrypts) a key | ✓ | ✗ |
| `C_UnwrapKey` | Unwraps (decrypts) a key | ✓ | ✗ |
| `C_DeriveKey` | Derives a key from a base key | ✓ | ✗ |
| `C_EncryptInit` | Initializes an encryption operation | ✗ | ✓ |
| `C_Encrypt` | Encrypts single-part data | ✗ | ✓ |
| `C_EncryptUpdate` | Continues a multiple-part encryption operation | ✗ | ✓ |
| `C_EncryptFinal` | Finishes a multiple-part encryption operation | ✗ | ✓ |
| `C_DecryptInit` | Initializes a decryption operation | ✗ | ✓ |
| `C_Decrypt` | Decrypts single-part encrypted data | ✗ | ✓ |
| `C_DecryptUpdate` | Continues a multiple-part decryption operation | ✗ | ✓ |
| `C_DecryptFinal` | Finishes a multiple-part decryption operation | ✗ | ✓ |

| *PKCS#11 Function* | *Description* | *Permitted user* | |
|---|---|---|---|
| | | *KM* | *KU* |
| `C_DigestInit` | Initializes a message-digesting operation | ✕ | ✓ |
| `C_Digest` | Digests single-part data | ✕ | ✓ |
| `C_DigestUpdate` | Continues a multiple-part digesting operation | ✕ | ✓ |
| `C_DigestKey` | Digests a key | ✕ | ✓ |
| `C_DigestFinal` | Finishes a multiple-part digesting operation | ✕ | ✓ |
| `C_SignInit` | Initializes a signature operation | ✕ | ✓ |
| `C_Sign` | Signs single-part data | ✕ | ✓ |
| `C_SignUpdate` | Continues a multiple-part signature operation | ✕ | ✓ |
| `C_SignFinal` | Finishes a multiple-part signature operation | ✕ | ✓ |
| `C_SignRecoverInit` | Initializes a signature operation, where the data can be recovered from the signature | ✕ | ✓ |
| `C_SignRecover` | Signs single-part data, where the data can be recovered from the signature | ✕ | ✓ |
| `C_VerifyInit` | Initializes a verification operation | ✕ | ✓ |
| `C_Verify` | Verifies a signature on single-part data | ✕ | ✓ |
| `C_VerifyUpdate` | Continues a multiple-part verification operation | ✕ | ✓ |
| `C_VerifyFinal` | Finishes a multiple-part verification operation | ✕ | ✓ |
| `C_VerifyRecoverInit` | Initializes a verification operation where the data is recovered from the signature | ✕ | ✓ |
| `C_VerifyRecover` | Verifies a signature on single-part data where the data is recovered from the signature | ✕ | ✓ |
| `C_DigestEncryptUpdate` | Continues simultaneous multiple-part digesting and encryption operations | ✕ | ✓ |
| `C_DecryptDigestUpdate` | Continues simultaneous multiple-part decryption and digesting operations | ✕ | ✓ |

| *PKCS#11 Function* | *Description* | *Permitted user* | |
|---|---|---|---|
| | | *KM* | *KU* |
| `C_SignEncryptUpdate` | Continues simultaneous multiple-part signature and encryption operations | ✕ | ✓ |
| `C_DecryptVerifyUpdate` | Continues simultaneous multiple-part decryption and verification operations | ✕ | ✓ |

Table 13: List of PKCS#11 standard functions for KM and KU

# 11 Vendor Defined PKCS#11 Extensions

Definitions for all vendor defined extensions are provided with the include `pkcs11t_cs.h` file.

## 11.1 CryptoServer Defined Mechanisms

The CryptoServer implements the following mechanisms which are not included in the PKCS#11 standard (see [PKCS11ICMS]).

| *Name* | *Description* |
|---|---|
| `CKM_ECDSA_SHA3_224` | ECDSA signature generation using SHA3-224 hash algorithm. |
| `CKM_ECDSA_SHA3_256` | ECDSA signature generation using SHA3-256 hash algorithm. |
| `CKM_ECDSA_SHA3_384` | ECDSA signature generation using SHA3-384 hash algorithm. |
| `CKM_ECDSA_SHA3_512` | ECDSA signature generation using SHA3-512 hash algorithm. |
| `CKM_ECDSA_RIPEMD160` | ECDSA signature generation using RIPEMD-160 hash algorithm. |
| `CKM_DSA_RIPEMD160` | DSA signature generation using RIPEMD-160 hash algorithm. |
| `CKM_DES3_RETAIL_MAC` | Triple DES Retail-MAC with 0-Padding (see chapter 11.3). |
| `CKM_RSA_PKCS_MULTI` | Generate multiple signatures with `CKM_RSA_PKCS` mechanism (see chapter 11.4). |
| `CKM_RSA_X_509_MULTI` | Generate multiple signatures with `CKM_RSA_X_509` mechanism (see chapter 11.4). |
| `CKM_ECDSA_MULTI` | Generate up to 16 signatures with `CKM_ECDSA` mechanism (see chapter 11.4). |
| `CKM_DES_CBC_WRAP` | Enhanced DES key wrapping mechanism (see definition for structure `CK_WRAP_PARAMS`). |
| `CKM_AES_CBC_WRAP` | Enhanced AES key wrapping mechanism (see definition for structure `CK_WRAP_PARAMS`). |
| `CKM_ECKA` | EC secret agreement according to BSI-TR-03111 (returns secret point without hashing). |
| `CKM_ECDSA_ECIES` | ECDSA crypt algorithm (see chapter 11.2). |

Table 14: Description of CryptoServer specific mechanisms

## 11.2 Encryption with the “Elliptic Curve Integrated Encryption Scheme” (ECIES)

The mechanism `CKM_ECDSA_ECIES` can be used in single `C_Encrypt()` or `C_Decrypt()` function calls to cipher data according to the “Elliptic Curve (Augmented) Encryption Scheme” of [ANSI-X9.63] or “Elliptic Curve Integration Encryption Scheme (ECIES)” of [SEC1].

### Example for mechanism CKM_ECDSA_ECIES:

```
CK_SESSION_HANDLE     sid;
CK_OBJECT_HANDLE      hPublicKey;  // CKK_ECDSA
CK_OBJECT_HANDLE      hPrivateKey; // CKK_ECDSA
CK_BYTE               plain[BUFFERSIZE];
CK_ULONG              l_plain = sizeof(plain);
CK_BYTE               encrypt[BUFFERSIZE];
CK_BYTE               decrypt[BUFFERSIZE];
CK_ULONG              l_encrypt = sizeof(encrypt);
CK_ULONG              l_decrypt = sizeof(decrypt);

CK_ECDSA_ECIES_PARAMS ecies_para;

CK_MECHANISM mechanism = {
    CKM_ECDSA_ECIES, &ecies_para, sizeof(ecies_para)
};

...

ecies_para.hashAlg         = CKM_SHA_1;
ecies_para.cryptAlg        = CKM_AES_CBC;
ecies_para.cryptOpt        = 16;
ecies_para.macAlg          = CKM_SHA_1_HMAC;
ecies_para.macOpt          = 0;
ecies_para.pSharedSecret1  = "top";
ecies_para.ulSharetSecret1 = 3;
ecies_para.pSharedSecret2  = "secret";
ecies_para.ulSharetSecret2 = 6;

err = C_EncryptInit(sid, &mechanism, hPublicKey);
err = C_Encrypt(sid, plain, l_plain, encrypt, &l_encrypt);
err = C_DecryptInit(sid, &mechanism, hPrivateKey);
err = C_Decrypt(sid, encrypt, l_encrypt, decrypt, &l_decrypt);
```

The following parameters can be used:

```
hashAlg:  CKM_SHA_1, CKM_SHA224, CKM_SHA256, CKM_SHA384,
          CKM_SHA512, CKM_RIPEMD160, CKM_MD5

cryptAlg: CKM_AES_ECB, CKM_AES_ECB, CKM_AES_ECB, CKM_AES_CBC,
```

CKM_AES_CBC, CKM_AES_CBC, CKM_ECDSA_ECIES_XOR

| | |
|---|---|
| `cryptOpt:` | Key Length of `cryptAlg`. (0 for `CKM_ECDSA_ECIES_XOR`) |
| `macAlg:` | `CKM_SHA_1_HMAC, CKM_SHA224_HMAC, CKM_SHA256_HMAC, CKM_SHA384_HMAC, CKM_SHA512_HMAC, CKM_MD5_HMAC, CKM_RIPEMD160_HMAC` |
| `macOpt:` | currently ignored |

## 11.3 Sign and Verify Using the DES Retail-MAC

The mechanism `CKM_DES3_RETAIL_MAC` can be used in `C_Sign()` and `C_Verify()` to calculate a CBC Retail-MAC according to [ISO-9797] and [ANSI-X9.19].

