SKS (Secure Key Store)
API and Architecture

Note: This is an early version of a system in development. That is, the specification is incomplete and may also change considerably before finalization. However, it might give you a fairly good idea about the “air-tight” provisioning concept.

Feedback is encouraged!

Table of Contents

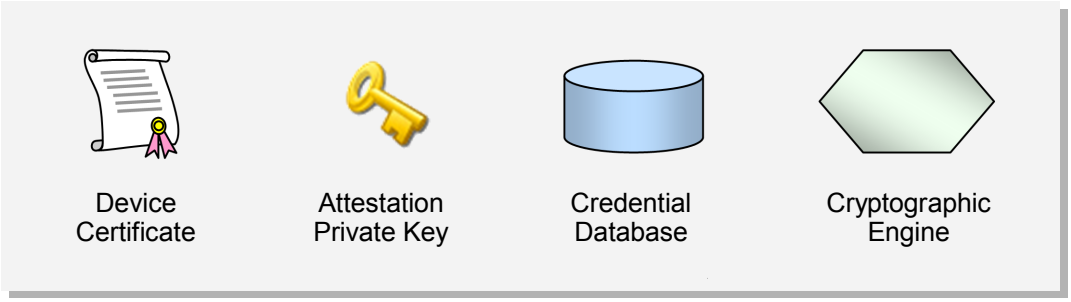
Introduction....3
Architecture....3
Provisioning API....3
User API....3
Objects....4
  Key Protection Objects....5
  Key Data Objects....5
Data Types....6
Return Values....6
Error Codes....6
Encrypted Data....7
MAC Operations....7
SKS Attestations....7
PIN and PUK Formats....7
PIN Grouping Control....8
PIN Input Methods....8
PIN Pattern Restrictions....8
Algorithm Support....9
Key Usage....10
Method List....10
  createProvisioningSession (1)....11
  closeProvisioningSession (2)....13
  abortProvisioningSession (3)....14
  createPUKPolicy (5)....15
  createPINPolicy (6)....16
  createKeyPair (7)....17
  setCertificatePath (8)....20
  setPiggybackedSymmetricKey (19)....22
  restorePrivateKey (20)....23
Sample Session....24
Security Considerations....24
Intellectual Property Rights....24
References....25
Acknowledgments....26
Author....26

Introduction

This document describes the API (Application Programming Interface) and architecture of a system 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 an SKS are 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 its associated *Attestation Private Key* form the foundation for the mechanism that facilitates secure provisioning of keys, also when 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.

A vital part of the *Cryptographic Engine* is a high quality random number generator since the integrity of the entire provisioning scheme is relying on this.

The *Credential Database* holds keys and other data that is related to keys such as protection and extension objects.

Provisioning API

Although SKS may be regarded as a “component”, it actually comprises of three associated systems: The KeyGen2 protocol, the SKS architecture, and the provisioning API described in this document. These items are *tightly 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 existing protocols, APIs, and smart cards. The answer is simply: NOT AT ALL.

A reason why SKS still may serve a purpose is that few of the current protocols, APIs and smart cards support secure on-line provisioning to end-users. in fact, *smart cards are almost exclusively personalized by more or less proprietary software used by specific card administrators or by automated production facilities*. It is evident that (at least) 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”

User API

In this document User API refers to operations that are required by security applications like TLS client-certificate authentication, S/MIME, and Kerberos (PKINIT). The 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 User API is fully mappable to the subset of CryptoAPI, PKCS #11, and JCE that most PKI-using applications rely on.

The standard User API does not rely on 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 User API is used the only requirement is that the key objects created by the provisioning API, are compatible with the former.

Objects

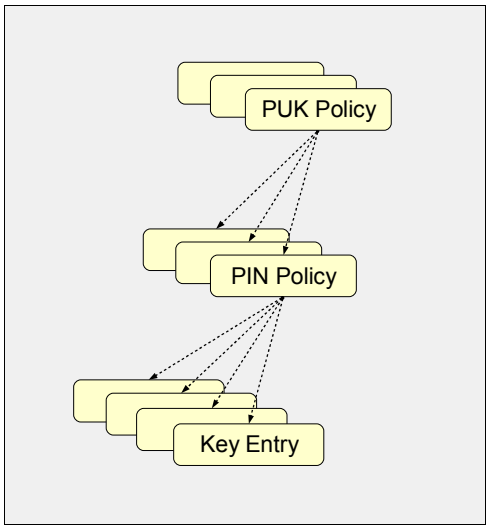
The SKS API (as well as its companion protocol KeyGen2), assumes that objects are arranged in a specific fashion in order to work. At the core of the system are the cryptographic keys which are not only used for authentication, signing etc. but also for key life-cycle management and management of key attributes.

