GO PKCS#11 Wrapper

1.1.1

Integration Guide

CryptoServer HSM

SecurityServer V4.60.0.0

utimaco®

Imprint

Copyright 2026	Utimaco IS GmbH Krefelder Straße 220 52070 Aachen Germany
Phone	AMERICAS +1-844-UTIMACO (+1 844-884-6226) EMEA +49 800-627-3081 APAC +81 800-919-1301
Internet	https://support.hsm.utimaco.com/
e-mail	support@utimaco.com
Document Version	1.0.0
Date	2026-07-27
Status	PUBLISHED
Document No.	IG-2026-0084

Table of Contents

1 Introduction 4
1.1 About This Guide 4
1.2 Target Audience for This Guide 4
1.3 Document Conventions 4
1.4 Abbreviations 5
2 Overview 7
2.1 GO PKCS#11 Wrapper 7
2.2 Utimaco SecurityServer HSM 7
2.3 Utimaco u.trust Anchor 7
3 Integration Requirements and Prerequisites 8
3.1 Tested Versions 8
3.2 Software Requirements 8
3.3 Hardware Requirements 9
3.4 Prerequisites 9
4 Installing and Configuring Utimaco SecurityServer Software 11
4.1 Download and Install Utimaco Software 11
4.2 SecurityServer PKCS#11 Configuration 12
4.3 Create SO User and Initialize a Slot 13
5 Integrating GO PKCS#11 Wrapper with Utimaco HSM 15
5.1 Install GO PKCS11 Package on the Server 15
5.2 Perform Cryptographic Operations using GO PKCS#11 Wrapper 15
5.2.1 AES Key 15
5.2.2 DES3 Key 21
5.2.3 RSA Key 27
5.2.4 ECC Key 33
6 Troubleshooting 38
7 Further Information 39
8 References 40
9 Contact and Support Information 41

1 Introduction

This guide is part of the information and support provided by Utimaco. Additional documentation produced to support your Utimaco SecurityServer product can be found in the document directory of the Utimaco SecurityServer product bundle.

All Utimaco SecurityServer product documentation is available from Utimaco's website at *https://utimaco.com/*.

1.1 About This Guide

This guide describes how to integrate an Utimaco CryptoServer Hardware Security Module (HSM) with GO PKCS#11 Wrapper. With the help of GO PKCS#11 wrapper you can securely generate keys on Utimaco HSMs and perform cryptographic operations.

1.2 Target Audience for This Guide

This guide is intended for GO PKCS#11 and Utimaco HSM administrators.

1.3 Document Conventions

The following conventions are used in this guide:

Convention	Use	Example
Bold	Items of the Graphical User Interface (GUI), e.g., menu options	Select **Details** and click on **Properties** button
`Monospaced`	Code that is given for explanation or as an example, file paths	`certreq.exe -new request.inf IISCertRequest.csr`
Italic	References and important terms	Operating system listed in *Tested Versions*

Table 1: Document conventions

We use special icons to highlight the most important notes and information.

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

Here you will find additional notes or supplementary information.

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

1.4 Abbreviations

The following abbreviations are used in this guide:

Abbreviation	Meaning
AES	Advanced Encryption Standard
API	Application Programming Interface
CSADM	CryptoServer Command-line Administration Tool
CSAR	Cloud Service Architecture
DES3	Triple Data Encryption Standard
ECC	Elliptic Curve Cryptography
GUI	Graphical User Interface

Abbreviation	Meaning
HSM	Hardware Security Module
IP	Internet Protocol
LAN	Local Area Network
P11CAT	PKCS#11 CryptoServer Administration Tool
PCIe	PCI Express Interface
PKCS#11	Public-Key Cryptography Standard #11
RSA	Rivest, Shamir, Adleman (cryptosystem)
SO	Security Officer
URL	Uniform Resource Locator
VM	Virtual Machine

Table 2: List of abbreviations

2 Overview

2.1 GO PKCS#11 Wrapper

GO PKCS#11 Wrapper is Go implementation of the PKCS#11 API. It wraps the library closely but uses Go idiom to seamlessly add the capability to use hardware cryptography using the PKCS#11 Cryptographic Token Interface standard.

2.2 Utimaco SecurityServer HSM

SecurityServer is a hardware security module developed by Utimaco IS GmbH. SecurityServer is a physically protected specialized computer unit designed to perform sensitive cryptographic tasks and to securely manage, as well as store, cryptographic keys and data. It can be used as a universal, independent security component for heterogeneous computer systems.

2.3 Utimaco u.trust Anchor

u.trust Anchor is the next generation hardware security module platform developed by Utimaco IS GmbH. u.trust Anchor is a physically protected specialized computer unit designed for true multi-tenancy, performing sensitive cryptographic tasks, and to securely manage as well as store cryptographic keys and data. It can be used as a universal, independent security component for heterogeneous computer systems.

3 Integration Requirements and Prerequisites

Ensure the system environment you will be using meets the following hardware and software requirements.

3.1 Tested Versions

The integrations that have been successfully tested with the Utimaco HSM with GO PKCS11.