### Example for mechanism CKM_DES3_RETAIL_MAC:

```
CK_SESSION_HANDLE      sid;
CK_OBJECT_HANDLE       hSecretKey; //CKK_DES3 handle
CK_MECHANISM           signMechanism;
CK_BYTE                data[BUFFERSIZE];
CK_ULONG               l_data = sizeof(data);
CK_BYTE                signature[BUFFERSIZE];
CK_ULONG               l_signature = sizeof(signature);

signMechanism.mechanism = CKM_DES3_RETAIL_MAC;
signMechanism.pParameter = NULL;
signMechanism.ulParameterLen = 0;

...

err = C_SignInit(sid, &signMechanism, hSecretKey);
err = C_Sign(sid, data, l_data, signature, &l_signatur);
```

- If the `pParameter` of the mechanism structure is set to `NULL` (or `8`), the result has the length of 8 bytes according to the [ISO-9797] specification.
- If the `pParameter` is set to `4`, the result has the length of 4 bytes according to the [ANSI-X9.19] specification.

## 11.4 Multiple Signature Mechanisms

To achieve the maximum performance for signature generation, multiple signatures using the same key can be generated with the vendor defined mechanisms `CKM_RSA_PKCS_MULTI`, `CKM_RSA_X_509_MULTI` and `CKM_ECDSA_MULTI.` These mechanisms behave like `CKM_RSA_PKCS`, `CKM_RSA_X_509` and `CKM_ECDSA` respectively, except that an array of data is given as input to the `C_Sign()` function and that the returned data is an array of signatures. Only single part operations are allowed with these mechanisms.

### Input Data Format

The input data given to the `C_Sign()` function has the following format:

| k | data_1 | data_2 | … | data_k |
|---|---|---|---|---|
| 2 byte | n byte | n byte | … | n byte |

### Output Data Format

On success the function returns an array of signatures of equal length:

| signature_1 | signature_2 | … | signature_k |
|---|---|---|---|
| m byte | m byte | … | m byte |

The following table explains the meaning of the particular fields in the input and output data structures.

| *Field* | *Description* |
|---|---|
| `k` | Number of signatures to be calculated (k ≤ 50). Must be in big-endian format. |
| `data_1` | First input data to be signed. All data parts must have the same length. |
| `data_k` | Last input data to be signed. All data parts must have the same length. |
| `signature_1` | First signature generated by the function calculated over `data_1`. All signatures have the same length. |
| `signature_k` | Last signature generated by the function, calculated over `data_k`. All signatures have the same length. |

Table 15: Fields of the input and output data structures

## 11.5 Configuration Objects

The behavior of the CryptoServer PKCS#11 library can be configured by special objects, called configuration objects. They can be neither created nor deleted and are referenced by unique object handles. The only valid operations are functions to read or to change an attribute value of a configuration object:

```
C_GetAttributeValue
C_SetAttributeValue
```

### 11.5.1 Local Configuration Object

Local configuration objects are used to configure the instance of the CryptoServer PKCS#11 library. They are operative in the currently started instance of the CryptoServer PKCS#11 library, and are referenced by the handle `P11_CFG_LOCAL_HDL`.

| *Attribute* | *Description* |
|---|---|
| `CKA_UTIMACO_CFG_PATH` | Type: `CK_BYTE_PTR`<br>Value: Path to the configuration file<br>Default:<br>■ For the SecurityServer product CD:<br>  ▣ As of version 4.20: `C:\ProgramData\cs_pkcs11_R2.cfg`<br>  ▣ Earlier than version 4.20: `C:\Program Files\Utimaco\CryptoServer\Lib\cs_pkcs11_R2.cfg`<br>■ For the CryptoServer SDK product CD: `C:\Utimaco\CryptoServer\Lib\cs_pkcs11_R2.cfg`<br>read only |

Table 16: Attribute CKA_UTIMACO_CFG_PATH - details

### 11.5.2 Global CryptoServer Configuration Object

Global CryptoServer configuration objects are used to configure settings that affect the whole CryptoServer. They are operative for all instances of the CryptoServer PKCS#11 library that are using the CryptoServer where the object is configured. They are referenced by the handle `P11_CFG_GLOBAL_HDL`.

The attributes can be read by the users ADMIN (or CryptoServer administrators with min. permission 2 in the user group 7, 20000000), SO, USER, key manager and key user.

Write access to global configuration objects is only granted to the default CryptoServer Administrator ADMIN or users with CryptoServer User Administrator role (min. permission 2 in the user group 7, 20000000).

Changes on the attributes of a global configuration object are stored in the database `CXIKEY.db,` which is deleted on alarm occurrence and when the `Clear` command (see Chapter "The Clear Functionality" in [CSADMIN]) is performed. We highly recommend to create a backup of the Global CryptoServer Configuration Object resp. of the `CXIKEY.db` with the csadm command `BackupDatabase` described in Chapter "BackupDatabase" of the [CSADMIN] or with the CAT as described in Chapter "Creating a Database Backup" of the [CSMSADM], so you can easily restore your global PKCS#11 configuration.

The following attributes for global configuration are available on the CryptoServer:

- `CKA_CFG_ALLOW_SLOTS`

  This attribute enables the Security Officer (SO) to configure slots.

  Possible values:

  - `CK_TRUE` - the SO is permitted to configure slots.
  - `CK_FALSE` (default) - the SO is not permitted to configure slots.

- `CKA_CFG_CHECK_VALIDITY_PERIOD`

  This attribute checks the validity period of the key.

  The validity period of a key is only checked, if the following functions are to be performed using the key: `C_SignInit (), C_EncryptInit (), C_DecryptInit (), C_DeriveInit (), C_WrapKey (), C_UnwrapKey ()`

  Possible values:

  - `CK_TRUE` - the validity period of a key is checked, if the key has the attributes `CKA_START_DATE` and `CKA_END_DATE`.
  - `CK_FALSE` (default) - the validity period of a key is not checked.

- `CKA_CFG_AUTH_PLAIN_MASK`

  This attribute defines the permissions required to import and export a key in plaintext.

  Default value: 0x00000002 - corresponds to the permissions of the Cryptographic User, who is already set up in the CryptoServer.

  IMPORTANT:
  If you change the default setting, you must also use the CAT or csadm administration tools to set up the corresponding user in your CryptoServer. This user must be assigned the permissions specified here. For step-by-step instructions on how to create a new user with CAT, please read Chapter "Creating a New User" in [CSMSADM]. Examples for creating different users with csadm are provided in Chapter "AddUser" of the [CSADMIN].

- `CKA_CFG_WRAP_POLICY`

  This attribute applies a key wrapping policy specifying how keys are encrypted so they can be securely exported outside the CryptoServer.

  Possible values:

  - `CK_TRUE` - a strong key (for example, 256-bit AES) cannot be encrypted with a weak key (for example, 1024-bit RSA).
  - `CK_FALSE` (default) - a strong key can be encrypted with a weak key.

- `CKA_CFG_AUTH_KEYM_MASK`

  This attribute defines the authentication status of the key manager who, by default, has the same permissions as the User (00000002).

  Default value: 0x00000002 - corresponds to the permission of the Cryptographic User, who is already set up in the CryptoServer.

  You can change this permission for the key manager here to 00000020, and split the User role into two roles: key user and key manager.

  IMPORTANT:
  If you change the default value, you must use the user management functions in CAT or csadm administration tools to set up a key manager in CryptoServer, who is assigned the permission 2 in the user group 1 corresponding to the authentication status 00000020 specified here. For step-by-step instructions on how to create a new user with the key manager role with CAT and P11CAT, please read Chapter "Creating a New User" in [CSMSADM], and Chapter "Changing the Global Configuration" in [CS_PKCS11CAT]. An example for creating a user with the key manager role with csadm is provided in Chapter "AddUser" of the [CSADMIN].

