

CryptoServer

PKCS#11 R3

Developer Guide



Imprint

Copyright 2024	Utimaco IS GmbH Germanusstr. 4 D-52080 Aachen Germany
Phone	AMERICAS +1-844-UTIMACO (+1 844-884-6226) EMEA +49 800-627-3081 APAC +81 800-919-1301
Internet e-mail	https://support.hsm.utimaco.com/ support@utimaco.com
Document Version	1.3.18
Product Version	6.0.0
Date	2024-10-18
Document No.	2012-0007
Status	PUBLISHED

All rights reserved	<p>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.</p> <p>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. Any mention of the company name Utimaco in this documents refers to the Utimaco IS GmbH.</p> <p>All trademarks and registered trademarks are the property of their respective owners.</p>
---------------------	--

Table of Contents

1	Introduction	5
1.1	About this Document.....	5
1.1.1	Target Audience for this Manual	5
1.1.2	Document Conventions	5
1.2	Recommended Reading.....	6
2	The PKCS#11 R3 Interface - Overview	7
3	Setup and Requirements	9
3.1	Required Firmware Package.....	9
3.2	Location of the Configuration File cs_pkcs11_R3.cfg	10
4	Configuration	12
4.1	The Parameter Device	16
4.2	Logging.....	18
5	Operating Modes.....	20
5.1	Load Balancing Modes	20
5.2	Failover Mode.....	22
6	Internal and External Key Storage	24
7	Development of a PKCS#11 Application	25
7.1	Libraries.....	25
8	Runtime	27
8.1	Initialization.....	27
8.2	Limited Data Length	27
8.2.1	Key Wrapping with AES GCM/CCM	27
8.2.2	Initialization Vector Length for AES GCM	28
8.2.3	Data Length for Key Wrapping with AES GCM/CCM	28
8.3	Multithreading	28
9	Authentication Concept.....	30
9.1	Standard Authentication Concept	30
9.2	Enhanced Authentication Concept	34
9.2.1	Create Users with Other Authentication Mechanisms	34
9.2.2	Login User with Other Authentication Mechanisms	36
9.2.3	Change PIN for Other Authentication Mechanisms	38
9.2.4	Authentication via Configuration File	39

9.2.5	Automatic Login of Administrator via Configuration File	42
9.2.6	Authentication According to the Two-Person Rule	43
9.2.6.1	Extended Login.....	44
9.2.7	Separated Key Manager and Key User Role	45
10	Vendor Defined PKCS#11 Extensions	47
10.1	CryptoServer Defined Mechanisms	47
10.2	Handling CKM_ECKA	49
10.3	Encryption with the “Elliptic Curve Integrated Encryption Scheme” (ECIES)	50
10.4	Sign and Verify Using the DES Retail-MAC	51
10.5	Multiple Signature Mechanisms	52
10.6	Configuration Objects.....	53
10.6.1	Local Configuration Object	53
10.6.2	Global CryptoServer Configuration Object.....	54
10.6.3	CryptoServer Slot Configuration Objects.....	61
10.7	Switching from the DCC VDM Module to the Updated GBCS Integration	62
10.8	OSCCA Module.....	63
11	Key Management Functions and Cryptographic Operations in PKCS#11	68
12	Supported Mechanisms and Function Mapping	72
12.1	PKCS#11 Defined Mechanisms, Functions and Key Types	72
12.1.1	PKCS #11 Functions.....	90
12.1.2	PKCS #11 Key Types	95
12.2	Vendor Defined Mechanisms.....	98
12.3	Public Object Support.....	98
13	PKCS#11 API in FIPS Mode	99
13.1	Padding Mechanisms in FIPS Mode	99
13.2	Key Usage in FIPS Mode.....	100
13.3	Mechanisms Supported in FIPS Mode for CryptoServer CSe and Se Gen2	100
13.4	Mechanisms Supported in FIPS 140-3 Mode	107
14	References	113

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 3, as provided by Utimaco's hardware security module CryptoServer with SecurityServer version 4.40 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 Document Conventions

We use the following document conventions:

Convention	Use	Example
Bold	Items of the Graphical User Interface (GUI), e.g., menu options	Press OK
<code>Monospaced</code>	Code that is given for explanation or as an example, file paths	<code>chsm-create</code>
<i>Italic</i>	References and important terms	See <i>Sample Chapter</i> in the <i>CryptoServer - Sample Manual</i>

Table 1: Document conventions

We use special 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.



This message marks the result expected after the successful execution of an instruction.

1.2 Recommended Reading

We highly recommend to read also [\[CS_PKCS11HD \(p. 113\)\]](#) (PKCS11_HandsOn.pdf) provided on the SecurityServer product CD in the same directory as this manual:

`\Documentation\Crypto_APIS\PKCS11_R3` . There you can learn how to develop PKCS#11 applications.

2 The PKCS#11 R3 Interface - Overview

PKCS#11 is a general purpose Public Key Cryptography Standard originally developed by RSA Security [\[PKCS11\]](#) (p. 113) and currently maintained by the OASIS PKCS11 Technical Committee. It defines an interface between an application and a cryptographic device.

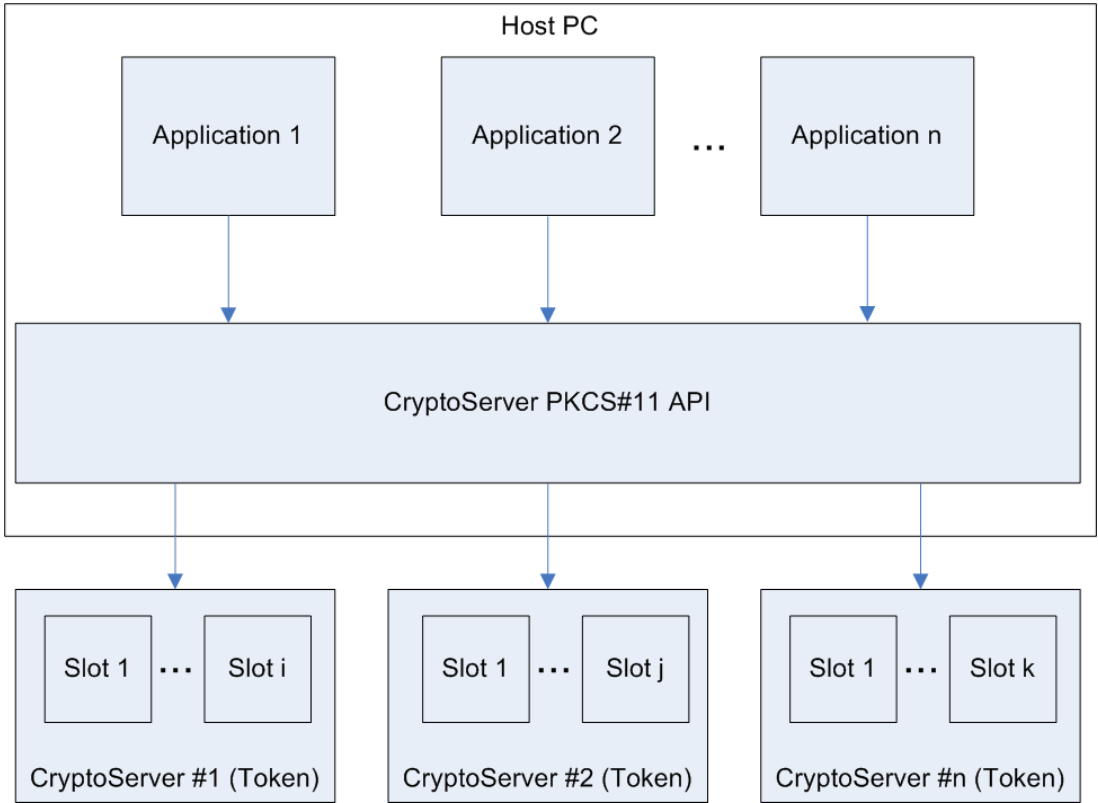
The CryptoServer provides a PKCS#11 interface. To use this interface a dedicated firmware package including the CXI firmware module must be loaded into the CryptoServer. Additionally, it is required that the PKCS#11 application is linked against the CryptoServer PKCS#11 library or it must be able to 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 maximum number of PKCS#11 sessions that the PKCS#11 R3 provider can handle is determined by the maximum number of Secure Messaging connections that a CryptoServer HSM supports. See section *Secure Messaging* in the *CryptoServer - csadm Manual* for the maximum number of Secure Messaging connections (session keys). Consider that the number of Secure Messaging connections, and thus the number of PKCS#11 sessions, sums up in case that multiple applications open PKCS#11 sessions to the HSM. For example, 5 applications opening PKCS#11 sessions to the HSMs may in total open the supported maximum number of Secure Messaging connections.

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 PCIe 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 Setup and 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) - or the CryptoServer Simulator is installed and running.
- 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 section [Required Firmware Package \(p. 9\)](#)).
- 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_R3.cfg` is configured correctly (see [Configuration \(p. 12\)](#)) and the CryptoServer PKCS#11 library is able to find and access it (see [Location of the Configuration File cs_pkcs11_R3.cfg \(p. 10\)](#))

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 the *CryptoServer - csadm Manual* or 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 directory `...\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 (product CD version 4.40 or higher) and `<Version>` describes the version of the package. To load the corresponding firmware package into the CryptoServer, see sections *Commands for Firmware Management* and *Commands for Firmware Packaging* in the *CryptoServer - csadm Manual* 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_R3.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_R3_CFG environment variable to the correct path and location of the configuration file.

Example for a Windows system:

```
#> set CS_PKCS11_R3_CFG="C:\My Documents\utimaco\cs_pkcs11_R3.cfg"
```