Operating System	GO PKCS#11 Wrapper Version	Utimaco Security Server Version	Utimaco HSM
RHEL 8	1.1.1	SecurityServer V4.60.0.0	CryptoServer CSe-Series/Se-Series u.trust Anchor Se*k and u.trust Anchor CSAR

Table 3: List of tested versions

3.2 Software Requirements

Software	Software Requirements
GO Version	1.19.10
Host VM	Redhat 8 and above
HSM software	Utimaco SecurityServer Software 4.60.0.0

Software	Software Requirements
P11tool2	p11tool2 (3.1.1) from product package Utimaco SecurityServer 4.60.0.0

Table 4: List of software requirements

3.3 Hardware Requirements

Hardware	Hardware Requirements
Utimaco LAN HSM	CryptoServer CSe-Series/Se-Series LAN with firmware SecurityServer 4.60.0.0 or higher u.trust Anchor Se*k and CSAR Series LAN with firmware 4.60.0.0 or higher
Utimaco PCI-e HSM	CryptoServer CSe-Series/Se-Series PCI-e with firmware SecurityServer 4.60.0.0 or higher u.trust Anchor Se*k and CSAR Series PCI-e with firmware 4.60.0.0 or higher

Table 5: List of hardware requirements

Set up an account on the Utimaco support portal and request download access at the following URL https://support.hsm.utimaco.com/ .

3.4 Prerequisites

Before you begin, please ensure that:

- The operating system installed/set up is listed in *Tested Versions*.
- The SecurityServer installed/set up is listed in *Tested Versions*.
- The SecurityServer Default Admin has been replaced with a new admin user.

- The SecurityServer is set up and configured. Refer to the SecurityServer documentation to set up the HSM.
- The PKCS#11 library is set up and configured as per your environment. Refer to the SecurityServer documentation to set up and configure the PKCS#11 library.
- You get yourself familiar with GO PKCS#11 Wrapper documentation.
- There is a user with root privilege as it is required to install software.

4 Installing and Configuring Utimaco SecurityServer Software

4.1 Download and Install Utimaco Software

If you have not already done so, please create and request an Utimaco Support Portal Account. This will allow you to download the software components needed for this installation.

1. Copy the downloaded software at the appropriate location on the GO PKCS#11 Server.
2. Create a utimaco folder under `/opt` directory and further create 2 directories `/opt/utimaco/bin` and `/opt/utimaco/lib`.

›_ Console

```
# mkdir -p /opt/utimaco/bin

# mkdir /opt/utimaco/lib
```

3. Copy pkcs11 library file `libcs_pkcs11_R3.so` from Utimaco SecurityServer software to the `/opt/utimaco/lib` directory.

›_ Console

```
# cp ~/path_to_application_folder/lib/libcs_pkcs11_R3.so /opt/utimaco/lib
```

4. Copy the csadm and p11tool2 files from Utimaco SecurityServer software to `/opt/utimaco/bin` directory and make both the files executable.

›_ Console

```
# cd ~/path_to_application_folder

# cp csadm p11tool2 /opt/utimaco/bin

# chmod +x /opt/utimaco/bin/csadm /opt/utimaco/bin/p11tool2
```

4.2 SecurityServer PKCS#11 Configuration

1. Create the directory `/etc/utimaco`. Locate the Utimaco PKCS#11 configuration file in your SecurityServer directory, `Linux/x86-64/Crypto_APIS/PKCS11_R3/sample`.

Copy the Utimaco PKCS#11 configuration file `cs_pkcs11_R3.cfg` into `/etc/utimaco` directory.

›_ Console

```
# mkdir /etc/utimaco

# cd <install directory>/Software/Linux/x86-64/Crypto_APIs/PKCS11_R3/sample # cp
cs_pkcs11_R3.cfg /etc/utimaco # cd /etc/utimaco
```

2. Edit the `cs_pkcs11_R3.cfg` file and make the appropriate changes to the file.

›_ Console

```
[Global]
# For unix:
Logpath = /tmp
# Loglevel (0 = NONE; 1 = ERROR; 2 = WARNING; 3 = INFO; 4 = TRACE)
Logging = 1
Keepalive = true
# Set the Device to connect with
[CryptoServer]
# Device specifier
Device = <HSM_IP>
```

For more information regarding the commands and command parameters please check the CryptoServer documentation. The device may be a CryptoServer (PCIe or LAN) device.

The device line will follow one of these patterns, based on the HSM form-factor:
Device = 288@<HSM IP address> Hardware (LAN) HSM

OR

Device = /dev/cs2.0 Hardware (PCIe) HSM

To make your testing easier, it would be good to enable the PKCS#11 log file.

That can be enabled by editing the **Logging Loglevel**. Set the **LogPath** and **Logging Loglevel** to 1. For testing you may want to increase it to 4. The added **LogPath** points to a writable directory, not to a file.

If you encounter problems, check the log file named `cs_pkcs11_R3.log` in the **LogPath** defined directory. When you are done testing, you should change **Logging** to 1 or 2. This will limit the logging to only critical and important messages.

4.3 Create SO User and Initialize a Slot

You must initialize a slot with a custom label using p11tool2.

First, using p11tool2 create the SO or Security Officer and then using p11tool2 command initialize the slot that you want to use, and the slot user as shown below.

›_ Console

```
# p11tool2 slot=<slot_no.> Label=<token_label> Login=ADMIN,<ADMIN.key>
InitToken=<ask>

# p11tool2 slot=<slot_no.> LoginSO=<ask> SetPin=<ask><ask>

# p11tool2 slot=<slot_no.> LoginSO=<ask> InitPIN=ask

# p11tool2 slot=<slot_no.> LoginUser=<ask> SetPIN=ask,ask
```

