CODE-SIGNING TOOL – HSM BACKEND
Architecture Overview

According 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 and how public certificates and private keys are loaded from the HSM to the appropriate data structure X509, RSA and EVP_PKEY. Optionally SHA digest functions can be re-written also to be performed at the HSM level.

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.

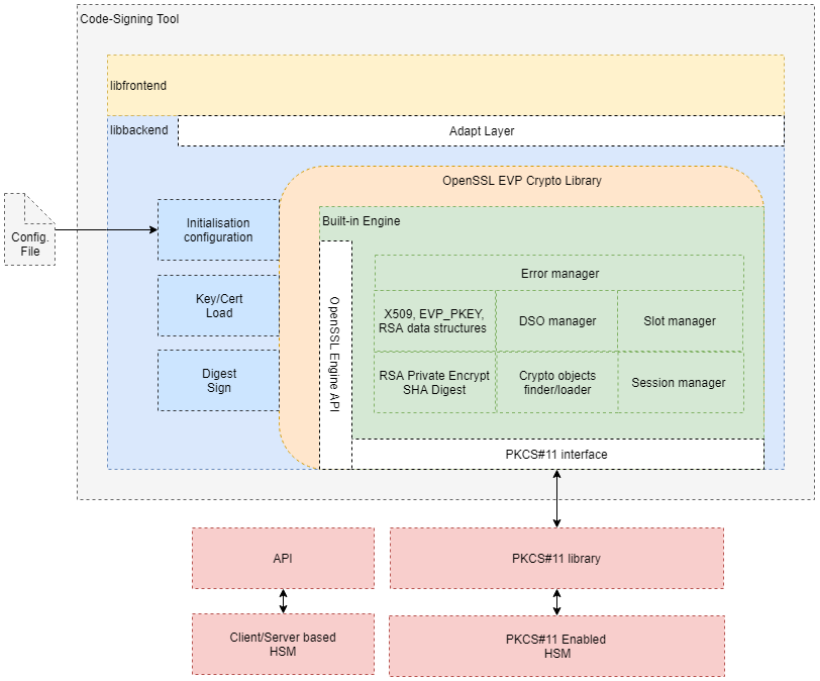

Figure 1 CST-HSM Architecture

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)

Client/Server based HSM

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/certifcates on HSM by an identifier, name or label. This identifier will be kept across cryptographic operation.
- ✓ Load parial 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 parametres for priuvate 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 paramateres, you should patch OpenSSL to ignore ***X509_check_private_key*** function.
- ✓ Write an implemnetation 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 initialisation and destruction.
- ✓ In case of statefull service, add a session manager to your implemnetation.

Based on the cst-hsm source code the following functions should be implemented and adapted to your non-standard HSM. In **e_hsm_adapt.c** we removed all functions related to a standard HSM. You should implement the remaining functions.

pre_init_hsm: Perform any pre-initialization actions, test connectivity to HSM etc …
return 1 in success 0 if error

hsm_init: initialize the engine, in case of statefull service you can perform login to your HSM and create a session wrapper to be used to perform any further operation later.
In addition, you should register new RSA internal data:
hsmSession: Index to store session identifier
rsaPrivKey: Index to store private key identifier
If everything is fine, set HSM_Initialized flag to 1. You can check later this flag for consistency.
return 1 in success of if error

hsm_finish: finalize the engine, you can perform logout here and free any allocated resources.

hsm_ctrl: control handlers, you can define any custom command and add the corresponding handler here. you should keep the command to load the certificate from HSM.

hsm_find_certificate: loads a certificate from the HSM by Id
return X509 object if success otherwise NULL

hsm_load_privkey: loads a private key from the HSM by id
return EVP_PKEY if success otherwise NULL

hsm_RSA_init: Here you can get a session handler and store it as internal data in RSA object

hsm_RSA_private_encrypt: perform RSA encryption (signature) and returns the signature length.
You can retrieve the identifier or the handler of the private key from the RSA object in memory using

RSA_get_ex_data(rsa, rsaPrivKey);

If you want to perform digesting at your HSM, you should implement the following functions:
get_hsm_digests: Get list of Digest algorithms supported by the HSM
hsm_digest_init
hsm_digest_update
hsm_digest_finish
hsm_digest_copy: You should check the status of the digesting operation here before and after copying.
hsm_digest_cleanup

hsm_engine_destructor: Free all allocated memory