Example for a Linux system:

```
#> CS_PKCS11_R3_CFG=~/.utimaco/cs_pkcs11_R3.cfg
```

```
#> export CS_PKCS11_R3_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_R3.cfg` configuration file in the following order:

On Windows Operating Systems

1. First it checks if the `CS_PKCS11_R3_CFG` environment variable is set and if it contains the name and location of the configuration file.



The CS_PKCS11_R3_CFG environment variable is set to the default name and location of the configuration file during the installation of the product CD: C:\ProgramData\Utimaco\PKCS11_R3\cs_pkcs11_R3.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 checks if the `CS_PKCS11_R3_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_R3.cfg`).
3. If not it checks if the configuration file is located in the current working directory.
4. If not, it checks 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`

On other Unix Operating Systems

1. First it checks if the `CS_PKCS11_R3_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_R3.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 following 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_R3.cfg` configuration file. For details about the location of the configuration file, see [Location of the Configuration File cs_pkcs11_R3.cfg \(p. 10\)](#). 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.
- `[KeyStorage]` – (optional) section for specifying an external keystore

The following table gives an overview of all parameter that can be configured in the `cs_pkcs11_R3.cfg` configuration file.

<i>Parameter</i>	<i>Description</i>
Logging	Specifies the log level 0, 1, 2, 3 or 4 (see Logging (p. 18) details).
Logpath	Specifies the path where the logfile shall be created. In case of <code>/tmp</code> 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 <code>Logsize</code> 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. The external key storage is specified by <code>KeyStorageType</code> and <code>KeyStorageConfig</code> .
KeyStore	Deprecated as of SecurityServer/CryptoServer SDK 4.40.0. It has been replaced by <code>KeyStorageConfig</code> in the <code>[KeyStorage]</code> section. Specifies the path to the external keystore file (e.g. <code>C:\ProgramData\Utlimaco\PKCS11_R3\P11.sdb</code>). Network (UNC) paths like <code>\\network\path\to\P11.sdb</code> are supported as well. This parameter must be set if <code>KeysExternal = true</code> .
SlotMultiSession	Specifies session connection behavior. If <code>true</code> , every session establishes its own connection to the CryptoServer.

Parameter	Description
SlotLoginRestriction	If <code>false</code> , multiple users are allowed to log in as one session, leading to the user permissions being added up. The <code>SlotLoginRestriction</code> parameter in the <code>[Slot]</code> section overwrites the <code>SlotLoginRestriction</code> parameter in the <code>[Global]</code> section.
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.
FallbackInterval	Configures load balancing mode (<code>FallbackInterval = 0</code> , default) or failover mode (<code>FallbackInterval > 0</code>). Time interval [s] after which a reconnection attempt to the primary CryptoServer should be started.
KeepAlive	Keep sessions alive and prevent them from expiring after 15 minutes idle time (<code>true</code> or <code>false</code>). The <code>KeepAlive</code> parameter only works for Secure Messaging sessions.
Device	Device address to connect a CryptoServer device (see The Parameter Device (p. 16))
ConnectionTimeout	Specifies the maximum time in milliseconds to wait before the connection establishment is aborted if the device is not responding. In practice, the timeout can reach approximately $2 \cdot n \cdot T$. Legend: - n: Number of HSMs specified by the <code>Device</code> parameter - T: The timeout value specified by the <code>ConnectionTimeout</code> parameter.
CommandTimeout	Specifies the maximum time in milliseconds to wait for the answer from CryptoServer after sending a command. In practice, the timeout can reach approximately $2 \cdot n \cdot T$. Legend: - n: Number of HSMs specified by the <code>Device</code> parameter - T: The timeout value specified by the <code>CommandTimeout</code> parameter.
SlotNumber	Number of the slot to be configured
<username>	Defines the authentication mechanism of the user with name <username> (see Authentication via Configuration File (p. 39))
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

Parameter	Description
KeyStorageType	Specifies the type of an external keystore. <code>KeyStorageType</code> must be set if <code>KeysExternal = true</code> . The value of <code>KeyStorageType</code> is either ODBC or Legacy (default). <code>KeyStorageType</code> is only available as of SecurityServer/CryptoServer SDK 4.40.0.
KeyStorageConfig	<ul style="list-style-type: none"> If <code>KeyStorageType = ODBC</code>: <code>KeyStorageConfig</code> must be the server configuration, e.g., <code>DSN=Key Store</code>. If <code>KeyStorageType = Legacy</code>: <code>KeyStorageConfig</code> specifies the path to the external keystore file (SDB file) <ul style="list-style-type: none"> Windows E.g., <code>C:\ProgramData\Utimaco\PKCS11_R3\P11.sdb</code>. Network (UNC) paths like <code>\\network\path\to\P11.sdb</code> are supported as well. Linux E.g., <code>/tmp/P11.sdb</code> <p><code>KeyStorageConfig</code> must be set if <code>KeysExternal = true</code>. <code>KeyStorageConfig</code> is only available as of SecurityServer/CryptoServer SDK 4.40.0.</p>
KeyStorageReconnect	<p>A client application may connect to the PKCS#11 provider to perform key operations in an external key storage. The PKCS#11 provider supports the usage of an external key storage by using an ODBC provider. When the connection from the PKCS#11 provider to the ODBC provider is lost, the PKCS#11 provider tries to re-establish the connection.</p> <p><code>KeyStorageReconnect</code> specifies how many reconnection attempts are performed in case of a connection loss to the ODBC provider. The default value is <code>0</code>.</p> <p>To make the setting of <code>KeyStorageReconnect</code> valid, the following settings must be performed as well: <code>KeysExternal = true</code> and <code>KeyStorageType = ODBC</code>.</p> <p><code>KeyStorageReconnect</code> is supported as of SecurityServer/CryptoServer SDK 4.60.</p> <p>Each failed reconnection attempt initiates an error message.</p>
MultiInitReturnsCKR_OK	If set to <code>true</code> (default <code>false</code>) applications not strictly following the PKCS#11 standard can call <code>C_Initialize</code> and <code>C_Finalize</code> multiple times without the provider returning error <code>CKR_CRYPTOKI_ALREADY_INITIALIZED</code> or finalizing at the first <code>C_Finalize</code> call.

Parameter	Description
ForceOSLocking	If set to <code>true</code> (default <code>false</code>) the PKCS#11 provider is forced to apply internal locking using operating system primitives independent of the parameters passed to the <code>C_Initialize</code> call by applications not strictly following the PKCS#11 standard.

Table 2: Configuration parameter in the `cs_pkcs11_R3.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
Logging	<code>[Global]</code>	0 (NONE)
Logpath	<code>[Global]</code>	No default value (no logfile is created per default).
Logsize	<code>[Global]</code>	1 000 000
KeysExternal	<code>[Global]</code> , <code>[CryptoServer]</code> , <code>[Slot]</code>	false
KeyStore (Deprecated)	<code>[Global]</code>	No default path (no external key storage is created per default)
SlotLoginRestriction	<code>[Global]</code> , <code>[Slot]</code>	true
SlotCount	<code>[Global]</code> , <code>[CryptoServer]</code>	10
FallbackInterval	<code>[Global]</code>	0
KeepAlive	<code>[Global]</code> , <code>[CryptoServer]</code> , <code>[Slot]</code>	false
Device	<code>[CryptoServer]</code>	No default value (value must be defined)

Parameter	Allowed Sections	Default Value
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
KeyStorageType	[KeyStorage]	Legacy
KeyStorageConfig	[KeyStorage]	No default path (no external key storage is created per default)
KeyStorageReconnect	[KeyStorage]	0
MultiInitReturnsCKR_OK	[Global]	false
ForceOSLocking	[Global]	false

Table 3: Default settings for the parameters in the cs_pkcs11_R3.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 where n = {0, 1, 2, ..., 7} /dev/cs2.n.m where n = 0 and m = {1, 2, ..., 12}	Local CryptoServer No. n+1 on a UNIX system. The maximum number of eight CryptoServer PCIe cards can be changed in the source of the Linux driver. n+1: No. of local u.trust Anchor PCIe card on a UNIX system. m: No. of cHSM on a local u.trust Anchor PCIe card on a UNIX system.

Address	Description
PCI:n where n = {0, 1, 2, ..., 31}	Local CryptoServer No. n+1 on a Windows system (not available for u.trust Anchor)
TCP:288@194.168.4.107	IP address and port number of a u.trust Anchor/CryptoServer LAN 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 u.trust Anchor/CryptoServer LAN (default: port=288) 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 u.trust Anchor/CryptoServer LAN (default: protocol=TCP, port=288) In commands, always use IP addresses without leading zeros although they are shown in the CryptoServer LAN display, e.g., 194.168.004.107. When working on a cHSM (containerized Hardware Security Module) on a u.trust Anchor instead of working on a CryptoServer, Device = 4006@192.168.11.215 is an example of the Device parameter of a cHSM with 4006 indicating the port representing the cHSM slot 6. Do not confuse cHSM slots with PKCS slots. For details, see the u.trust Anchor documentation.
TCP:288@cslan01	Host name and port number of a u.trust Anchor/CryptoServer LAN (using DNS request to resolve host name)
TCP:cslan01	Host name of a u.trust Anchor/CryptoServer LAN (using DNS request to resolve host name, default: port=288)
cslan01	Host name of a u.trust Anchor/CryptoServer LAN (using DNS request to resolve host name, default: protocol=TCP, port=288)
TCP:3001@127.0.0.1 or 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 chapter <i>CryptoServer Simulator</i> in the <i>CryptoServer - Administration Manual</i> for further details. No simulator for u.trust Anchor.
3001@127.0.0.1 or 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 chapter <i>CryptoServer Simulator</i> in the <i>CryptoServer - Administration Manual</i> for further details. No simulator for u.trust Anchor.

Table 4: Examples for setting the Device parameter

Example of the configuration file `cs_pkcs11_R3.cfg`:

```

[Global]
Logging = 3
Logpath = c:/tmp
Logsize = 10mb
KeysExternal = true
KeepAlive = true
ConnectionTimeout = 7000
CommandTimeout = 70000
SlotLoginRestriction = true
# CryptoServer is a CryptoServer LAN
[CryptoServer]
Device = 192.168.4.137
CommandTimeout = 80000
SlotCount = 3
# first slot
[Slot]
SlotNumber = 0
SlotLoginRestriction = true
KeysExternal = true
[KeyStorage]
KeyStorageType = Legacy
KeyStorageConfig = C:/ProgramData/Utlimaco/PKCS11_R3/P11.sdb

# Number of reconnection attempts to ODBC database
#KeyStorageReconnect = 0

```



When working on a cHSM (containerized Hardware Security Module) on a u.trust Anchor instead of working on a CryptoServer, the following applies:

Do not confuse cHSM slots with PKCS slots. In the example, `SlotCount`, `[Slot]` and `SlotNumber` indicate a PKCS slot while `Device = 4006@192.168.11.215` is an example of the `Device` parameter of a cHSM with `4006` indicating the port representing the cHSM slot 6. For details, see the u.trust Anchor documentation.

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 section *Clustering for Load Balancing and Failover* in the *CryptoServer - Administration Manual* 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 section *Clustering for Load Balancing and Failover* in the *CryptoServer - Administration Manual* (provided on the delivered product CD in the directory ...\\Documentation\\Administration Guides) are also fulfilled.

5.1 Load Balancing Modes

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_R3.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_R3.cfg` configuration file in load balancing mode could look for example as follows:

```
[Global]
Logging = 3
Logpath = C:/tmp
Logsize = 10mb
KeysExternal = true
# 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
[KeyStorage]
KeyStorageType = Legacy
KeyStorageConfig = C:/ProgramData/Utimaco/PKCS11_R3/P11.sdb

# Number of reconnection attempts to ODBC database
#KeyStorageReconnect = 0
```

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 next free CryptoServer using round-robin scheduling.

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



Make sure that all CryptoServer devices in a single cluster have the same active MBK as well as the same P11 user with the same credentials.

Key generation is done on one device, then the key is transferred to the other devices in the cluster via backup/restore. If this fails, e.g. due to a connection issue to one of the devices, the successful devices are written to the P11 log file, given the configured log level is 3 or higher. If this happens, you should manually transfer the generated key to the remaining cluster devices to make sure they won't be left in an asynchronous state.

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 reconnection attempt to the primary CryptoServer should be started.

A `cs_pkcs11_R3.cfg` configuration file in failover mode could look for example as follows:

```
[Global]
Logging = 3
Logpath = c:/tmp
Logsize = 10mb
KeysExternal = true
# Configures load balancing mode ( == 0 ) or failover mode ( > 0 ).
# Time interval [s] after which a reconnection attempt to the primary
# CryptoServer should be started.
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
[KeyStorage]
KeyStorageType = Legacy
KeyStorageConfig = C:/ProgramData/Utimaco/PKCS11_R3/P11.sdb

# Number of reconnection attempts to ODBC database
#KeyStorageReconnect = 0
```



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

<i>CryptoServer Device (logical)</i>	<i>Slot ID (deviceID slot ID)</i>
192.168.4.136 192.168.0.137	0x 0000 0000
	0x 0000 0001
	0x 0000 0002
	0x 0000 0003
	0x 0000 0004

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.

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.

If `KeysExternal` is set to `true` and `KeyStorageType` as well as `KeyStorageConfig` are set in the configuration file (see [Configuration \(p. 12\)](#)) this external keystore is used when searching for keys.

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 [\[PKCS11BS\] \(p. 113\)](#) and [\[PKCS11CMS\] \(p. 113\)](#).

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

- `cryptoki.h`
- `pkcs11.h`
- `pkcs11f.h`
- `pkcs11t.h`
- `pkcs11t_cs.h`

These files can be downloaded from [\[PKCS11\] \(p. 113\)](#) 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:

- All requirements in [Setup and Requirements \(p. 9\)](#) must be fulfilled.
- The application shall include the header file `cryptoki.h`.

7.1 Libraries

For development the following libraries exist:

- For Microsoft Windows operating systems: Dynamic Link Library (DLL) `cs_pkcs11_R3.dll`. The library is built with Microsoft Visual Studio components.
- For Linux, and other UNIX systems: shared `library libcs_pkcs11_R3.so` and static library `libcs_pkcs11_R3_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_R3.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 [Operating Modes](#) (p. 20).

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. In case the limit is exceeded, `CKR_ARGUMENTS_BAD` is returned.

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 `uIvLen`. A length of 12 byte is recommended by the NIST. The maximum length is limited to 64 kbyte – 1 (0xFFFF) because no auto-chunking of IV is supported.

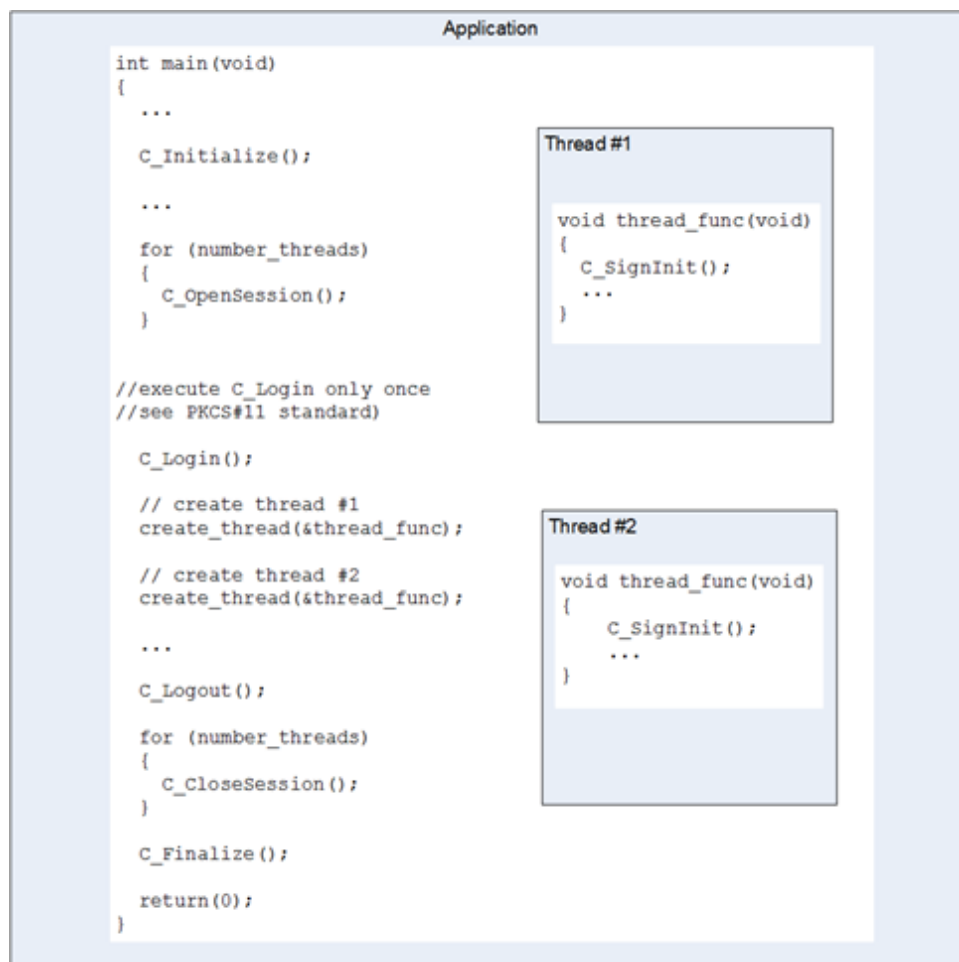
8.2.3 Data Length for Key Wrapping with AES GCM/CCM

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.



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 R3 library is described. For information about the general authentication concept of the CryptoServer (HMAC password authentication, RSA and ECDSA signature authentication and RSA smartcard authentication), see the section *Authentication Mechanisms* in the *CryptoServer - Administration Manual*.

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 [Standard Authentication Concept \(p. 30\)](#) for details.

A PKCS#11 Security Officer (SO) or a PKCS#11 user uses an HMAC password-based key-derived function (HMAC-PBKDF) for authentication. The HMAC-PBKDF according to NIST SP 800-132 does not use the HMAC password itself for authentication but a function derived from the HMAC password. For HMAC-PBKDF, 1000 iterations are used here. This number of iterations, as used for the key derivation from a given password, is fix and not configurable. HMAC-PBKDF is only applied if on both sides, the host side and the firmware side, HMAC-PBKDF is applied. If it is not available on one of these sides, the legacy version of the HMAC password-based mechanism is applied, whereby the user's password is used directly as an HMAC key. In FIPS mode, using HMAC-PBKDF is mandatory.

For non-standard requirements like special authentication mechanisms, authentication according to the two-person rule or a dedicated key manager role, see [Enhanced Authentication Concept \(p. 34\)](#).

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 into the CryptoServer.

Additionally, Utimaco's PKCS#11 R3 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. Log in to a CryptoServer user with a permission of at least 20000000. For instance, log in as the CryptoServer default user ADMIN with the `C_Login` command.

2. Create the SO executing the `C_InitToken` command.
3. Log out the CryptoServer administrator with the `C_Logout` command.
4. Log in as the SO with the `C_Login` command.
5. Set a new PIN for SO with the `C_SetPIN` command.
This step is mandatory for users with HMAC password authentication to perform commands needing to be authenticated.
6. Log out the SO with the `C_Logout` command.
7. Log in as the SO with the `C_Login` command with the new PIN.
8. Create the USER executing the `C_InitPIN` command.
9. Log out the SO with the `C_Logout` command.
10. Log in as the USER with the `C_Login` command.
11. Set a new PIN for USER with the `C_SetPIN` command.
This step is mandatory for users with HMAC password authentication to perform commands needing to be authenticated.
12. Log out the USER 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 = "12345678";  
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 = "12345678";
```

```
CK_ULONG ulPinLen = strlen(pPin);  
err = C_Login(hSession, CKU_SO, pPin, ulPinLen);
```

```
//5.  
CK_UTF8CHAR_PTR pOldPin = " 12345678";  
CK_ULONG ulOldLen = strlen(pOldPin);  
CK_UTF8CHAR_PTR pNewPin = "87654321";  
CK_ULONG ulNewLen = strlen(pNewPin);  
err = C_SetPIN(hSession, pOldPin, ulOldLen , pNewPin, ulNewLen);
```