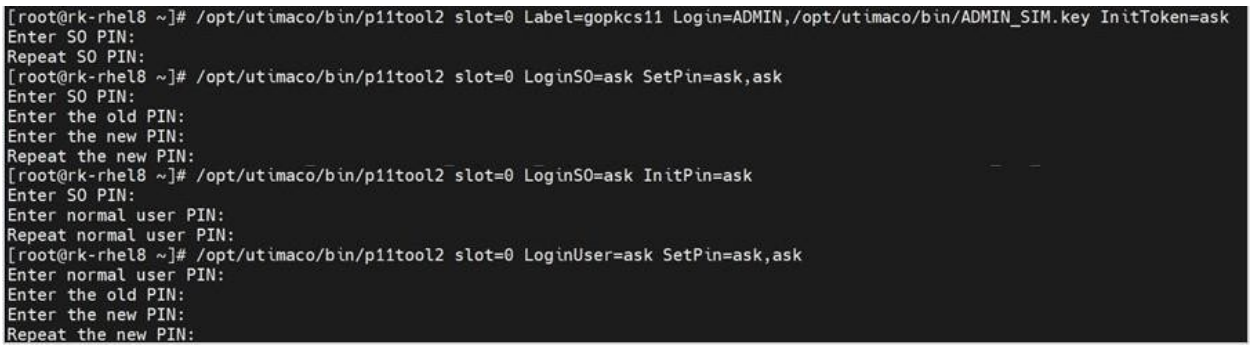

Figure 1 : Slot initialization output

5 Integrating GO PKCS#11 Wrapper with Utimaco HSM

5.1 Install GO PKCS11 Package on the Server

1. Install GO first if it is not installed.

›_ Console

```
# dnf install golang
```

Figure 2 : Installing Golang on RHEL 8

2. Install PKCS#11 package.

›_ Console

```
# go install github.com/miekg/pkcs11@latest
```

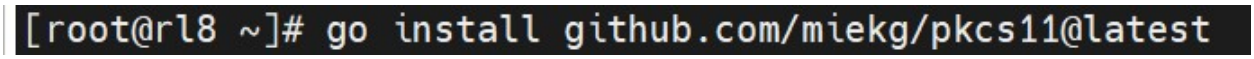

Figure 3 : Installing pkcs11 package

5.2 Perform Cryptographic Operations using GO PKCS#11 Wrapper

5.2.1 AES Key

1. Create a `aes.go` file and add the contents as shown below.

aes.go

```
package main
import (
 "bytes"
 "crypto/rand"
 "crypto/sha256"
 "encoding/base64"
 "encoding/binary"
 "fmt"
 "log"
 "github.com/miekg/pkcs11"
)
const (
 pin = "87654321"
)
var ()
func main() {
 // Init PKCS11
 p := pkcs11.New("/opt/utimaco/lib/libcs_pkcs11_R3.so")
 err := p.Initialize()
 if err != nil {
 panic(err)
 }
 defer p.Destroy()
 defer p.Finalize()
 slots, err := p.GetSlotList(true)
 if err != nil {
 panic(err)
 }
 session, err := p.OpenSession(slots[0],
pkcs11.CKF_SERIAL_SESSION|pkcs11.CKF_RW_SESSION)
 if err != nil {
 panic(err)
 }
 defer p.CloseSession(session)
 err = p.Login(session, pkcs11.CKU_USER, pin)
 if err != nil {
 panic(err)
 }
 defer p.Logout(session)
 info, err := p.GetInfo()
 if err != nil {
 panic(err)
 }
 fmt.Printf("CryptokiVersion.Major %v", info.CryptokiVersion.Major)
 fmt.Println()
 // Create AES key
 buf := new(bytes.Buffer)
 var num uint16 = 1
```

aes.go

```
 err = binary.Write(buf, binary.LittleEndian, num)
 if err != nil {
 fmt.Println("binary.Write failed:", err)
 }
 id := buf.Bytes()
 aesKeyTemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_AES),
 pkcs11.NewAttribute(pkcs11.CKA_ENCRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_DECRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_EXTRACTABLE, true), // we don't
need to extract this..
 pkcs11.NewAttribute(pkcs11.CKA_SENSITIVE, true),
 pkcs11.NewAttribute(pkcs11.CKA_VALUE_LEN, 32), /* KeyLength */
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "AESKey"), /*
Name of Key */
 }
 aesKey, err := p.GenerateKey(session,
 []*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_KEY_GEN
, nil)},
 aesKeyTemplate)
 if err != nil {
 panic(fmt.Sprintf("GenerateKey() failed %s\n", err))
 }
 log.Printf("Created AES Key: %v", aesKey)
 ktemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_ID, id),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "AESKey"),
 }
 aesKeyTemplate2 := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_AES),
 pkcs11.NewAttribute(pkcs11.CKA_ENCRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_DECRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_DECRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_WRAP, true),
 pkcs11.NewAttribute(pkcs11.CKA_UNWRAP, true),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_EXTRACTABLE, true), // we don't
need to extract this..
 pkcs11.NewAttribute(pkcs11.CKA_SENSITIVE, true),
 pkcs11.NewAttribute(pkcs11.CKA_VALUE_LEN, 32), /* KeyLength */
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "AESKeyToWrap"), /*
Name of Key */
 }
 aesKey2, err := p.GenerateKey(session,
 []*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_KEY_GEN
, nil)},
```

