SKS (Secure Key Store)
API and Architecture

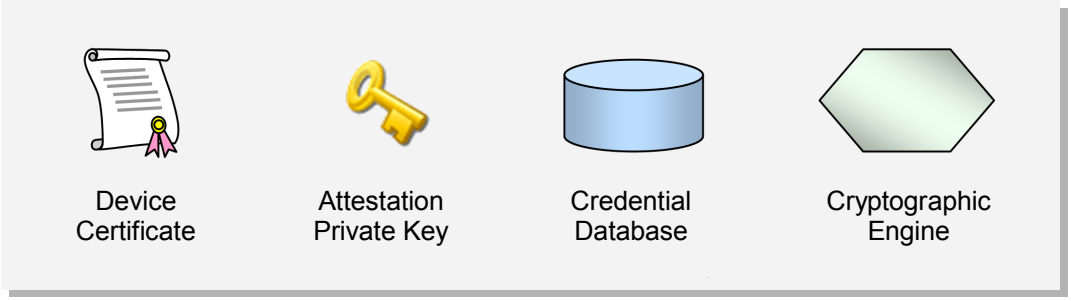
Table of Contents

Introduction

This document describes the API (Application Programming Interface) and architecture of an electronic component called SKS (Secure Key Store). SKS is essentially an enhanced smart card that is optimized for on-line provisioning of cryptographic keys and associated attributes.

Architecture

Below is a picture showing the components in the SKS architecture:



All operations inside of a SKS is supposed to be protected from tampering by malicious external entities but the degree of internal protection may vary depending on the environment that the SKS is running in. That is, an SKS housed in a smart card which may be inserted in an arbitrary computer must keep all data within its protected memory, while an SKS that is an integral part of a mobile phone processor may store credential data in the same external Flash where programs are stored, but sealed by an SKS-resident “master key”.

The *Device Certificate* and the associated *Attestation Private Key* form the foundation for the mechanism that secure provisioning of keys, which remains secure even if the surrounding middleware (for self-contained SKSes NB) and network are unsecured.

The *Cryptographic Engine* performs in addition to standard cryptographic operations on private and secret keys, the core of the provisioning operations which from an API point-of-view are considerably more complex than the former.

Provisioning API

The core of SKS is the cooperation between three associated systems: The KeyGen2 protocol, the SKS architecture, and the provisioning API described in this document. i.e. *these items must be matched* in order to create a secure and interoperable system. A question that arises is of course how compatible this scheme is with respect to current protocols, APIs, and smart cards. The simple answer is: NOT. The reason why SKS still makes sense is that none of the common protocols, APIs and smart cards actually support secure on-line provisioning to end-users because *the current generation of smart cards are almost exclusively personalized by fairly proprietary software used by specific card administrators or by automated production facilities*. It is clear that mobile phones need a scheme that is more consistent with the on-line paradigm since SIM-cards due to operator-bindings do not scale particularly well. *“On the internet anybody can be an operator of something”*.

“Key User” API

In this document Key User API refers to operations that are used by security applications like SSL-client-certificate authentication, S/MIME, and Kerberos (PKINIT). The Key User API is not a core SKS facility but its implementation is anyway RECOMMENDED, particularly for SKSes that are featured in connectable containers such as smart cards since card middleware have proved to be a major stumbling block for wide-spread adoption of PKI cards for consumers. The described Key User API is fully mappable to the subset of CryptoAPI, PKCS#11, and JCE that most existing PKI-using applications rely on.

The Key User API does not use authenticated sessions like featured in TPM 1.2 because this is a *local security option*, while the provisioning API has its own self-contained (mandatory) authentication scheme.

If another Key User API is used the only requirement is that the things that are provisioned by the Provisioning API, are compatible.

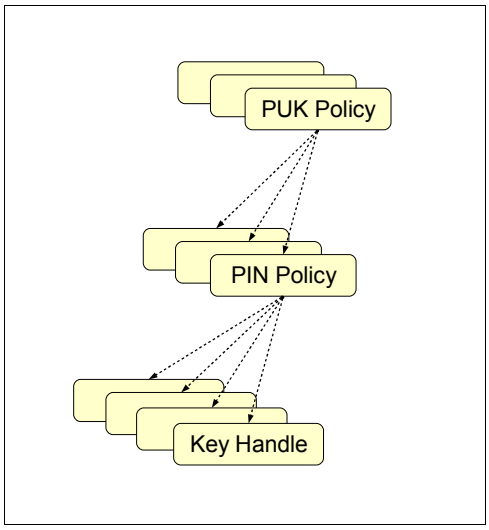
Objects

The SKS API (as well as its companion protocol KeyGeg2), assumes that objects are arranged in a specific fashion in order to work. At the core of the system are the cryptographic keys which unlike most other cryptographic APIs are not only used for authentication, signing etc. but also for key life-cycle management and management of key attributes. All provisioned keys, included symmetric dittos, are identified and managed through an X.509 certificate. The reason for this arrangement is that this facilitates *universal object management* as well as supporting the POI (Proof Of Issuance) concept for enabling *secure remote object management by independent issuers*.