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

```
//7.  
CK_UTF8CHAR_PTR pPin = "87654321";  
CK_ULONG ulPinLen = strlen(pPin);  
err = C_Login(hSession, CKU_SO, pPin, ulPinLen);
```

```
//8.  
CK_UTF8CHAR_PTR pPin = "123456789";  
CK_ULONG ulPinLen = strlen(pPin);  
err = C_InitPIN(hSession, pPin, ulPinLen);
```

```
//9.  
Err = C_Logout(hSession);
```

```
//10.  
CK_UTF8CHAR_PTR pPin = "123456789";  
CK_ULONG ulPinLen = strlen(pPin);  
err = C_Login(hSession, CKU_USER, pPin, ulPinLen);
```



```
//11.
CK_UTF8CHAR_PTR pOldPin = " 123456789";
CK_ULONG ulOldLen = strlen(pOldPin);
CK_UTF8CHAR_PTR pNewPin = "987654321";
CK_ULONG ulNewLen = strlen(pNewPin);
err = C_SetPIN(hSession, pOldPin, ulOldLen , pNewPin, ulNewLen);
```

```
//12.
Err = C_Logout(hSession);
```

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

Name	Permission	Mechanism	Attributes
ADMIN	220000000	RSA sign	I[0]
SO_0000	00000200	HMAC passwd	I[1]A[CXI_GROUP=SLOT_0000]
USR_0000	00000002	HMAC passwd	I[1]A[CXI_GROUP=SLOT_0000]

The ADMIN user in the initial state at delivery cannot perform commands he/she needs an authentication for (except for an ADMIN user on a CryptoServer Simulator), except for the `csadm ChangeUser` command. The credentials (i.e., the `ADMIN.key` file) of the ADMIN user are unchanged in this state. The `csadm ListUser` command shows this user with the `I[1]` attribute value. To enable the ADMIN user to perform commands he/she needs an authentication for, the ADMIN user must change the credentials by performing the `csadm ChangeUser` command. The `csadm ListUser` command then shows this user with the `I[0]` attribute value.

If a user with HMAC password authentication is created, this user cannot perform commands he/she needs an authentication for (except for the `csadm ChangeUser` command). The `csadm ListUser` command shows this user with the `I[1]` attribute value. To enable the user to perform commands he/she needs an authentication for, the user must change his/her credentials. To do so, this user must perform the `csadm ChangeUser` command. If the user is a PKCS#11 Security Officer (SO) or a PKCS#11 normal user, he/she may perform the `p11tool2 SetPIN` command as an alternative. It is important that the changes are

performed by the user himself/herself and not, for example, by an administrator. The `csadm ListUser` command then shows this user with the `I[0]` attribute value.

9.2 Enhanced Authentication Concept

9.2.1 Create Users with Other Authentication Mechanisms

The default authentication mechanism used with PKCS#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` .:

In the following, `<hash>` stands for the hash algorithm to be used for RSA smartcard authentication, RSA signature authentication, ECDSA signature authentication or HMAC password authentication mechanism in case that not the default hash algorithm shall be used for this user, which is SHA-1 for RSA smartcard authentication and SHA-256 for RSA signature authentication, ECDSA signature authentication and HMAC password authentication.

The value can be one of the following algorithms:

- SHA-1, SHA-224, SHA-256, SHA-384 or SHA-512
- MD5
- RIPEMD-160

The defined value is assigned to the user as the attribute `H[]` , for example, `H[SHA-256]` . In FIPS mode, only the default hash algorithms are supported.

Authentication mechanism	Syntax (pPin parameter of C_InitToken or C_InitPIN)
RSA signature authentication with a keyfile	<pre>CKU_VENDOR:RSASign=[{<hash>}]<filename> CKU_VENDOR:RSASign=[{<hash>}]<filename>#<password></pre> <ul style="list-style-type: none"> ▪ <code><filename></code> - Name of the file containing the RSA key for the user incl. path to the file ▪ <code><password></code> - Password of the keyfile, if encrypted

Authentication mechanism	Syntax (pPin parameter of C_InitToken or C_InitPIN)
RSA signature authentication with a smartcard (directly connected to a host)	CKU_VENDOR:RSASign=[{<hash>}]<key-specifier> <key-specifier> - Description of smartcard, PIN pad and interface (example: :cs2:auto:USB0). See the sections <i>Key Specifiers</i> and <i>Using a Local PIN Pad for a Remote CryptoServer</i> in the <i>CryptoServer - Administration Manual</i> for details about key specifiers.
RSA smartcard authentication (with a smartcard connected to the CryptoServer PCIe card)	CKU_VENDOR:RSASC=[{<hash>}]<key-specifier> <key-specifier> - Description of smartcard, PIN pad and interface (example: :cs2:auto:USB0). See the section <i>Key Specifiers</i> in the <i>CryptoServer - csadm Manual</i> for details about key specifiers. Section <i>Using a Local PIN Pad for a Remote CryptoServer</i> in the <i>CryptoServer - Administration Manual</i> does not apply here because for RSA smartcard authentication, the PIN pad must be directly connected to the CryptoServer PCIe card. See section <i>Authentication Mechanisms</i> in the <i>CryptoServer - Administration Manual</i> for details.
HMAC password authentication	CKU_VENDOR:HMACPwd=[{<hash>}]<password> <password> - Password of the user
ECDSA signature authentication with a keyfile	CKU_VENDOR:ECDSA=[{<hash>}]<filename> CKU_VENDOR:ECDSA=[{<hash>}]<filename>#<password> <ul style="list-style-type: none"> <filename> - Name of the file containing the RSA key of the user incl. path to the file <password> - Password of keyfile, if encrypted
ECDSA signature authentication with a smartcard (connected to a host)	CKU_VENDOR:ECDSA=[{<hash>}]<key-specifier> <key-specifier> - Description of smartcard, PIN pad and interface (example: :cs2:auto:USB0). See sections <i>Key Specifiers</i> and <i>Using a Local PIN Pad for a Remote CryptoServer</i> in the <i>CryptoServer - Administration Manual</i> 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 authentication with a smartcard directly connected to a 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 authentication with a 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 authentication):

```
CK_SLOT_ID slotID = 0;
CK_UTF8CHAR_PTR pPin = "CKU_VENDOR:HMACPwd=12345678";
CK_ULONG ulPinLen = strlen(pPin);
CK_UTF8CHAR_PTR pLabel = "Testlabel";
err = C_InitToken(slotID, pPin, ulPinLen, pLabel);
```

Example (SO using ECDSA signature authentication with a smartcard directly connected to a 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 authentication with a 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 (<i>pPin</i> parameter of <i>C_Login</i>)
RSA signature authentication with a keyfile	<code><username>,<filename></code> <code><username>,<filename>#<password></code> <code><filename></code> - Name of the file containing the user's RSA key incl. path <code><password></code> - Password of keyfile, if encrypted
RSA signature authentication with a smartcard (directly connected to a host) PIN is read via the PIN pad.	<code><username>,<key-specifier></code> <code><key-specifier></code> - Description of smartcard, PIN pad and interface (example: <code>:cs2:auto:USB0</code>). See sections <i>Key Specifiers</i> and <i>Using a Local PIN Pad for a Remote CryptoServer</i> in the <i>CryptoServer - Administration Manual</i> for details about key specifiers.
RSA smartcard authentication (with a smartcard connected to the CryptoServer) PIN is read via the PIN pad.	<code><username></code> <code><key-specifier></code> - Description of smartcard, PIN pad and interface (example: <code>:cs2:auto:USB0</code>). See sections <i>Key Specifiers</i> and <i>Using a Local PIN Pad for a Remote CryptoServer</i> in the <i>CryptoServer - Administration Manual</i> (p. 113) for details about key specifiers.
HMAC password authentication	<code><username>,<password></code> <code><password></code> - Password of the user
ECDSA signature authentication with a keyfile	<code><username>,<filename></code> <code><username>,<filename>#<password></code> <code><filename></code> - Name of the file containing the user's ECDSA incl. path to the file <code><password></code> - Password of keyfile, if encrypted
ECDSA signature authentication with a smartcard (connected to a host) PIN is read in over the PIN pad.	<code><username>,<key-specifier></code> <code><key-specifier></code> - Description of smartcard, PIN pad and interface (example: <code>:cs2:auto:USB0</code>). See sections <i>Key Specifiers</i> and <i>Using a Local PIN Pad for a Remote CryptoServer</i> in the <i>CryptoServer - Administration Manual</i> for details about key specifiers.

Table 9: Authentication mechanisms and corresponding syntax

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

```
CK_UTF8CHAR_PTR pPin = "SO_0000,12345678";
CK_ULONG ulPinLen = strlen(pPin);
err = C_Login(hSession, CKU_CS_GENERIC, pPin, ulPinLen);
```

Example (login of SO on slot 0) using RSA signature authentication with a 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 authentication with a smartcard directly connected to a 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 authentication with a 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 authentication with a smartcard directly connected to a 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 HMAC password authentication mechanism (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` :

<i>Authentication mechanism</i>	<i>Syntax (pPin parameter of C_SetPIN)</i>
HMAC password authentication	CKU_VENDOR:<password> <password> - Password of the user

Table 10: Syntax for changing the user password

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

```
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 HMAC password authentication mechanism 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_R3.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.

In the following, `<hash>` stands for the hash algorithm to be used for RSA smartcard authentication, RSA signature authentication, ECDSA signature authentication or HMAC password authentication mechanism in case that not the default hash algorithm shall be used for this user, which is SHA-1 for RSA smartcard authentication and SHA-256 for RSA signature authentication, ECDSA signature authentication and HMAC password authentication.

The value can be one of the following algorithms:

- SHA-1, SHA-224, SHA-256, SHA-384 or SHA-512
- MD5
- RIPEMD-160

The defined value is assigned to the user as the attribute `H[]`, for example, `H[SHA-256]`.

In FIPS mode, only the default hash algorithms are supported.

Authentication mechanism	Syntax in configuration file (" <code><key> = <value></code> ")
RSA signature authentication with a keyfile	<code><username> = "RSASign=[{<hash>}]<filename>"</code> <code><filename></code> - Name of the file containing the user's RSA key incl. path to the file
RSA signature authentication with a smartcard (connected to a host)	<code><username> = "RSASign=[{<hash>}]<key-specifier>"</code> <code><key-specifier></code> - Description of smartcard, PIN pad and interface (example: <code>:cs2:auto:USB0</code>). See sections <i>Key Specifiers</i> and <i>Using a Local PIN Pad for a Remote CryptoServer</i> in the <i>CryptoServer - Administration Manual</i> for details about key specifiers.

Authentication mechanism	Syntax in configuration file (" <key> = <value> ")
RSA smartcard authentication (with a smartcard directly connected to the CryptoServer)	<code><username> = "RSASC = [{<hash>}] <key-specifier>"</code> <code><key-specifier></code> - Description of smartcard, PIN pad and interface (example: :cs2:auto:USB0). See sections <i>Key Specifiers</i> and <i>Using a Local PIN Pad for a Remote CryptoServer</i> in the <i>CryptoServer - Administration Manual</i> for details about key specifiers.
HMAC password authentication	<code><username> = "HMACPwd= (<{hash}>) #<password>"</code>
ECDSA signature authentication with a keyfile	<code><username> = "ECDSA = [{<hash>}] <filename>"</code> <code><filename></code> - Name of the file containing the user's ECDSA incl. path to the file
ECDSA signature authentication with a smartcard (connected to a host)	<code><username> = "ECDSA = [{<hash>}] <key-specifier>"</code> <code><key-specifier></code> - Description of smartcard, PIN pad and interface (example: :cs2:auto:USB0). See sections <i>Key Specifiers</i> and <i>Using a Local PIN Pad for a Remote CryptoServer</i> in the <i>CryptoServer - Administration Manual</i> for details about key specifiers.

Table 11: Syntax for defining user's authentication mechanism in cs_pkcs11_R3.cfg

Example configuration file (default SO of slot 0 who uses RSA signature authentication with a 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 the RSA signature authentication with a smartcard connected to a 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 sections *Key Specifiers* and *Using a Local PIN Pad for a Remote CryptoServer* in the *CryptoServer - Administration Manual* 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:

Authentication mechanism	C_InitToken, C_InitPIN	C_Login	C_SetPin
RSA signature authentication with a keyfile	NULL_PTR or password of keyfile	NULL_PTR or password of keyfile	NOT ALLOWED
RSA signature authentication with a smartcard (connected to a host)	NULL_PTR	NULL_PTR	NOT ALLOWED
RSA smartcard authentication (with a smartcard connected to the CryptoServer)	NULL_PTR	NULL_PTR	NOT ALLOWED
HMAC password authentication	password	password	password
ECDSA signature authentication with a keyfile	NULL_PTR or password of keyfile	NULL_PTR or password of keyfile	NOT ALLOWED
ECDSA signature authentication with a 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_R3.cfg`

9.2.5 Automatic Login of Administrator via Configuration File

Sometimes, it might not be possible to log in 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 a configuration file. The user's name is "AD", and he uses the HMAC password authentication mechanism with minimum Authentication Concept 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	220000000	RSA sign	I[0]
AD	200000000	HMAC passwd	I[1]

This AD user has to perform the command `csadm ChangeUser` to change his/her credentials to be enabled to authenticate commands he/she wants to perform. This is indicated by `I[0]` in the output of the command `csadm ListUser`.

Example:

Name	Permission	Mechanism	Attributes
ADMIN	220000000	RSA sign	I[0]
AD	200000000	HMAC passwd	I[0]

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>"
```

`<hash>` stands for the hash algorithm to be used for the HMAC password authentication mechanism in case that not the default hash algorithm shall be used for this user, which is SHA-256 for HMAC password authentication. The value can be one of the following algorithms:

- SHA-1, SHA-224, SHA-256, SHA-384 or SHA-512
- MD5
- RIPEMD-160

The defined value is assigned to the user as the attribute `H[]`, for example, `H[SHA-256]`. In FIPS mode, only the default hash algorithms are supported.

