

Utimaco HSM Simulator

Learning PKCS#11 in Half a Day

Using the Utimaco HSM Simulator



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	1.2.14
Product Version	6.0.0
Date	2024-10-23
Document No.	2015-0008
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	Contents of this Guide	5
1.2	Target Audience	6
1.3	Document Conventions	6
1.4	Abbreviations	7
2	Utimaco's HSM Simulator	8
2.1	Obtaining the HSM Simulator	8
2.2	Installing the HSM Simulator	8
2.3	Configuring for Use by PKCS#11	11
2.4	PKCS#11 - Basic Points to Know About	12
2.4.1	Token	12
2.4.2	Cryptoki	12
2.4.3	Slot	12
2.4.4	User Types	13
2.4.4.1	Security Officer	13
2.4.4.2	User	13
2.4.4.3	Utimaco's Extended User Concept	13
2.4.5	Objects	14
2.5	P11CAT - Get a First Feeling for PKCS#11	15
2.5.1	Create SO and User	15
2.5.2	Create an Object	16
2.5.3	Delete SO, User and Objects	16
3	Writing PKCS#11 Applications	18
3.1	Prerequisites	18
3.1.1	Location of Header and Library Files	18
3.1.2	Nomenclature	19
3.1.3	Compiling the Sample Code	20
3.1.4	Editable Sample Code	21
3.2	Slot Conditioning	21
3.2.1	Initialization	21
3.2.2	The PKCS#11 Session	24
3.2.3	User Creation and Login	24
3.3	Writing the RSA Sample Application	26

3.3.1	The Template Concept.....	26
3.3.2	The Mechanism Concept	26
3.3.3	RSA Key Pair Generation	27
3.3.4	Signing Data	30
3.3.5	Verifying Data	33
4	References	37

1 Introduction

The PKCS#11 standard [\[2, 4\] \(p. 37\)](#) defines a platform-independent API to cryptographic “tokens”, a class of devices that includes things like hardware security modules (HSM), trusted platform modules (TPM) and smart cards.

PKCS#11 itself specifies a C-library implementation. The standard and its included API define data representations and a ‘CRUD’ lifecycle (create, read, update, delete) for the most commonly used cryptographic objects. The term “PKCS#11” refers to the standard; the actual API is named “Cryptoki”.

This guide provides an easy-to-comprehend introduction on how to use the PKCS#11 C-library functions. In order to be effective, however, the guide requires that there be a cryptographic hardware token available to target.

HSMs are computing devices which securely store digital keys and perform cryptographic operations in a secured environment. They differ from other cryptographic devices (TPMs, smart cards) in their ability to react to physical attacks and environmental changes, such as temperature or voltage variations.

Utimaco’s [FIPS 140-2 \(p. 37\)](#) certified HSMs offer active tamper response (FIPS 140-2 Level 4 for physical security, the highest defined). In the event of physical attack (temperature, voltage, chemical or direct), the device will zeroize internally stored secrets, making key material maintained by the HSM inaccessible to an attacker. The HSMs also implement protections against side-channel attacks, for example against differential power analysis, and provide for true, entropy-driven random number generation

For development, training and research purposes, not production environments, Utimaco IS GmbH (referred to below as Utimaco) provides a simulator that emulates an HSM’s behavior (see [EULA \(p. 37\)](#)). This HSM Simulator is available for download from Utimaco’s website (hsm.utimaco.com), and is a suitable token.

While this guide is written with the assumption that the simulator is being used, there is no difference in the code between using a simulator, and using an Utimaco HSM. To change between the simulator and an HSM, the only change required is to a configuration file.

1.1 Contents of this Guide

This guide demonstrates how to develop PKCS#11 applications in general, and how to do so using Utimaco provided equipment in specific. Except for a desktop computer or laptop (Windows or Linux), no additional hardware is required.

The guide is in two parts. The first part explains how to obtain and install Utimaco’s HSM Simulator, and how to perform Utimaco-specific HSM administrative tasks using the Utimaco supplied GUI-based tool “P11CAT” (PKCS#11 CryptoServer Administration Tool).

The second and main part of the guide describes the philosophy behind PKCS#11, and demonstrates, step-by-step, how to develop a PKCS#11 application which will generate an RSA key pair, and then use the keys to sign and verify data.

1.2 Target Audience

The audience for this guide is everyone interested in learning how to perform cryptographic operations via the PKCS#11 Cryptoki API. Whether one is an engineer with some PKCS#11 experience and just want to get an impression of Utimaco's PKCS#11 implementation, or whether one starts out and has no background in the Cryptoki API at all, this guide provides all the necessary steps.

Recommended prerequisite is knowledge of the C programming language, and access to a preferred C development environment

1.3 Document Conventions

We use the following conventions in this manual:


Convention	Use
Monospaced	Code that is given for explanation or as an example, file paths
	This icon will mark Utimaco specific features which go beyond the scope of the PKCS#11 standard.

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.

1.4 Abbreviations

<i>Abbreviation</i>	<i>Meaning</i>
HSM	Hardware Security Module
PKCS	Public Key Cryptography Standards
PKCS#11	PKCS Part 11: The Cryptographic Token Interface Standard
SO	The PKCS#11 cryptographic slot Security Officer
USER	The PKCS#11 cryptographic slot User
ADMIN	The Utimaco administrative user name
API	Application Programming Interface
PIN	A Personal Identification "Number" = a sequence with minimum length of four containing any character of the 7bit ASCII table (in the decimal code range between 32 and 126)
<install_dir>	Utimaco's product installation directory. Default installation directory under Windows is C:\Program Files\Utimaco\SecurityServer.

2 Utimaco's HSM Simulator

Utimaco offers software which simulates their HSMs. From a PKCS#11 perspective, both in terms of cryptographic operations and administrative tasks, the simulator behaves like a real device. Of course, the simulator can't provide protection against physical attacks and its processing speed differs from that of a real HSM.

2.1 Obtaining the HSM Simulator

Go to Utimaco IS GmbH's website (<https://hsm.utimaco.com>), click on "Downloads" and then "Utimaco Portal". After registration and Utimaco's check of accordance with their export regulations, you will be able to download the current simulator version. The zip file will be named `SecurityServerEvaluation-Vx.xx.zip` with the version title given in the file name. This guide uses x.xx to indicate the most current version.