All provisioned keys, included symmetric dittos (see setPiggybackedSymmetricKey), are identified and managed through an X.509 certificate. The reason for this somewhat unusual arrangement is that this enables *universal object management* as well as supporting the PoI (Proof of Issuance) concept for enabling *secure remote object management by independent issuers*.

Note: unlike 7816-compatible smart cards, an SKS has no visible file system, only objects.

Key Protection Objects

Keys may optionally be protected by PIN-codes. Each PIN-protected key maintains a separate PIN error-counter, but a single PIN policy object may govern multiple keys. A PIN-policy and its associated keys may in turn be 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 XML extract shows a matching key generation (provisioning) request in KeyGen2:

```
<CreateObject>
  <PUKPolicy ID="PUK.1" Format="numeric" RetryLimit="3" EncryptedValue="mjRKrcuO ... 1O/e9mgMf3qw">
    <PINPolicy ID="PIN.1" 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.

See PIN and PUK Formats, createPUKPolicy, createPINPolicy and createKeyPair.

Key Data Objects

Provisioned keys always have an associated X.509 certificate, while other objects are optional.

Lots of other stuff: TBD

Data Types

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

Type	Length	Comment
byte	1	Unsigned byte (0 - 0xFF)
bool	1	Byte containing 0x01 (true) or 0x00 (false)
short	2	Unsigned two-byte integer (0 - 0xFFFF)
int	4	Unsigned four-byte integer (0 - 0xFFFFFFFF)
byte[]	2 + length	Array of bytes with a leading “short” holding the length of the data
blob	4 + length	Long array of bytes with a leading “int” holding the length of the data

If an array is followed by a number in brackets (“byte[32]”) it means that the array MUST be exactly of that length.

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 shown below:

Name	Type	Comment
Status	byte	Non-zero (error) value
ErrorString	byte[]	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 TBD
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

Encrypted Data

During provisioning encrypted data is occasionally exchanged between the issuer and the SKS using a key based on the session variables established during the createProvisioningSession call. The encryption key is created by the following key derivation scheme:

EncryptionKey = HMAC-SHA256 (**SK**, **ClientSessionID** ||
ServerSessionID ||
IssuerURI ||
"Encryption Key")

The **EncryptionKey** is used with the AES256-CBC algorithm.

MAC Operations

In order to verify the integrity of provisioned data, most of the provisioning methods requires that the data-carrying arguments are included in a MAC (Message Authentication Code) operation as well. Unless stated otherwise, MAC operations are based on the session variables established during the createProvisioningSession call and use the following scheme:

MAC = HMAC-SHA256 (*MethodName* || **SK** || **ClientSessionID** || **ServerSessionID** || **IssuerURI**, *Data*...)

The *MethodName* is simply the string literal of the target method like **"closeProvisioningSession"**, while *Data* represent the arguments in declaration order unless otherwise noted.

Argument data that is to be included in MAC operations MUST only include the content data, not length etc. See Data Types.

SKS Attestations

Except for the createProvisioningSession call, SKS attestations during provisioning sessions are using symmetric keys derived as for MAC Operations where *MethodName* is **"SKS Attestation"**.

PIN and PUK Formats

PIN and PUK codes MUST adhere to one of formats described in the following table:

Name	Value	Comment
Numeric	0x00	0 - 9
Alphanumeric	0x01	0 - 9, A - Z
UTF-8	0x02	Any valid UTF-8 string
Binary	0x03	Binary value, typically issued as hexadecimal data

Note that format specifiers only deal with how PINs and PUKs are treated in GUIs; internally key protection data is always stored as strings of bytes.

Length of the clear-text binary value MUST NOT exceed 100 bytes.

See the **Format** attribute in createPINPolicy and createPUKPolicy.

PIN Grouping Control

A PIN policy object may govern multiple keys. The **`Grouping`** policy attribute (see createPINPolicy) controls how PIN codes to the different keys may relate to each other according to the following table:

Name	Value	Comment
None	0x00	No restrictions
Shared	0x01	All keys share the *same* PIN (synchronized)
Signature+Standard	0x02	Keys with Key Usage = Signature share one PIN while all other keys share *another* PIN