- `CKA_CFG_SECURE_DERIVATION`

*This security relevant attribute is only available as from SecurityServer 4.01 (CXI firmware module version 2.1.11.1).*

  This attribute prohibits the usage of the following key derivation mechanisms, and prevents Reduced Key Space attacks:

  - `CKM_XOR_BASE_AND_DATA`
  - `CKM_CONCATENATE_DATA_AND_BASE`
  - `CKM_CONCATENATE_BASE_AND_DATA`
  - `CKM_CONCATENATE_BASE_AND_KEY`
  - `CKM_EXTRACT_KEY_FROM_KEY`

  For a detailed description of the mechanisms see [PKCS11ICMS].

  Possible values:

  - `CK_TRUE` – none of the key derivation mechanisms listed above can be used by the function `C_Derive ()`.

- `CK_FALSE` (default) – the key derivation mechanisms listed above can be used by the function `C_Derive ()` for key derivation.

- `CKA_CFG_SECURE_IMPORT`

*This security relevant attribute is only available as from SecurityServer 4.01 (CXI firmware module version 2.1.11.1).*

This attribute prevents simple Key Extraction attacks by performing additional strict checks on wrapping keys.

Possible values:

- `CK_FALSE (default)` – no additional strict checks on wrapping keys are performed.
- `CK_TRUE` – the key wrapping and unwrapping functions perform the following additional strict checks on wrapping keys.
  - `C_CreateObject` checks that for public keys the attribute `CKA_WRAP` is set to `CK_FALSE`. If this check fails, the error code `B0680204` is written to the `cs_pkcs11_R2.log` logfile. This prevents wrapping with potentially untrustworthy keys, since we have no control over the corresponding private key.
  - `C_WrapKey` and `C_EncryptInit` prohibit the use of the `CKM_RSA_PKCS` mechanism. If `CKM_RSA_PKCS` is provided as key wrapping mechanism, the error code `B068002D` is written to the `cs_pkcs11_R2.log` logfile. This mitigates the Bleichenbacher Padding Oracle attack on wrapped keys.
  - `C_WrapKey`:

    Prohibits the use of public keys as wrapping keys. If the wrapping key is a public one, the error code `B0680205` is written to the `cs_pkcs11_R2.log` logfile. This prevents wrapping with potentially untrustworthy keys, since we have no control over the corresponding private key.

    Checks that for wrapping keys the attribute `CKA_DECRYPT` is set to `CK_FALSE`. B0680200 is written to the `cs_pkcs11_R2.log` logfile. This prevents simple Key Extraction attacks.

    Checks that for wrapping keys the attribute `CKA_ALWAYS_SENSITIVE` is set to `CK_TRUE`. If this check fails, the error code `B0680202` is written to the `cs_pkcs11_R2.log` logfile. This prevents wrapping with potentially untrustworthy keys.
  - `C_UnwrapKey`:

Checks that for unwrapping keys the `CKA_ENCRYPT` attribute is set to `CK_FALSE`. If this check fails, the error code `B0680201` is written to the `cs_pkcs11_R2.log` logfile. This prevents simple Key Extraction attacks.

Checks that templates used for unwrapping keys contain the attribute `CKA_CHECK_VALUE` (obtains its value through `C_GetAttributeValue` when exporting keys). The check value of the unwrapped components is then compared to the provided value. If this check fails, the error code `B0680206` is written to the `cs_pkcs11_R2.log` logfile. This checks the integrity of reimported keys to prevent Key Binding attacks and Unwrap Fault attacks.

Checks that for unwrapped keys the `CKA_WRAP` attribute is set to `CK_FALSE`. If this check fails, the error code `B0680204` is returned is written to the `cs_pkcs11_R2.log` logfile. This prevents wrapping with potentially untrustworthy keys.

Checks that for unwrapped keys the attribute `CKA_SENSITIVE` is set to `CK_TRUE`. If this check fails, the error code `B0680203` is returned is written to the `cs_pkcs11_R2.log` logfile. This prevents simple Key Extraction attacks.

*See chapter 4.2, "Logging" for details about how to configure the PKCS#11 API logfile (cs_pkcs11_R2.log).*

- `CKA_CFG_SECURE_RSA_COMPONENTS`

*This security relevant attribute is only available as from SecurityServer/CryptoServer SDK 4.01 (CXI firmware module version 2.1.11.1).*

This attribute applies restrictions on the length of the public exponent used for the generation of RSA keys.

Possible values:

- `CK_TRUE (default) –` new RSA keys cannot be created with very low, smaller than 0x10001, public exponents.
- `CK_FALSE –` new RSA keys can be created with very low public exponents.

- `CKA_CFG_P11R2_BACKWARDS_COMPATIBLE`

*This security relevant attribute is only available as from SecurityServer/CryptoServer SDK 4.01 (CXI firmware module version 2.1.11.1).*

This attribute determines whether keys can be used by default as base keys for key derivation or not.

Possible values:

- `CK_TRUE` – keys generated by using an ECC scheme or Diffie-Hellman algorithm can be used as base keys for key derivation (PKCS#11 standard non-compliant legacy); may be necessary for some integrations.
- `CK_FALSE` (default) –newly generated or imported keys cannot be used by default as base keys for key derivation.

- `CKA_CFG_ENFORCE_BLINDING`

*This security relevant attribute is only available as from SecurityServer/CryptoServer SDK 4.10 (CXI firmware module version 2.2.1.0).*

This attribute prevents side-channel analysis (SCA) attacks by enabling/disabling CryptoServer-specific software measures for SCA resistance. These software measures imply changing the internal computations of RSA and ECC keys in a way that simple and differential power analysis, as well as electromagnetic and timing analysis measurements on cryptographic keys do not reveal information any longer.

However, the measures for SCA resistance negatively affect the performance of the cryptographic operations on RSA and ECDSA keys. Therefore, they are disabled by default, and can be enabled, if necessary.

Possible values:

- `CK_TRUE - software measures for SCA resistance are used for` cryptographic operations on RSA and ECDSA keys.
- `CK_FALSE (default) - normal (without software measures for SCA resistance)` cryptographic operations on RSA and ECDSA keys are used.

- `CKA_CFG_SECURE_SLOT_BACKUP`

*This security relevant attribute is only available as from SecurityServer/CryptoServer SDK 4.10 (CXI firmware module version 2.2.1.0).*

This attribute enforces the usage of an individual backup key (Tenant Backup Key, TBK) per slot instead of the MBK to protect external keys and key backups. By default, only MBK-protected external key storage and key backup is enabled.

Possible values:

- `CK_TRUE` – use slot-individual backup keys (TBKs) derived from the CryptoServer's MBK to encrypt external keys and key backups.
- `CK_FALSE` (default) – use the CryptoServer's MBK to encrypt external keys and key backups

*Make sure you have set this configuration attribute according to your security policy before your CryptoServer production environment gets operational.*

*If you use SecurityServer/CryptoServer SDK 4.10, then create an external key or key backup by using an MBK, then enable the usage of slot-individual backup keys by setting the `CKA_CFG_SECURE_SLOT_BACKUP` configuration attribute to the `CK_TRUE` value, trying to restore the external key or key backup fails and the external key and key backups become inaccessible. The error message "invalid mac of key blob" (error code: 0xB0680026) is created.*
*This applies as well if you have upgraded to SecurityServer/CryptoServer SDK 4.20 or later before trying to restore the external key or key backups.*
*However, if you upgrade to SecurityServer/CryptoServer SDK 4.20 before creating the external key or key backup using an MBK, restoring them with a slot-individual backup key succeeds.*

To further individualize your slot-individual backup key, you can optionally define a slot specific passphrase to be used for the derivation of that backup key. This is done by setting the `CKA_CFG_SLOT_BACKUP_PASS_HASH` slot configuration attribute prior to enabling the usage of slot-individual backup keys with the `CKA_CFG_SECURE_SLOT_BACKUP` global configuration attribute set to `CK_TRUE`. See Chapter 11.5.3, "CryptoServer Slot Configuration

Objects", for a detailed description of the `CKA_CFG_SLOT_BACKUP_PASS_HASH` slot configuration attribute.