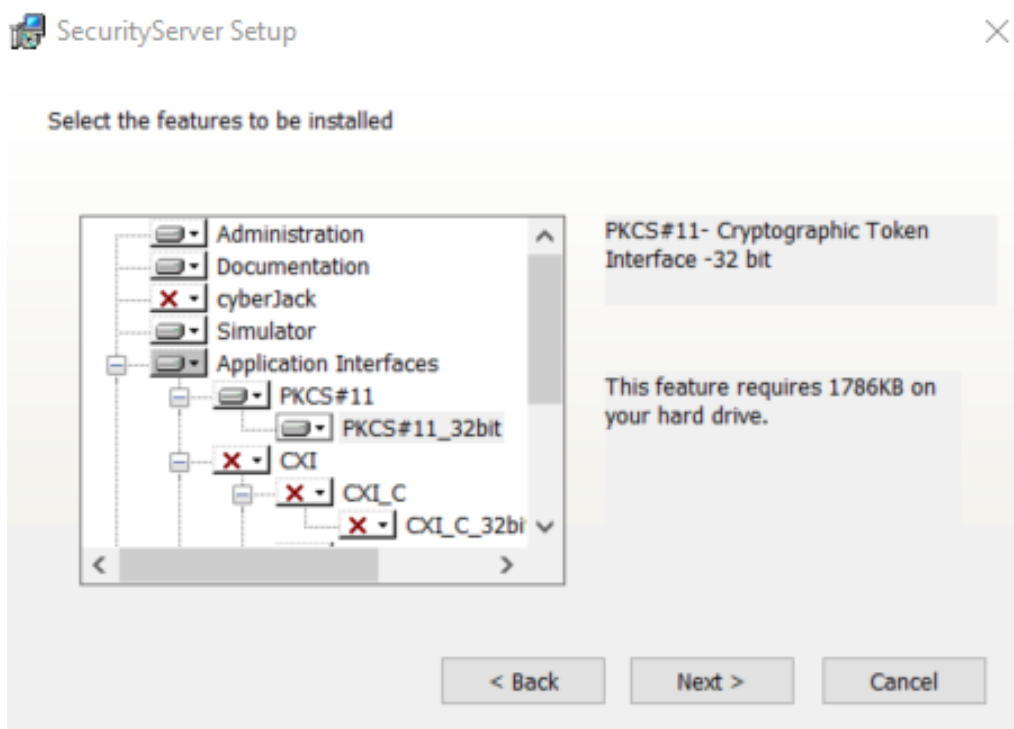
Note that the export regulations check is partially a manual process. Please allow up to 48 hours for the download link email to arrive, and remember to check spam/virus filters.

2.2 Installing the HSM Simulator

Extract `SecurityServerEvaluation-Vx.xx.zip` and start the installer application `CryptoServer-x.xx.msi`. In the Setup-Wizard dialog, chose the Setup Type **Custom**.

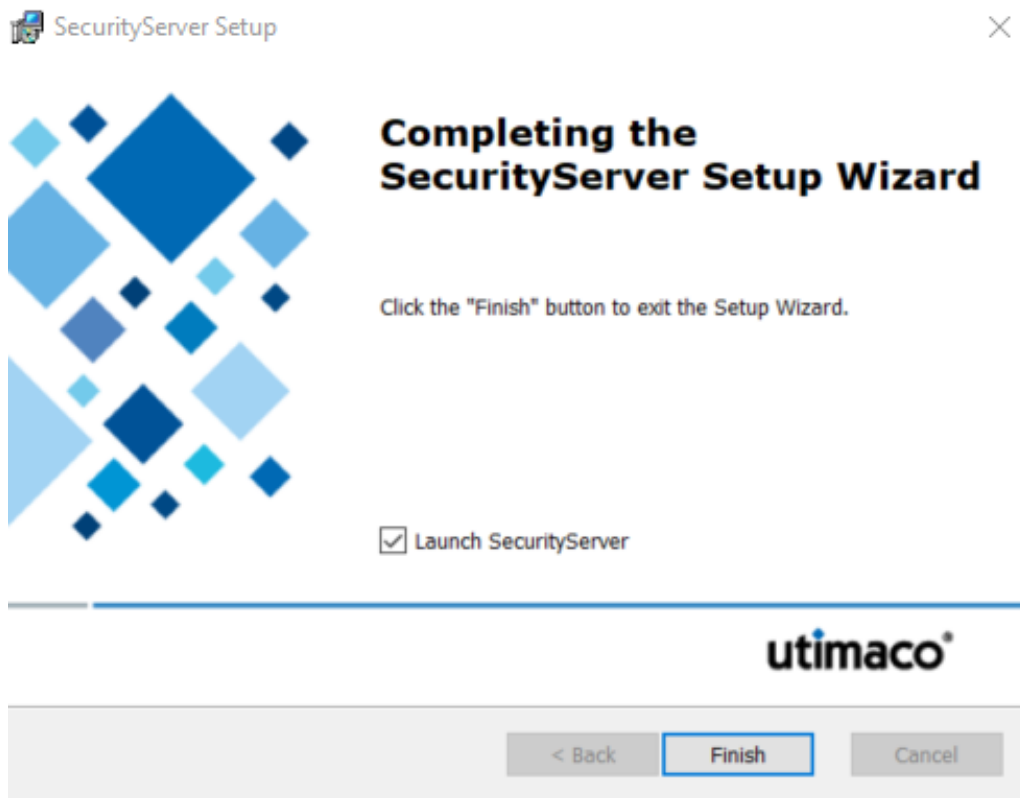


Manually choose to install the CryptoServer Simulator and the PKCS#11 Cryptographic Token Interface.



Proceed with the Installation until it has been completed successfully.

Unmark the "Launch CryptoServer Administration" check box – this guide only focuses on the PKCS#11 interface.



When the installation is finished, the simulator icon shown to the right will be visible on the user desktop.



Start the simulator by double-clicking the icon, and then minimize the window that appears. The simulator will silently run in the background, waiting to receive cryptographic commands for processing.

The simulator is not yet configured for use by PKCS#11. To do so, go to the next section.

2.3 Configuring for Use by PKCS#11

The installer of the HSM Simulator creates an environment variable called `CS_PKCS11_R3_CFG`. It contains the path to Utimaco's PKCS#11 configuration file.

◆ By default, the installer copies the file to the location `C:\ProgramData\Utimaco\PKCS11_R3\cs_pkcs11_R3.cfg` and sets `CS_PKCS11_R3_CFG` to point to that location. This is an Utimaco-specific way of handing over initialization parameters to the PKCS#11 application programming interface (API).

In order to be able to access the HSM Simulator via PKCS#11, do the following:

- Open the configuration file in a plain text editor (notepad, vi, or similar)
- Edit line 53. Replace `Device = 192.168.0.1` by `Device = 3001@127.0.0.1`
Remark:
Utimaco HSM network appliances (CryptoServer LAN) are accessed by a default TCP/IP port at their IP address, a real HSM PCIe card device by `PCI:0` (first PCIe card slot on windows; `PCI:1` for the second slot etc.) or `/dev/cs2.0` (first PCIe card slot on Linux/Unix; `/dev/cs2.1` for the second slot etc.), and the HSM simulator at `<port>@localhost`. Default TCP/IP port for the HSM is 288, default for the simulator is 3001. Utimaco convention is to assume that if no port is given, to use 288. See the [CryptoServer - csadm Manual](#) (p. 37), chapter *Syntax of csadm* for details.

To log all PKCS#11 activities, change the following lines in the configuration file:

- Windows:
Uncomment line 6 (`Logpath = c:/tmp`) and set in line 9 `Logging=3`. Ensure the directory `c:/tmp` If it doesn't, create it
- Linux:
Uncomment line 4 (`Logpath = /tmp`) and set in line 9 `Logging=3`. Ensure the directory `/tmp` If it does not, create it.