aes.go

```
 aesKeyTemplate2)
 if err != nil {
 panic(fmt.Sprintf("GenerateKey() failed %s\n", err))
 }
 log.Printf("Created AES Key: %v", aesKey2)
 if err := p.FindObjectsInit(session, ktemplate); err != nil {
 panic(err)
 }
 if err != nil {
 panic(err)
 }
 if err = p.FindObjectsFinal(session); err != nil {
 panic(err)
 }
 iv := make([]byte, 16)
 _, err = rand.Read(iv)
 if err != nil {
 panic(err)
 }
 // Encryption
 err = p.EncryptInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_CBC_PAD, iv)}, aesKey)
 if err != nil {
 panic(fmt.Sprintf("EncryptInit() failed %s\n", err))
 }


 ct, err := p.Encrypt(session, []byte("foo"))
 if err != nil {
 panic(fmt.Sprintf("Encrypt() failed %s\n", err))
 }
 // append the IV to the ciphertext
 cdWithIV := append(iv, ct...)
 log.Printf("Encrypted IV+Ciphertext %s",
base64.RawStdEncoding.EncodeToString(cdWithIV))
 // Decryption
 err = p.DecryptInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_CBC_PAD,
cdWithIV[0:16])}, aesKey)
 if err != nil {
 panic(fmt.Sprintf("EncryptInit() failed %s\n", err))
 }
 pt, err := p.Decrypt(session, ct[:16])
 if err != nil {
 panic(fmt.Sprintf("Encrypt() failed %s\n", err))
 }
 log.Printf("Decrypt %s", string(pt))
 // Signing
 msg := []byte("foo")
 digest := sha256.Sum256(msg)
```

aes.go

```
 log.Printf("Decrypt %s", string(pt))
 err = p.SignInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_CBC, nil)}, aesKey)

 sig, err := p.Sign(session, msg)
 // Verification
 err = p.VerifyInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_CBC, nil)}, aesKey)
 err = p.Verify(session, digest[:], sig)
 // Wrapping
 wrappedPrivBytes, err := p.WrapKey(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_KEY_WRAP, nil)},
aesKey, aesKey2)
 if err != nil {
 log.Fatalf("failed to wrap privatekey : %s\n", err)
 }
 buf = new(bytes.Buffer)
 num = 6
 err = binary.Write(buf, binary.LittleEndian, num)
 if err != nil {
 log.Fatalf("binary.Write failed: %v", err)
 }
 importedPrivID := buf.Bytes()
 aesKeyTemplate2 = []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_AES),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "UnwrappedAESKey"), /*
Name of Key */
 pkcs11.NewAttribute(pkcs11.CKA_ID, importedPrivID),
 }
 // Unwrapping
 ik, err := p.UnwrapKey(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_KEY_WRAP, nil)},
aesKey, wrappedPrivBytes, aesKeyTemplate2)
 _ = ik
 if err != nil {
 log.Fatalf("Unwrap Failed: %v", err)
 }
}
```

Update Slot PIN and PKCS#11 library path according to your environment.

2. Execute the go code using the command below.

›_ Console

```
#  go run aes.go
```

3. Verify that keys are generated on HSM using below command.

›_ Console

```
#   p11tool2 Slot=<slot_no.> LoginUser=ask ListObjects
```

```
[root@rk-rhel8 pkcs11@v1.1.1]# /opt/utimaco/bin/p11tool2 Slot=0 LoginUser=ask ListObjects
Enter normal user PIN:


CKO_SECRET_KEY:

+ 1.1
  CKA_KEY_TYPE                  = CKK_AES
  CKA_UNIQUE_ID                 = 25F4EA11-8C8A-4A19-AF8E-7B439A88EED5
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_TRUE
  CKA_LABEL                     = AESKey
  CKA_ID                        =


+ 1.2
  CKA_KEY_TYPE                  = CKK_AES
  CKA_UNIQUE_ID                 = 866FED68-7BD2-4B02-B071-F2BF51071568
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_TRUE
  CKA_LABEL                     = AESKeyToWrap
  CKA_ID                        =


+ 1.3
  CKA_KEY_TYPE                  = CKK_AES
  CKA_UNIQUE_ID                 = 7E6F3A95-47DF-46A1-978A-D8A80BF6177D
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_FALSE
  CKA_LABEL                     = UnwrappedAESKey
  CKA_ID                        = 0x0600 ( )
[root@rk-rhel8 pkcs11@v1.1.1]#
```

Figure 4 : List keys

5.2.2 DES3 Key

1. Create a `des3.go` file and add the contents as shown below.

des3.go

```
package main
import (
 "bytes"
 "crypto/sha256"
 "crypto/rand"
 "encoding/base64"
 "encoding/binary"
 "fmt"
 "log"
 "github.com/miekg/pkcs11"
)
const (
 pin = "87654321"
)
var ()
func main() {
 // Init PKCS11
 p := pkcs11.New("/opt/utimaco/lib/libcs_pkcs11_R3.so")
 err := p.Initialize()
 if err != nil {
 panic(err)
 }
 defer p.Destroy()
 defer p.Finalize()
 slots, err := p.GetSlotList(true)
 if err != nil {
 panic(err)
 }
 session, err := p.OpenSession(slots[0],
pkcs11.CKF_SERIAL_SESSION|pkcs11.CKF_RW_SESSION)
 if err != nil {
 panic(err)
 }
 defer p.CloseSession(session)
 err = p.Login(session, pkcs11.CKU_USER, pin)
 if err != nil {
 panic(err)
 }
 defer p.Logout(session)
 info, err := p.GetInfo()
 if err != nil {
 panic(err)
 }
 fmt.Printf("CryptokiVersion.Major %v", info.CryptokiVersion.Major)
 fmt.Println()
 // Create DES3 key
 // first lookup the key
 buf := new(bytes.Buffer)
```