### 11.5.3 CryptoServer Slot Configuration Objects

The CryptoServer's slot configuration objects are used to configure the current slot. They are operative for all instances of the CryptoServer PKCS#11 library that are using the specific PKCS#11 slot. The handle they are referenced by is `P11_CFG_SLOT_HDL`.

These objects can only be changed by the SO of the slot after the global attribute `CKA_CFG_ALLOW_SLOTS` has been set to `CK_TRUE` by the default user ADMIN or a user(s) with permissions in the user group 7 (min. required authentication status is 20000000). The SO can configure all attributes previously described in chapter 11.5.2, except for the `CKA_CFG_ALLOW_SLOTS` attribute.

The CryptoServer slot configuration objects can be read by the SO, USER, key manager and key user (if configured as mentioned in chapter 9.2.9).

Changes on the attributes of a slot configuration object are stored in the database `CXIKEY.db`, which is deleted on alarm occurrence and when the `Clear` command (see Chapter "The Clear Functionality" in [CSADMIN]) is performed. We highly recommend to create a backup of the Slot CryptoServer Configuration Object with the P11CAT tool (see Chapter "Creating a Slot Configuration Backup" in the [CS_PKCS11CAT]) or the `p11tool2 BackupConfig` command (see [CS_PKCS11T2]) so you can easily restore your PKCS#11 slot configuration.

In addition to the attributes described in chapter 11.5.2, the `CKA_CFG_SLOT_BACKUP_PASS_HASH` slot-individual attribute can be configured in a slot configuration object. This attribute stores the SHA-256 hash value of a passphrase which is only used for the derivation of a slot-individual backup key (see `CKA_CFG_SECURE_SLOT_BACKUP` in chapter 11.5.2).

*If you want to use an individual passphrase for the derivation of the slot-individual backup key, make sure you have set the `CKA_CFG_SLOT_BACKUP_PASS_HASH` configuration attribute prior to the activation of the `CKA_CFG_SECURE_SLOT_BACKUP` attribute and before your CryptoServer production environment gets operational.*
*Changing the `CKA_CFG_SLOT_BACKUP_PASS_HASH` configuration attribute for a slot that is currently in use causes previously generated external keys and their backups to become inaccessible.*
*If you use SecurityServer/CryptoServer SDK 4.10 when creating the external key and their backups and you try to restore them with a changed individual passphrase, the error message "invalid mac of key blob" (error code: 0xB0680026) is created.*
*This applies as well if you have upgraded to SecurityServer/CryptoServer SDK 4.20 or later before trying to restore the external key or key backups.*
*However, if you upgrade to SecurityServer/CryptoServer SDK 4.20 before creating the external key or key backup and you try to restore them with a changed individual passphrase, the error message "wrong TBK passphrase for this key blob" (error code: 0xB0680081) is created.*

The `CKA_CFG_SLOT_BACKUP_PASS_HASH` attribute can be set by the SO to any string even if the `CKA_CFG_ALLOW_SLOTS` attribute is set to `CK_FALSE`. The default value is the SHA-256 hash of the empty string.

# 12 Supported Mechanisms and Function Mapping

This chapter contains an overview about the mechanisms currently supported by the CryptoServer PKCS#11 library.

## 12.1 PKCS#11 Defined Mechanisms

The following table shows the PKCS#11 standard mechanisms provided by the CryptoServer and the PKCS#11 API functions supporting them.

| *Mechanism* | *Functions* | | | | | | |
|---|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | SR & VR[1] | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_RSA_PKCS_OAEP | ✓[2] | | | | | ✓ | |
| CKM_RSA_PKCS_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_RSA_X9_31_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_RSA_PKCS | ✓[2] | ✓[2] | ✓ | | | ✓ | |
| CKM_RSA_PKCS_PSS | | ✓[2] | | | | | |
| CKM_RSA_X_509 | ✓[2] | ✓[2] | ✓ | | | | |
| CKM_RSA_X9_31 | | ✓[2] | | | | | |
| CKM_MD5_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA1_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA224_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA256_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA384_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA512_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA3_224_RSA_PKCS* | | ✓ | | | | | |
| CKM_SHA3_256_RSA_PKCS* | | ✓ | | | | | |
| CKM_SHA3_384_RSA_PKCS* | | ✓ | | | | | |
| CKM_SHA3_512_RSA_PKCS* | | ✓ | | | | | |
| CKM_RIPEMD160_RSA_PKCS | | ✓ | | | | | |
| CKM_SHA1_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA224_RSA_PKCS_PSS | | ✓ | | | | | |

| *Mechanism* | *Functions* | | | | | | |
|---|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | SR & VR[1] | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA256_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA384_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA512_RSA_PKCS_PSS | | ✓ | | | | | |
| CKM_SHA3_224_RSA_PKCS_PSS* | | ✓ | | | | | |
| CKM_SHA3_256_RSA_PKCS_PSS* | | ✓ | | | | | |
| CKM_SHA3_384_RSA_PKCS_PSS* | | ✓ | | | | | |
| CKM_SHA3_512_RSA_PKCS_PSS* | | ✓ | | | | | |
| CKM_SHA1_RSA_X9_31 | | ✓ | | | | | |
| CKM_DSA | | ✓[2] | | | | | |
| CKM_DSA_SHA1 | | ✓ | | | | | |
| CKM_DSA_SHA224 | | ✓ | | | | | |
| CKM_DSA_SHA256 | | ✓ | | | | | |
| CKM_DSA_SHA384 | | ✓ | | | | | |
| CKM_DSA_SHA512 | | ✓ | | | | | |
| CKM_DSA_SHA3_224* | | ✓ | | | | | |
| CKM_DSA_SHA3_256* | | ✓ | | | | | |
| CKM_DSA_SHA3_384* | | ✓ | | | | | |
| CKM_DSA_SHA3_512* | | ✓ | | | | | |
| CKM_DSA_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_DSA_PARAMETER_GEN | | | | | ✓ | | |
| CKM_DH_PKCS_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_DH_PKCS_DERIVE | | | | | | | ✓ |
| CKM_X9_42_DH_KEY_PAIR_GEN | | | | | ✓ | | |
| CKM_X9_42_DH_PKCS_PARAMETER_GEN | | | | | ✓ | | |
| CKM_X9_42_DH_DERIVE | | | | | | | ✓ |
| CKM_EC_KEY_PAIR_GEN (CKM_ECDSA_KEY_PAIR_GEN) | | | | | ✓ | | |
| CKM_ECDSA | | ✓[2] | | | | | |
| CKM_ECDSA_SHA1 | | ✓ | | | | | |

| *Mechanism* | *Functions* | | | | | | |
|---|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | SR & VR[1] | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_ECDSA_SHA224 | | ✓ | | | | | |
| CKM_ECDSA_SHA256 | | ✓ | | | | | |
| CKM_ECDSA_SHA384 | | ✓ | | | | | |
| CKM_ECDSA_SHA512 | | ✓ | | | | | |
| CKM_ECDH1_DERIVE | | | | | | | ✓ |
| CKM_ECDH1_COFACTOR_DERIVE | | | | | | | ✓ |
| CKM_GENERIC_SECRET_KEY_GEN | | | | | ✓ | | |
| CKM_AES_KEY_GEN | | | | | ✓ | | |
| CKM_AES_ECB | ✓ | | | | | ✓ | |
| CKM_AES_CBC | ✓ | | | | | ✓ | |
| CKM_AES_CBC_PAD | ✓ | | | | | ✓ | |
| CKM_AES_CTR | ✓ | | | | | | |
| CKM_AES_CCM | ✓ | | | | | ✓ | |
| CKM_AES_GCM | ✓ | | | | | ✓ | |
| CKM_AES_MAC_GENERAL | | ✓ | | | | | |
| CKM_AES_MAC | | ✓ | | | | | |
| CKM_AES_CMAC | | ✓ | | | | | |
| CKM_AES_GMAC* | | ✓ | | | | | |
| CKM_AES_OFB | ✓ | | | | | ✓ | |
| CKM_AES_KEY_WRAP | ✓[2] | | | | | ✓ | |
| CKM_AES_KEY_WRAP_PAD | ✓[2] | | | | | ✓ | |
| CKM_AES_KEY_WRAP_KWP | ✓[2] | | | | | ✓ | |
| CKM_DES_KEY_GEN | | | | | ✓ | | |
| CKM_DES_ECB | ✓ | | | | | ✓ | |
| CKM_DES_CBC | ✓ | | | | | ✓ | |
| CKM_DES_CBC_PAD | ✓ | | | | | ✓ | |
| CKM_DES_MAC_GENERAL | | ✓ | | | | | |
| CKM_DES_MAC | | ✓ | | | | | |