(For more details on what else can be set in the configuration file, refer to [CryptoServer - PKCS#11 R3 - Developer Guide](#) (p. 37)).

- Save the edited configuration file.

2.4 PKCS#11 - Basic Points to Know About

With the simulated Hardware Security Module running, start writing PKCS#11 applications. It is helpful to be familiar with PKCS#11's frequently used vocabulary and mechanisms (see [2, 4 \(p. 37\)](#) for details).

2.4.1 Token

The PKCS#11 "token" represents a device which is able to store sensitive information such as cryptographic keys. Such devices perform cryptographic operations without revealing these keys [\[1\] \(p. 37\)](#). Examples of tokens include portable devices such as smart cards and more complex devices such as Hardware Security Modules.

2.4.2 Cryptoki

Cryptoki, pronounced "crypto-key" is short for "CRYPTographic TOfken Interface", the invented name for the application programming interface (API) specified by the PKCS#11 standard. Data types, functions and mechanisms are defined using the ANSI C programming language. It was designed with the following goals in mind [\[1, 2\] \(p. 37\)](#):

- Emphasis is on cryptography. (Other possible functions of the device, such as user or key lifecycle management, are left to other interfaces.)
- Cryptoki offers a logical view onto the device from the application. It is independent of the type and number of underlying hardware devices, and is designed to be isolated from the details of the cryptographic device.
- It is intended for cryptographic devices associated with a single credential user.

Keep these three design goals in mind while proceeding.

2.4.3 Slot

A PKCS#11 "slot" is the interface through which application and device connect. Slots may be best defined by their properties:

- Only one user can be logged onto a slot.
- A slot can be connected to several tokens.
- A token can have several slots to connect to.

- Slots do not share memory space with other slots (Applications running on different slots do not know about or interfere with each other – specifically, key or other cryptographic materials present on one slot are not visible or accessible from users logged into other slots.)

The name “slot” links to PKCS#11’s historical context. PKCS#11 was originally developed for portable, single-user cryptographic devices such as smartcards. The token represents the smartcard and the slot the smartcard reader through which an application can connect to the card.

Cryptoki is designed in an abstract way, isolated from the details of the underlying hardware. This means that the picture of a card and its slot may not exactly match a complex, multi-user capable cryptographic device such as an HSM, where a slot and token are no longer separate physical entities.

An HSM can have several logical slots, and each slot has one administrative and one normal user. These users have rights restricted to only that single slot, and the applications and objects are restricted to slot-specific memory, which is not accessible from within other slots.

2.4.4 User Types

The Cryptoki API itself only knows two types of users. The first is the Security Officer (SO) and the second is the normal user (USER). Each entity has its own, different areas of responsibility. Per each slot, only one SO and one USER entity can exist.

2.4.4.1 Security Officer

The Security Officer (SO) is responsible for administrative tasks such as initializing the slot and assigning a PIN to the USER. The SO is not allowed to access private objects on the slot. When a slot is reinitialized by the SO, the user and all user objects on the slot will be destroyed.

2.4.4.2 User

The USER can access private objects on the slot and perform authenticated cryptographic operations. The user cannot log in to the slot, without first being given a PIN by the SO.

2.4.4.3 Utimaco's Extended User Concept



Utimaco’s HSMs support an extended user concept for additional flexibility and security. This goes hand in hand with an extended authentication scheme that allows for more-secure

access control. These topics are beyond the scope of this Cryptoki-focused document. Please refer to [\[9, 10\] \(p. 37\)](#) for more information.

Have a look in `<install_dir>\Software\PKCS11_R3\doc` and `<install_dir>\Documentation\Administration Guides`.

For now it suffices to remember that there is a layer of administrative control on top of the Security Officer (SO). The “generic” user ADMIN creates the Security Officer on a slot. Or, spoken in PKCS#11 language: the ADMIN initializes the slot in case it hasn’t been initialized yet. The generic user ADMIN can delete and recreate the SO without deleting the user and its objects.



2.4.5 Objects

Cryptoki defines three types of storage objects which can be created, stored and deleted [\[2\] \(p. 37\)](#):

- Data (`CKO_DATA`)
- Certificates (`CKO_CERTIFICATES`)
- Keys (`CKO_PRIVATE_KEY` , `CKO_PUBLIC_KEY` , `CKO_SECRET_KEY`)

In the Cryptoki API, objects are classified according to their lifetime and visibility [\[2\] \(p. 37\)](#):

- Token objects:
 - visible to all applications, and:
 - public objects: no authentication required
 - private objects: authentication required
 - remain on the token, even if a session is closed
- Session objects: (session is explained below later in [The PKCS#11 Session \(p. 24\)](#))
 - visible only to the application which created them
 - are automatically destroyed when the session is closed

In addition to these storable objects, there are objects of type “mechanism” and “hardware feature” which are also defined in the `CK_OBJECT_CLASS` data type. Notionally, all objects

(keys, mechanisms, etc.) consist of a set of attributes. The guide focuses on this in [Writing the RSA Sample Application](#) (p. 26).

2.5 P11CAT - Get a First Feeling for PKCS#11


P11CAT is a graphical user interface tool provided by Utimaco. It provides administrative PKCS#11 tasks like creation and deletion of the Security Officers and Users, as well as lifecycle maintenance of objects. It is a convenient way of understanding the PKCS#11 way of thinking. It visualizes data described in the previous section. (For more details on the functionality of P11CAT see [\[11\]](#) (p. 37).)

2.5.1 Create SO and User

This section assumes that an application will be using slot ID=0; the steps below describe briefly how to create the Security Officer (SO) and User on the slot using P11CAT.

1. If it isn't already running, start the HSM simulator by clicking on the simulator icon on the desktop.
2. Open P11CAT by clicking on the P11 icon on the desktop.



3. In the left sidebar click on the `0000_0000` row under "Slot ID" to select it.
4. Click on "Login."
5.  Login as user ADMIN of type Generic [\[9\]](#) (p. 37):
Select "Login Generic". In section "User Name" type ADMIN. Select "Keyfile". Select "Browse". Go to `<install_dir>\Administration\ADMIN.key`. Click "Login" without giving a password.
6. Click "Slot Management."
7. Create the slot's Security Officer (SO):
Select "Init Token". Set "SO PIN" and confirm. Select "InitToken."
8. Click "Logout". Then, only necessary in releases 3.30: Click "Logout All" to logout from the Generic ADMIN user.
9. Click "Login."

10. Login as SO:
Select "Login SO". Enter SO PIN. Select "Login."
11. Click "Slot Management."
12. Create User:
Select "Init PIN." Set "User PIN" and confirm. Select "Init PIN."
13. Click "Logout," then "Logout All"¹ to logout from the SO.

2.5.2 Create an Object

Once the SO and USER for slot ID=0 are created, the USER can, for example, create an RSA key pair on the slot.

1. Select the slot with ID=0:
Under "Slot ID" click on `0000_0000`.
2. Select "Login."
3. Login as User:
Select "Login User". Enter User PIN. Select "Login."
4. Select "Object Management."
5. Click "Generate."
6. Generate an RSA key pair:
Select "Generate Key Pair." Select Generate.

