Code-Signing Tool – HSM

User's Guide

Rev. 3.3.1

Aug 2020

Revision Sheet

Release No.	Date	Revision Description
Rev. 0	24/05/2018	Initial Work
Rev. 1	27/09/2018	Added support for AHAB and for OpenSSL 1.1.0
Rev. 2	13/08/2020	Add Copyright

USER'S GUIDE

TABLE OF CONTENTS

Page #

1.0 GENERAL INFORMATION

This document provides the information necessary for the user to effectively use Code-Signing Tool with Hardware Security Module backend. It is primarily intended for users who are familiar with CST tool to sign codes for the NXP High Assurance Boot (HAB) and NXP Advanced High Assurance Boot (AHAB).

A. GETTING STARTED

Refereeing to "Appendix B, Replacing the CST Backend Implementation" of Code-Signing Tool User's Guide, NXP has architected the Code-Signing Tool in two parts a Front-End and a Back-End. The Front-End contains all the NXP proprietary operations, while the Back-End containing all standard cryptographic operations. Users can write a replacement for the reference backend to interface with a HSM.

The reference backend uses OpenSSL to perform HAB signature generation and encrypted data generation. OpenSSL in his turn, exposes an Engine API, which makes it possible to plug in alternative implementations for some of the cryptographic operations implemented by OpenSSL. We can take advantage of this to reuse the reference backend with a HSM by offloading cryptographic operations involved during signature generation to HSM.

The engine should re-write an implementation of RSA private encrypt function, ECDSA sign function and how public certificates and private keys are loaded from the HSM to the appropriate data structure X509, RSA, EVP_PKEY and EC_KEY. Optionally SHA digest functions can be re-written also to be performed at the HSM level.

1.1 PKCS#11 enabled HSM

PKCS#11 is a standardized interface for cryptographic tokens which exposes an API called Cryptoki. Cryptographic token manufacturers provide shared libraries (.so or .dll) which implements PKCS#11 standard. Those libraries can be used as bridge between the HSM and the CST Back-End. Typically, the engine loads the shared PKCS#11 module and invoke the appropriate functions on it.

The following figure illustrates the architecture of Code-Signing Tool with a Backend replacement to interface with a PKCS#11 enabled HSM.

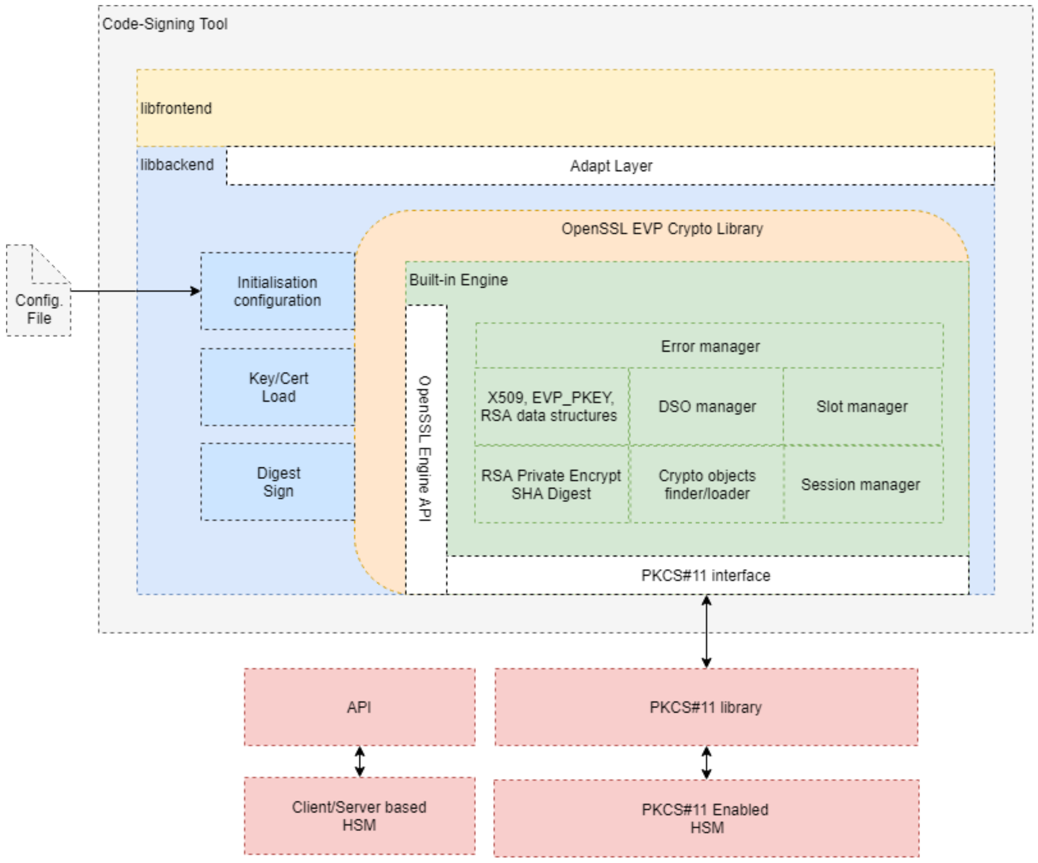


Figure 1HSM-backed CST architecture

The green part of the architecture represents a custom built-in OpenSSL engine which uses PKCS#11 interface to interact with HSM via PKCS#11 implementation library provided by the HSM vendor. It implements all required functions to manage session and tokens, load public certificates, private keys, sign and hash.

Slight modification should be done at the reference backend (blue part) to initialize the engine, perform control command and configuration. In the reference backend public certificates and private keys are loaded from file system. This should be replaced respectively by ENGINE_load_certificate and ENGINE_load_private_key to load cryptographic material from HSM into appropriate data structures which can be used later. Those functions are bound to low-level functions implemented in the engine.