Unique	0x03	All keys must have *different* PIN codes

During provisioning the middleware MUST maintain the PIN policy and optionally ask the user to create another PIN if there is a policy mismatch because createKeyPair will return an error if it fed with inappropriate arguments.

PIN Input Methods

The **`InputMethod`** policy attribute (see createPINPolicy) tells how PIN codes SHOULD be inputted to the SKS according to the following table:

Name	Value	Comment
Any	0x00	No restrictions
Programmatic	0x01	PINs SHOULD only be issued through the SKS User API
Trusted GUI	0x02	Keys SHOULD only be used through a trusted GUI that does the actual PIN request and API invocation

Note that this policy attribute requires that the middleware is “cooperative” to be enforced.

PIN Pattern Restrictions

The **`PatternRestrictions`** policy attribute (see createPINPolicy) specifies how PIN codes MUST NOT be designed according to the following table:

Name	Mask	Comment
Two-in-a-row	0x01	Flags 1124
Three-in-a-row	0x02	Flags 1114
Sequence	0x04	Flags 1234, 9876, etc
Repeated	0x08	All PIN bytes MUST be *unique*
Missing-group	0x10	Flags 135674 for an alphanumeric PIN. See PIN and PUK Formats

Note that this policy attribute contains a byte holding a *set of bits*. That is, 0x00 means that there are no pattern restrictions, while 0x06 imposes two constraints. Also note that pattern policy checking is supposed to be applied at the *binary* level which has implications for the binary PIN format (see PIN and PUK Formats).

For organizations having very strict or unusual requirements on PIN patterns, it is RECOMMENDED letting the user define PINs during enrollment in a web application and then deploy issuer-set PIN codes during provisioning.

Algorithm Support

Algorithm support in SKS MUST as a *minimum* include the following items:

URI	Comment
Symmetric Key Encryption	
http://www.w3.org/2001/04/xmlenc#aes128-cbc	See ?
http://www.w3.org/2001/04/xmlenc#aes256-cbc	See ?
internal:AES/ECB/NoPadding	See ? Works with 128, 192, and 256 bit keys
internal:AES/ECB/PKCS5Padding	See ? Works with 128, 192, and 256 bit keys
HMAC Operations	
http://www.w3.org/2000/09/xmldsig#hmac-sha1	See ?
http://www.w3.org/2001/04/xmldsig-more#hmac-sha256	See
Asymmetric Key Encryption	
http://www.w3.org/2001/04/xmlenc#rsa-1_5	See ?
Asymmetric Key Signatures	
http://www.w3.org/2000/09/xmldsig#rsa-sha1	See ?
http://www.w3.org/2001/04/xmldsig-more#rsa-sha256	See ?
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256	See ?
HMAC Operations	
http://www.w3.org/2000/09/xmldsig#hmac-sha1	See ?
http://www.w3.org/2001/04/xmldsig-more#hmac-sha256	See ?
TBD	
	See ?

TBD.

Note that algorithms in SKI methods are always specified in local representation. See TBD

Key Usage

The **`KeyUsage`** policy attribute (see createKeyPair) specifies how keys are supposed to be used both during provisioning and during actual usage according to the following table:

Name	Value	Comment
Signature	0	The key MUST only be used in signature applications like S/MIME
Authentication	1	The key MUST only be used in authentication applications
Encryption	2	The key MUST only be used for PKCS #1 or Diffie-Hellman encryption operations
Universal	3	There are no restrictions on private key usage
Transport	4	The private key is *disabled*
Piggybacked-symmetric-key	5	The key MUST include a “piggybacked” symmetric key during provisioning. The private key is *disabled.* See setPiggybackedSymmetricKey

Note that the purpose of the Signature and Authentication attributes is aiding the GUI middleware to request the proper PIN for the user. In most real-world deployments they will coincide with the X.509 **`nonRepudiation`** and **`digitalSignature`** bits respectively. Also see PIN Grouping Control.

Method List

This section provides a (*not very complete...*) list of the SKS methods. The number in parenthesis holds the decimal value used to identify the method in a call. Method calls are formatted as strings of bytes where the first byte is the method ID and the succeeding bytes the applicable argument data.

createProvisioningSession (1)

Input

Name	Type	Comment
ServerSessionID	byte[32]	Server nonce value
ClientSessionID	byte[32]	Client nonce value
IssuerURI	byte[]	UTF-8 encoded URI identifying the issuer. In KeyGen2 this is the URL to which the result of this method is POSTed. The string MUST NOT exceed 1024 bytes