By default a 2048 bit RSA key has been generated. Click on "List Attributes" to see the key's details.

2.5.3 Delete SO, User and Objects



Specific objects can be deleted in the user's object management area. To view what objects exist on the slot, login as the USER and select "Object Management." In order to delete the SO, the user and all its objects on the slot with ID=0, take the following steps:

1. If still logged in: Select "Logout," then, only necessary in releases > 3.30, click "Logout All".

2. Select "Slot Management."
3. Select "Init Token."
4. Enter SO PIN.
5. Confirm re-initialization of the slot. User and user objects will be deleted.
6. Select "Login."
7. Login as user ADMIN of type Generic – see [Create SO and User \(p. 15\)](#).
8. Select "Slot Management."
9. Select "Delete SO" and confirm the pop-up window with "Yes."
10. Select "Logout," then, only necessary in releases > 3.30, click "Logout All."



3 Writing PKCS#11 Applications

Step-by-step, this guide will develop a simple PKCS#11 application through implementation of the following three tasks:

- Generate an RSA key pair
- Sign data
- Verify signed data

Setting up a build environment is outside the scope of this guide. In general, the compiler and tool suite need to be able to locate the header and library files necessary to compile and link the application. [Prerequisites \(p. 18\)](#) takes some time to mention the necessary library and header files as well as to clarify some of the PKCS#11 standard's naming conventions, before moving on to the "session" and "template" concepts.

The supplied code walks through what was done using the P11CAT in the previous section: it will programmatically initialize a slot, create the SO and USER, and finally log in. Once connected, the application will generate an RSA key pair, and then use the pair to sign and verify a data file.

The code walk-through in the guide does not include error handling. This, as well as a makefile for Linux and Visual Studio project files for Windows can be found in

```
<install_dir>\Software\PKCS11_R3\sample\PKCS11_HandsOn .
```

3.1 Prerequisites

3.1.1 Location of Header and Library Files

With the installation of Utimaco's HSM Simulator and the PKCS#11 component, header files and a dynamic PKCS#11 library file have been installed on your computer.

The necessary header files are found in `<install_dir>\Software\PKCS11_R3\include`. The directory includes the header files given by the standard, as well as a vendor specific header file containing additional types, attributes, mechanisms, structures and functions defined and provided by Utimaco.

- OASIS standard declarations are found in: `h`, `pkcs11.h`, `pkcs11t.h`, `pkcs11f.h`
- Utimaco (vendor) specific declarations are found in: `h`

For Windows, the dynamic link library (`cs_pkcs11_R3.dll`) and matching import library (`cs_pkcs11_R3.lib`) can be found at `<install_dir>\Lib` . For Linux-like operating systems a shared object library (`libcs_pkcs11_R3.so`) and a static library (`libcs_pkcs11_R3_m.a`) are provided.

3.1.2 Nomenclature

PKCS#11 has a rigorous naming convention using prefixes and even standard ANSI C data types receive a new name. Being aware of these prefixes allows one to understand PKCS#11 code more easily. Prefixes used frequently in the introductory applications are highlighted in bold.

C_	Function
CK_	Data type or general constant
CKA_	Attribute
CKC_	Certificate type
CKD_	Key derivation function
CKF_	Bit flag
CKG_	Mask generation function
CKH_	Hardware feature type
CKK_	Key type
CKM_	Mechanism type
CKN_	Notification
CKO_	Object class
CKP_	Pseudo-random function
CKS_	Session State
CKR_	Return value
CKU_	User type
CKZ_	Salt/Encoding parameter source
h	a handle
ul	a CK_ULONG
p	a pointer
pb	a pointer to a CK_BYTE
ph	a pointer to a handle
pul	a pointer to a CK_ULONG

3.1.3 Compiling the Sample Code

The walk-through code printed in this guide compiles with the Microsoft C/C++ compiler, and is valid. If copied, pasted and compiled, it runs. But in order to focus on the essentials, it does not include error checking/handling. However, this guide is accompanied by ready-typed source code (see [Editable Sample Code \(p. 21\)](#)) including proper error handling.

The walk-through code consists of two files which will grow as the tutorial is followed. It is recommended to follow the sections in order.

- The file `pkcs11_handson.c` serves as a repository and contains the functions for the tasks of Initialization, User Creation and Login, Key Pair Generation, Signing Data and Verifying Signed Data.
- The file `main.c` includes `pkcs11_handson.c` via `#include` and is simply the main routine. If building manually make sure to compile only `main.c` since otherwise errors will be thrown.

If running the printed walk-through code “as is”, pay attention to the following:

- Make sure that the include directory for `windows.h` is set. For simplicity sake the printed walk-through code in the guide addresses Windows only. Out of `windows.h` only the functions for the dynamic loading of the PKCS#11 runtime library `cs_pkcs11_R3.dll` are used. A Linux version of the sample code is included in the files accompanying this guide (see [Editable Sample Code \(p. 21\)](#)). There, the Linux counterpart `dlopen.h` is used to dynamically load the PKCS#11 Linux library file `cs_pkcs11_R3.so`.
- Place the six PKCS#11 header files (see [Location of Header and Library Files \(p. 18\)](#)) in the directory where the C applications are being compiled and written or specify an include path via the build tools’ native methods.
- Place copies of the dynamic library file `cs_pkcs11_R3.dll` (see [Location of Header and Library Files \(p. 18\)](#)) and the `ADMIN.key` file in the directory where the generated executable will be started.
(The key file of the HSM Simulator Administrator can be found in `<install_dir>\Administration`)

3.1.4 Editable Sample Code

For convenience, the sample code below is available both in print, as well in an editable form. The editable code includes proper error handling, as well as Project files for Visual C++ and makefiles suitable for compilation under Linux. Several versions of Visual Studio are supported.

The editable C-files and further supplementary material can be found in

`<install_dir>\Software\PKCS11_R3\sample\PKCS11_HandsOn\`. Please read the `readme.txt` file and follow its instructions.

Differences compared to the printed code:

- The location of the dynamic library file isn't hard coded, but will be entered on the command line through the option `-LIB <Path to library file>\cs_pkcs11_R3.dll`
- If it can't be found, the program asks during runtime for the location of the `ADMIN.key` file. The user then needs to enter the location.
- Linux is supported. The sample files can as well be compiled on a Linux machine. Please see `<install_dir>\Software\PKCS11_R3\sample\PKCS11_HandsOn\readme.txt` for further details.
- Error handling is included.

3.2 Slot Conditioning

Before being able to perform any kind of cryptographic operation, the Cryptoki library has to be initialized and a user account has to be created for the given slot. This can be done using the P11CAT tool as shown above, or programmatically in code.

The code below checks for an initialized slot, and if it is not configured, it will initialize it accordingly.

3.2.1 Initialization

The code will grow from section to section, and every new step needs the previous steps to run correctly. Create two files, one named `p_kcs11_handson.c` and the other `main.c`.

Start by implementing the function `Initialize()` which will:

- Load all PKCS#11 functions.
- Make all PKCS#11 functions available for use in form of a function list.