Below is a non-exhaustive list of well-known smartcards/tokens which support PKCS#11 and the suggested library filename to use.

- aetpkss1.dll (for G&D StarCos and Rainbow iKey 3000)
- cs2_pkcs11.dll (for Utimaco CryptoServer LAN)
- CccSigIT.dll (for IBM MFC)
- pk2priv.dll (for GemSAFE, old version)
- gclib.dll (for GemSAFE, new version)
- dspkcs.dll (for Dallas iButton)

- slbck.dll (for Schlumberger Cryptoflex and Cyberflex Access)
- SetTokI.dll (for SeTec)
- acpkcs.dll (for ActivCard)
- psepkcs11.dll (for A-Sign Premium)
- id2cbox.dll (for ID2 PKCS#11)
- smartp11.dll (for SmartTrust PKCS#11)
- pkcs201n.dll (for Utimaco Cryptoki for SafeGuard)
- dkck201.dll (for DataKey and Rainbow iKey 2000 series)
- cryptoki.dll (for Eracom CSA)
- AuCryptoki2-0.dll (for Oberthur AuthentIC)
- eTpkcs11.dll (for Aladdin eToken, and some Siemens Card OS cards)
- cknfast.dll (for nCipher nFast or nShield)
- cryst201.dll (for Chrysalis LUNA)
- cryptoki.dll (for IBM 4758)
- softokn3.dll (for the Mozilla or Netscape crypto module)
- iveacryptoki.dll (for Rainbow CryptoSwift HSM)
- sadaptor.dll (for Eutron CryptoIdentity or Algorithmic Research MiniKey)
- pkcs11.dll (for TeleSec)
- siecap11.dll (for Siemens HiPath SIcurity Card API)
- asepkcs.dll (for Athena Smartcard System ASE Card)
- /opt/SUNWconn/cryptov2/lib/libvpkcs11.so (for SUN Crypto Accelerator 4000, 32-bit libraries)
- /opt/SUNWconn/cryptov2/lib/sparcv9/libvpkcs11.so (for SUN Crypto Accelerator 4000, 64-bit libraries)
- /opt/SUNWconn/crypto/lib/libpkcs11.so (for SUN Crypto Accelerator 1000, 32-bit libraries)
- /opt/SUNWconn/crypto/lib/sparcv9/libpkcs11.so (for SUN Crypto Accelerator 1000, 64-bit libraries)

1.2 Client/Server based HSM

The HSM-backend is mainly written for PKCS#11 enabled HSMs. In case of Client/Server based HSM the built-in engine can be adapted to implement a client to consume a server exposed API for example. The following requirement should be fulfilled:

- Locate keys/certificates on HSM by an identifier, name or label. This identifier will be kept across cryptographic operation.
- Load partial attributes of private key. In case of RSA should be able to get the value of the modulus and the exponent. It is mandatory to populate RSA structure with those parameters for private keys. OpenSSL uses that for consistency check between certificate and its corresponding private key. In case your HSM is not able to provide partial private keys parameters, you should patch OpenSSL to ignore X509_check_private_key function.
- Write an implementation for RSA init / finish methods and rsa_priv_encrypt method.
- Optionally implement SHA diget methods if you want to perform digesting at HSM level. Methods are: digest_init, digest_update, digest_finish, digest_copy and digest_cleanup
- Engine initialization and destruction.
- In case of stateful service, add a session manager to your implementation.

2.0 Install Code-Signing Tool - HSM

B. INSTALLATION

2.1 Dependencies

The CST-HSM backend depends on:

- OpenSSL library
- Libconfig

Make is required for building the software.

Linux

1) Install OpenSSL

```
$ apt-get install openssl libssl-dev
```

libconfig is C/C++ library for processing structured configuration files. It is used by CST to load HSM related configuration.

2) Install libconfig

```
$ apt-get install libconfig-dev
```

Windows

Under Windows all setup can be done in top of MSYS2 and MinGW.

1) Install MinGW toolchain

For 32-bit:

```
$ pacman -S mingw-w64-i686-gcc make
```

For 64-bit:

```
$ pacman -S mingw-w64-x86_64-gcc make
```

2) Install OpenSSL libraries

You can search the repositories by doing:

```
$ pacman -Ss openssl-devel
```

Then install using

```
 $ pacman -Ss openssl-devel
```

3) Install libconfig libraries

Search for libconfig-devel and install it by doing:

```
$  pacman -Ss libconfig
```

For 32-bit:

```
$ pacman -S mingw-w64-i686-libconfig
```

For 64-bit:

```
$ pacman -S mingw-w64-x86_64-libconfig
```

2.2 Get sources

The HSM backend source code for CST can be found in the official CST package or checked-out internally from Bitbucket repository.

2.3 Compile sources

To compile backend and CST from souce:

2.3.1 Compile backend

1) Verify that the compiler is working by doing:

```
 $ gcc -v
```

For Windows open a MSYS2 MinGW 64-bit or a MSYS2 MinGW 32-bit shell.

2) Compile the backend source code using the following command:

From CST package:

```
$ tar xzf cst-release.tgz
$ cd release/code/back_end-hsm/src/
$ make
```

From Bitbucket repository:

```
$ git clone ssh://git@bitbucket.sw.nxp.com/micrse/cst-hsm.git
$ cd cst-hsm/src
$ make
```

If you have a custom OpenSSL version or installation and you want to compile the backend against it you can use OPENSSL_PATH attribute and specifies the path to it.

```
$ make OPENSSL_PATH=/opt/openssl-1.1.0
```

The result is **libbackend.a** static library present in the current working folder.

2.3.1 Compile Code-Signing tool

Taking into consideration the target architecture, copy **libfrontend.a** to current working directory (where libbackend.a resides) and then compile CST using the following command:

```
$ cp ../../../linux64/lib/libfrontend.a .
$ make all
```

The result is **./cst** binary file created in the current working directory .