As an alternative to the HMAC password authentication, the RSA signature authentication with a keyfile is supported as well:

```
AD = "RSASign=<path to *.key file>[#<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 '12345678'
AD = "HMACPwd=#12345678"
```

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`.

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

```
CK_UTF8CHAR_PTR pPinS01 = "S01_0000,12345678";
CK_ULONG ulPinLenS01 = strlen(pPinS01);
CK_UTF8CHAR_PTR pPinS02 = "S02_0000,87654321";
CK_ULONG ulPinLenS02 = strlen(pPinS02);
err = C_Login(hSession, CKU_CS_GENERIC, pPinS01, ulPinLenS01);
err = C_Login(hSession, CKU_CS_GENERIC, pPinS02, ulPinLenS02);
```

9.2.6.1 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 [Authentication According to the Two-Person Rule \(p. 43\)](#). To activate extended login for a slot, an entry shall be set in the corresponding slot section of the configuration file `cs_pkcs11_R3.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 [Login User with Other Authentication Mechanisms \(p. 36\)](#).

Login pattern:

```
ExtendedLoginSO/ExtendedLoginUSER = {
  <username1>,<keyfile-path/smartcard-spec>,<password>
  <username2>,<keyfile-path/smartcard-spec>,<password>
  <usernameN>,<keyfile-path/smartcard-spec>,<password>
}
```

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 USR_0000 must authenticate via the
# password; then the user SomeUser must authenticate via the ADMIN.key file
ExtendedLoginSO = {
  USR_0000,12345678
  SomeUser,C:/ADMIN.key,
}
```

9.2.7 Separated 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:

- Key manager (KM) with permission mask 00000020 to manage cryptographic keys
- Key user (KU) with the permission mask 00000002 to use cryptographic keys.

Splitting the key manager and the key user into two separate roles can be achieved by setting the global or local configuration object `CKA_CFG_AUTH_KEYM_MASK`. See [Configuration Objects \(p. 53\)](#) for more details.

Initialize the slot as usual using `p11tool2` (see section *InitToken* in [\[CS_PKCS11T2\] \(p. 113\)](#)), `P11CAT` (see section *Setting up a Slot (Init Token)* in [\[CS_PKCS11CAT\] \(p. 113\)](#)) or the PKCS#11 API function `C_InitToken` (see [Standard Authentication Concept \(p. 30\)](#)).

However, the key manager and the key user have to be created manually. For example, perform these steps for slot 1 to do so:

1. Create a key manager (user `KM_0001`) with the permission mask 00000020 and the attribute `CXI_GROUP=SLOT_0001`. This command requires authentication by a user with the user administrator role (minimum permissions 20000000). This key manager must be created out of the scope of the CryptoServer PKCS#11 API and tools with the CryptoServer's administration tools `csadm` or `CAT`.

For example, using csadm:

```
csadm Dev=3001@127.0.0.1 LogonSign=ADMIN,:cs2:cjo:USB0  
AddUser=KM_0001,000000020{CXI_GROUP=SLOT_0001},hmacpwd,12345678
```

For details on how to create a user, see section *AddUser* in the *CryptoServer - Administration Manual* when using csadm or section *Creating a New User* in the *CryptoServer - CAT Manual* when using CAT.

2. Perform the analog step to create the `USR_0001` user of slot 1.

For example, using csadm:

```
csadm Dev=3001@127.0.0.1 LogonSign=ADMIN,:cs2:cjo:USB0  
AddUser=USR_0001,000000002{CXI_GROUP=SLOT_0001},hmacpwd,12345678
```



Do not initialize the slot PIN using p11tool2, P11CAT or the PKCS#11 API. The "Init PIN" step would create a `USR_000<x>` user with the permission mask of a key manager and a key user (00000022) instead of the permission mask of a key user (00000002).

Now, the key manager can log in as user `KM_0001` with the `C_Login` function and user type `CKU_CS_GENERIC` (see [Login User with Other Authentication Mechanisms \(p. 36\)](#) for the syntax) to perform key management functions. The key user can log in as usual with user type `CKU_USER`.

10 Vendor Defined PKCS#11 Extensions

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

10.1 CryptoServer Defined Mechanisms

The CryptoServer implements the following mechanisms that are not included in the PKCS#11 standard (see [\[PKCS11CMS\] \(p. 113\)](#)).

<i>Name</i>	<i>Description</i>
CKM_ECDSA_SHA3_224	ECDSA signature generation using SHA3-224 hash algorithm. This mechanism is defined in the PKCS#11 specification 3.00. The standardized mechanisms constants shall be used. The Utimaco Vendor Defined Mechanisms are only supported for backward compatibility and may be discontinued in a future version.
CKM_ECDSA_SHA3_256	ECDSA signature generation using SHA3-256 hash algorithm. This mechanism is defined in the PKCS#11 specification 3.00. The standardized mechanisms constants shall be used. The Utimaco Vendor Defined Mechanisms are only supported for backward compatibility and may be discontinued in a future version.
CKM_ECDSA_SHA3_384	ECDSA signature generation using SHA3-384 hash algorithm. This mechanism is defined in the PKCS#11 specification 3.00. The standardized mechanisms constants shall be used. The Utimaco Vendor Defined Mechanisms are only supported for backward compatibility and may be discontinued in a future version.
CKM_ECDSA_SHA3_512	ECDSA signature generation using SHA3-512 hash algorithm. This mechanism is defined in the PKCS#11 specification 3.00. The standardized mechanisms constants shall be used. The Utimaco Vendor Defined Mechanisms are only supported for backward compatibility and may be discontinued in a future version.
CKM_ECDSA_RIPEMD160	ECDSA signature generation using RIPEMD-160 hash algorithm. CKM_ECDSA_RIPEMD160 and CKM_DSA_RIPEMD160 use the same calling conventions as CKM_ECDSA_SHA1 and CKM_DSA_SHA1.
CKM_DSA_RIPEMD160	DSA signature generation using RIPEMD-160 hash algorithm. CKM_ECDSA_RIPEMD160 and CKM_DSA_RIPEMD160 use the same calling conventions as CKM_ECDSA_SHA1 and CKM_DSA_SHA1.
CKM_DES3_RETAIL_MAC	Triple DES Retail-MAC with 0-Padding (see Sign and Verify Using the DES Retail-MAC (p. 51)). Triple DES (TDES) is blocked for FIPS on u.trust Anchor.
CKM_RSA_PKCS_MULT1	Generate multiple signatures with <code>CKM_RSA_PKCS</code> mechanism (see Multiple Signature Mechanisms (p. 52)).
CKM_RSA_X_509_MULT1	Generate multiple signatures with <code>CKM_RSA_X_509</code> mechanism (see Multiple Signature Mechanisms (p. 52)).
CKM_ECDSA_MULT1	Generate multiple signatures with <code>CKM_ECDSA</code> mechanism (see Multiple Signature Mechanisms (p. 52)).

Name	Description
CKM_DES_CBC_WRAP	<p>Enhanced DES key wrapping mechanism, see the structure CK_WRAP_PARAMS:</p> <pre>typedef struct CK_WRAP_PARAMS { unsigned char *pIv; // the starting IV unsigned int ulIvLen; // length of the starting IV unsigned char *pPrefix; // prefix data unsigned int ulPrefixLen; // length of the prefix data unsigned char *pPostfix; // postfix data unsigned int ulPostfixLen; // length of the postfix data } CK_WRAP_PARAMS;</pre>
CKM_AES_CBC_WRAP	<p>Enhanced AES key wrapping mechanism, see the structure CK_WRAP_PARAMS:</p> <pre>typedef struct CK_WRAP_PARAMS { unsigned char *pIv; // the starting IV unsigned int ulIvLen; // length of the starting IV unsigned char *pPrefix; // prefix data unsigned int ulPrefixLen; // length of the prefix data unsigned char *pPostfix; // postfix data unsigned int ulPostfixLen; // length of the postfix data } CK_WRAP_PARAMS;</pre>
CKM_ECKA	EC secret agreement according to BSI-TR-03111 (returns secret point without hashing; see Handling CKM_ECKA (p. 49)).
CKM_ECDSA_ECIES	ECDSA crypt algorithm (see 2012-0007 Encryption with the “Elliptic Curve Integrated Encryption Scheme” (ECIES) (p. 50)).
CKM_UTIMACO_SM2	SM2 algorithm without hashing (input is expected to be hashed before) (see OSCCA Module (p. 63))
CKM_UTIMACO_SM2_SHA256	SM2 algorithm with SHA256 hash algorithm (see OSCCA Module (p. 63))
CKM_UTIMACO_SM2_SHA384	SM2 algorithm with SHA384 hash algorithm (see OSCCA Module (p. 63))
CKM_UTIMACO_SM2_SHA512	SM2 algorithm with SHA512 hash algorithm (see OSCCA Module (p. 63))
CKM_UTIMACO_SM2_SHA3_256	SM2 algorithm with SHA3_256 hash algorithm (see OSCCA Module (p. 63))

<i>Name</i>	<i>Description</i>
CKM_UTIMACO_SM2_SHA3_384	SM2 algorithm with SHA3_384 hash algorithm (see OSCCA Module (p. 63))
CKM_UTIMACO_SM2_SHA3_512	SM2 algorithm with SHA3_512 hash algorithm (see OSCCA Module (p. 63))
CKM_UTIMACO_SM2_SM3	SM2 algorithm with SM3 algorithm (see OSCCA Module (p. 63))
CKM_UTIMACO_SM2_KEY_PAIR_GEN	SM2 key generation (see OSCCA Module (p. 63))
CKM_UTIMACO_SM3	SM3 digest algorithm (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_KEY_GEN	SM4 key generation (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_ECB	SM4 cipher in ECB mode without padding (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_ECB_PAD	SM4 cipher in ECB mode with PKCS7 padding (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_CBC	SM4 cipher in CBC mode without padding (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_CBC_PAD	SM4 cipher in CBC mode with PKCS7 padding (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_CFB	SM4 cipher in CFB mode (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_OFB	SM4 cipher in OFB mode (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_CTR	SM4 cipher in CTR mode (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_GCM	SM4 cipher in GCM mode (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_CCM	SM4 cipher in CCM mode (see OSCCA Module (p. 63))
CKM_UTIMACO_SM4_GMAC	SM4 GMAC signature with 12 byte IV (see OSCCA Module (p. 63))

Table 13: Description of CryptoServer-specific mechanisms

10.2 Handling CKM_ECKA

The mechanism `CKM_ECKA` can be used in the function `CS_AgreeSecret()` to calculate a shared secret from two ECDH or ECDSA keys as specified in BSI TR 03116-1.

```

/**
 * CS_AgreeSecret()
 *
 * Calculates a shared secret from two ECDH or ECDSA keys as described in BSI-TR-03116-1.
 *
 * \param[in] hSession session handle
 * \param[in] pMechanism secret calculation mechanism
 * \param[in] hPrivateKey handle of private key used for shared secret calculation
 * \param[in] pPublicKey public key used for shared secret calculation
 * \param[in] ulPublicKeyLen length public key used for shared secret calculation
 * \param[out] pSharedSecret gets the shared secret
 * \param[out] pulSharedSecretLen gets shared secret length
 */
CK_PKCS11_FUNCTION_INFO(CS_AgreeSecret)
#ifdef CK_NEED_ARG_LIST
(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hPrivateKey,
    CK_BYTE_PTR pPublicKey,
    CK_ULONG_PTR pulPublicKeyLen,
    CK_BYTE_PTR pSharedSecret,
    CK_ULONG_PTR pulSharedSecretLen
);

```

As typically done in other PKCS#11 functions, the function `CS_AgreeSecret()` can be called with `pSharedSecret = NULL_PTR` to determine the length of the result (shared secret). With the introduction of this function, the mechanism `CKM_ECKA` is not supported with `C_SignInit()` function anymore and will be responded with `CKR_MECHANISM_INVALID` return code.

The public key used for the shared secret calculation is provided as the `pPublicKey` Parameter (as plain data). The mechanism parameter can be optionally provided as a `CK_BBOOL` value (`CK_TRUE` or `CK_FALSE`) which indicates if the calculated shared secret should be MBK encrypted (The default value is `CK_FALSE`, i.e., the shared secret is returned/provided as plain value.). Also, if no mechanism parameter is provided, no encryption is used.

10.3 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 [\[ANSI-X9.63\] \(p. 113\)](#) or *Elliptic Curve Integration Encryption Scheme (ECIES)* of [\[SEC1\] \(p. 113\)](#).

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_CBC, Vendor Defined PKCS#11 Extensions , 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

10.4 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 [\[ISO-9797\] \(p. 113\)](#) and [\[ANSI-X9.19\] \(p. 113\)](#).

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_signature);
```

- If the `pParameter` of the mechanism structure is set to `NULL` (or `8`), the result has the length of 8 bytes according to the [\[ISO-9797\] \(p. 113\)](#) specification.
- If the `pParameter` is set to `4`, the result has the length of 4 bytes according to the [\[ANSIX9.19\] \(p. 113\)](#) specification.