- Initialize the Cryptoki library (also known as the “initialize the token” step). Please be aware that the code below, for the sake of brevity, omits error handling. Please see the code in [Editable Sample Code \(p. 21\)](#) for proper error handling

pkcs11_handson.c – Initialize()

```
#include <windows.h>
#include <stdio.h>
#include "cryptoki.h"
#include "pkcs11t_cs.h"

void Initialize(CK_FUNCTION_LIST_PTR_PTR ppFunctions,
HMODULE *phModule)
{
    char szDllName[] = "cs_pkcs11_R3.dll";
    CK_C_GetFunctionList pC_GetFunctionList = NULL;

    // load PKCS#11 library
    (*phModule) = LoadLibrary(szDllName);

    // get the address of the C_GetFunctionList function
    pC_GetFunctionList=(CK_C_GetFunctionList)GetProcAddress(
    (*phModule),"C_GetFunctionList");

    // get addresses of all the remaining PKCS#11 functions
    pC_GetFunctionList(ppFunctions);

    // initialize token
    (*ppFunctions)->C_Initialize(NULL);
}
```

- From `windows.h` only the two functions `LoadLibrary()` and `GetProcAddress()` are needed. The header file `cryptoki.h` contains the PKCS#11 declarations from the standard while `pkcs11t_cs.h` contains the Utimaco specific PKCS#11 declarations.
- The PKCS#11 library file `cs_pkcs11_R3.dll` is searched according to the Windows default search order, which includes the same directory as the executable being generated and the directories in the PATH. The installer places the DLL in `<install_dir>\Lib` and adds this directory to the path. Alternatively, an absolute path to the library can be specified on this line. The variable `pC_GetFunctionList` will receive the pointer to the function `C_GetFunctionList()`.
- `LoadLibrary(szDllName)` Loads the dynamic PKCS#11 library.

- `GetFunctionList(ppFunctions)` makes all PKCS#11 functions available in form of a function list by returning in `ppFunctions` the pointer to that list.
- `C_Initialize(NULL)` uses the first PKCS#11 function, `C_Initialize()`, to initialize the Cryptoki library.

According to the PKCS#11 standard the parameters handed over to `C_Initialize()` specifies multi-threading behavior for the library [1] (p. 37). Since this application is single-threaded `NULL` is passed. A multi-threaded application must pass a pointer to a valid `CK_C_INITIALIZE_ARGS` struct. See the PKCS#11 specification for more details.

Open a new file – `main.c` – and add the following lines. If using the supplied C source files from the installation, multiple versions of the `main.c` file are supplied, named after the section of the guide they appear in. This `main.c` is named `main_3.2.1_Initialization.c` in the source directory.

`main.c`

```
#include "pkcs11_handson.c"

int main(int argc, char *argv[])
{
    CK_FUNCTION_LIST_PTR pFunctions      = NULL;
    HMODULE               hModule        = NULL;

    // initialize
    Initialize(&pFunctions,&hModule);

    // finalize
    if (pFunctions != NULL) pFunctions->C_Finalize(NULL);
    if (hModule != NULL) FreeLibrary(hModule);
}
```

- `#include "pkcs11_handson.c"` includes the `Initialize()` function. “#include” source code is used in the guide for the sake of brevity and pedagogical simplicity. The full source code in the accompanying package (see [Editable Sample Code \(p. 21\)](#)) includes a proper split into header and source code files, along with proper error handling and other improvements not directly relevant to the understanding of PKCS#11 concepts.
- `CK_FUNCTION_LIST_PTR` declares and defines the variables needed. The variable `pFunctions` will contain the address to a structure holding all PKCS#11 function pointers. The handle `hModule` will receive the file handle to the PKCS#11 library file.
- `Initialize(&pFunctions,&hModule)` calls the `Initialize()` function written previously in `pkcs11_handson.c`.

- Finalize the token by calling `C_Finalize()`.

3.2.2 The PKCS#11 Session

A session establishes a connection between an application and a slot. A session can be:

- Read/write (R/W) = application can create, modify and destroy token objects (see [Objects \(p. 14\)](#))
- Read-only (R/O) = application can only read token objects

Important notice: both types of sessions (R/W and R/O) can create, read, write and destroy session objects (see [Objects \(p. 14\)](#)).

A single session can perform only one operation at a time. The following types of operations are possible in an open session:

- Administrative operations such as logging in
- Object management operations such as creating or destroying an object on the token
- Cryptographic operations such as signing data

A session is identified by a session handle, a Cryptoki-assigned value. It is somewhat similar to a file handle. Functions receive this handle in order to know which session to act on [\[1\] \(p. 37\)](#).

3.2.3 User Creation and Login

For accessing private token objects as well as for performing cryptographic operations a user has to exist on the desired slot. Above, the guide walked through the creation of a user, using the graphical user interface P11CAT (see [P11CAT - Get a First Feeling for PKCS#11 \(p. 15\)](#)). Below, the code achieves the same task programmatically.

The code below extends `pkcs11_handson.c` by providing a function `EnsureUserExistence()`.

pkcs11_handson.c – EnsureUserExistence()

```
#include <windows.h>
```



```
#include <stdio.h>
#include "cryptoki.h"
#include "pkcs11t_cs.h"

void Initialize(CK_FUNCTION_LIST_PTR_PTR ppFunctions,
HMODULE *phModule)
{ ... }

void EnsureUserExistence(CK_FUNCTION_LIST_PTR pFunctions,
char *userPIN,
CK_ULONG slotID)
{
CK_UTF8CHAR slotLabel[32] = "PKCS11 Simulator Token";
char *soPIN = "12345678";
char *adminPIN = "ADMIN,ADMIN.key";
CK_ULONG lenSoPIN = (CK_ULONG)strlen(soPIN);
CK_ULONG lenAdminPIN = (CK_ULONG)strlen(adminPIN);
CK_TOKEN_INFO tinfo;
CK_SESSION_HANDLE hSession = 0;

pFunctions->C_GetTokenInfo(slotID, &tinfo);

pFunctions->C_OpenSession(slotID, CKF_SERIAL_SESSION |
CKF_RW_SESSION, NULL, NULL, &hSession);

// check if SO exists
if((tinfo.flags & CKF_TOKEN_INITIALIZED) == 0)
{
pFunctions->C_Login(hSession, CKU_CS_GENERIC,
(CK_UTF8CHAR_PTR)adminPIN, lenAdminPIN);
pFunctions->C_InitToken(slotID, (CK_UTF8CHAR_PTR)soPIN,
lenSoPIN, slotLabel);
pFunctions->C_Logout(hSession);
}
else printf("-> SO already exists on slot %lu.\n",slotID);

// check if USER exists
if((tinfo.flags & CKF_USER_PIN_INITIALIZED) == 0)
{
pFunctions->C_Login(hSession, CKU_SO,(CK_UTF8CHAR_PTR)soPIN,
lenSoPIN);
pFunctions->C_InitPIN(hSession,(CK_UTF8CHAR_PTR)userPIN,
(CK_ULONG)strlen(userPIN));
pFunctions->C_Logout(hSession);
}
else printf("-> USER already exists on slot %lu.\n",slotID);

pFunctions->C_CloseSession(hSession);
}
```

- The method `EnsureUserExistence()` expects the pointer to the PKCS#11 function list `pFunctions`, the authentication PIN of the user through the variable `userPIN` and the identification number of the slot through `slotID`. The structure `CK_TOKEN_INFO` receives information about a token, like version number, amount of free memory, model, serial number, maximum number of sessions, etc.