Note: unlike smart cards, SKS has no visible file system, only objects.

Key Protection Objects

Keys may optionally be protected by PIN-codes. Each PIN-protected key must have a separate error-counter, but a single PIN policy object may govern multiple keys. A PIN-policy and its associated keys are in turn governed by a PUK (Personal Unlock Key) policy object that can be used to reset error-counters that have passed the limit as defined by the PIN policy.



PIN and PUK policy objects are not directly addressable after provisioning; in order to read PIN and/or PUK policy data, you need to use an associated key handle as input.

The following extract shows a matching key generation (provisioning) request in KeyGen2:

```
<CreateObject>
  <PUKPolicy Format="numeric" RetryLimit="3" EncryptedValue="mjRKrcuO ... 1O/e9mgMf3qw">
    <PINPolicy Format="numeric" Grouping="shared" MaxLength="8" MinLength="4"
               PatternRestrictions="three-in-a-row sequence" RetryLimit="3">
      <KeyPair ID="Key.1" KeyUsage="encryption">
        <RSA KeySize="1024"/>
      </KeyPair>
      <KeyPair ID="Key.2" KeyUsage="authentication">
        <RSA KeySize="2048"/>
      </KeyPair>
    </PINPolicy>
  </PUKPolicy>
</CreateObject>
```

This sequence should be interpreted as a request for two RSA keys to be generated, protected by user-defined (within specified policy limits) PINs (the same for both keys), where the PINs are governed by an issuer-defined, and (protocol-wise) secret PUK.

Data Types

The table below shows the data types used by the SKS API. Note that all values are stored in big-endian fashion.

Type	Length	Comment
byte	1	Unsigned byte (0 - 0xFF)
short	2	Unsigned two-byte integer (0 - 0xFFFF)
int	4	Unsigned four-byte integer (0 - 0xFFFFFFFF)
array	2 + length	Array of bytes with a leading “short” holding the length of the array
long_array	4 + length	Array of bytes with a leading “int” holding the length of the array

Return Values

All methods return a single-byte status code. In case the status is <> 0 there is an error and any expected succeeding values MUST NOT be read as they are not supposed to be available. Instead there is a second return value containing an UTF-8 encoded description in English to be used for logging and debugging purposes as show below:

Name	Type	Comment
status	byte	Non-zero (error) value
error_string	array	A human-readable error description

Error Codes

The following table shows the standard SKS error-codes:

Name	Value	Comment
ERROR_AUTHENTICATION	1	This error is returned when there is something wrong with a supplied PIN-code. For more detailed information, see HHH and FFF
ERROR_STORAGE	2	There is no persistent storage available for the operation
ERROR_MAC	3	MAC does not match supplied data
ERROR_CRYPTO	4	Various cryptographic errors
ERROR_NO_SESSION	5	Session not found
ERROR_SESSION_VERIFY	6	The final step in the provisioning session failed to verify
ERROR_NO_KEY	7	Key not found
ERROR_ALGORITHM	8	Unknown or not fitting algorithm

Method List

This section provides a list of all the SKS methods. In the number in parenthesis is the decimal value used to identify the method in a call. Method calls are formatted as byte-arrays where the first byte contains the method ID and the succeeding part contains applicable argument data.

createProvisioningSession [1]

Input

Name	Type	Comment
server_session_id	byte[32]	Server nonce value
client_session_id	byte[32]	Client nonce value
provisioning_uri	string	The issuer's way to identify itself. In KeyGen2 this is the URL to which the result of this method is POSTed
server_public_key	byte[]	The RSA server key used to encrypt the session key (SK)
life_time	int	How long the provisioning session is valid (in seconds)

Output

Name	Type	Comment
status	byte	See Return Values
encrypted_session_key	byte[]	The encrypted SK
attested_session_key	byte[]	The SK attestation signature
provisioning_id	int	A handle to the created provisioning session

This is a sample API method m

SKS_Deactivate [25]

Input

Name	Type	Comment
status	byte	See Return Values

Output

Name	Type	Comment
	byte	
aa		

This is a sample API method.

You may call it from Error: Reference source not found hhh