Optionally, copy or create a symlink to the created cst binary file on linux64/bin folder.

```
$ ln -s ./cst ../../../linux64/bin/cst-hsm
```

3.0 DEPLOY Code-Signing Tool

C. DEPOLY

This section provides instructions and prerequisites to deploy Code-Signing Tool with 2 HSMs, these are SoftHSM2 and Utimaco HSM simulator.

3.1 Install HSM

3.1.1 SoftHSM2

Install SoftHSM2

```
  $ apt-get install softhsm2
```

Use either softhsm2-util to initialize a token. The SO PIN can e.g. be used to re-initialize the token and the user PIN is handed out to the application so it can interact with the token.

```
  $ softhsm2-util --init-token --slot 0 --label "CST-HSM"
```

Type in SO PIN and user PIN.

```
=== SO PIN (4-255 characters) ===
Please enter SO PIN: ******
Please reenter SO PIN: ******
=== User PIN (4-255 characters) ===
Please enter user PIN: ******
Please reenter user PIN: ******
The token has been initialized and is reassigned to slot 66362367
```

Initialized tokens will be reassigned to another slot (based on the token serial number). It is recommended to find and interact with the token by searching for the token label or serial number in the slot list / token info.

3.1.2 Utimaco HSM Simulator

Download and extract SecurityServerEvaluation package.

```
$ cd SecurityServerEvaluation-
V4.20.0.4/Software/Linux/Simulator/sim5_linux/bin
$ ./cs_sim.sh
```

Copy cs_pkcs11_R2.cfg to /etc

```
$ cp /home/nxf45729/SecurityServerEvaluation-V4.20.0.4/Software/Linux/x86-
64/Crypto_APIs/PKCS11_R2/sample/cs_pkcs11_R2.cfg /etc
```

Edit **/etc/cs_pkcs11_R2.cfg** configuration file and

- Uncomment Logpath = /tmp in [Global] section
- Add Device = 3001@127.0.0.1 to [CryptoServer] section
- Save the file

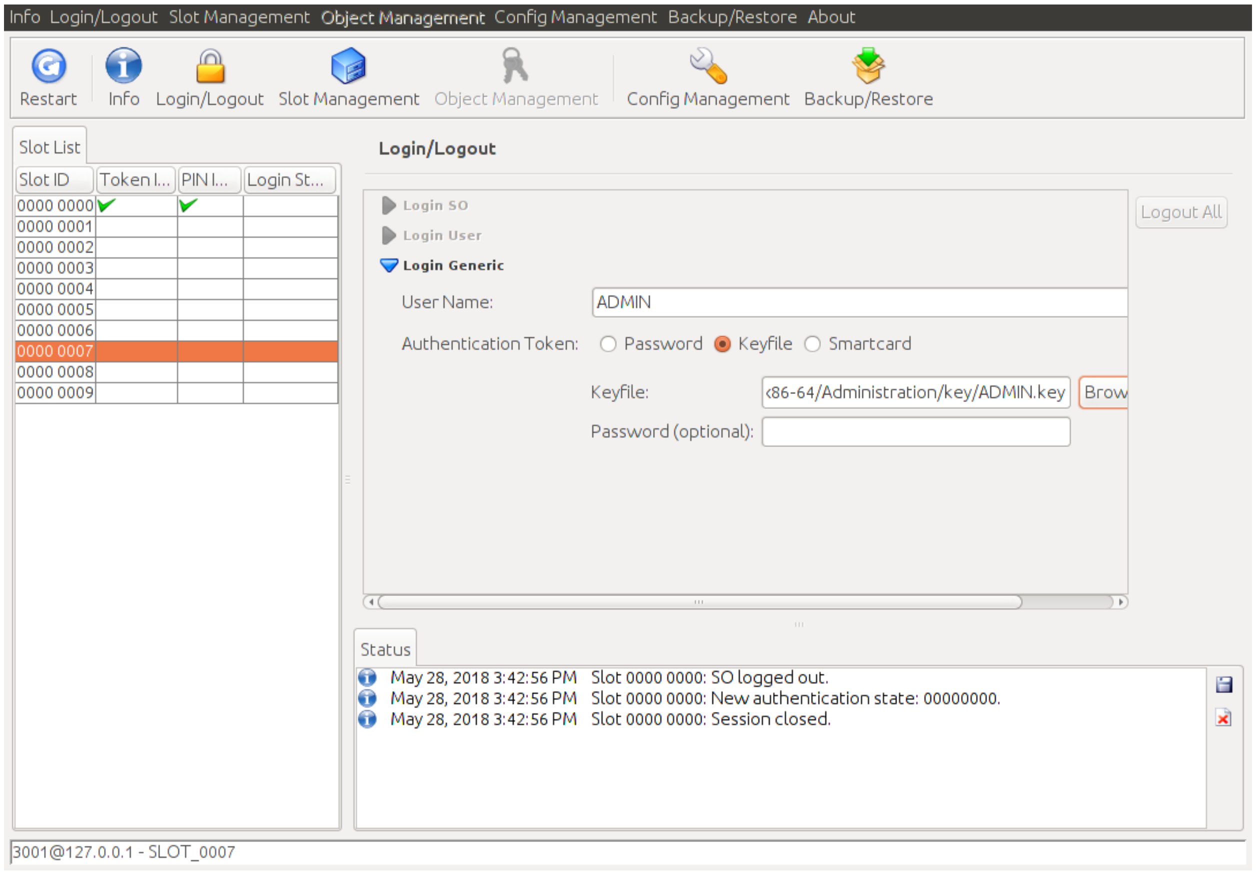
Install java if it is not installed on your system

```
$ apt-get install default-jre
```

Launch p11cat

```
$ java -jar /home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-64/Administration/p11cat.jar
```