| *Mechanism* | *Functions* | | | | | | |
|---|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | SR & VR[1] | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_DES2_KEY_GEN | | | | | ✓ | | |
| CKM_DES3_KEY_GEN | | | | | ✓ | | |
| CKM_DES3_ECB | ✓ | | | | | ✓ | |
| CKM_DES3_CBC | ✓ | | | | | ✓ | |
| CKM_DES3_CBC_PAD | ✓ | | | | | ✓ | |
| CKM_DES3_MAC_GENERAL | | ✓ | | | | | |
| CKM_DES3_MAC | | ✓ | | | | | |
| CKM_DES_ECB_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_DES_CBC_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_DES3_ECB_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_DES3_CBC_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_AES_ECB_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_AES_CBC_ENCRYPT_DATA | | | | | | | ✓ |
| CKM_MD5 | | | | ✓ | | | |
| CKM_MD5_HMAC_GENERAL | | ✓ | | | | | |
| CKM_MD5_HMAC | | ✓ | | | | | |
| CKM_MD5_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA_1 | | | | ✓ | | | |
| CKM_SHA_1_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA_1_HMAC | | ✓ | | | | | |
| CKM_SHA1_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA224 | | | | ✓ | | | |
| CKM_SHA224_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA224_HMAC | | ✓ | | | | | |
| CKM_SHA224_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA256 | | | | ✓ | | | |
| CKM_SHA256_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA256_HMAC | | ✓ | | | | | |

| *Mechanism* | *Functions* | | | | | | |
|---|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | SR & VR[1] | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_SHA256_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA384 | | | | ✓ | | | |
| CKM_SHA384_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA384_HMAC | | ✓ | | | | | |
| CKM_SHA384_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA512 | | | | ✓ | | | |
| CKM_SHA512_HMAC_GENERAL | | ✓ | | | | | |
| CKM_SHA512_HMAC | | ✓ | | | | | |
| CKM_SHA512_KEY_DERIVATION | | | | | | | ✓ |
| CKM_SHA3_224* | | | | ✓ | | | |
| CKM_SHA3_224_HMAC_GENERAL* | | ✓ | | | | | |
| CKM_SHA3_224_HMAC* | | ✓ | | | | | |
| CKM_SHA3_224_KEY_DERIVATION* | | | | | | | ✓ |
| CKM_SHA3_256* | | | | ✓ | | | |
| CKM_SHA3_256_HMAC_GENERAL* | | ✓ | | | | | |
| CKM_SHA3_256_HMAC* | | ✓ | | | | | |
| CKM_SHA3_256_KEY_DERIVATION* | | | | | | | ✓ |
| CKM_SHA3_384* | | | | ✓ | | | |
| CKM_SHA3_384_HMAC_GENERAL* | | ✓ | | | | | |
| CKM_SHA3_384_HMAC* | | ✓ | | | | | |
| CKM_SHA3_384_KEY_DERIVATION* | | | | | | | ✓ |
| CKM_SHA3_512* | | | | ✓ | | | |
| CKM_SHA3_512_HMAC_GENERAL* | | ✓ | | | | | |
| CKM_SHA3_512_HMAC* | | ✓ | | | | | |
| CKM_SHA3_512_KEY_DERIVATION* | | | | | | | ✓ |
| CKM_RIPEMD160 | | | | ✓ | | | |
| CKM_RIPEMD160_HMAC_GENERAL | | ✓ | | | | | |
| CKM_RIPEMD160_HMAC | | ✓ | | | | | |

| *Mechanism* | *Functions* | | | | | | |
|---|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | SR & VR[1] | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| `CKM_XOR_BASE_AND_DATA` | | | | | | | ✓ |
| `CKM_CONCATENATE_BASE_AND_KEY` | | | | | | | ✓ |
| `CKM_CONCATENATE_BASE_AND_DATA` | | | | | | | ✓ |
| `CKM_CONCATENATE_DATA_AND_BASE` | | | | | | | ✓ |
| `CKM_EXTRACT_KEY_FROM_KEY` | | | | | | | ✓ |
| `CKM_UTI_AES_KEY_WRAP` | ✓[2] | | | | | ✓ | |
| `CKM_UTI_AES_KEY_WRAP_PAD` | ✓[2] | | | | | ✓ | |
| `CKM_UTI_AES_KEY_WRAP_KWP*` | ✓[2] | | | | | ✓ | |

Table 17: List of supported mechanisms defined in the PKCS#11 standard

1 SR = SignRecover, VR = VerifyRecover

2 Single-part operations only

3 Single-part sign operations only

4 Wrap only

* As specified in PKCS #11 Cryptographic Token Interface Current Mechanisms Specification 3.00 Draft Version

## 12.2 Vendor Defined Mechanisms

The following table shows the PKCS#11 mechanisms provided by the CryptoServer, which are not included in the PKCS#11 standard, and the PKCS#11 API functions supporting them.

| *Mechanism* | *Functions* | | | | | | |
|---|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | SR & VR[1] | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| `CKM_ECDSA_SHA3_224` | | ✓ | | | | | |
| `CKM_ECDSA_SHA3_256` | | ✓ | | | | | |
| `CKM_ECDSA_SHA3_384` | | ✓ | | | | | |
| `CKM_ECDSA_SHA3_512` | | ✓ | | | | | |

| *Mechanism* | *Functions* | | | | | | |
|---|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | SR & VR[1] | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| CKM_ECDSA_RIPEMD160 | | ✓ | | | | | |
| CKM_DSA_RIPEMD160 | | ✓ | | | | | |
| CKM_DES3_RETAIL_MAC | | ✓ | | | | | |
| CKM_RSA_PKCS_MULTI | | ✓[4, 5] | | | | | |
| CKM_RSA_X_509_MULTI | | ✓[4, 5] | | | | | |
| CKM_ECDSA_MULTI | | ✓[4, 5] | | | | | |
| CKM_DES_CBC_WRAP | | | | | | ✓ | |
| CKM_AES_CBC_WRAP | | | | | | ✓ | |
| CKM_ECKA | | ✓[4] | | | | | |
| CKM_ECDSA_ECIES | ✓[2] | | | | | | |

Table 18: List of vendor defined mechanisms

1 SR = SignRecover, VR = VerifyRecover

2 Single-part operations only

3 Mechanism can only be used for wrapping, not for unwrapping

4 Single-part sign operations only

5 Mechanism can only be used for signing, not for verification

## 12.3 Public Object Support

Currently only `CKK_RSA` and `CKK_EC` public objects (`CKA_PRIVATE == CK_FALSE`) are supported for the following operations:

```
C_GetAttributeValue
C_EncryptInit
C_Encrypt
C_EncryptUpdate
C_DecryptInit
C_Decrypt
C_DecryptUpdate
```

```
C_SignInit
C_Sign
C_SignUpdate
C_VerifyInit
C_Verify
C_VerifyUpdate
C_WrapKey
C_UnwrapKey
```

# 13 PKCS#11 API in FIPS Mode

In this chapter you find information about important restrictions applying when the CryptoServer is used in FIPS mode. Additionally, a list of mechanisms defined in the PKCS#11 standard as well as some vendor specific ones that are supported by the different cryptographic operations provided by the CryptoServer in FIPS mode is given.

The (validated) FIPS mode can only become active if the FIPS140 firmware module (`FLASH\fips140.msc`) is loaded into the CryptoServer, and if additionally the dedicated FIPS-validated firmware package has been loaded. The FIPS mode is set by the FIPS140 firmware module.

If the FIPS140 firmware module has not been loaded, the CXI module starts in normal mode.