3.3 Writing the RSA Sample Application

The following section shows how to implement the sign and verify cryptographic functions, given the existing administrative steps above have been completed.

3.3.1 The Template Concept

Objects are characterized by their attributes in PKCS#11. All attribute types start with “CKA_”. General attributes apply to all objects; like the distinction between public and private data. Other attributes are specific to a particular type of object, e.g. the exponent for RSA keys. PKCS#11 object attributes are always single-valued [2] (p. 37).

The data type `CK_ATTRIBUTE` is a structure. It contains type, value and length of an attribute. An array of `CK_ATTRIBUTE`s is called a “template.”

Quite a lot of PKCS#11 functions use templates as input and output mechanism, as demonstrated in the code below. The object attributes are read, created or manipulated by the function. The template presents a “bundled and selective way” of dealing with the object’s attributes

3.3.2 The Mechanism Concept

A mechanism specifies the algorithm behind a certain cryptographic operation. The PKCS#11 standard [4] (p. 37) presents the default set of mechanisms supported by Cryptoki along with their respective parameters (see [4] (p. 37) or `pkcs11t.h`). Vendors can implement additional mechanisms. Utimaco’s specific mechanism can be found in `pkcs11t_cs.h`.

The data type `CK_MECHANISM` is a structure which is used to specify a particular mechanism and its parameters.

For example, data signing can be performed by a lot of different mechanisms. The information on which of the algorithms is to be used is handed over to the PKCS#11 `C_Sign()` function by means of an argument of type `CK_MECHANISM`.

3.3.3 RSA Key Pair Generation

The sample code better demonstrates the template and mechanism concepts. The first section supplies a `GenerateKeyPair()` function.

pkcs11_handson.c – GenerateKeyPair()

```
#include <windows.h>
#include <stdio.h>
#include "cryptoki.h"
#include "pkcs11t_cs.h"

void Initialize(CK_FUNCTION_LIST_PTR_PTR ppFunctions,
               HMODULE *phModule)
{ ... }

void EnsureUserExistence(CK_FUNCTION_LIST_PTR pFunctions,
                        char *userPIN,
                        CK_ULONG slotID)
{ ... }

void GenerateKeyPair(CK_FUNCTION_LIST_PTR pFunctions,
                    CK_SESSION_HANDLE hSession,
                    CK_OBJECT_HANDLE_PTR phPublicKey,
                    CK_OBJECT_HANDLE_PTR phPrivateKey)
{
    CK_ULONG modulusBits = 2048;
    CK_BYTE publicExponent[] = { 0x01, 0x00, 0x01 };
    CK_BYTE label[] = {"RSA key pair"};
    CK_BYTE keyID[] = {"0"};
    CK_BBOOL bTrue = CK_TRUE;

    CK_ATTRIBUTE publicKeyTemplate[] =
    {
        {CKA_LABEL, label, sizeof(label)-1},
        {CKA_ID, keyID, sizeof(keyID)-1},
        {CKA_TOKEN, &bTrue, sizeof(bTrue)},
        {CKA_ENCRYPT, &bTrue, sizeof(bTrue)},
        {CKA_VERIFY, &bTrue, sizeof(bTrue)},
        {CKA_WRAP, &bTrue, sizeof(bTrue)},
        {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
        {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}
    };

    CK_ATTRIBUTE privateKeyTemplate[] =
    {
        {CKA_LABEL, label, sizeof(label)-1},
        {CKA_ID, keyID, sizeof(keyID)-1},
        {CKA_TOKEN, &bTrue, sizeof(bTrue)},
```

```

{CKA_PRIVATE, &bTrue, sizeof(bTrue)},
{CKA_SENSITIVE, &bTrue, sizeof(bTrue)},
{CKA_DECRYPT, &bTrue, sizeof(bTrue)},
{CKA_SIGN, &bTrue, sizeof(bTrue)},
{CKA_UNWRAP, &bTrue, sizeof(bTrue)}
};

CK_MECHANISM mechanism;
mechanism.mechanism = CKM_RSA_PKCS_KEY_PAIR_GEN; 4
mechanism.pParameter = NULL;
mechanism.ulParameterLen = 0;

pFunctions->C_GenerateKeyPair(hSession, &mechanism, 5
publicKeyTemplate,
sizeof(publicKeyTemplate)/sizeof(CK_ATTRIBUTE),
privateKeyTemplate,
sizeof(privateKeyTemplate)/sizeof(CK_ATTRIBUTE),
phPublicKey, phPrivateKey);
}

```

- The `GenerateKeyPair()` function receives the session handles and the PKCS#11 function list pointer and returns - in case of success - a non-zero object handle to the generated public and private key. Object handles are similar to session handles.
- The RSA public key object's attributes are defined by means of the template `publicKeyTemplate`. The RSA specific attributes are `CKA_MODULUS_BITS` and `CKA_PUBLIC_EXPONENT`. RSA public key objects belong to the object class `CKO_PUBLIC_KEY` and key type `CKK_RSA`. The other attributes in the `publicKeyTemplate` are attributes common to all objects belonging to the object class `CKO_PUBLIC_KEY`. The `CKA_TOKEN` attribute set to `CK_TRUE` ensures that the key remains on the token and won't be destroyed after the session closes.
- The RSA private key object's attributes are defined similar to the public key object. Two of the attributes common to the `CKO_PRIVATE_KEY` class are `CKA_SENSITIVE` and `CKA_EXTRACTABLE`. If `CKA_SENSITIVE` is set to `CK_TRUE` the private key cannot be revealed in plaintext off the token. Sensitive keys can be extracted when they are wrapped (encrypted) though. The strength of this encryption hinges on a number of factors including the wrapping key, its attributes, and the encryption mechanism used. If `CKA_EXTRACTABLE` is set to `CK_FALSE`, then the private key cannot be extracted from a token even when encrypted.
- The type of algorithm to be used for key pair generation is specified in `mechanism`. For details on the mechanisms see [\[4\] \(p. 37\)](#).

- The attributes of private and public key objects (bundled in form of templates) are handed over to the `C_GenerateKeyPair()` function. The function returns handles to the generated key pair.

The following code extends the previously written main-routine in `main.c`. In the installation, the comparable source code file is noted as `main_3.3.3_GenerateKeyPair.c`.