From the menu bar click **Login/Logout**
Select a slot from the **Slot List grid**
Select **Login Generic**
Type ***ADMIN*** as UserName
Select Keyfile as Authenticiation Token
Click browse button and fin the **ADMIN.key** file
The keyfile is under SecurityServerEvaluation-V4.20.0.4/Software/Linux/x86-64/Administration/key/ADMIN.key
Click Login



Slot ID	Token I...	PIN I...	Login St...
0000 0000	✓	✓	
0000 0001			
0000 0002			
0000 0003			
0000 0004			
0000 0005			
0000 0006			
0000 0007			
0000 0008			
0000 0009			

User Name: ADMIN

Authentication Token: ○ Password [✓ Keyfile] ○ Smartcard

Keyfile: ‹86-64/Administration/key/ADMIN.key Brow

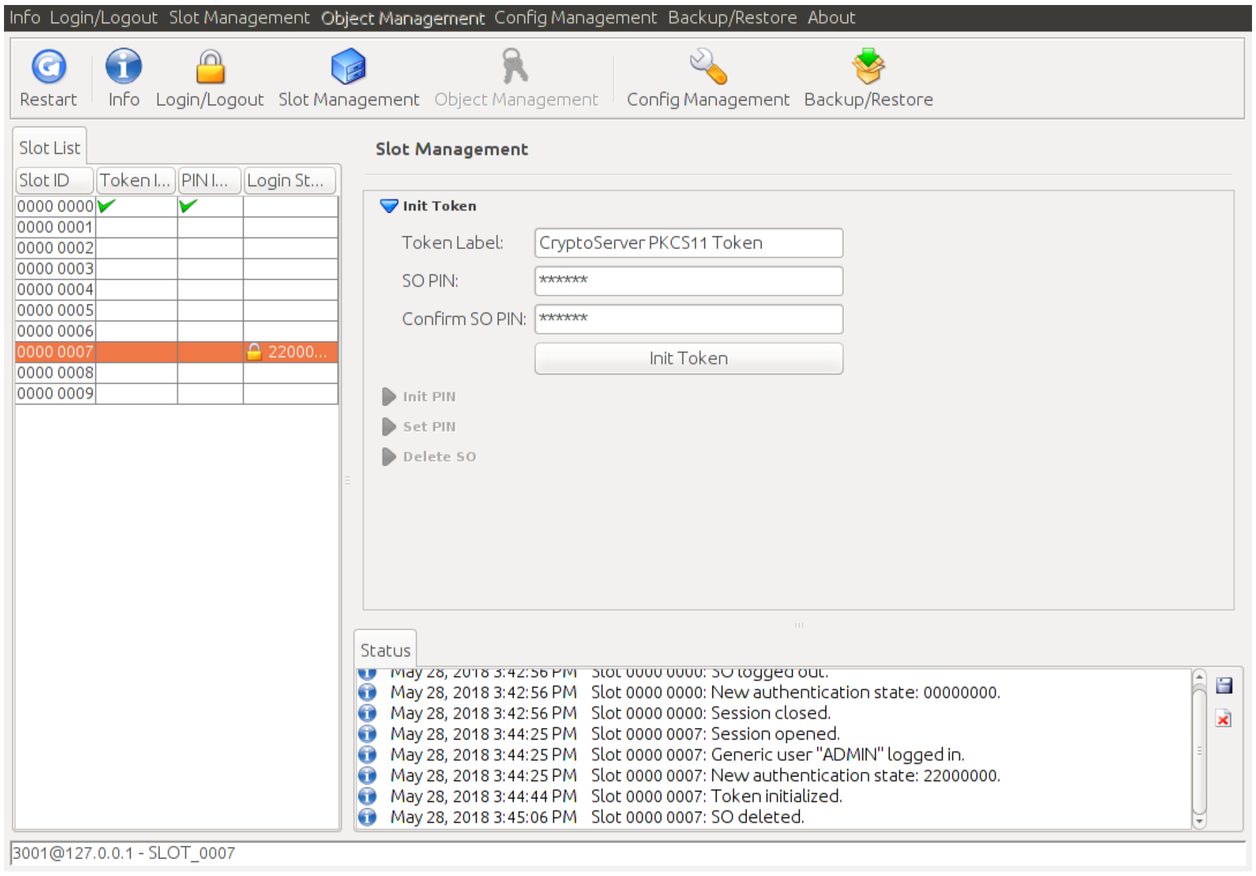
Password (optional):

```
May 28, 2018 3:42:56 PM   Slot 0000 0000: SO logged out.
May 28, 2018 3:42:56 PM   Slot 0000 0000: New authentication state: 00000000.
May 28, 2018 3:42:56 PM   Slot 0000 0000: Session closed.
```

3001@127.0.0.1 - SLOT_0007

From **Slot Management** menu click

In **Init Token** form define the **Token Label** and **SO PIN** then click **Init Token**