des3.go

```
 var num uint16 = 1
 err = binary.Write(buf, binary.LittleEndian, num)
 if err != nil {
 fmt.Println("binary.Write failed:", err)
 }
 id := buf.Bytes()
 desKeyTemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_DES3),
 pkcs11.NewAttribute(pkcs11.CKA_ENCRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_DECRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_EXTRACTABLE, true), // we don't
need to extract this..
 pkcs11.NewAttribute(pkcs11.CKA_SENSITIVE, true),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "DESKey"), /*
Name of Key */
}

 desKey, err := p.GenerateKey(session,

[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_DES3_KEY_GEN , nil)},
 desKeyTemplate)
 if err != nil {
 panic(fmt.Sprintf("GenerateKey() failed %s\n", err))
 }
 log.Printf("Created DES3 Key: %v", desKey)
 ktemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_ID, id),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "DESKey"),
 }
 if err := p.FindObjectsInit(session, ktemplate); err != nil {
 panic(err)
 }
 aesKeyTemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_AES),
 pkcs11.NewAttribute(pkcs11.CKA_ENCRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_DECRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_DECRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_WRAP, true),
 pkcs11.NewAttribute(pkcs11.CKA_UNWRAP, true),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_EXTRACTABLE, true), // we don't
need to extract this..
 pkcs11.NewAttribute(pkcs11.CKA_SENSITIVE, true),
 pkcs11.NewAttribute(pkcs11.CKA_VALUE_LEN, 32), /* KeyLength */
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "AESKeyToWrap"), /*
Name of Key */
```

des3.go

```
 }
 aesKey, err := p.GenerateKey(session,
 []*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_KEY_GEN
, nil)},
aesKeyTemplate)
 if err != nil {
 panic(fmt.Sprintf("GenerateKey() failed %s\n", err))
 }
 log.Printf("Created AES Key: %v", aesKey)
 // Wrapping
 wrappedPrivBytes, err := p.WrapKey(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_DES3_CBC, nil)}, desKey,
aesKey)
 buf = new(bytes.Buffer)
 num = 6
 err = binary.Write(buf, binary.LittleEndian, num)
 if err != nil {
 log.Fatalf("binary.Write failed: %v", err)
 }
 importedPrivID := buf.Bytes()
 aesKeyTemplate = []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_AES),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "UnwrappedDES3Key"), /*
Name of Key */
 pkcs11.NewAttribute(pkcs11.CKA_ID, importedPrivID),
 }
 // Unwrapping
ik, err := p.UnwrapKey(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_DES3_CBC, nil)}, desKey,
wrappedPrivBytes, aesKeyTemplate)
 _ = ik
 if err != nil {
 log.Fatalf("Unwrap Failed: %v", err)
 }
 if err != nil {
 log.Fatalf("failed to wrap privatekey : %s\n", err)
 }
 if err != nil {
 panic(err)
 }
 if err = p.FindObjectsFinal(session); err != nil {
 panic(err)
 }
 iv := make([]byte, 8)
 _, err = rand.Read(iv)
 if err != nil {
 panic(err)
 }
```

des3.go

```
 // Encryption
 err = p.EncryptInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_DES3_CBC, iv)}, desKey )
 if err != nil {
 panic(fmt.Sprintf("EncryptInit() failed %s\n", err))
 }
 ct, err := p.Encrypt(session, []byte("fooooooo"))
 if err != nil {
 panic(fmt.Sprintf("Encrypt() failed %s\n", err))
 }
 // append the IV to the ciphertext
 cdWithIV := append(iv, ct...)
 log.Printf("Encrypted IV+Ciphertext %s",
base64.RawStdEncoding.EncodeToString(cdWithIV))
 // Decryption
err = p.DecryptInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_DES3_CBC, cdWithIV[0:8])},
desKey)
 if err != nil {
 panic(fmt.Sprintf("EncryptInit() failed %s\n", err))
 }
 pt, err := p.Decrypt(session, ct[:8])
 if err != nil {
 panic(fmt.Sprintf("Encrypt() failed %s\n", err))
 }
 log.Printf("Decrypt %s", string(pt))
 // Signing
 msg := []byte("foo")
 digest := sha256.Sum256(msg)
 log.Printf("Decrypt %s", string(pt))
 err = p.SignInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_DES3_CBC, nil)}, desKey)
 sig, err := p.Sign(session, msg)
 // Verification
 err = p.VerifyInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_DES3_CBC, nil)}, desKey)
 err = p.Verify(session, digest[:], sig)
}
```

Update Slot PIN and PKCS#11 library path according to your environment.

2. Execute the go code using the command below.

›_ Console

```
# go run des3.go
```

3. Verify that keys are generated on HSM using below command.

›_ Console

```
# p11tool2 Slot=<slot_no.> LoginUser=ask ListObjects
```

```
[root@rk-rhel8 pkcs11@v1.1.1]# /opt/utimaco/bin/p11tool2 Slot=0 LoginUser=ask ListObjects
Enter normal user PIN:


CKO_SECRET_KEY:

+ 1.1
  CKA_KEY_TYPE                  = CKK_DES3
  CKA_UNIQUE_ID                 = FAFDF76D-F0F7-431E-A03D-2B1EF085F4A2
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_TRUE
  CKA_LABEL                     = DESKey
  CKA_ID                        =


+ 1.2
  CKA_KEY_TYPE                  = CKK_AES
  CKA_UNIQUE_ID                 = E2DFA1D4-32C7-4ED6-9C5D-C06FE41EBB24
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_TRUE
  CKA_LABEL                     = AESKeyToWrap
  CKA_ID                        =


+ 1.3
  CKA_KEY_TYPE                  = CKK_AES
  CKA_UNIQUE_ID                 = 9DFDF9B3-CEA9-4D5C-B491-BB7F58F1D5FF
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_FALSE
  CKA_LABEL                     = UnwrappedDES3Key
  CKA_ID                        = 0x0600 ()
[root@rk-rhel8 pkcs11@v1.1.1]#
```