10.5 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.

<i>Field</i>	<i>Description</i>
k	Number of signatures to be calculated ($k \leq 50$ for RSA, $k \leq 16$ for ECDSA). 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 <code>data_1</code> . All signatures have the same length.
signature_k	Last signature generated by the function, calculated over <code>data_k</code> . All signatures have the same length.

Table 14: Fields of the input and output data structures

10.6 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

10.6.1 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`.

<i>Attribute</i>	<i>Description</i>
CKA_UTIMACO_CFG_PATH	Type: <code>CK_BYTE_PTR</code> Value: Path to the configuration file Default: <code>C:\ProgramData\Utimaco\PKCS11_R3\cs_pkcs11_R3.cfg</code> read only

Table 15: Attribute CKA_UTIMACO_CFG_PATH - details

10.6.2 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 section *The Clear Functionality* in the *CryptoServer - csadm Manual*) is performed. We highly recommend to create a backup of the Global CryptoServer Configuration Object resp. of the `CXIKEY.db` with the `csadm BackupDatabase` command described in section *BackupDatabase* of the *CryptoServer - csadm Manual* or with the CAT as described in section "Backing up Databases" of the *CryptoServer - CAT Manual*, 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.



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 section *Creating a New User* in the *CryptoServer - CAT Manual*. Examples for creating different users with csadm are provided in section *AddUser* of the *CryptoServer - Administration Manual*.

- **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.



If the User role has been split into the key user role and the key manager role, the steps in [Separated Key Manager and Key User Role \(p. 45\)](#), must be performed.

- **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 [PKCS11CMS].

Possible value:

- `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_R3.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_R3.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_R3.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_R3.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_R3.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_R3.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_R3.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_R3.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_R3.log logfile`. This prevents simple Key Extraction attacks.



See [Logging \(p. 18\)](#) for details about how to configure the PKCS#11 API logfile (`cs_pkcs11_R3.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_P11R3_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 MBKprotected 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 [CryptoServer Slot Configuration Objects \(p. 61\)](#), for a detailed description of the `CKA_CFG_SLOT_BACKUP_PASS_HASH` slot configuration attribute.

- `CKA_CFG_ENFORCE_EXT_KEYS`



This parameter is supported by Security Server 4.30 /CryptoServer SDK 4.30 and later (CXI firmware module version 2.4.0.0 and later).

If `CKA_CFG_ENFORCE_EXT_KEYS` (default: `CK_FALSE`) is set to `CK_TRUE`, an object (for example, a cryptographic key) is always created in the external keystore. A cryptographic key is created by the following actions: generate a key, generate a key pair, derive a key, restore a key, import a key and unwrap a key. `CKA_CFG_ENFORCE_EXT_KEYS` is irrelevant for key usage functions. For example,

signing with an already existing internal key is supported even if

`CKA_CFG_ENFORCE_EXT_KEYS` is set to `CK_TRUE`.

If `CKA_CFG_ENFORCE_EXT_KEYS` is set to `CK_TRUE`, the `KeysExternal` parameter in the `cs_pkcs11_R3.cfg` file must be set to true and the `KeyStorageType` and the `KeyStorageConfig` parameter in the same file must be set as well. See chapter *Editing the cs_pkcs11_R3.cfg Configuration File* in [\[CS_PKCS11CAT\] \(p. 113\)](#) and [Configuration \(p. 12\)](#) for details. If `CKA_CFG_ENFORCE_EXT_KEYS` is set to `CK_TRUE` and `CKA_CFG_ENFORCE_EXT_KEYS` is set to `CK_FALSE`, no cryptographic key but an error message is generated.

- `CKA_CFG_ALLOW_WEAK_DES_KEYS`



This parameter is supported by Security Server 4.30 /CryptoServer SDK 4.30 and later (CXI firmware module version 2.4.0.0 and later).

If `CKA_CFG_ALLOW_WEAK_DES_KEYS` (default: `CK_FALSE`) is set to `CK_TRUE`, importing weak DES keys is supported. `CKA_CFG_ALLOW_WEAK_DES_KEYS` might be set for a key group (slot) or for the entire CryptoServer. A weak DES key is a 2DES key that has two equal parts or a 3DES key that has two parts that are pairwise equal.

10.6.3 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 [Global CryptoServer Configuration Object \(p. 54\)](#) 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 Error! Reference source not found.).

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

section *Clear* in the *CryptoServer - csadm Manual*) is performed. We highly recommend to create a backup of the Slot CryptoServer Configuration Object with the P11CAT tool (see section *Creating a Slot Configuration Backup* in the [\[CS_PKCS11CAT\]](#) (p. 113)) or the `p11tool2 BackupConfig` command (see [\[CS_PKCS11T2\]](#) (p. 113)) so you can easily restore your PKCS#11 slot configuration.

In addition to the attributes described in [Global CryptoServer Configuration Object](#) (p. 54), 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 [Global CryptoServer Configuration Object](#) (p. 54)).



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. `CKA_CFG_ALLOW_SLOTS` attribute is set to `CK_FALSE`. The default value is the SHA-256 hash of the empty string.

10.7 Switching from the DCC VDM Module to the Updated GBCS Integration

With the release of SecurityServer 4.50, the CKM_ECDSA_SHA256_DCC mechanism, which was previously available in the DCC VDM module, has been integrated into the base firmware to support the calculation of an ECDSA per-message secret number, according to the Great Britain Companion Specification (GBCS).

If you want to use this integration, some adjustments are necessary to use the base firmware's implementation since the latter uses a different value for CKM_ECDSA_SHA256_DCC than the DCC VDM module. If you are currently using the previous DCC VDM module, proceed as follows to update the mechanisms:

1. Update the SecurityServer to the newest version.

This does not delete the DCC VDM module, unless you add `ForceClear` to the `csadm LoadPkg` command. If you did, load the DCC VDM module again to ensure your existing applications work as intended before you continue.

2. Adjust your application code:

- Remove `#define CKM_ECDSA_SHA256_DCC 0xC00D0001` from your code.
- If you defined your own name, replace all of its occurrences with `CKM_ECDSA_SHA256_DCC`.
- Example for mechanism `CKM_ECDSA_SHA256_DCC`

```
CK_SESSION_HANDLE    sid;
CK_OBJECT_HANDLE     eccPrivateKey;
CK_MECHANISM         signMechanism;
CK_BYTE              data[BUFFERSIZE];
CK_ULONG              l_data = sizeof(data);
CK_BYTE              signature[64]; // r + s = 2 * 256 bits -> 64
bytes
CK_ULONG              l_signature = sizeof(signature);

signMechanism.mechanism = CKM_ECDSA_SHA256_DCC;
signMechanism.pParameter = NULL;
signMechanism.ulParameterLen = 0;
...
err = C_SignInit(sid, &signMechanism, eccPrivateKey);
err = C_Sign(sid, data, l_data, signature, &l_signature);
```

3. Test if all functions work as needed.
4. Delete the DCC VDM module.
5. Restart the HSM.

10.8 OSCCA Module

Currently, the OSCCA module is part of the SecurityServer package. To use your own VDM or other VDMs, you need to delete OSCCA first.

Definitions for all OSCCA vendor defined extensions are provided with the inclusion of the `pkcs11t_cs.h` file.

Example for SM2 GenerateKeyPair

```
CK_SESSION_HANDLE  hSession;
CK_OBJECT_HANDLE_PTR  pHPublicKey;
CK_OBJECT_HANDLE_PTR  pHPrivateKey;

CK_OBJECT_CLASS      objClassPub  = CKO_PUBLIC_KEY;
CK_OBJECT_CLASS      objClassPrv  = CKO_PRIVATE_KEY;
CK_KEY_TYPE          keyType      = CKK_EC;
CK_BBOOL             b_true       = CK_TRUE;
CK_BYTE              keyID[]       = {"SM2"};
CK_BYTE              ecParams[]    = "sm2p256v1";
CK_BYTE              labelPub[]    = "SM2 Public Key";
CK_BYTE              labelPrv[]    = "SM2 Private Key";
```

```
CK_ATTRIBUTE  keyTemplatePublic[] =
{
{CKA_CLASS,      &objClassPub,  sizeof(objClassPub)},
{CKA_KEY_TYPE,   &keyType,      sizeof(keyType)},
{CKA_LABEL,      labelPub,      sizeof(labelPub)},
{CKA_TOKEN,      &b_true,       sizeof(b_true)},
{CKA_ID,         keyID,         sizeof(keyID)},
{CKA_VERIFY,     &b_true,       sizeof(b_true)},
{CKA_EC_PARAMS,  ecParams,      sizeof(ecParams)}
};
```

```
CK_ATTRIBUTE  keyTemplatePrivate[] =
{
{CKA_CLASS,      &objClassPrv,  sizeof(objClassPrv)},
{CKA_KEY_TYPE,   &keyType,      sizeof(keyType)},
{CKA_LABEL,      labelPrv,      sizeof(labelPrv)},
{CKA_TOKEN,      &b_true,       sizeof(b_true)},
{CKA_ID,         keyID,         sizeof(keyID)},
{CKA_PRIVATE,    &b_true,       sizeof(b_true)},
{CKA_SENSITIVE,  &b_true,       sizeof(b_true)},
{CKA_EXTRACTABLE, &b_true,      sizeof(b_true)},
{CKA_SIGN,       &b_true,       sizeof(b_true)}
};
```

```
CK_MECHANISM      mechanism;
```



```

mechanism.mechanism      = CKM_UTIMACO_SM2_KEY_PAIR_GEN;
mechanism.pParameter      = NULL;
mechanism.ulParameterLen  = 0;
err = C_GenerateKeyPair(hSession, &mechanism,
                        keyTemplatePublic, sizeof(keyTemplatePublic) /
sizeof(CK_ATTRIBUTE),
                        keyTemplatePrivate, sizeof(keyTemplatePrivate) /
sizeof(CK_ATTRIBUTE),
                        pHPublicKey, pHPrivateKey);

```

Example for SM4 GenerateKey

```

CK_SESSION_HANDLE      hSession;
CK_OBJECT_HANDLE_PTR   pHKey;
CK_OBJECT_CLASS         objClass = CKO_PRIVATE_KEY;
CK_KEY_TYPE            keyType   = CKK_VENDOR_DEFINED;
CK_BBOOL               b_true    = CK_TRUE;
CK_BYTE                keyID[]   = {"SM4"};
CK_BYTE                label[]   = "Secret SM4 Key";

```

```

CK_ATTRIBUTE keyTemplatePrivate[] =
{
    {CKA_CLASS,          &objClass,  sizeof(objClass)},
    {CKA_KEY_TYPE,       &keyType,   sizeof(keyType)},
    {CKA_LABEL,          label,       sizeof(label)},
    {CKA_TOKEN,          &b_true,     sizeof(b_true)},
    {CKA_ID,             keyID,       sizeof(keyID)},
    {CKA_PRIVATE,        &b_true,     sizeof(b_true)},
    {CKA_EXTRACTABLE,    &b_true,     sizeof(b_true)},
    {CKA_ENCRYPT,         &b_true,     sizeof(b_true)},
    {CKA_DECRYPT,         &b_true,     sizeof(b_true)}
};

```

```

CK_MECHANISM           mechanism;
mechanism.mechanism     = CKM_UTIMACO_SM4_KEY_GEN;
mechanism.pParameter     = NULL;
mechanism.ulParameterLen = 0;
err = C_GenerateKey(hSession, &mechanism,
                    keyTemplate, sizeof(keyTemplate) / sizeof(CK_ATTRIBUTE),
                    pHKey);

```

Example for SM2 Sign and Verify

```

CK_SESSION_HANDLE    hSession;
CK_OBJECT_HANDLE     hPublicKey;
CK_OBJECT_HANDLE     hPrivateKey;
CK_BYTE              signData[BUFFERSIZE];
CK_ULONG              signDataLength      = sizeof(signData);
CK_BYTE              signature[BUFFERSIZE];
CK_ULONG              signatureLength      = sizeof(signature);
CK_BYTE              userId[] = "MyUser";
CK_BYTE              userField[sizeof(userId) + 2];
CK_ULONG              userIdLen            = sizeof(userId);
userIdField[0]        = userIdLen >> 8U;
userIdField[1]        = userIdLen;
memcpy(userIdField+2, userId, userIdLen);

```

```

CK_MECHANISM          signMechanism;
signMechanism.mechanism = CKM_UTIMACO_SM2_SHA256;
signMechanism.pParameter = &userIdField;
signMechanism.ulParameterLen = sizeof(userIdField);
err = C_SignInit(hSession, &signMechanism, hPrivateKey)
err = C_Sign(hSession, signData, signDataLength, signature,
&signatureLength)
err = C_VerifyInit(hSession, &signMechanism, hPublicKey)
err = C_Verify(hSession, signData, signDataLength, signature,
signatureLength)

```

Example for SM4 Encrypt and Decrypt

```

CK_SESSION_HANDLE    hSession;
CK_OBJECT_HANDLE     hKey;

CK_BYTE              data[BUFFERSIZE];
CK_ULONG              dataLength          = sizeof(data);
CK_BYTE              encryptedData[BUFFERSIZE];
CK_ULONG              encryptedDataLength =
sizeof(encryptedData);
CK_BYTE              plainData[BUFFERSIZE];

```

```
CK_ULONG plainDataLength = sizeof(plainData);
```

```
CK_MECHANISM mechanism;  
mechanism.mechanism = CKM_UTIMACO_SM4_ECB;  
mechanism.pParameter = NULL;  
mechanism.ulParameterLen = 0;
```

```
err = C_EncryptInit(hSession, &mechanism, hKey)  
err = C_Encrypt(hSession, data, dataLength, encryptedData,  
&encryptedDataLength)  
err = C_DecryptInit(hSession, &mechanism, hKey)  
err = C_Decrypt(hSession, encryptedData, encryptedDataLength, plainData,  
&plainDataLength)
```

11 Key Management Functions and Cryptographic Operations 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 [\[PKCS11BS\] \(p. 113\)](#).