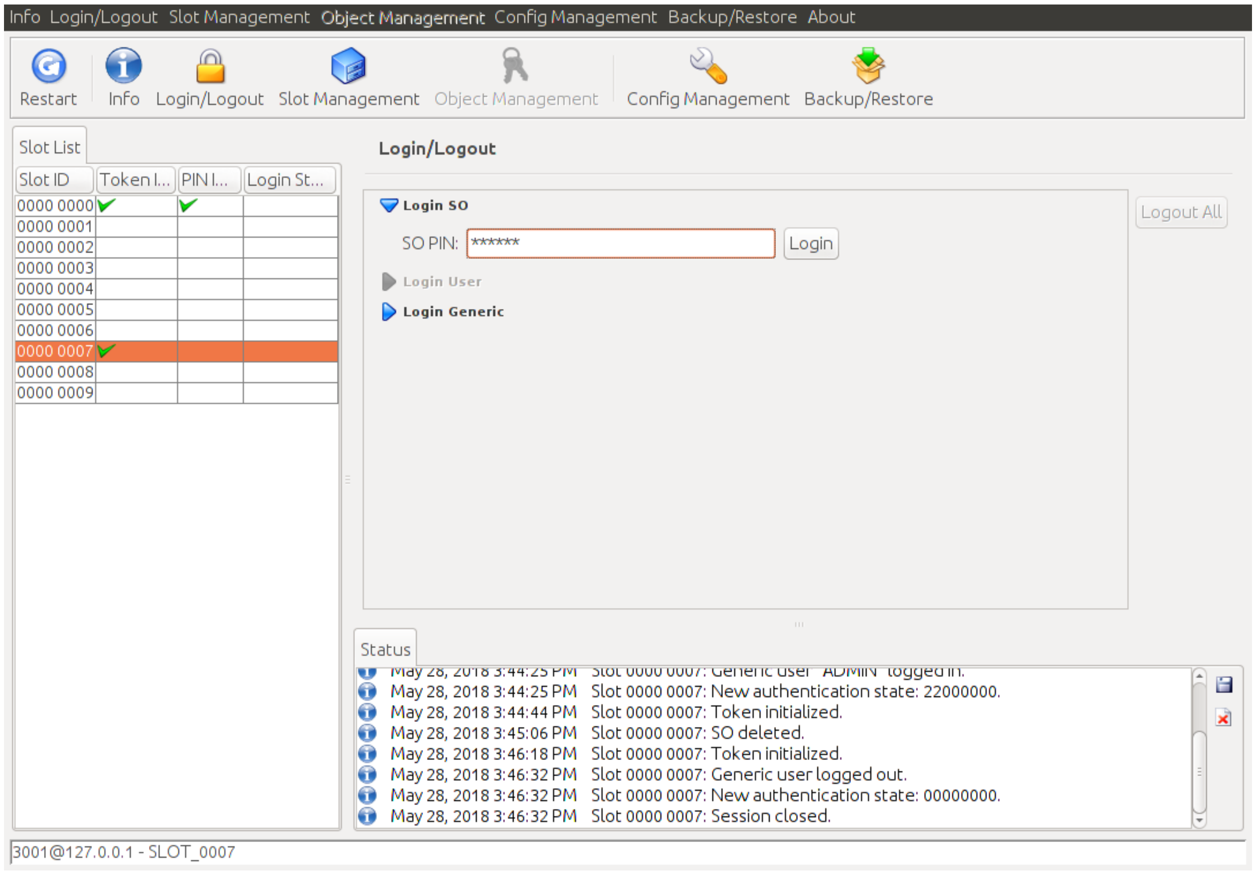
Slot ID	Token I...	PIN I...	Login St...
0000 0000	✓	✓	
0000 0001			
0000 0002			
0000 0003			
0000 0004			
0000 0005			
0000 0006			
0000 0007			22000...
0000 0008			
0000 0009			

Token Label:	CryptoServer PKCS11 Token
SO PIN:	******
Confirm SO PIN:	******
	Init Token

```
May 28, 2018 3:42:56 PM   Slot 0000 0000: SO logged out.
May 28, 2018 3:42:56 PM   Slot 0000 0000: New authentication state: 00000000.
May 28, 2018 3:42:56 PM   Slot 0000 0000: Session closed.
May 28, 2018 3:44:25 PM   Slot 0000 0007: Session opened.
May 28, 2018 3:44:25 PM   Slot 0000 0007: Generic user "ADMIN" logged in.
May 28, 2018 3:44:25 PM   Slot 0000 0007: New authentication state: 22000000.
May 28, 2018 3:44:44 PM   Slot 0000 0007: Token initialized.
May 28, 2018 3:45:06 PM   Slot 0000 0007: SO deleted.
```

3001@127.0.0.1 - SLOT_0007

From **Login/Logout** menu click **Logout All**
From **Login/Logout** menu click **Login SO**