Figure 5 : List keys

5.2.3 RSA Key

1. Create a `rsa.go` file and add the contents as shown below.

rsa.go

```
package main
import (
 "bytes"
 "encoding/base64"
 "crypto/sha256"
 "encoding/binary"
 "fmt"
 "log"
 "github.com/miekg/pkcs11"
)
const (
 pin = "87654321"
)
var ()
func main() {
 // Init PKCS11
 p := pkcs11.New("/opt/utimaco/lib/libcs_pkcs11_R3.so")
 err := p.Initialize()
 if err != nil {
 panic(err)
 }
 defer p.Destroy()
 defer p.Finalize()
 slots, err := p.GetSlotList(true)
 if err != nil {
 panic(err)
 }
 session, err := p.OpenSession(slots[0],
pkcs11.CKF_SERIAL_SESSION|pkcs11.CKF_RW_SESSION)
 if err != nil {
 panic(err)
 }
 defer p.CloseSession(session)
 err = p.Login(session, pkcs11.CKU_USER, pin)
 if err != nil {
 panic(err)
 }
 defer p.Logout(session)
 info, err := p.GetInfo()
 if err != nil {
 panic(err)
 }
 fmt.Printf("CryptokiVersion.Major %v", info.CryptokiVersion.Major)
 fmt.Println()
 buf := new(bytes.Buffer)
 buf = new(bytes.Buffer)
 var num uint16 = 1
 num = 1
```

rsa.go

```
 err = binary.Write(buf, binary.LittleEndian, num)
 if err != nil {
 log.Fatalf("binary.Write failed: %v", err)
 }
 pubID := buf.Bytes()
 buf = new(bytes.Buffer)
 num = 2
 err = binary.Write(buf, binary.LittleEndian, num)
 if err != nil {
 log.Fatalf("binary.Write failed: %v", err)
 }
 privID := buf.Bytes()
 publicKeyTemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_PUBLIC_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_RSA),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_VERIFY, true),
 pkcs11.NewAttribute(pkcs11.CKA_ENCRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_WRAP, true),
 pkcs11.NewAttribute(pkcs11.CKA_MODULUS_BITS, 2048),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "pub1"),
 pkcs11.NewAttribute(pkcs11.CKA_ID, pubID),
 }
 privateKeyTemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_PRIVATE_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_RSA),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_SIGN, true),
 pkcs11.NewAttribute(pkcs11.CKA_DECRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "priv1"),
 pkcs11.NewAttribute(pkcs11.CKA_PRIVATE, true),
 pkcs11.NewAttribute(pkcs11.CKA_SENSITIVE, true),
 pkcs11.NewAttribute(pkcs11.CKA_WRAP_WITH_TRUSTED, false),
 pkcs11.NewAttribute(pkcs11.CKA_UNWRAP, true),
 pkcs11.NewAttribute(pkcs11.CKA_EXTRACTABLE, true),
 pkcs11.NewAttribute(pkcs11.CKA_ID, privID),
 }
 pbk, pvk, err := p.GenerateKeyPair(session,

[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_RSA_PKCS_KEY_PAIR_GEN,
nil)},
 publicKeyTemplate, privateKeyTemplate)
 var (
 // OAEPLabel defines the label we use for OAEP encryption; this cannot
be changed
 OAEPLabel = []byte("")
 // OAEPSha256Params describes the OAEP parameters with sha256 hash
algorithm
 OAEPSha256Params = &pkcs11.OAEPParams{
 HashAlg: pkcs11.CKM_SHA256,
```

rsa.go

```
 MGF: pkcs11.CKG_MGF1_SHA256,
 SourceType: pkcs11.CKZ_DATA_SPECIFIED,
 SourceData: OAEPLabel,
 }
)
 // Encryption
 err = p.EncryptInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_RSA_PKCS_OAEP,
OAEPSha256Params)}, pbk )
 if err != nil {
 panic(fmt.Sprintf("EncryptInit() failed %s\n", err))
 }
 ct, err := p.Encrypt(session, []byte("fooooooo"))
 if err != nil {
 panic(fmt.Sprintf("Encrypt() failed %s\n", err))
 }
 // Decryption
 err = p.DecryptInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_RSA_PKCS_OAEP,
OAEPSha256Params)}, pvk)
 plaintext, err := p.Decrypt(session, ct)
 _ = plaintext
 _ = err
 // Signing
 msg := []byte("foo")
 fmt.Printf("Signing %d bytes with %s\n", len(msg), msg)
 err = p.SignInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_SHA256_RSA_PKCS, nil)},
pvk)
 if err != nil {
 log.Fatalf("Signing Initiation failed (%s)\n", err.Error())
 }
 sig, err := p.Sign(session, msg)
 if err != nil {
 err = fmt.Errorf("Signing failed (%s)\n", err.Error())
 return
 }
 log.Printf("Signature %s", base64.RawStdEncoding.EncodeToString(sig))
 // Verification
 digest := sha256.Sum256(msg)
 err = p.VerifyInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_SHA256_RSA_PKCS, nil)},
pbk)
 err = p.Verify(session, digest[:], sig)
 log.Printf(">>>>>> Signature Verified")
 id := buf.Bytes()
 //Create AES key which is enabled for wrapping
 aesKeyTemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_AES),
```