The following features are not yet available in FIPS mode:

- Blinding configuration attribute CXI_PROP_CFG_ENFORCE_BLINDING
- SHA-3 algorithms
- AES CCM mode
- Tenant Backup Keys

## 13.1 Padding Mechanisms in FIPS Mode

If the CryptoServer is operating in FIPS mode, the use of one RSA key with multiple padding mechanisms is not allowed. Therefore, the additional `CKA_ALLOWED_MECHANISMS` key attribute containing all supported padding mechanisms must be set prior to the key generation. It is only allowed to choose one padding mechanism to be used with all supported hash algorithms. The supported hash mechanisms must be defined explicitly.

The following table shows which combinations of padding mechanisms and hash algorithms are supported and which constants represent them.

| *Combination of padding mechanism and hash algorithm* | | *Constant* |
|---|---|---|
| PKCS1 padding mechanism | | |
| | SHA-224 hashing algorithm | `CKM_SHA224_RSA_PKCS` |
| | SHA-256 hashing algorithm | `CKM_SHA256_RSA_PKCS` |
| | SHA-384 hashing algorithm | `CKM_SHA384_RSA_PKCS` |
| | SHA-512 hashing algorithm | `CKM_SHA512_RSA_PKCS` |

| *Combination of padding mechanism and hash algorithm* | | *Constant* |
|---|---|---|
| PSS padding | | |
| | SHA-224 hashing algorithm | `CKM_SHA224_RSA_PKCS_PSS` |
| | SHA-256 hashing algorithm | `CKM_SHA256_RSA_PKCS_PSS` |
| | SHA-384 hashing algorithm | `CKM_SHA384_RSA_PKCS_PSS` |
| | SHA-512 hashing algorithm | `CKM_SHA512_RSA_PKCS_PSS` |

Table 19: Combinations of padding mechanisms and hash algorithms

The constants in the table represent combinations of hash algorithm and a padding mechanism, e.g., `CKM_SHA256_RSA_PKCS` represents the combination of the SHA-256 hash algorithm and the PKCS1 padding mechanism. A combination is an or value of the corresponding hash algorithm constant and the padding mechanism constant.

When creating a key, one or several of the above combinations may be set as an attribute of the key. Use either one or several of the combinations with PKCS1 padding or one or several of the combinations with PSS padding but never use two or more combinations with different padding mechanisms.

Example:

```
CK_MECHANISM_TYPE mechs[] = {CKM_SHA256_RSA_PKCS, CKM_SHA384_RSA_PKCS};
```

You must always explicitly use at least one combination.

The following example shows how to generate and use an RSA key pair applying the PKCS1 padding mechanism (`CKM_RSA_PKCS`, i.e., RSASSA-PKCS-V1_5 from the [PKCS#1] standard) and two hash algorithms, SHA-256 and SHA-384, i.e., `CKM_SHA256_RSA_PKCS` and `CKM_SHA384_RSA_PKCS` are applied.

```
// Key generation
CK_MECHANISM_TYPE mechs[] = {CKM_SHA256_RSA_PKCS, CKM_SHA384_RSA_PKCS};

  CK_ATTRIBUTE publicKeyTemplate[] =
  {
    {CKA_TOKEN,           &bTrue,         sizeof(bTrue)},
    {CKA_VERIFY,          &bTrue,         sizeof(bTrue)},
    {CKA_ALLOWED_MECHANISMS,   &mechs,         sizeof(mechs)},
    {CKA_MODULUS_BITS,    &modulusBits,   sizeof(modulusBits)},
    {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}
  };

  CK_ATTRIBUTE privateKeyTemplate[] =
  {
    {CKA_TOKEN,       &bTrue,   sizeof(bTrue)},
    {CKA_PRIVATE,     &bTrue,   sizeof(bTrue)},
    {CKA_ID,          id,       sizeof(id)},
    {CKA_SENSITIVE,   &bTrue,   sizeof(bTrue)},
    {CKA_SIGN,        &bTrue,   sizeof(bTrue)},
    {CKA_ALLOWED_MECHANISMS,   &mechs,         sizeof(mechs)},
    {CKA_LABEL,       label,    sizeof(label)}
  };

mechanism.mechanism = CKM_RSA_PKCS_KEY_PAIR_GEN;
mechanism.pParameter = NULL;
mechanism.ulParameterLen = 0;

    if ((err = pFunctions->C_GenerateKeyPair(hSession, &mechanism,
                                          publicKeyTemplate,
sizeof(publicKeyTemplate)/sizeof(CK_ATTRIBUTE),
                                          privateKeyTemplate,
sizeof(privateKeyTemplate)/sizeof(CK_ATTRIBUTE),
                                          &hPublicKey, &hPrivateKey)) != 0)
  {
    printf("C_GenerateKeyPair returned 0x%08x\n", err);
    goto cleanup;
  }

// Key usage preparation
// If you want to use SHA-256 hash algorithm and the PKCS1 padding
// mechanism, use the following line:
mechanism.mechanism = CKM_SHA256_RSA_PKCS;

// If you want to use SHA-384 hash algorithm and the PKCS1 padding
// mechanism, use the following command instead:
// mechanism.mechanism = CKM_SHA384_RSA_PKCS;

mechanism.pParameter = NULL;
mechanism.ulParameterLen = 0;

// Key usage: Signing
if ((err = pFunctions->C_SignInit(hSession, &mechanism, hPrivateKey)) != 0)
    printf("C_SignInit returned 0x%08x\n", err);
signatureLength = sizeof(signature);
if ((err = pFunctions->C_Sign(hSession, Data, dataLength, signature,
                              &signatureLength)) != 0)
    printf("C_Sign returned 0x%08x\n", err);

// Key usage: Verifying
if ((err = pFunctions->C_VerifyInit(hSession, &mechanism, hPublicKey)) !=
0)
    printf("C_VerifyInit returned 0x%08x\n", err);


if ((err = pFunctions->C_Verify(hSession, Data, dataLength, signature,
                                signatureLength)) != 0)
    printf("C_Verify returned 0x%08x\n", err);
```

Consider that the `mechs` variable defined in the following line

```
CK_MECHANISM_TYPE mechs[] = {CKM_SHA256_RSA_PKCS, CKM_SHA384_RSA_PKCS};
```

has to be used for the public key and the private key as shown in the following lines. Otherwise, an error would be generated during the signing or verifying process..

```
  CK_ATTRIBUTE publicKeyTemplate[] =
  {
    //…
    {CKA_ALLOWED_MECHANISMS,   &mechs,         sizeof(mechs)},
    //…
  };

  CK_ATTRIBUTE privateKeyTemplate[] =
  {
    //…
    {CKA_ALLOWED_MECHANISMS,   &mechs,         sizeof(mechs)},
    //…
  };
```

If you want to use PSS padding instead, replace the following line in the key generation section

```
CK_MECHANISM_TYPE mechs[] = {CKM_SHA256_RSA_PKCS, CKM_SHA384_RSA_PKCS};
```

by the following line, if you want to use SHA-256

```
CK_MECHANISM mechs[] = {CKM_SHA256_RSA_PKCS_PSS};
```

and replace the key usage preparation section by the following lines:

```
// Key usage preparation
CK_RSA_PKCS_PSS_PARAMS pssParam;
pssParam.hashAlg = CKM_SHA256;
pssParam.mgf = CKG_MGF1_SHA256;
pssParam.sLen = 16;

mechanism.mechanism = CKM_SHA256_RSA_PKCS_PSS;
mechanism.pParameter = &pssParam;
mechanism.ulParameterLen = sizeof(pssParam);
```

In the above examples, the public key is used for verifying (`CKA_VERIFY`) and the private key is used for signing (`CKA_SIGN`). If you want to use for example the public key for encryption and the private key for decryption instead, exchange `CKA_VERIFY` by `CKA_ENCRYPT` and `CKA_SIGN` by `CKA_DECRYPT` in the above code. The rest of the code remains unchanged. Consider the restrictions described in Chapter 13.2, “Key Usage in FIPS Mode”.

## 13.2 Key Usage in FIPS Mode

Each time a DSA/DH/DH_PKCS, RSA or EC (ECDSA or ECDH) key is generated or imported, it must be checked that its usage attribute is exactly one of the `{CKA_SIGN; CKA_VERIFY}` usage bit group or exactly one of the `{CKA_ENCRYPT, CKA_DECRYPT, CKA_DERIVE, CKA_WRAP, CKA_UNWRAP}` usage bit group. I.e., if key pairs are used for signature generation and verification, they must not be used for any other purpose (see [FIPS186-4]). The `CKA_*` values are defined in the `pkcs11t.h` file.