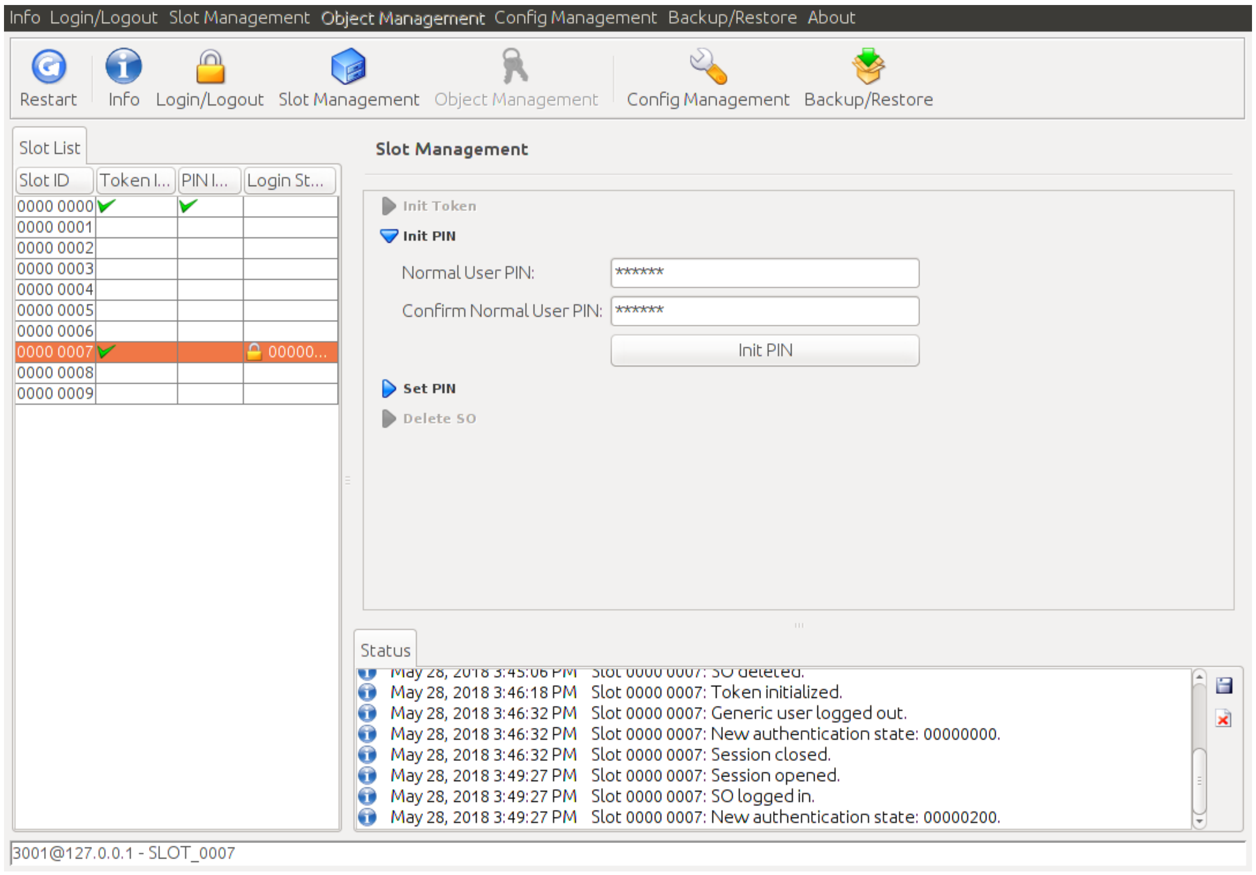


Slot ID	Token I...	PIN I...	Login St...
0000 0000	✓	✓	
0000 0001			
0000 0002			
0000 0003			
0000 0004			
0000 0005			
0000 0006			
0000 0007	✓		
0000 0008			
0000 0009			

```
May 28, 2018 3:44:25 PM   Slot 0000 0007: Generic user "ADMIN" logged in.
May 28, 2018 3:44:25 PM   Slot 0000 0007: New authentication state: 22000000.
May 28, 2018 3:44:44 PM   Slot 0000 0007: Token initialized.
May 28, 2018 3:45:06 PM   Slot 0000 0007: SO deleted.
May 28, 2018 3:46:18 PM   Slot 0000 0007: Token initialized.
May 28, 2018 3:46:32 PM   Slot 0000 0007: Generic user logged out.
May 28, 2018 3:46:32 PM   Slot 0000 0007: New authentication state: 00000000.
May 28, 2018 3:46:32 PM   Slot 0000 0007: Session closed.
```

3001@127.0.0.1 - SLOT_0007

From **Slot Management** menu click **Init PIN**
Set User PIN for example 123456



Slot ID	Token I...	PIN I...	Login St...
0000 0000	✓	✓	
0000 0001			
0000 0002			
0000 0003			
0000 0004			
0000 0005			
0000 0006			
0000 0007	✓		00000...
0000 0008			
0000 0009			

Normal User PIN: ******

Confirm Normal User PIN: ******

Init PIN

```
May 28, 2018 3:45:06 PM   Slot 0000 0007: SO deleted.
May 28, 2018 3:46:18 PM   Slot 0000 0007: Token initialized.
May 28, 2018 3:46:32 PM   Slot 0000 0007: Generic user logged out.
May 28, 2018 3:46:32 PM   Slot 0000 0007: New authentication state: 00000000.
May 28, 2018 3:46:32 PM   Slot 0000 0007: Session closed.
May 28, 2018 3:49:27 PM   Slot 0000 0007: Session opened.
May 28, 2018 3:49:27 PM   Slot 0000 0007: SO logged in.
May 28, 2018 3:49:27 PM   Slot 0000 0007: New authentication state: 00000200.
```

3001@127.0.0.1 - SLOT_0007

3.3 Create certificates & keys

Referring to **Key and Certificate Generation HAB Code Signing Tool User's guide**, generate keys and certificates which will be used to sign boot loader image.
If you already have the cryptographic material on the disk, please go directly to *push cryptographic material to HSM* section

3.3.1 Create serial file

serial - 8-digit OpenSSL uses the contents of this file for the certificate serial numbers.

```
$ cd release/keys
```

```
$ echo "28280581" > serial
```

3.3.2 Create passphrase file

key_pass.txt - Contains your pass phrase that will protect the HAB code signing private keys.

```
$ echo "maro1337" > key_pass.txt
$ echo "maro1337" >> key_pass.txt
```

3.3.3 Generate the PKI tree

HABv4

```
$ ./hab4_pki_tree.sh
```

When prompts please answer according to the following:

```
Do you want to use an existing CA key (y/n)?: n
Do you want to use Elliptic Curve Cryptography (y/n)?: n
Enter key length in bits for PKI tree: 2048
Enter PKI tree duration (years): 10
How many Super Root Keys should be generated? 4
Do you want the SRK certificates to have the CA flag set? (y/n)?: y
```

Private keys will generate on keys directory and corresponding Certificates are placed in the crts directory.

AHAB

```
$ ./ahab_pki_tree.sh
```

When prompts please answer according to the following:

```
Do you want to use an existing CA key (y/n)?: n
Do you want to use Elliptic Curve Cryptography (y/n)?: y
Enter length for elliptic curve to be used for PKI tree:
Possible values p256, p384, p521:  p384
Enter the digest algorithm to use: sha384
Enter PKI tree duration (years): 10
Do you want the SRK certificates to have the CA flag set? (y/n)?: y
```

3.3.4 Generate SRK table

Create fuse table and binary (SRK table) to be flashed.