rsa.go

```
 pkcs11.NewAttribute(pkcs11.CKA_ENCRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_DECRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_WRAP, true),
 pkcs11.NewAttribute(pkcs11.CKA_UNWRAP, true),
 pkcs11.NewAttribute(pkcs11.CKA_VERIFY, false),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_PRIVATE, true),
 pkcs11.NewAttribute(pkcs11.CKA_EXTRACTABLE, true), // we don't
need to extract this..
 pkcs11.NewAttribute(pkcs11.CKA_SENSITIVE, true),
 pkcs11.NewAttribute(pkcs11.CKA_VALUE_LEN, 32),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "WrappingAESKey"), /*
Name of Key */
 pkcs11.NewAttribute(pkcs11.CKA_ID, id),
 }
 importedPrivID := buf.Bytes()
 aesKeyTemplate2 := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_SECRET_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_AES),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "UnwrappedAESKey"), /*
Name of Key */
 pkcs11.NewAttribute(pkcs11.CKA_ID, importedPrivID),
 }
aesKey, err := p.GenerateKey(session,
 []*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_AES_KEY_GEN
, nil)},
 aesKeyTemplate)
 log.Printf("Created AES Key: %v", aesKey)
{
 // Wrapping
wrappedPrivBytes, err := p.WrapKey(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_RSA_PKCS, nil)}, pbk,
aesKey)
 // Unwrapping
 ik, err := p.UnwrapKey(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_RSA_PKCS, nil)}, pvk,
wrappedPrivBytes, aesKeyTemplate2)
 _ = ik
 _ = err
}
}
```

Update Slot PIN and PKCS#11 library path according to your environment.

2. Execute the go code using the command below.

›_ Console

```
# go run rsa.go
```

3. Verify that keys are generated on HSM using below command.

›_ Console

```
# p11tool2 Slot=<slot_no.> LoginUser=ask ListObjects
```

```
[root@rk-rhel8 pkcs11@v1.1.1]# /opt/utimaco/bin/p11tool2 Slot=0 LoginUser=ask ListObjects
Enter normal user PIN:


CKO_PUBLIC_KEY:

+ 1.1
  CKA_KEY_TYPE                  = CKK_RSA
  CKA_UNIQUE_ID                 = E194ED95-688B-41CE-81EE-15F5983D4B9C
  CKA_LABEL                     = pub1
  CKA_ID                        = 0x0100 ()


CKO_PRIVATE_KEY:

+ 2.1
  CKA_KEY_TYPE                  = CKK_RSA
  CKA_UNIQUE_ID                 = 3F6B5E98-EBA5-46A7-BD72-3963618AA88F
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_TRUE
  CKA_LABEL                     = priv1
  CKA_ID                        = 0x0200 ()


CKO_SECRET_KEY:

+ 3.1
  CKA_KEY_TYPE                  = CKK_AES
  CKA_UNIQUE_ID                 = EA9FC1ED-0ADA-47CF-8BFB-DD0AB1968CB3
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_FALSE
  CKA_LABEL                     = UnwrappedAESKey
  CKA_ID                        = 0x0200 ()

+ 3.2
  CKA_KEY_TYPE                  = CKK_AES
  CKA_UNIQUE_ID                 = C309FEA7-9E24-4D38-A518-8816642C09EA
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_TRUE
  CKA_LABEL                     = WrappingAESKey
  CKA_ID                        = 0x0200 ()
```

Figure 6 : List keys

5.2.4 ECC Key

1. Create a `ecckey.go` file and add the contents as shown below.

ecckey.go

```
# package main
import (
 "bytes"
 "crypto/sha256"
 "encoding/binary"
 "fmt"
 "log"
 "github.com/miekg/pkcs11"
)
const (
 pin = "87654321"
)
var ()
func main() {
 // Init PKCS11
 p := pkcs11.New("/opt/utimaco/lib/libcs_pkcs11_R3.so")
 err := p.Initialize()
 if err != nil {
 panic(err)
 }
 defer p.Destroy()
 defer p.Finalize()
 slots, err := p.GetSlotList(true)
 if err != nil {
 panic(err)
 }
 session, err := p.OpenSession(slots[0],
pkcs11.CKF_SERIAL_SESSION|pkcs11.CKF_RW_SESSION)
if err != nil {
 panic(err)
 }
 defer p.CloseSession(session)
 err = p.Login(session, pkcs11.CKU_USER, pin)
 if err != nil {
 panic(err)
 }
 defer p.Logout(session)
 info, err := p.GetInfo()
 if err != nil {
 panic(err)
 }
 fmt.Printf("CryptokiVersion.Major %v", info.CryptokiVersion.Major)
 fmt.Println()
 buf := new(bytes.Buffer)
 buf = new(bytes.Buffer)
 var num uint16 = 1
 num = 1
 err = binary.Write(buf, binary.LittleEndian, num)
```