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	✗	✓

PKCS#11 Function	Description	Permitted user	
		KM	KU
C_DecryptFinal	Finishes a multiple-part decryption operation	✗	✓
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	✗	✓
C_SignEncryptUpdate	Continues simultaneous multiple-part signature and encryption operations	✗	✓

PKCS#11 Function	Description	Permitted user	
		KM	KU
C_DecryptVerifyUpdate	Continues simultaneous multiple-part decryption and verification operations	✗	✓
C_SeedRandom	Mixes additional seed material into the token's random number generator	✗	✓
C_GenerateRandom	Generates random data	✗	✓
C_MessageEncryptInit	Initializes a message-based encryption process	✗	✓
C_EncryptMessage	Encrypts a single-part message	✗	✓
C_EncryptMessageBegin	Begins a multiple-part message encryption operation	✗	✓
C_EncryptMessageNext	Continues or finishes a multiple-part message encryption operation	✗	✓
C_EncryptMessageFinal	Finishes a message-based encryption process	✗	✓
C_MessageDecryptInit	Initializes a message-based decryption process	✗	✓
C_DecryptMessage	Decrypts a single-part encrypted message	✗	✓
C_DecryptMessageBegin	Begins a multiple-part message decryption operation	✗	✓
C_DecryptMessageNext	Continues or finishes a multiple-part message decryption operation	✗	✓
C_DecryptMessageFinal	Finishes a message-based decryption process	✗	✓
C_MessageSignInit	Initializes a message-based signature process	✗	✓
C_SignMessage	Signs as single-part message	✗	✓
C_SignMessageBegin	Begins a multiple-part message signature operation	✗	✓
C_SignMessageNext	Continues or finishes a multiple-part message signature operation	✗	✓
C_MessageSignFinal	Finishes a message-based signature process	✗	✓
C_MessageVerifyInit	Initializes a message-based verification process	✗	✓
C_VerifyMessage	Verifies a signature on a single-part message	✗	✓

PKCS#11 Function	Description	Permitted user	
		KM	KU
C_VerifyMessageBegin	Begins a multiple-part message verification operation	✗	✓
C_VerifyMessageNext	Continues or finishes a multiple-part message verification operation	✗	✓
C_MessageVerifyFinal	Finishes a message-based verification process	✗	✓
CS_AgreeSecret	Calculates a shared secret	✓	✗

Table 16: List of PKCS#11 standard functions for KM and KU

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, Functions and Key Types

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

Mechanisms marked with ^h after the mechanism name were defined in earlier versions than PKCS #11 3.1 but are no longer in general use and have been moved to the Historical Mechanisms Specification. It is not recommended to use them in new applications. Beside these historical mechanisms explicitly mentioned in the tables below, no other historical mechanisms (for example, BATON, CAST, CDMF, FASTHASH, FORTEZZA timestamp, IDEA, JUNIPER, PBE, RIPE-MD 128, SKIPJACK) are supported.

The mechanisms table is followed by a table of Vendor Defined Mechanisms that are not included in the PKCS#11 standard, and the PKCS#11 API functions supporting them.

Mechanisms	Function						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key (Pair)	Wrap & Unwrap	Derive
PKCS #11 2.40							
CKM_RSA_PKCS_KEY_PAIR_GEN					✓		
CKM_RSA_X9_31_KEY_PAIR_GEN					✓		
CKM_RSA_PKCS	✓ ²	✓ ²	✓			✓	
CKM_RSA_PKCS_OAEP	✓ ²					✓	
CKM_RSA_PKCS_PSS		✓ ²					
CKM_RSA_9796		✗	✗				
CKM_RSA_X_509	✓ ²	✓ ²	✓			✓	
CKM_RSA_X9_31		✓ ²					
CKM_MD2_RSA_PKCS		✗					
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_RIPEMD128_RSA_PKCS		✗					
CKM_RIPEMD160_RSA_PKCS		✓					

CKM_SHA1_RSA_PKCS_PSS		✓					
CKM_SHA224_RSA_PKCS_PSS		✓					
CKM_SHA256_RSA_PKCS_PSS		✓					
CKM_SHA384_RSA_PKCS_PSS		✓					
CKM_SHA512_RSA_PKCS_PSS		✓					
CKM_SHA1_RSA_X9_31		✓					
CKM_RSA_PKCS_TPM_1_1	–					–	
CKM_RSA_PKCS_OAEP_TPM_1_1	–					–	
CKM_RSA_AES_KEY_WRAP						–	
CKM_DSA_KEY_PAIR_GEN					✓		
CKM_DSA_PARAMETER_GEN					✓		
CKM_DSA_PROBABLISTIC_PARAMETER_GEN					–		
CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN					–		
CKM_DSA_FIPS_G_GEN					–		
CKM_DSA		✓ ²					
CKM_DSA_SHA1		✓					
CKM_DSA_SHA224		✓					
CKM_DSA_SHA256		✓					
CKM_DSA_SHA384		✓					
CKM_DSA_SHA512		✓					

CKM_EC_KEY_PAIR_GEN (CKM_ECDSA_KEY_PAIR_GEN)					✓		
CKM_ECDSA		✓ ²					
CKM_ECDSA_SHA1		✓					
CKM_ECDSA_SHA224		✓					
CKM_ECDSA_SHA256		✓					
CKM_ECDSA_SHA384		✓					
CKM_ECDSA_SHA512		✓					
CKM_ECDH1_DERIVE							✓
CKM_ECDH1_COFACTOR_DERIVE							✓
CKM_ECMQV_DERIVE							–
CKM_ECDH_AES_KEY_WRAP						–	
CKM_DH_PKCS_KEY_PAIR_GEN					✓		
CKM_DH_PKCS_PARAMETER_GEN					✓		
CKM_DH_PKCS_DERIVE							✓
CKM_X9_42_DH_KEY_PAIR_GEN					✓		
CKM_X9_42_DH_PARAMETER_GEN					✓		
CKM_X9_42_DH_DERIVE							✓
CKM_X9_42_DH_HYBRID_DERIVE							–
CKM_X9_42_MQV_DERIVE					✓		–
CKM_GENERIC_SECRET_KEY_GEN					✓		
CKM_AES_KEY_GEN					✓		
CKM_AES_ECB	✓					✓	

CKM_AES_CBC	✓					✓	
CKM_AES_CBC_PAD	✓					✓	
CKM_AES_MAC_GENERAL		✓					
CKM_AES_MAC		✓					
CKM_AES_OFB	✓					✓	
CKM_AES_CFB64	✗					✗	
CKM_AES_CFB8	✗					✗	
CKM_AES_CFB128	✗					✗	
CKM_AES_CFB1	✗					✗	
CKM_AES_XCBC_MAC		✗					
CKM_AES_XCBC_MAC_96		✗					
CKM_AES_CTR	✓					✗	
CKM_AES_CTS	✗					✗	
CKM_AES_GCM	✓					✓	
CKM_AES_CCM	✓					✓	
CKM_AES_GMAC		✓					
CKM_AES_CMAC_GENERAL		✗					
CKM_AES_CMAC		✓					
CKM_AES_KEY_WRAP	✓ ²					✓	
CKM_AES_KEY_WRAP_PAD	✓ ²					✓	
CKM_DES_KEY_GEN ^h					✓		
CKM_DES_ECB ^h	✓					✓	

CKM_DES_CBC ^h	✓					✓	
CKM_DES_CBC_PAD ^h	✓					✓	
CKM_DES_MAC_GENERAL ^h		✓					
CKM_DES_MAC ^h		✓					
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_OFB64	✗						
CKM_DES_OFB8	✗						
CKM_DES_CFB64	✗						
CKM_DES_CFB8	✗						
CKM_DES3_CMAC_GENERAL		✗					
CKM_DES3_CMAC		✗					
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 ^h				✓			
CKM_MD5_HMAC_GENERAL ^h		✓					
CKM_MD5_HMAC ^h		✓					
CKM_MD5_KEY_DERIVATION ^h							✓
CKM_SHA_1				✓			
CKM_SHA_1_HMAC_GENERAL		✓					
CKM_SHA_1_HMAC		✓					
CKM_SHA1_KEY_DERIVATION							✓
CKM_SHA224				✓			
CKM_SHA224_HMAC		✓					
CKM_SHA224_HMAC_GENERAL		✓					
CKM_SHA224_KEY_DERIVATION							✓
CKM_SHA256				✓			
CKM_SHA256_HMAC_GENERAL		✓					
CKM_SHA256_HMAC		✓					
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_SHA512_224				–			
CKM_SHA512_224_HMAC_GENERAL		–					
CKM_SHA512_224_HMAC		–					
CKM_SHA512_224_KEY_DERIVATION							–
CKM_SHA512_256				–			
CKM_SHA512_256_HMAC_GENERAL		–					
CKM_SHA512_256_HMAC		–					
CKM_SHA512_256_KEY_DERIVATION							–
CKM_SHA512_T				–			
CKM_SHA512_T_HMAC_GENERAL		–					
CKM_SHA512_T_HMAC		–					
CKM_SHA512_T_KEY_DERIVATION							–
CKM_RIPEMD160 ^h				–			
CKM_RIPEMD160_HMAC_GENERAL ^h		–					
CKM_RIPEMD160_HMAC ^h		–					
CKM_PBE_SHA1_DES3_EDE_CBC					–		
CKM_PBE_SHA1_DES2_EDE_CBC					–		
CKM_PBA_SHA1_WITH_SHA1_HMAC					–		
CKM_PKCS5_PBKD2					–		

CKM_SSL3_PRE_MASTER_KEY_GEN					–		
CKM_SSL3_MASTER_KEY_DERIVE							–
CKM_SSL3_MASTER_KEY_DERIVE_DH							–
CKM_SSL3_KEY_AND_MAC_DERIVE							–
CKM_SSL3_MD5_MAC		–					
CKM_SSL3_SHA1_MAC		–					
CKM_TLS_MASTER_KEY_DERIVE							–
CKM_TLS_MASTER_KEY_DERIVE_DH							–
CKM_TLS_KEY_AND_MAC_DERIVE							–
CKM_TLS_PRFF							–
CKM_TLS_KDF							–
CKM_TLS_MAC		–					
CKM_TLS12_MASTER_KEY_DERIVE							–
CKM_TLS12_MASTER_KEY_DERIVE_DH							–
CKM_TLS12_KEY_AND_MAC_DERIVE							–
CKM_TLS12_KEY_SAFE_DERIVE							–
CKM_TLS12_KDF							–
CKM_TLS12_MAC		–					
CKM_WTLS_PRE_MASTER_KEY_GEN					–		
CKM_WTLS_MASTER_KEY_DERIVE							–
CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC							–
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE							–

CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE							–
CKM_WTLS_PRF							–
CKM_CONCATENATE_BASE_AND_KEY							✓
CKM_CONCATENATE_BASE_AND_DATA							✓
CKM_CONCATENATE_DATA_AND_BASE							✓
CKM_XOR_BASE_AND_DATA							✓
CKM_EXTRACT_KEY_FROM_KEY							✓
CKM_CMS_SIG		–	–				
CKM_BLOWFISH_CBC	–					–	
CKM_BLOWFISH_CBC_PAD	–					–	
CKM_BLOWFISH_KEY_GEN					–		
CKM_TWOFISH_CBC	–					–	
CKM_TWOFISH_CBC_PAD	–					–	
CKM_TWOFISH_KEY_GEN					–		
CKM_CAMELLIA_KEY_GEN					–		
CKM_CAMELLIA_ECB	–					–	
CKM_CAMELLIA_CBC	–					–	
CKM_CAMELLIA_CBC_PAD	–					–	
CKM_CAMELLIA_CTR	–					–	
CKM_CAMELLIA_MAC_GENERAL		–					
CKM_CAMELLIA_MAC		–					
CKM_CAMELLIA_ECB_ENCRYPT_DATA							–

CKM_CAMELLIA_CBC_ENCRYPT_DATA							–
CKM_ARIA_KEY_GEN					–		
CKM_ARIA_ECB	–					–	
CKM_ARIA_CBC	–					–	
CKM_ARIA_CBC_PAD	–					–	
CKM_ARIA_MAC_GENERAL		–					
CKM_ARIA_MAC		–					
CKM_ARIA_ECB_ENCRYPT_DATA							–
CKM_ARIA_CBC_ENCRYPT_DATA							–
CKM_SEED_KEY_GEN					–		
CKM_SEED_ECB			–				
CKM_SEED_CBC			–				
CKM_SEED_CBC_PAD	–					–	
CKM_SEED_MAC_GENERAL			–				
CKM_SEED_MAC				–			
CKM_SEED_ECB_ENCRYPT_DATA							–
CKM_SEED_CBC_ENCRYPT_DATA							–
CKM_SECURID_KEY_GEN					–		
CKM_SECURID		–					
CKM_HOTP_KEY_GEN					–		
CKM_HOTP		–					
CKM_ACTI_KEY_GEN					–		
CKM_ACTI		–					