HABv4

```
$ cd ../crts
```

```
$ ../linux64/bin/srktool -h 4 -t SRK_1_2_3_4_table.bin -e
SRK_1_2_3_4_fuse.bin -d sha256 -c
./SRK1_sha256_2048_65537_v3_ca_crt.pem,./SRK2_sha256_2048_65537_v3_ca_crt.pem
,./SRK3_sha256_2048_65537_v3_ca_crt.pem,./SRK4_sha256_2048_65537_v3_ca_crt.pe
m -f 1
```

SRK_1_2_3_4_table.bin - SRK table contents with HAB data
SRK_1_2_3_4_fuse.bin - contains SHA256 result to be burned to fuse

AHAB

```
$ cd ../crts
```

```
$ ../linux64/bin/srktool -a -s sha384 -t SRK1234table.bin -e SRK1234fuse.bin
-f 1 -c
SRK1_sha384_secp384r1_v3_ca_crt.pem,SRK2_sha384_secp384r1_v3_ca_crt.pem,SRK3_
sha384_secp384r1_v3_ca_crt.pem,SRK4_sha384_secp384r1_v3_ca_crt.pem
```

3.3.4 Command Sequence File (CSF)

CST tool requires CSF script. This file describes certificates, keys and the data ranges used in sign and encryption functions.

HAB

Create **u-boot.csf** file.

```
$ cd ../linux64/bin/
$ nano u-boot.csf
```

```
[Header]
Version = 4.1
Security Configuration = Open
Hash Algorithm = sha256
Engine Configuration = 0
Certificate Format = X509
Signature Format = CMS
Engine = CAAM

[Install SRK]
File = "../../crts/SRK_1_2_3_4_table.bin"
Source index = 0
```

```
[Install CSFK]
File = "../../crts/CSF1_1_sha256_4096_65537_v3_usr_crt.pem"

[Authenticate CSF]

[Install Key]
# Key slot index used to authenticate the key to be installed
Verification index = 0

# Key to install
Target index = 2
File = "../../crts/IMG1_1_sha256_4096_65537_v3_usr_crt.pem"

[Authenticate Data]
Verification index = 2
Blocks = 0x00000000 0 0xd4454 "flash.bin"
```

AHAB

The CSF contains the input files and keys used by CST to create the final image already signed. Please report the offsets noted previously when run mkimage.

```
[Header]
Target = AHAB
Version = 1.0
[Install SRK]
# SRK table generated by srktool
File = "../../crts/SRK1234table.bin"
# Public key certificate in PEM format
Source = "../../crts/SRK1_sha384_secp384r1_v3_ca_crt.pem"
# Index of the public key certificate within the SRK table (0 .. 3)
Source index = 0
# Type of SRK set (NXP or OEM)
Source set = OEM
# bitmask of the revoked SRKs
Revocations = 0x0
[Authenticate Data]
# Binary to be signed generated by mkimage
File = "flash.bin"
# Offsets = Container header  Signature block (printed out by mkimage)
Offsets   = 0x400             0x590
```

3.4 Push cryptographic material to HSM

3.4.1 Install p11tool

To perform interact with HSM you can use p11tool.

```
$ apt-get install gnutls-bin
```

3.4.2 Find Token URL

You need to locate your HSM vendor's PKCS#11 interface implementation in order to use it with p11tool and CST later.

Utimaco pkcs#11 library is *SecurityServerEvaluation-V4.20.0.4/Software/Linux/x86-64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so*

SoftHSM2 pkcs11# library is */usr/lib/softhsm/libsofthsm2.so*

Find the Token URL to interact with the token.

```
$ p11tool --provider <pkcs11_lib_path> --list-tokens
```

Example using SoftHSM2

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so --list-tokens
```

```
Token 0:
        URL:
pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;serial=9c1aeb00e05a3
48b;token=CST-HSM
        Label: CST-HSM
        Type: Generic token
        Manufacturer: SoftHSM project
        Model: SoftHSM v2
        Serial: 9c1aeb00e05a348b
        Module: (null)
```

In this case Token URL is *pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;serial=9c1aeb00e05a348b;token=CST-HSM*

3.4.3 Push certificate & key

To match a certificate to a private key, both should have the same ID, for some other HSM implementations they should have the same label.

HAB

1) Push CSF certificate and private key

Utimaco HSM Simulator

Private Key

```
$ p11tool --provider /home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so
"pkcs11:model=CryptoServer;manufacturer=Utimaco%20IS%20GmbH;serial=UTIMACO%20
CS000000;token=CryptoServer%20PKCS11%20Token" so  --login --write --load-
privkey ../../keys/CSF1_1_sha256_2048_65537_v3_usr_key.pem --label
CSF1_1_sha256_2048_65537_v3_usr --id ec705018e9bf8ad60096e13cb2f0fbad
```

Certificate

```
$ p11tool --provider /home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so
"pkcs11:model=CryptoServer;manufacturer=Utimaco%20IS%20GmbH;serial=UTIMACO%20
CS000000;token=CryptoServer%20PKCS11%20Token" so  --login --write --load-
certificate ../../crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem --label
CSF1_1_sha256_2048_65537_v3_usr --id ec705018e9bf8ad60096e13cb2f0fbad
```

SoftHSM2

Private Key

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so  --login --write
"<token-url>" --load-privkey
../../keys/CSF1_1_sha256_2048_65537_v3_usr_key.pem --label
CSF1_1_sha256_2048_65537_v3_usr --id ec705018e9bf8ad60096e13cb2f0fbad
```

Certificate

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so  --login --write
"<token-url>" --load-certificate
../../crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem --label
CSF1_1_sha256_2048_65537_v3_usr --id ec705018e9bf8ad60096e13cb2f0fbad
```

2) Push IMG certificate & key

Utimaco HSM Simulator

Private key

```
$ p11tool --provider /home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so
"pkcs11:model=CryptoServer;manufacturer=Utimaco%20IS%20GmbH;serial=UTIMACO%20
CS000000;token=CryptoServer%20PKCS11%20Token" so  --login --write --load-
privkey ../../keys/IMG1_1_sha256_2048_65537_v3_usr_key.pem --label
IMG1_1_sha256_2048_65537_v3_usr --id a0c8cac03985fb6dced29c97dc83aef7
```

Certificate