ecckey.go

```
 if err != nil {
 log.Fatalf("binary.Write failed: %v", err)
 }
 pubID := buf.Bytes()
 buf = new(bytes.Buffer)
 num = 2
 err = binary.Write(buf, binary.LittleEndian, num)
 if err != nil {
 log.Fatalf("binary.Write failed: %v", err)
 }
 privID := buf.Bytes()
 publicKeyTemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_PUBLIC_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_ECDSA),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_VERIFY, true),
 pkcs11.NewAttribute(pkcs11.CKA_PRIVATE, true),
 pkcs11.NewAttribute(pkcs11.CKA_ENCRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_WRAP, false),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "pub1"),
 pkcs11.NewAttribute(pkcs11.CKA_ID, pubID),
 pkcs11.NewAttribute(pkcs11.CKA_EC_PARAMS, []byte{0x06, 0x08,
0x2a, 0x86, 0x48, 0xce, 0x3d, 0x03, 0x01, 0x07}),
 }
 privateKeyTemplate := []*pkcs11.Attribute{
 pkcs11.NewAttribute(pkcs11.CKA_CLASS, pkcs11.CKO_PRIVATE_KEY),
 pkcs11.NewAttribute(pkcs11.CKA_KEY_TYPE, pkcs11.CKK_ECDSA),
 pkcs11.NewAttribute(pkcs11.CKA_TOKEN, true),
 pkcs11.NewAttribute(pkcs11.CKA_SIGN, true),
 pkcs11.NewAttribute(pkcs11.CKA_DECRYPT, true),
 pkcs11.NewAttribute(pkcs11.CKA_LABEL, "priv1"),
 pkcs11.NewAttribute(pkcs11.CKA_PRIVATE, true),
 pkcs11.NewAttribute(pkcs11.CKA_SENSITIVE, true),
 pkcs11.NewAttribute(pkcs11.CKA_WRAP_WITH_TRUSTED, false),
 pkcs11.NewAttribute(pkcs11.CKA_UNWRAP, false),
 pkcs11.NewAttribute(pkcs11.CKA_EXTRACTABLE, false),
 pkcs11.NewAttribute(pkcs11.CKA_ID, privID),
 }
 pbk, pvk, err := p.GenerateKeyPair(session,

[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_ECDSA_KEY_PAIR_GEN, nil)},
 publicKeyTemplate, privateKeyTemplate)
 _ = pvk
 _ = pbk
 if err != nil {
 log.Fatalf("failed to generate exportable keypair: %s\n", err)
 }
 // Signing
 msg := []byte("foo")
 digest := sha256.Sum256(msg)
```

ecckey.go

```
 err = p.SignInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_ECDSA_SHA1, nil)}, pvk)
 sig, err := p.Sign(session, msg)
 // Verification
  err = p.VerifyInit(session,
[]*pkcs11.Mechanism{pkcs11.NewMechanism(pkcs11.CKM_ECDSA_SHA1, nil)}, pbk)
 err = p.Verify(session, digest[:], sig)
}
```

Update Slot PIN and PKCS#11 library path according to your environment.

2. Execute the go code using the command below.

›_ Console

```
# go run ecckey.go
```

3. Verify that keys are generated on HSM using below command.

›_ Console

```
# p11tool2 Slot=<slot_no.> LoginUser=ask ListObjects
```

```
[root@rk-rhel8 pkcs11@v1.1.1]# /opt/utimaco/bin/p11tool2 Slot=0 LoginUser=ask ListObjects
Enter normal user PIN:


CKO_PUBLIC_KEY:

+ 1.1
  CKA_KEY_TYPE                  = CKK_ECDSA
  CKA_UNIQUE_ID                 = 848CF138-1D5F-45F1-BEA1-9D07EF85F9E3
  CKA_LABEL                     = pub1
  CKA_ID                        = 0x0100 ()


CKO_PRIVATE_KEY:

+ 2.1
  CKA_KEY_TYPE                  = CKK_ECDSA
  CKA_UNIQUE_ID                 = DE06F2E9-B853-484D-86E0-96A5D49F6BB8
  CKA_SENSITIVE                 = CK_TRUE
  CKA_EXTRACTABLE               = CK_FALSE
  CKA_LABEL                     = priv1
  CKA_ID                        = 0x0200 ()
[root@rk-rhel8 pkcs11@v1.1.1]#
```

Figure 7 : List keys

This completes the Integration for GO PKCS#11 Wrapper with Utimaco SecurityServer.

6 Troubleshooting

Error	Diagnosis
p11_login: C_Login [type=1] returned Error 0x00000102 (CKR_USER_PIN_NOT_INITIALIZED)	PKCS#11 Slot is not initialized.
The CryptoServer PKCS#11 Library R3 is not initialized. Error CKR_CRYPTOKI_NOT_INITIALIZED occurred	PKCS#11 Slot is not initialized.

Table 6: List of errors and their diagnoses

7 Further Information

This document forms a part of the information and support which is provided by Utimaco IS GmbH. Additional documentation can be found on the product CD in the Documentation directory.

All CryptoServer product documentation is also available at the Utimaco IS GmbH website: *https://utimaco.com/*.

8 References

Reference	Title/Company	Document No.
[CSADMIN]	CryptoServer – csadm Manual/Utimaco IS GmbH	2009-0003
[CSTrSh]	CryptoServer Troubleshooting/Utimaco IS GmbH	M011-0008-en
[CSADMIN2]	CryptoServer_csadm_Manual_Systemadministrators.pdf	2009-0003
[CSP11Tool2]	CryptoServer_p11tool2_Manual.pdf	2012-0004
[CSPKCSM]	CryptoServer - PKCS#11 P11CAT Manual	M013-0001-en
[CSLAN5]	CryptoServerLAN_Manual_Systemadministrators.pdf	2018-0004

Table 7: List of references

9 Contact and Support Information

You can reach us from Monday to Friday, 09.00 a.m. to 05.00 p.m., Central European Time (CET).

Utimaco IS GmbH
Krefelder Str. 220
52070 Aachen
Germany

RMA Query

If you need to send the device back to Utimaco IS GmbH, please open a new RMA case (Return Merchandise Authorization). We request that you use the following web address. RMA cases cannot be opened by email or phone.

https://support.hsm.utimaco.com/support/rma/new

Other Support Queries

- Mail (preferred contact method)
  support@utimaco.com
  Attach the diagnostic information to your email.
- Web portal
  https://support.hsm.utimaco.com/support/cases/new/
  The diagnostic information will be requested in our response if necessary.
- By phone
  AMERICAS +1-844-UTIMACO (+1 844-884-6226)
  EMEA +49 800-627-3081
  APAC +81 800-919-1301
  The diagnostic information will be requested in our response if necessary.