`main.c`

```
#include "pkcs11_handson.c"
int main(int argc, char *argv[])
{
    CK_FUNCTION_LIST_PTR pFunctions = NULL;
    HMODULE hModule = NULL;

    char *userPIN = "12345678";
    CK_ULONG lenUserPIN = (CK_ULONG)strlen(userPIN);
    CK_ULONG slotID = 0;
    CK_SESSION_HANDLE hSession = 0;
    CK_OBJECT_HANDLE hPublicKey = 0;
    CK_OBJECT_HANDLE hPrivateKey = 0;
    // initialize
    Initialize(&pFunctions,&hModule);
    // check for users on slot 0
    EnsureUserExistence(pFunctions, userPIN, slotID);
    // open session
    pFunctions->C_OpenSession(slotID, CKF_SERIAL_SESSION |
    CKF_RW_SESSION, NULL, NULL, &hSession);
1
Writing PKCS#11 Applications
Page 26 of 32 Document version: 1.2.2 Document No.: 2015-0008
    // login as user
    pFunctions->C_Login(hSession, CKU_USER,
    (CK_UTF8CHAR_PTR)userPIN, lenUserPIN);
2
    // generate 2048 bit RSA key pair and return keys
    GenerateKeyPair(pFunctions,hSession, &hPublicKey, &hPrivateKey); 3
    // logout
    pFunctions->C_Logout(hSession); 4

    // close session
    pFunctions->C_CloseSession(hSession); 5
    // finalize
    if (pFunctions != NULL) pFunctions->C_Finalize(NULL);
    if (hModule != NULL) FreeLibrary(hModule);
}
```

- `OpenSession` opens a session. Returns session handle in `hSession`, which will be nonzero on success.
- `Login` logs in as user, who may create key objects.
- `GenerateKeyPair` generates an RSA key pair. On success, returns nonzero public and private key object handles.
- `Logout` logs out as user.
- `CloseSession` closes the session. Verify whether a key pair has been generated by using P11CAT from [P11CAT - Get a First Feeling for PKCS#11 \(p. 15\)](#).

The results of the code above may be generated through the use of P11CAT as well. The following transactions require programming and cannot be done via P11CAT.

3.3.4 Signing Data

Signing data is the process of cryptographically hashing an input file, and then “signing” the resulting hash with a private key. Only the possessor of a specific private key can sign data, such that the signature can be verified through use of the matching public key. The following code extends the `pkcs11_handson.c` file, providing a `SignData()` function.

pkcs11_handson.c – SignData()

```
#include <windows.h>
#include <stdio.h>
#include "cryptoki.h"
#include "pkcs11t_cs.h"

void Initialize(CK_FUNCTION_LIST_PTR_PTR ppFunctions,
               HMODULE *phModule)
{ ... }

void EnsureUserExistence(CK_FUNCTION_LIST_PTR pFunctions,
                        char *userPIN,
                        CK_ULONG slotID)
{ ... }

void GenerateKeyPair(CK_FUNCTION_LIST_PTR pFunctions,
                    CK_SESSION_HANDLE hSession,
                    CK_OBJECT_HANDLE_PTR phPublicKey,
                    CK_OBJECT_HANDLE_PTR phPrivateKey)
```

```

{ ... }

void SignData(CK_FUNCTION_LIST_PTR pFunctions,
              CK_SESSION_HANDLE hSession,
              CK_OBJECT_HANDLE hPrivateKey,
              CK_BYTE *Data,
              CK_ULONG lenData,
              CK_BYTE **signature,
              CK_ULONG *lenSignature)
{
    CK_MECHANISM mechanism;

    mechanism.mechanism = CKM_SHA256_RSA_PKCS;
    mechanism.pParameter = NULL;
    mechanism.ulParameterLen = 0;

    pFunctions->C_SignInit(hSession, &mechanism, hPrivateKey);

    pFunctions->C_Sign(hSession, Data, lenData, NULL, lenSignature);
    *signature=(CK_BYTE_PTR)malloc(sizeof(CK_BYTE)*(*lenSignature));

    pFunctions->C_Sign(hSession, Data, lenData, *signature,
                      lenSignature);
}

```

- In addition to the function list pointer and session handle, a `SignData()` function needs the private key handle and the data to be signed. The signature and information about its length will be returned.
- The signing mechanism is defined here by the `mechanism.mechanism` section.
- `C_SignInit()` initializes a signature operation. `C_SignInit()` can be followed either by `C_Sign()`, which signs in a single part, or by `C_SignUpdate()`, which allows to sign in multiple parts (chunked input). `C_SignUpdate()` needs a call of `C_SignFinal()` to actually obtain the signature. This "Init", "Update" and "Final" sequence applies to many PKCS#11 functions, for example `C_Verify()`, `C_Encrypt()`, etc.
- `pFunctions->C_Sign` Shows another common behavior frequently used in PKCS#11. A function, in this case `C_Sign()`, is called twice. In the first call a NULL pointer is handed over to `C_Sign()`. If this is the case, it only returns the length of the signature in `lenSignature`. Now, the exact size of memory can be allocated to signature using `malloc`.
- In a second call of `C_Sign()` signature has become a non-NULL pointer to memory specifically allocated for the signature in the preceding line of code.

Extend the main method by inserting a call to the `SignData()` function. The source is supplied in the `main_3.3.4_SignData.c`.

main.c

```

#include "pkcs11_handson.c"
int main(int argc, char *argv[])
{
    CK_FUNCTION_LIST_PTR    pFunctions    =    NULL;
    HMODULE                  hModule       =    NULL;

    char                     *userPIN      =    "12345678";
    CK_ULONG                 lenUserPIN    =    (CK_ULONG)strlen(userPIN);
    CK_ULONG                 slotID        =    0;

    CK_SESSION_HANDLE        hSession      =    0;
    CK_OBJECT_HANDLE         hPublicKey    =    0;

    CK_OBJECT_HANDLE         hPrivateKey   =    0;
    CK_BYTE                  *signature    =    NULL ;
    CK_ULONG                 lenSignature  =    0;
    char                     *Data         =    "The standard PKCS11
specifies"

                                "an application programming"
                                "interface (API), called"
                                "Cryptoki, to devices which"
                                "hold cryptographic

information"

                                "and perform cryptographic"
                                "functions.";
    CK_ULONG                 lenData       =    (CK_ULONG)strlen(Data);

    // initialize
    Initialize(&pFunctions,&hModule);

    // check for users on slot
    EnsureUserExistence(pFunctions,userPIN, slotID);

    // open session
    pFunctions->C_OpenSession(slotID, CKF_SERIAL_SESSION |
                             CKF_RW_SESSION, NULL, NULL, &hSession);
    // login as user
    pFunctions->C_Login(hSession, CKU_USER,(CK_UTF8CHAR_PTR)userPIN,
                       lenUserPIN);
    // generate 2048 bit RSA key pair and return keys
    GenerateKeyPair(pFunctions,hSession, &hPublicKey, &hPrivateKey);
    // sign data and return signature
    SignData(pFunctions,hSession,hPrivateKey,(CK_BYTE_PTR)Data,
             lenData, &signature, &lenSignature);

    // logout
    pFunctions->C_Logout(hSession);

```



```
// close session
pFunctions->C_CloseSession(hSession);

// finalize
if (signature != NULL) free(signature);
if (pFunctions != NULL) pFunctions->C_Finalize(NULL);
if (hModule != NULL) FreeLibrary(hModule);
}
```

- `CK_BYTE *signature` declares a variable to hold the signature. Define a variable to hold the data to be signed. It is more likely that this step will involve reading in a file, into that variable.
- `SignData()` calls the sign data function after having created a RSA key pair.