```
$ p11tool --provider /home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so
"pkcs11:model=CryptoServer;manufacturer=Utimaco%20IS%20GmbH;serial=UTIMACO%20
CS000000;token=CryptoServer%20PKCS11%20Token" so  --login --write --load-
certificate ../../crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem --label
IMG1_1_sha256_2048_65537_v3_usr --id a0c8cac03985fb6dced29c97dc83aef7
```

SoftHSM2

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so  --login --write
"<token-url>"  --load-privkey
../../keys/IMG1_1_sha256_2048_65537_v3_usr_key.pem --label
IMG1_1_sha256_2048_65537_v3_usr --id a0c8cac03985fb6dced29c97dc83aef7
```

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so  --login --write
"<token-url>"  --load-certificate
../../crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem --label
IMG1_1_sha256_2048_65537_v3_usr --id a0c8cac03985fb6dced29c97dc83aef7
```

AHAB

For AHAB you should extract the EC public key from the private key and push it to HSM with same ID/Label as private key.

1) Push Private key

```
$ p11tool --provider /home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so
--login --write "<token-url>"  --load-privkey
../keys/SRK1_sha384_secp384r1_v3_ca_key.pem --label
SRK1_sha384_secp384r1_v3_ca_key --id b0c8cac03985fb6dced29c97dc83aef7
```

2) Push certificate

```
$ p11tool --provider /home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so
--login --write "<token-url>"   --load-certificate
./SRK1_sha384_secp384r1_v3_ca_crt.pem --label SRK1_sha384_secp384r1_v3_ca_crt
--id b0c8cac03985fb6dced29c97dc83aef7
```

3) Push Public key

```
$ openssl ec -in relase/keys/SRK1_sha384_secp384r1_v3_ca_key.pem -pubout -out
SRK1_sha384_secp384r1_v3_ca_pubkey.pem
```

```
$ p11tool --provider /home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so
--login --write "<token-url>"  --load-pubkey
```

```
SRK1_sha384_secp384r1_v3_ca_pubkey.pem --label
SRK1_sha384_secp384r1_v3_ca_pubkey --id b0c8cac03985fb6dced29c97dc83aef7
```

3.4.5 Verification

Verify that all cryptographic materials were written correctly to token by running.

```
$ p11tool --provider “<pkcs11-module>” "<token-url>"  --list-all --login
```

Enter PIN when prompts

3.5 Pull certificates from HSM

3.5.1 Install p11tool

To perform interact with HSM you can use p11tool.

```
$ apt-get install gnutls-bin
```

3.5.1 Find Token URL

Find the Token URL to interact with the token.

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so --list-tokens
```

```
Token 0:
        URL: 
pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;serial=9c1aeb00e05a3
48b;token=CST-HSM
        Label: CST-HSM
        Type: Generic token
        Manufacturer: SoftHSM project
        Model: SoftHSM v2
        Serial: 9c1aeb00e05a348b
        Module: (null)
```

In this case Token URL is
pkcs11:model=SoftHSM%20v2;manufacturer=SoftHSM%20project;serial=9c1aeb00e05a348b;token=CST-HSM

3.4.3 Pull Certificates

HABv4

Pull CSF & IMG certificate

List all certificates on HSM

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so "<token-url>"  --list-
all-certs
```

Pull CSF certificate from HSM

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so --export "<cert-url>" --
outfile CSF1_1_sha256_2048_65537_v3_ca_crt.pem
```

Pull IMG certificate from HSM

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so --export "<cert-url>" --
outfile IMG1_1_sha256_2048_65537_v3_ca_crt.pem
```

Pull SRK certificate

```
$ p11tool --provider /usr/lib/softhsm/libsofthsm2.so --export "<cert-url>" --
outfile SRK1_sha384_secp384r1_v3_ca_crt.pem
```

3.6 Create HSM configuration file

Create a **hsm.cfg** file in the same folder as CST. The configuration file specifies the following parameters.

- **hsm.module** contains the path to vendor's PKCS#11 implementation library
- **hsm.pin** specifies the User PIN
- **hsm.slot** specifies the SLOT ID
- **hsm.objects** is a lookup table which bind a cryptographic material on the file system to its Object ID on the HSM. This is needed since the current frontend implementation loads and checks certificates from the file system and then provide as argument the certificate path to the backend. The backend can see a path only. To be load the required certificate and its private key from the HSM, the backend uses this lookup.

```
$ nano hsm.cfg
```

HABv4

```
# hsm stuff
hsm:
{
  module = "/home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-
64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so";
  pin = "123456";
  slot = 0;
  objects = (
          { file  = "./CSF1_1_sha256_2048_65537_v3_usr_crt.pem";
            id = "ec705018e9bf8ad60096e13cb2f0fbad";
          },
          { file  = "./IMG1_1_sha256_2048_65537_v3_usr_crt.pem";
            id = "a0c8cac03985fb6dced29c97dc83aef7";
          });
};
```

For SRK tool you may need to pull all SRK certificates.

AHAB

```
# hsm stuff
hsm:
{
module = "/home/nxf45729/SecurityServerEvaluation-
V4.20.0.4/Software/Linux/x86-
64/Crypto_APIs/PKCS11_R2/lib/libcs_pkcs11_R2.so";
pin = "12345678";
slot = 0;
objects = (
{ file  = "./SRK1_sha384_secp384r1_v3_ca_crt.pem";
id = "b0c8cac03985fb6dced29c97dc83aef7";
});
};
```

D. RUN

Generate the CSF binary signature

```
$ ./cst -i u-boot.csf -o u-boot.csf.bin
CSF Processed successfully and signed data available in u-boot.csf.bin
```