The following applies:

- At least one of the usage bits of one of the groups is set, and all usage bits of the other group must be zero.
- If both usage bit groups contain bits that are set, the key generation or key import is rejected and the command returns an error (B0680109 "Key usage is restricted in FIPS mode").
- If key usage is not set, it is set to default `{CKA_SIGN; CKA_VERIFY}`.

Example for the `{CKA_SIGN; CKA_VERIFY}` usage bit group:

A private key is used for signing and a public key is used for verifying.

```
//…
  CK_ATTRIBUTE publicKeyTemplate[] =
  {
    //…
    {CKA_VERIFY,          &bTrue,         sizeof(bTrue)},
    //…
  };

  CK_ATTRIBUTE privateKeyTemplate[] =
  {
    //…
    {CKA_SIGN,        &bTrue,   sizeof(bTrue)},
    //…
  };
//…
```

Example for the `{CKA_ENCRYPT, CKA_DECRYPT, CKA_DERIVE, CKA_WRAP, CKA_UNWRAP}` usage bit group:

A public key is used for encrypting and a private key is used for decrypting.

```
//…
  CK_ATTRIBUTE publicKeyTemplate[] =
  {
    //…
    {CKA_ENCRYPT,          &bTrue,         sizeof(bTrue)},
    //…
  };

  CK_ATTRIBUTE privateKeyTemplate[] =
  {
    //…
    {CKA_DECRYPT,        &bTrue,   sizeof(bTrue)},
    //…
  };
//…
```

The rest of the code in these two examples may be set according to the example in Chapter 13.1, "Padding Mechanisms in FIPS Mode".

## 13.3 Mechanisms Supported in FIPS Mode for CryptoServer Se

The following table lists all mechanisms – defined in the PKCS#11 standard and the vendor specific ones – supported by the CryptoServer Se-Series if it is operated in FIPS mode.

| *Mechanism* | *Functions* | | | | | |
|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| *PKCS#11 Defined Mechanisms* | | | | | | |
| CKM_RSA_PKCS_OAEP | | | | | ✓[2] | |
| CKM_RSA_PKCS_KEY_PAIR_GEN | | | | ✓[2] | | |
| CKM_RSA_X9_31_KEY_PAIR_GEN | | | | ✓[2] | | |
| CKM_RSA_PKCS_PSS | | ✓[2] | | | | |
| CKM_RSA_X9_31 | | ✓[2] | | | | |
| CKM_SHA224_RSA_PKCS_PSS | | ✓[2] | | | | |
| CKM_SHA256_RSA_PKCS_PSS | | ✓[2] | | | | |
| CKM_SHA384_RSA_PKCS_PSS | | ✓[2] | | | | |
| CKM_SHA512_RSA_PKCS_PSS | | ✓[2] | | | | |
| CKM_EC_KEY_PAIR_GEN (CKM_ECDSA_KEY_PAIR_GEN) | | | | ✓[1] | | |
| CKM_ECDSA | | ✓[1] | | | | |
| CKM_ECDH1_COFACTPR_DERIVE | | | | | | ✓[1] |
| CKM_GENERIC_SECRET_KEY_GEN | | | | ✓ | | |
| CKM_AES_KEY_GEN | | | | ✓ | | |
| CKM_AES_ECB | ✓ | | | | ✓ | |
| CKM_DES2_KEY_GEN | | | | ✓[3] | | |
| CKM_DES3_KEY_GEN | | | | ✓[3] | | |
| CKM_DES_ECB | ✓[3] | | | | ✓[3] | |

| *Mechanism* | *Functions* | | | | | |
|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| *PKCS#11 Defined Mechanisms* | | | | | | |
| CKM_DES3_CBC | ✓[3] | | | | ✓[3] | |
| CKM_DES3_CBC_PAD | ✓[3] | | | | ✓[3] | |
| CKM_DES3_MAC_GENERAL | | ✓[3] | | | | |
| CKM_DES3_MAC | | ✓[3] | | | | |
| CKM_SHA224 | | | ✓ | | | |
| CKM_SHA224_HMAC_GENERAL | | ✓ | | | | |
| CKM_SHA224_HMAC | | ✓ | | | | |
| CKM_SHA256 | | | ✓ | | | |
| CKM_SHA256_HMAC_GENERAL | | ✓ | | | | |
| CKM_SHA256_HMAC | | ✓ | | | | |
| CKM_SHA384 | | | ✓ | | | |
| CKM_SHA384_HMAC_GENERAL | | ✓ | | | | |
| CKM_SHA384_HMAC | | ✓ | | | | |
| CKM_SHA512 | | | ✓ | | | |
| CKM_SHA512_HMAC_GENERAL | | ✓ | | | | |
| CKM_SHA512_HMAC | | ✓ | | | | |
| *Vendor Defined Mechanisms* | | | | | | |
| CKM_ECDSA_SHA224 | | ✓[1] | | | | |
| CKM_ECDSA_SHA25 | | ✓[1] | | | | |
| CKM_ECDSA_SHA384 | | ✓[1] | | | | |
| CKM_ECDSA_SHA512 | | ✓[1] | | | | |
| CKM_AES_CMAC | | ✓ | | | | |
| CKM_RSA_PKCS_MULTI | | ✓[2] | | | | |
| CKM_RSA_X_509_MULTI | | ✓[2] | | | | |
| CKM_ECKA | | ✓[1] | | | | |
| CKM_AES_OFB | ✓ | | | | ✓ | |

✓ The mechanism is available in FIPS mode.

1 NIST approved curves allowed for ECDSA and ECDH: P-192, P-224, P-256, P-384, P-521, K-163, K-233, K-283, K-409, K-571, B-163, B-233, B-283, B-409, B-571 (see [FIPS186-2], Appendix 6)

2 In FIPS mode the key length of an RSA key must be min. 1024 bit.

3 Only DES keys with key length of min. 112 bit are supported.

## 13.4 Mechanisms Supported in FIPS Mode for CryptoServer CSe and Se Gen2

The following table lists all mechanisms – defined in the PKCS#11 standard and the vendor specific ones – supported by the CryptoServer CSe-Series and Se-Series Gen2 if it is operated in FIPS mode.

| *Mechanism* | *Functions* | | | | | |
|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| *PKCS#11 Defined Mechanisms* | | | | | | |
| CKM_RSA_PKCS | | | | | ✓8, 9 | |
| CKM_RSA_PKCS_OAEP | | | | | ✓8, 9 | |
| CKM_RSA_PKCS_KEY_PAIR_GEN | | | | ✓1, 4 | | |
| CKM_RSA_X9_31_KEY_PAIR_GEN | | | | ✓1, 4 | | |
| CKM_RSA_X9_31 | | ✓4, 7 | | | | |
| CKM_SHA1_RSA_PKCS | | ✓7, 10 | | | | |
| CKM_SHA224_RSA_PKCS | | ✓4, 7 | | | | |
| CKM_SHA256_RSA_PKCS | | ✓4, 7 | | | | |
| CKM_SHA384_RSA_PKCS | | ✓4, 7 | | | | |
| CKM_SHA512_RSA_PKCS | | ✓4, 7 | | | | |
| CKM_SHA1_RSA_PKCS_PSS | | ✓7, 10 | | | | |
| CKM_SHA224_RSA_PKCS_PSS | | ✓4, 7 | | | | |
| CKM_SHA256_RSA_PKCS_PSS | | ✓4, 7 | | | | |
| CKM_SHA384_RSA_PKCS_PSS | | ✓4, 7 | | | | |
| CKM_SHA512_RSA_PKCS_PSS | | ✓4, 7 | | | | |
| CKM_DSA | | ✓11, 12 | | | | |