CKM_KIP_DERIVE							–
CKM_KIP_WRAP						–	
CKM_KIP_MAC		–					
CKM_GOST28147_KEY_GEN					–		
CKM_GOST28147_ECB	–					–	
CKM_GOST28147	–					–	
CKM_GOST28147_MAC		–					
CKM_GOST28147_KEY_WRAP						–	
CKM_GOSTR3411				–			
CKM_GOSTR3411_HMAC		–					
CKM_GOSTR3410_KEY_PAIR_GEN					–		
CKM_GOSTR3410		–					
CKM_GOSTR3410_WITH_GOSTR3411		–					
CKM_GOSTR3410_KEY_WRAP						–	
CKM_GOSTR3410_DERIVECKM_ECDSA_SHA3_384							–
PKCS #11 3.0							
CKM_SHA3_384_RSA_PKCS_PSS		✓					
CKM_SHA3_512_RSA_PKCS_PSS		✓					
CKM_DSA_SHA3_224		✓					
CKM_DSA_SHA3_256		✓					
CKM_DSA_SHA3_384		✓					
CKM_DSA_SHA3_512		✓					
CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS					–		

CKM_EC_EDWARDS_KEY_PAIR_GEN					✓		
CKM_EC_MONTGOMERY_KEY_PAIR_GEN					✗		
CKM_ECDSA_SHA3_224		✓					
CKM_ECDSA_SHA3_256		✓					
CKM_ECDSA_SHA3_384		✓					
CKM_ECDSA_SHA3_512		✓					
CKM_EDDSA		✓					
CKM_XEDDSA		✗					
CKM_X3DH_INITIALIZE							✗
CKM_X3DH_RESPOND							✗
CKM_X2RATCHET_INITIALIZE							✗
CKM_X2RATCHET_RESPOND							✗
CKM_X2RATCHET_ENCRYPT		✗				✗	
CKM_X2RATCHET_DECRYPT		✗				✗	
CKM_AES_XTS		✗				✗	
CKM_AES_XTS_KEY_GEN					✗		
CKM_AES_KEY_WRAP_KWP	✓ ²					✓	
CKM_SHA_1_KEY_GEN					✗		
CKM_SHA224_KEY_GEN					✗		
CKM_SHA256_KEY_GEN					✗		
CKM_SHA384_KEY_GEN					✗		
CKM_SHA512_KEY_GEN					✗		
CKM_SHA512_224_KEY_GEN					✗		

CKM_SHA512_256_KEY_GEN					–		
CKM_SHA512_T_KEY_GEN					–		
CKM_SHA3_224				✓			
CKM_SHA3_224_HMAC		✓					
CKM_SHA3_224_HMAC_GENERAL		✓					
CKM_SHA3_224_KEY_DERIVATION							✓
CKM_SHA3_224_KEY_GEN					–		
CKM_SHA3_256				✓			
CKM_SHA3_256_HMAC		✓					
CKM_SHA3_256_HMAC_GENERAL		✓					
CKM_SHA3_256_KEY_DERIVATION							✓
CKM_SHA3_256_KEY_GEN					–		
CKM_SHA3_384				✓			
CKM_SHA3_384_HMAC		✓					
CKM_SHA3_384_HMAC_GENERAL		✓					
CKM_SHA3_384_KEY_DERIVATION							✓
CKM_SHA3_384_KEY_GEN					–		
CKM_SHA3_512				✓			
CKM_SHA3_512_HMAC		✓					
CKM_SHA3_512_HMAC_GENERAL		✓					
CKM_SHA3_512_KEY_DERIVATION							✓
CKM_SHA3_512_KEY_GEN					–		
CKM_SHAKE_128_KEY_DERIVATION							–

CKM_SHAKE_256_KEY_DERIVATION							–
CKM_BLAKE2B_160				–			
CKM_BLAKE2B_160_HMAC		–					
CKM_BLAKE2B_160_HMAC_GENERAL		–					
CKM_BLAKE2B_160_KEY_DERIVE							–
CKM_BLAKE2B_160_KEY_GEN					–		
CKM_BLAKE2B_256				–			
CKM_BLAKE2B_256_HMAC		–					
CKM_BLAKE2B_256_HMAC_GENERAL		–					
CKM_BLAKE2B_256_KEY_DERIVE							–
CKM_BLAKE2B_256_KEY_GEN					–		
CKM_BLAKE2B_384				–			
CKM_BLAKE2B_384_HMAC		–					
CKM_BLAKE2B_384_HMAC_GENERAL		–					
CKM_BLAKE2B_384_KEY_DERIVE							–
CKM_BLAKE2B_384_KEY_GEN					–		
CKM_BLAKE2B_512				–			
CKM_BLAKE2B_512_HMAC		–					
CKM_BLAKE2B_512_HMAC_GENERAL		–					
CKM_BLAKE2B_512_KEY_DERIVE							–
CKM_BLAKE2B_512_KEY_GEN					–		
CKM_SP800_108_COUNTER_KDF							–
CKM_SP800_108_FEEDBACK_KDF							–

CKM_SP800_108_DOUBLE_PIPELINE_KDF							–
CKM_CHACHA20_KEY_GEN					–		
CKM_CHACHA20	–					–	
CKM_SALSA20_KEY_GEN					–		
CKM_SALSA20	–					–	
CKM_POLY1305_KEY_GEN					–		
CKM_POLY1305		–					
CKM_CHACHA20_POLY1305	–						
CKM_SALSA20_POLY1305	–						
CKM_HKDF_DERIVE							–
CKM_HKDF_DATA							–
CKM_HKDF_KEY_GEN					–		
CKM_NULL	–	–	–	–		–	–

Vendor Defined Mechanisms

Mechanisms	Function						
	Encrypt & Decrypt	Sign & Verify	SR & VR¹	Digest	Gen. 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}					

Mechanisms	Function						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key (Pair)	Wrap & Unwrap	Derive
CKM_ECDSA_MULT1		✓ ^{4,5}					
CKM_DES_CBC_WRAP						✓	
CKM_AES_CBC_WRAP						✓	
CKM_ECKA		✓ ⁴					
CKM_ECDSA_ECIES	✓ ²						
CKM_ECDSA_SHA256_DCC		✓					
CKM_UTIMACO_SM2		✓					
CKM_UTIMACO_SM2_SHA256		✓					
CKM_UTIMACO_SM2_SHA384		✓					
CKM_UTIMACO_SM2_SHA512		✓					
CKM_UTIMACO_SM2_SHA3_256		✓					
CKM_UTIMACO_SM2_SHA3_384		✓					
CKM_UTIMACO_SM2_SHA3_512		✓					
CKM_UTIMACO_SM2_SM3		✓					
CKM_UTIMACO_SM2_KEY_PAIR_GEN					✓		
CKM_UTIMACO_SM3				✓			
CKM_UTIMACO_SM4_KEY_GEN					✓		
CKM_UTIMACO_SM4_ECB	✓						
CKM_UTIMACO_SM4_ECB_PAD	✓						
CKM_UTIMACO_SM4_CBC	✓						

Mechanisms	Function						
	Encrypt & Decrypt	Sign & Verify	SR & VR¹	Digest	Gen. Key (Pair)	Wrap & Unwrap	Derive
CKM_UTIMACO_SM4_CBC_PAD	✓						
CKM_UTIMACO_SM4_CFB	✓						
CKM_UTIMACO_SM4_OFB	✓						
CKM_UTIMACO_SM4_CTR	✓						
CKM_UTIMACO_SM4_GCM	✓ ^{2, 6}						
CKM_UTIMACO_SM4_CCM	✓ ^{2, 6}						
CKM_UTIMACO_SM4_GMAC	✓						

SR = SignRecover

VR = VerifyRecover

2 Single-part operations only

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



4 Single-part sign operations only
















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

6 The length of the data to be encrypted must not be zero.

Supported Functions

12.1.1 PKCS #11 Functions

The following table is based on PKCS #11 base specifications version 2.40 and 3.0. It lists all functions as defined by the standard. Functions supported by the SecurityServer are marked with a  character. Functions not supported by the SecurityServer are implemented as function stub which returns CKR_FUNCTION_NOT_SUPPORTED; they are marked with a  character. Functions introduced with PKCS #11 3.0 specification are marked with "(3.0)" behind the function name.

<i>Function</i>	<i>Description</i>	<i>Supported</i>
General purpose functions		
C_Initialize	initializes Cryptoki	
C_Finalize	clean up miscellaneous Cryptoki associated resources	
C_GetInfo	obtains general information about Cryptoki	
C_GetFunctionList	obtains entry points of Cryptoki library functions	
C_GetInterfaceList (3.0)	obtains entry points of Cryptoki library functions	
C_GetInterface (3.0)	obtains entry points of Cryptoki library functions	
Slot and token management functions		
C_GetSlotList	obtains a list of slots in the system	
C_GetSlotInfo	obtains information about a particular slot	
C_GetTokenInfo	obtains information about a particular token	
C_WaitForSlotEvent	waits for a slot event (token insertion, removal, etc.) to occur	
C_GetMechanismList	obtains a list of mechanisms supported by a token	
C_GetMechanismInfo	obtains information about a particular mechanism	
C_InitToken	initializes a token	
C_InitPIN	initializes the normal user's PIN	
C_SetPIN	modifies the PIN of the current user	

<i>Function</i>	<i>Description</i>	<i>Supported</i>
Session management functions		
C_OpenSession		✓
C_CloseSession		✓
C_CloseAllSessions		✓
C_GetSessionInfo		✓
C_SessionCancel (3.0)		✗
C_GetOperationState		✗
C_SetOperationState		✗
C_Login		✓
C_LoginUser	logs a user into a token	✓
C_Logout	logs out from a token	✓
Object management functions		
C_CreateObject	creates an object	✓
C_CopyObject	creates a copy of an object	✓
C_DestroyObject	destroys an object	✓
C_GetObjectSize	obtains the size of an object in bytes	✓
C_GetAttributeValue	obtains an attribute value of an object	✓
C_SetAttributeValue	modifies an attribute value of an object	✓
C_FindObjectsInit	initializes an object search operation	✓
C_FindObjects	continues an object search operation	✓
C_FindObjectsFinal	finishes an object search operation	✓
Encryption functions		

Function	Description	Supported
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	✓
Message-based encryption functions (3.0)		
C_MessageEncryptInit (3.0)	prepares a session for message-based encryption	✓
C_EncryptMessage (3.0)	encrypts a message in a single part	✓
C_EncryptMessageBegin (3.0)	begins a multiple-part encryption operation	✓
C_EncryptMessageNext (3.0)	continues a multiple-part encryption operation	✓
C_MessageEncryptFinal (3.0)	finishes a multiple-part encryption operation	✓
Decryption functions		
C_DecryptInit	initializes a decryption operation	✓
C_Decrypt	decrypts single-part encrypted data	✓
C_DecryptUpdate	continues a multiple-part decryption operation	✓
C_DecryptFinal finishes	finishes a multiple-part decryption operation	✓
Message-based decryption functions (3.0)		
C_MessageDecryptInit (3.0)	prepares a session for message-based decryption	✓
C_DecryptMessage (3.0)	decrypts a message in a single part	✓
C_DecryptMessageBegin (3.0)	begins a multiple-part decryption operation	✓
C_DecryptMessageNext (3.0)	continues a multiple-part decryption operation	✓
C_MessageDecryptFinal (3.0)	finishes a multiple-part decryption operation	✓
Message digesting functions		

<i>Function</i>	<i>Description</i>	<i>Supported</i>
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	✓
Signing and MACing functions		
C_SignInit	initializes a signature operation	✓
C_Sign	signs single-part data	✓
C_SignUpdate	continues a multiple-part signature	✓
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	✓
Message-based signing and MACing functions (3.0)		
C_MessageSignInit (3.0)	prepares a session for message-based signing	—
C_SignMessage (3.0)	signs a message in a single part	—
C_SignMessageBegin (3.0)	begins a multiple-part signing operation	—
C_SignMessageNext (3.0)	continues a multiple-part signing operation	—
C_MessageSignFinal (3.0)	finishes a multiple-part signing operation	—
Functions for verifying signatures and MACs		
C_VerifyInit	initializes a verification operation	✓
C_Verify	verifies a signature on single-part data and MACs	✓

<i>Function</i>	<i>Description</i>	<i>Supported</i>
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	✓
Message-based functions for verifying signatures and MACs (3.0)		
C_MessageVerifyInit (3.0)	prepares a session for message-based verifying	—
C_VerifyMessage (3.0)	verifies a signature in a single part	—
C_VerifyMessageBegin (3.0)	begins a multiple-part verifying operation	—
C_VerifyMessageNext (3.0)	continues a multiple-part verifying operation	—
C_MessageVerifyFinal (3.0)	finishes a multiple-part verifying operation	—
Dual-function cryptographic functions		
C_DigestEncryptUpdate	continues simultaneous multiple-part digesting and encryption operations	✓
C_DecryptDigestUpdate	continues simultaneous multiple-part decryption and digesting operations	✓
C_SignEncryptUpdate	continues simultaneous multiple-part signature and encryption operations	✓
C_DecryptVerifyUpdate	continues simultaneous multiple-part decryption and verification operations	✓
Key management functions		
C_GenerateKey	generates a secret key	✓
C_GenerateKeyPair	generates a public-key/private-key pair	✓

<i>Function</i>	<i>Description</i>	<i>Supported</i>
C_WrapKey	wraps (encrypts) a key	✓
C_UnwrapKey	unwraps (decrypts) a key	✓
C_DeriveKey	derives a key from a base key	✓
Random number generation functions		
C_SeedRandom	mixes in additional seed material to the random number generator	✓
C_GenerateRandom	generates random data	✓
Parallel function management functions		
C_GetFunctionStatus	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL	—
C_CancelFunction	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL	—

12.1.2 PKCS #11 Key Types

<i>Key Type</i>	<i>Description</i>	<i>Supported</i>
CKK_RSA	RSA public or private key	✓
CKK_DSA	DSA public or private key	✓
CKK_EC	EC public or private key	✓
CKK_EC_MONTGOMERY	Montgomery EC public or private key	—
CKK_EC_EDWARDS	Edwards EC public or private key	✓
CKK_DH	Diffie-Hellmann public or private key	✓
CKK_X9_42_DH	X9.42 Diffie-Hellman public or private key	✓
CKK_X2RATCHET	Double Ratchet secret key	—