3.3.5 Verifying Data

The last operation will be verifying the signed data. Extend the `pkcs11_handson.c` file by the function `VerifySignedData()`.

pkcs11_handson.c - VerifySignedData()

```
#include <windows.h>
#include <stdio.h>
#include "cryptoki.h"
#include "pkcs11t_cs.h"

void Initialize(CK_FUNCTION_LIST_PTR_PTR ppFunctions,
               HMODULE                    *phModule)
{ ... }

void EnsureUserExistence(CK_FUNCTION_LIST_PTR pFunctions,
                        char                    *userPIN,
                        CK_ULONG               slotID)
{ ... }

void GenerateKeyPair(CK_FUNCTION_LIST_PTR pFunctions,
                    CK_SESSION_HANDLE hSession,
                    CK_OBJECT_HANDLE_PTR phPublicKey,
                    CK_OBJECT_HANDLE_PTR phPrivateKey)
{ ... }

void SignData(CK_FUNCTION_LIST_PTR pFunctions,
              CK_SESSION_HANDLE hSession,
              CK_OBJECT_HANDLE hPrivateKey,
```

```

        CK_BYTE *Data,
        CK_ULONG lenData,
        CK_BYTE **signature,
        CK_ULONG *lenSignature)
{ ... }

void VerifySignedData(CK_FUNCTION_LIST_PTR pFunctions,
                     CK_SESSION_HANDLE hSession,
                     CK_OBJECT_HANDLE hPublicKey,
                     CK_BYTE *Data,
                     CK_ULONG lenData,
                     CK_BYTE *signature,
                     CK_ULONG lenSignature)
{
    CK_MECHANISM mechanism;
    int err;

    mechanism.mechanism          = CKM_SHA256_RSA_PKCS;
    mechanism.pParameter         = NULL;
    mechanism.ulParameterLen     = 0;

    pFunctions->C_VerifyInit(hSession, &mechanism, hPublicKey);

    err = pFunctions->C_Verify(hSession, Data, lenData, signature,
                              lenSignature);

    if (err != CKR_OK)
        printf("[VerifySignedData]: C_Verify returned 0x%08x\n", err);
}

```

- `CK_OBJECT_HANDLE hPublicKey` defines the RSA public key object needed for verification.
- `mechanism.mechanism` sets the mechanism.
- `pFunctions->C_VerifyInit` hands over the mechanism.
- `pFunctions->C_Verify` calls the verify function.
- `printf("[VerifySignedData]` checks whether verification succeeded. In the main-routine a fail case will be triggered on purpose.

To test the `VerifySignedData()` function, extend the main-routine in `main.c`, called `main_3.3.5_VerifySignedData.c` in the C source files accompanying this guide.

main.c

```

#include "pkcs11_handson.c"

int main(int argc, char *argv[])
{
    int err = 0;
    CK_FUNCTION_LIST_PTR pFunctions = NULL;
    HMODULE hModule = NULL;

    char *userPIN = "12345678";
    CK_ULONG lenUserPIN = (CK_ULONG)strlen(userPIN);
    CK_ULONG slotID = 0;

    CK_SESSION_HANDLE hSession = 0;
    CK_OBJECT_HANDLE hPublicKey = 0;
    CK_OBJECT_HANDLE hPrivateKey = 0;

    CK_BYTE *signature = NULL ;
    CK_ULONG lenSignature = 0;
    char *Data = "The standard PKCS11 specifies"
                "an application programming"
                "interface (API), called"
                "Cryptoki,"
                "hold"
                "Cryptoki, to devices which"
                "cryptographic information and"
                "perform cryptographic"
                "functions.";
    CK_ULONG lenData = (CK_ULONG)strlen(Data);

    // initialize
    Initialize(&pFunctions,&hModule);

    // check for users on slot
    EnsureUserExistence(pFunctions,userPIN, slotID);

    // open session and login
    pFunctions->C_OpenSession(slotID, CKF_SERIAL_SESSION |
                             CKF_RW_SESSION, NULL, NULL, &hSession);

    // login as user
    pFunctions->C_Login(hSession, CKU_USER,(CK_UTF8CHAR_PTR)userPIN,
                       lenUserPIN);

    // generate 2048 bit RSA key pair and return keys
    GenerateKeyPair(pFunctions,hSession, &hPublicKey, &hPrivateKey);

    // sign data and return signature
    SignData(pFunctions,hSession,hPrivateKey,(CK_BYTE_PTR)Data,
             lenData, &signature,&lenSignature);

```

```
// verify signed data - PASS CASE
VerifySignedData(pFunctions,hSession,hPublicKey,
                 (CK_BYTE_PTR)Data, lenData,signature,lenSignature);

// verify signed data - FAIL CASE
VerifySignedData(pFunctions,hSession,hPublicKey, "FAIL CASE",
                 strlen("FAIL CASE"),signature,lenSignature);

// logout
pFunctions->C_Logout(hSession);

// close session
pFunctions->C_CloseSession(hSession);

// finalize
if (signature != NULL) free(signature);
if (pFunctions != NULL) pFunctions->C_Finalize(NULL);
if (hModule != NULL) FreeLibrary(hModule);
}
```

- For *Verify signed Data - Pass Case*, no additional variable declaration is necessary. Just add the `VerifySignedData()` function described above.
- *Verify signed data – FAIL CASE*: Trigger failure case by handing over text other than used for signing.

Congratulations ! You have successfully mastered this PKCS#11 tutorial and may now explore it further on your own.

4 References

[2] "PKCS #11 Cryptographic Token Interface Base Specification Version 2.40," edited by Chris Zimman and Dieter Bong, OASIS Standard, 29 May 2019. [Online]. Available: <https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/csprd01/pkcs11-base-v3.0-csprd01.html>.

[4] "PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40," edited by Susan Gleeson and Chris Zimman, OASIS Standard, 14 April 2015. [Online]. Available: <http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/os/pkcs11-curr-v2.40-os.html>.

[6] "National Institute of Standards and Technology, Security Requirements for Cryptographic Modules.," [Online]. Available: <http://csrc.nist.gov/groups/STM/cmvp/standards.html>.

[7] "EULA Utimaco IS GmbH," [Online]. Available: <https://hsm.utimaco.com/documents/eula-utimaco-is-gmbh.pdf>.

[8] HSM-Dokumentation, CryptoServer - Administration Manual, Utimaco IS GmbH, see <install_dir>\Documentation\Administration Guides.

[9] HSM-Dokumentation, CryptoServer - csadm Manual, Utimaco IS GmbH, see <install_dir>\Documentation\Administration Guides.

[10] HSM-Dokumentation, CryptoServer - PKCS#11 R3 - Developer Guide, Utimaco IS GmbH, see <install_dir>\Software\PKCS11_R3\doc.

[11] HSM-Dokumentation, CryptoServer - PKCS#11 P11CAT - Manual, Utimaco IS GmbH, 2015, see <install_dir>\Documentation\Administration Guides.