| *Mechanism* | *Functions* | | | | | |
|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| *PKCS#11 Defined Mechanisms* | | | | | | |
| CKM_DSA_SHA1 | | ✓[10, 11, 12] | | | | |
| CKM_DSA_SHA224 | | ✓[11, 12] | | | | |
| CKM_DSA_SHA256 | | ✓[11, 12] | | | | |
| CKM_DSA_SHA384 | | ✓[11, 12] | | | | |
| CKM_DSA_SHA512 | | ✓[11, 12] | | | | |
| CKM_DSA_KEY_PAIR_GEN | | | | ✓[11, 12] | | |
| CKM_DSA_PARAMETER_GEN | | | | ✓[11, 12] | | |
| CKM_EC_KEY_PAIR_GEN (CKM_ECDSA_KEY_PAIR_GEN) | | | | ✓[2] | | |
| CKM_ECDSA | | ✓[2, 3] | | | | |
| CKM_ECDH1_COFACTPR_DERIVE | | | | | | ✓[2, 11] |
| CKM_GENERIC_SECRET_KEY_GEN | | | | ✓ | | |
| CKM_AES_KEY_GEN | | | | ✓ | | |
| CKM_AES_ECB | ✓ | | | | ✓ | |
| CKM_AES_CBC | ✓ | | | | ✓ | |
| CKM_AES_CBC_PAD | ✓ | | | | ✓ | |
| CKM_AES_CMAC | | ✓ | | | ✓ | |
| CKM_AES_KEY_WRAP | ✓[2] | | | | ✓ | |
| CKM_AES_KEY_WRAP_PAD | ✓[2] | | | | ✓ | |
| CKM_AES_KEY_WRAP_KWP | ✓[2] | | | | ✓ | |
| CKM_DES3_KEY_GEN | | | | ✓[5] | | |
| CKM_DES_ECB | ✓[5, 6] | | | | ✓[5, 6] | |
| CKM_DES3_CBC | ✓[5, 6] | | | | ✓[5, 6] | |
| CKM_DES3_CBC_PAD | ✓[5, 6] | | | | ✓[5, 6] | |
| CKM_DES3_MAC | | ✓[5, 6] | | | | |
| CKM_DH_PKCS_DERIVE | | | | | | ✓[11] |
| CKM_X9_42_DH_DERIVE | | | | | | ✓[11] |
| CKM_SHA224 | | | ✓ | | | |

| *Mechanism* | *Functions* | | | | | |
|---|---|---|---|---|---|---|
| | Encrypt & Decrypt | Sign & Verify | Digest | Gen. Key/Key Pair | Wrap & Unwrap | Derive |
| *PKCS#11 Defined Mechanisms* | | | | | | |
| CKM_SHA224_HMAC_GENERAL | | ✓ | | | | |
| CKM_SHA224_HMAC | | ✓ | | | | |
| CKM_SHA256 | | | ✓ | | | |
| CKM_SHA256_HMAC_GENERAL | | ✓ | | | | |
| CKM_SHA256_HMAC | | ✓ | | | | |
| CKM_SHA384 | | | ✓ | | | |
| CKM_SHA384_HMAC_GENERAL | | ✓ | | | | |
| CKM_SHA384_HMAC | | ✓ | | | | |
| CKM_SHA512 | | | ✓ | | | |
| CKM_SHA512_HMAC_GENERAL | | ✓ | | | | |
| CKM_SHA512_HMAC | | ✓ | | | | |
| *Vendor Defined Mechanisms* | | | | | | |
| CKM_ECDSA_SHA1 | | ✓[2, 3 , 10] | | ✓[2] | | |
| CKM_ECDSA_SHA224 | | ✓[2, 3] | | ✓[2] | | |
| CKM_ECDSA_SHA25 | | ✓[2, 3] | | ✓[2] | | |
| CKM_ECDSA_SHA384 | | ✓[2, 3] | | ✓[2] | | |
| CKM_ECDSA_SHA512 | | ✓[2, 3] | | ✓[2] | | |
| CKM_AES_CMAC | | ✓ | | | | |
| CKM_RSA_PKCS_MULTI | | ✓[4] | | | | |
| CKM_RSA_X_509_MULTI | | ✓[4] | | | | |
| CKM_ECKA | | ✓[2] | | | | |
| CKM_AES_CBC_WRAP | | | | | ✓ | |
| CKM_AES_OFB | ✓ | | | | ✓ | |

✓ The mechanism is available in FIPS mode.

1 Restrictions on RSA padding mechanisms as described above in chapter 13.1.

2 NIST approved curves allowed for EC key generation, ECDSA signing and ECDH key derivation: P-224, P-256, P-384, P-521, K-233, K-283, K-409, K-571, B-233, B-283, B-409, B-571 (see [FIPS186-4], Appendix 6)

3 NIST approved curves allowed for ECDSA signature verification: P-192, P-224, P-256, P-384, P-521, K-163, K-233, K-283, K-409, K-571, B-163, B-233, B-283, B-409, B-571 (see [FIPS186-2], Appendix 6)

4 Only RSA key with length of 2048 or 3072 bits is allowed for key and signature generation.

5 For DES key generation, encryption, MAC generation and key wrapping only key length of 24 byte is supported.

6 For DES decryption, MAC verification and key unwrapping only key length of 24 byte is supported.

7 For RSA signature verification only RSA key length of min. 1024 bit is allowed.

8 For RSA key wrapping only RSA key length of min. 2048 bit is allowed.

9 For RSA key unwrapping only RSA key length of min. 1024 bit is allowed.

10 The mechanism is only allowed for signature verification.

11 For key and signature generation and key derivation only the following parameter length pairs are allowed: |P|/|Q| = 2048/224, 2048/256 or 3072/256

12 For DSA signature verification, a |P|/|Q| parameter length of min. 1024/160 is allowed.

## References

| *Reference* | *Title/Company* | *Document No.* |
|---|---|---|
| [ANSI-X9.19] | ANSI X9.19: Financial Institution Retail Message Authentication, 1996/ANSI (American National Standards Institute) | |
| [ANSI-X9.63] | ANSI X9.63: Public Key Cryptography for the Financial Services Industry - Key Agreement and Key Transport using Elliptic Curve Cryptography, 2001/ANSI (American National Standards Institute). | |
| [CSADMIN] | CryptoServer – csadm Manual /Utimaco IS GmbH. | 2009-0003 |
| [CSMSADM] | CryptoServer – Administration Manual /Utimaco IS GmbH. | M010-0001-en |
| [CS_PKCS11CAT] | CryptoServer – PKCS#11 P11CAT - Manual /Utimaco IS GmbH. | M013-0001-en |
| [CS_PKCS11T2] | CryptoServer – PKCS#11 p11tool2 – Reference Manual/Utimaco IS GmbH. | 2012-0014 |
| [FIPS186-2] | FIPS PUB 186-2, Digital Signature Standard/National Institute of Standards and Technology (NIST), January 2000. | |
| [FIPS186-4] | FIPS PUB 186-4, Digital Signature Standard/National Institute of Standards and Technology (NIST), July 2013. | |
| [ISO-9797] | ISO/IEC 9797-1:1999 - Information technology -- Security techniques -- Message Authentication Codes (MACs) -- Part 1: Mechanisms using a block cipher/International Organization for Standardization, Geneva, Switzerland. | |
| [PKCS#3] | PKCS#3: Diffie-Hellman Key Agreement Standard v1.4, November 1, 1993/RSA Laboratories. Available: http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-3-diffie-hellman-key-agreement-standar.htm | |
| [PKCS11] | PKCS#11: Cryptographic Token Interface Standard v2.20, June 28, 2004/RSA Laboratories. Available: http://www.emc.com/emc-plus/rsa-labs/standards- | |

| *Reference* | *Title/Company* | *Document No.* |
|---|---|---|
| | initiatives/pkcs-11-cryptographic-token-interface-standard.htm | |
| [PKCS11BS] | "PKCS #11 Cryptographic Token Interface Base Specification Version 2.40," Committee Specification 01, September 16, 2014/OASIS Standard. Available: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/cs01/pkcs11-base-v2.40-cs01.html | |
| [PKCS11ICMS] | "PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40," Committee Specification 01, September 16, 2014/OASIS Standard. Available: http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/cs01/pkcs11-curr-v2.40-cs01.html | |
| [PKCS#1] | PKCS#1: RSA Cryptography Standard v2.1, June 14, 2002/RSA Laboratories. Available: http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-rsa-cryptography-standard.htm | |
| [SEC1] | SEC1: Elliptic Curve Cryptography – Certicom Research – May 21, 2009, Version 2.0. | |