Key Type	Description	Supported
CKK_GENERIC_SECRET	Generic secret key	✓
CKK_AES	AES secret key	✓
CKK_AES_XTS	Double-length AES secret key	✗
CKK_DES ^h	DES secret key	✓
CKK_DES2	Double-length DES secret key	✓
CKK_DES3	Triple-length DES secret key. Triple DES (TDES) is blocked for FIPS on u.trust Anchor.	✓
CKK_SHA_1_HMAC	Secret key to be used with the SHA-1-HMAC mechanisms	✗
CKK_SHA224_HMAC	Secret key to be used with the SHA-224-HMAC mechanisms	✗
CKK_SHA256_HMAC	Secret key to be used with the SHA-256-HMAC mechanisms	✗
CKK_SHA384_HMAC	Secret key to be used with the SHA-384-HMAC mechanisms	✗
CKK_SHA512_HMAC	Secret key to be used with the SHA-512-HMAC mechanisms	✗
CKK_SHA512_224_HMAC	Secret key to be used with the SHA-512/224 mechanisms	✗
CKK_SHA512_256_HMAC	Secret key to be used with the SHA-512/256 mechanisms	✗
CKK_SHA512_T_HMAC	Secret key to be used with the SHA-512/t mechanisms	✗
CKK_SHA3_224_HMAC	Secret key to be used with the SHA3-224-HMAC mechanisms	✗
CKK_SHA3_256_HMAC	Secret key to be used with the SHA3-256-HMAC mechanisms	✗
CKK_SHA3_384_HMAC	Secret key to be used with the SHA3-384-HMAC mechanisms	✗
CKK_SHA3_512_HMAC	Secret key to be used with the SHA3-512-HMAC mechanisms	✗
CKK_BLAKE2B_160_HMAC	Secret key to be used with the BLAKE2B-160-HMAC mechanisms	✗
CKK_BLAKE2B_256_HMAC	Secret key to be used with the BLAKE2B-256-HMAC mechanisms	✗
CKK_BLAKE2B_384_HMAC	Secret key to be used with the BLAKE2B-384-HMAC mechanisms	✗

<i>Key Type</i>	<i>Description</i>	<i>Supported</i>
CKK_BLAKE2B_512_HMAC	Secret key to be used with the BLAKE2B-512-HMAC mechanisms	—
CKK_BLOWFISH	Blowfish secret key	—
CKK_TWOFISH	Twofish secret key	—
CKK_CAMELLIA	Camellia secret key	—
CKK_ARIA	ARIA secret key	—
CKK_SEED	SEED secret key	—
CKK_SECURID	RSA SecurID secret key	—
CKK_HOTP	OATH HOTP secret key	—
CKK_ACTI	ActivIdentity ACTI secret key	—
CKK_GOST28147	GOST 28147-89 secret key	—
CKK_GOSTR3411	GOST R 34.11-94 domain parameters	—
CKK_GOSTR3410	GOST R 34.10-2001 public or private key	—
CKK_CHACHA20	ChaCha20 secret key	—
CKK_SALSA20	Salsa20 secret key	—
CKK_POLY1305	Poly1305 secret key	—
CKK_HKDF	HMAC-based KDF (HKDF) secret key	—

12.2 Vendor Defined Mechanisms

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

They are included at the end of the previous chapter in their own table.

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 FIPSvalidated 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 first operation performed on the key is locking the used padding mechanism to one of the options in the following table. Once locked only that specific padding mechanism can be combined with the supported hash algorithms for upcoming operations.

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.

<i>Combination of padding mechanism and hash algorithm</i>	<i>Constant</i>
PKCS1 padding mechanism	
SHA-224 hashing algorithm	<code>CKM_SHA224_RSA_PKCS</code>
SHA-256 hashing algorithm	<code>CKM_SHA256_RSA_PKCS</code>
SHA-384 hashing algorithm	<code>CKM_SHA384_RSA_PKCS</code>

<i>Combination of padding mechanism and hash algorithm</i>	<i>Constant</i>
SHA-512 hashing algorithm	CKM_SHA512_RSA_PKCS
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 17: Combinations of padding mechanisms and hash algorithms

13.2 Key Usage in FIPS Mode

If the CryptoServer operates in FIPS mode, the key usage for different purposes is not allowed (see [\[FIPS186-4\] \(p. 113\)](#)). Therefore, each time a key is generated or imported, its usage attributes are provided as required for a specific key type according to [\[PKCS11\] \(p. 113\)](#). The first operation performed on this newly created key is locking its usage to the operation being executed. I.e., key pairs are locked for signing/verification on the first sign/verify operation or for non-signing/non-verification on any other cryptographic operation.

13.3 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		✓ ^{4, 7}			✓ ^{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_SHA224_RSA_PKCS		✓ ^{4, 7}				
CKM_SHA256_RSA_PKCS		✓ ^{4, 7}				
CKM_SHA384_RSA_PKCS		✓ ^{4, 7}				
CKM_SHA512_RSA_PKCS		✓ ^{4, 7}				
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_SHA3_224_RSA_PKCS		✓ ^{4, 7}				
CKM_SHA3_256_RSA_PKCS		✓ ^{4, 7}				
CKM_SHA3_384_RSA_PKCS		✓ ^{4, 7}				
CKM_SHA3_512_RSA_PKCS		✓ ^{4, 7}				

CKM_SHA3_224_RSA_PKCS_PSS		✓ 4, 7				
CKM_SHA3_256_RSA_PKCS_PSS		✓ 4, 7				
CKM_SHA3_384_RSA_PKCS_PSS		✓ 4, 7				
CKM_SHA3_512_RSA_PKCS_PSS		✓ 4, 7				
CKM_DSA		✓ 11, 12				
CKM_DSA_SHA224		✓ 11, 12				
CKM_DSA_SHA256		✓ 11, 12				
CKM_DSA_SHA3_224		✓ 11, 12				
CKM_DSA_SHA3_256		✓ 11, 12				
CKM_DSA_KEY_PAIR_GEN				✓ 13		
CKM_DSA_PARAMETER_GEN				✓ 13		
CKM_EC_KEY_PAIR_GEN (CKM_ECDSA_KEY_PAIR_GEN)				✓ 2		
CKM_ECDSA		✓ 2, 3				
CKM_ECDSA_SHA224		✓ 2, 3				
CKM_ECDSA_SHA256		✓ 2, 3				
CKM_ECDSA_SHA384		✓ 2, 3				
CKM_ECDSA_SHA512		✓ 2, 3				
CKM_ECDH1_COFACTOR_DERIVE						✓ 2, 11
CKM_GENERIC_SECRET_KEY_GEN				✓		

CKM_AES_KEY_GEN				✓		
CKM_AES_ECB	✓				✓	
CKM_AES_CBC	✓				✓	
CKM_AES_CBC_PAD	✓				✓	
CKM_AES_CTR	✓					
CKM_AES_OFB	✓				✓	
CKM_AES_KEY_WRAP	✓				✓	
CKM_AES_KEY_WRAP_PAD	✓				✓	
CKM_AES_KEY_WRAP_KWP	✓				✓	
CKM_AES_CMAC		✓				
CKM_DES3_KEY_GEN				✓ ⁵		
CKM_DES3_ECB	✓ ^{5, 6}				✓ ^{5, 6}	
CKM_DES3_CBC_PAD	✓ ^{5, 6}				✓ ^{5, 6}	
CKM_DH_PKCS_DERIVE						✓ ¹¹
CKM_X9_42_DH_DERIVE						✓ ¹¹
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		✓				
CKM_SHA3_224			✓			
CKM_SHA3_224_HMAC		✓				
CKM_SHA3_224_HMAC_GENERAL		✓				
CKM_SHA3_256			✓			
CKM_SHA3_256_HMAC		✓				
CKM_SHA3_256_HMAC_GENERAL		✓				
CKM_SHA3_384			✓			
CKM_SHA3_384_HMAC		✓				
CKM_SHA3_384_HMAC_GENERAL		✓				
CKM_SHA3_512			✓			
CKM_SHA3_512_HMAC		✓				
CKM_SHA3_512_HMAC_GENERAL		✓				
CKM_ECDSA_SHA3_224		✓ ^{2,3}				
CKM_ECDSA_SHA3_256		✓ ^{2,3}				
CKM_ECDSA_SHA3_384		✓ ^{2,3}				
CKM_ECDSA_SHA3_512		✓ ^{2,3}				
Vendor Defined Mechanisms						

CKM_ECKA		✓ ²				
CKM_AES_CBC_WRAP					✓	

✓ The mechanism is available in FIPS mode

1 Restrictions on RSA padding mechanisms as described above in Section 13.1, "Padding Mechanisms in FIPS Mode".

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 [\[FIPS186-4\]](#) (p. 113), 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 [\[FIPS186-4\]](#), Appendix 6)

4 Only RSA key with a length of at least 2048 bit is allowed for key and signature generation.

5 For DES key encryption, MAC generation and key wrapping are not supported.

6 For DES key generation, DES decryption, MAC verification and key unwrapping only key length of 16 or 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, an additional |P|/|Q| parameter length of 1024/160, 2048/224, 2048/256 or 3072/256 is allowed.

13 For key generation, $|P|/|Q| \geq 2048/224$ is allowed.

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

13.4 Mechanisms Supported in FIPS 140-3 Mode

The following table lists all mechanisms – defined in the PKCS#11 standard and the vendor-specific ones – supported by u.trust Anchor if it is operated in FIPS 140-3 mode.

<i>Mechanism</i>	<i>Functions</i>					
	Encrypt & Decrypt	Sign & Verify	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
<i>PKCS#11 Defined Mechanisms</i>						
CKM_RSA_PKCS		✓ ^{4, 5}			✓ ^{6, 7}	
CKM_RSA_PKCS_OAEP					✓ ^{6, 7}	
CKM_RSA_PKCS_KEY_PAIR_GEN				✓ ^{1, 4}		
CKM_RSA_X9_31_KEY_PAIR_GEN				✓ ^{1, 4}		
CKM_RSA_X9_31		✓ ^{4, 5}				
CKM_SHA224_RSA_PKCS		✓ ^{4, 5}				
CKM_SHA256_RSA_PKCS		✓ ^{4, 5}				
CKM_SHA384_RSA_PKCS		✓ ^{4, 5}				
CKM_SHA512_RSA_PKCS		✓ ^{4, 5}				
CKM_SHA224_RSA_PKCS_PSS		✓ ^{4, 5}				
CKM_SHA256_RSA_PKCS_PSS		✓ ^{4, 5}				
CKM_SHA384_RSA_PKCS_PSS		✓ ^{4, 5}				
CKM_SHA512_RSA_PKCS_PSS		✓ ^{4, 5}				
CKM_SHA3_224_RSA_PKCS		✓ ^{4, 5}				
CKM_SHA3_256_RSA_PKCS		✓ ^{4, 5}				
CKM_SHA3_384_RSA_PKCS		✓ ^{4, 5}				

CKM_SHA3_512_RSA_PKCS		✓ ^{4,5}				
CKM_SHA3_224_RSA_PKCS_PSS		✓ ^{4,5}				
CKM_SHA3_256_RSA_PKCS_PSS		✓ ^{4,5}				
CKM_SHA3_384_RSA_PKCS_PSS		✓ ^{4,5}				
CKM_SHA3_512_RSA_PKCS_PSS		✓ ^{4,5}				
CKM_EC_KEY_PAIR_GEN (CKM_ECDSA_KEY_PAIR_GEN)				✓ ²		
CKM_ECDSA		✓ ^{2,3}				
CKM_ECDSA_SHA224		✓ ^{2,3}				
CKM_ECDSA_SHA256		✓ ^{2,3}				
CKM_ECDSA_SHA384		✓ ^{2,3}				
CKM_ECDSA_SHA512		✓ ^{2,3}				
CKM_ECDH1_COFACTOR_DERIVE						✓ ^{2,8}
CKM_GENERIC_SECRET_KEY_GEN				✓		
CKM_AES_KEY_GEN				✓		
CKM_AES_ECB	✓				✓	
CKM_AES_CBC	✓				✓	
CKM_AES_CBC_PAD	✓				✓	
CKM_AES_CTR	✓					
CKM_AES_OFB	✓				✓	
CKM_AES_KEY_WRAP	✓				✓	

CKM_AES_KEY_WRAP_PAD	✓				✓	
CKM_AES_KEY_WRAP_KWP	✓				✓	
CKM_AES_CMAC		✓				
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		✓				
CKM_SHA3_224			✓			
CKM_SHA3_224_HMAC		✓				
CKM_SHA3_224_HMAC_GENERAL		✓				
CKM_SHA3_256			✓			
CKM_SHA3_256_HMAC		✓				
CKM_SHA3_256_HMAC_GENERAL		✓				
CKM_SHA3_384			✓			

CKM_SHA3_384_HMAC		✓				
CKM_SHA3_384_HMAC_GENERAL		✓				
CKM_SHA3_512			✓			
CKM_SHA3_512_HMAC		✓				
CKM_SHA3_512_HMAC_GENERAL		✓				
CKM_ECDSA_SHA3_224		✓ ^{2,3}				
CKM_ECDSA_SHA3_256		✓ ^{2,3}				
CKM_ECDSA_SHA3_384		✓ ^{2,3}				
CKM_ECDSA_SHA3_512		✓ ^{2,3}				
Vendor Defined Mechanisms						
CKM_ECKA		✓ ²				
CKM_AES_CBC_WRAP					✓	

✓ The mechanism is available in FIPS mode

1 Restrictions on RSA padding mechanisms as described above in Section 13.1, "Padding Mechanisms in FIPS Mode".

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 [\[FIPS186-4\]](#) (p. 113), 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 [\[FIPS186-4\]](#), Appendix 6)

4 Only RSA key with a length of at least 2048 bit is allowed for key and signature generation.

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

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

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

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

14 References

<i>Reference</i>	<i>Titel/Company</i>	<i>Document No.</i>
[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_PKCS11HD]	Learning PKCS#11 in Half a Day – Using the Utimaco HSM Simulator/Utimaco IS GmbH.	2015-0008
[CS_PKCS11T2]	CryptoServer – PKCS#11 p11tool2 – Reference Manual/Utimaco IS GmbH.	2012-0014
[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-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	

Reference	Titel/Company	Document No.
[PKCS11CMS]	"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.	