IssuerPublicKey	byte[]	RSA server key (in X.509 DER format), for encrypting the session key (**SK**). The size of the key MUST match RSA capabilities
Updatable	bool	True if the session is supporting post provisioning updates
ClientOperationLimit	short	Constraint for thwarting cryptographic attacks on **SK** by limiting the number of externally visible SKS-generated signed and/or encrypted data objects
SessionLifeTime	int	Validity of the provisioning session in seconds

Output

Name	Type	Comment
Status	byte	See Return Values
EncryptedSessionKey	byte[]	Encrypted **SK**
SessionKeyAttestation	byte[]	**SK** attestation signature
ProvisioningHandle	int	Local handle to created provisioning session

createProvisioningSession is the foundation for provisioning keys in an SKS. It performs the following steps in an *atomic* fashion:

- Generates a *random*, *secret* 32-byte **SK** (Session Key).
- Internally stores **SK**, **ClientSessionID**, **ServerSessionID**, **IssuerURI**, **Updatable**, **ClientOperationLimit**, current time + **SessionLifeTime** and returns a handle to the storage location in **ProvisioningHandle**.
- **EncryptedSessionKey** = ***Encrypt*** (**IssuerPublicKey**, **SK**)
- **SessionKeyAttestation** = ***Sign*** (**AttestationPrivateKey**, // See Architecture
  HMAC-SHA256 (**SK**, **ClientSessionID** ||
  ServerSessionID ||
  IssuerPublicKey ||
  IssuerURI ||
  Updatable ||
  ClientOperationLimit ||
  SessionLifeTime))

The purpose of **createProvisioningSession** is creating a shared session key (**SK**) that is only known by the issuer and the SKS. In addition, the SKS is authenticated by the issuer.

SK is used for *authenticating* and *encrypting* data that is exchanged between the issuer and the SKS during subsequent steps in the provisioning session.

If any succeeding operation associated with the provisioning session (through **ProvisioningHandle**), is regarded as incorrect by the SKS, *the session is immediately terminated and removed from internal storage*.

An SKS SHOULD only constrain the number of simultaneous sessions due to lack of storage.

A provisioning session SHOULD NOT be terminated due to power down of an SKS.

Notes

Encrypt uses the PKCS #1 RSAES algorithm.

Sign uses DIAS or PKCS #1 RSASSA signatures for RSA keys and ECDSA for EC keys with SHA256 as the hash function.

The **`ProvisioningHandle`** is guaranteed to be unique and never reused.

closeProvisioningSession (2)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to a provisioning session
GeneratedKeys	short	*Expected result*. What the issuer considers “has been ordered”
DeletedKeys	short	
ClonedKeys	short	
ReplacedKeys	short	
ExtensionObjects	short	
MAC	byte[32]	Vouches for the authenticity of the *expected result* parameters. See MAC Operations

Output

Name	Type	Comment
Status	byte	See Return Values
AttestedResponse	byte[32]	Attestation of the string **"Success"**. See SKS Attestations

closeProvisioningSession terminates a provisioning session and returns a proof of successful operation to the issuer. However, success status will only be returned if *all* of the following conditions are valid:

- There is an open provisioning session associated with **ProvisioningHandle**
- **MAC** matches the *expected result* parameters
- The *expected result* matches the SKS' internal calculations
- All generated keys are fully provisioned which means that matching public key certificates have been deployed. See setCertificatePath

When a provisioning session has been successfully closed by this method, it remains stored until all associated keys have been deleted. However, a closed provisioned session will only be a target for updates if its **Updatable** flag is true.

abortProvisioningSession (3)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to a provisioning session

Output

Name	Type	Comment
Status	byte	See Return Values

abortProvisioningSession is intended to be used by provisioning middleware if an unrecoverable error occurs in the communication with the issuer, or if a user cancels a session. If there is a matching and still *open* provisioning session, all associated data is removed from the SKS, otherwise an error is returned.

createPUKPolicy (5)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* provisioning session
ID	byte[]	ID string with a length of 1-32 bytes holding an *external* name of the PUK policy object. PUK IDs MUST be unique *within* a provisioning session
EncryptedValue	byte[]	Encrypted PUK value. See Encrypted Data
Format	byte	Format of PUK strings. See PIN and PUK Formats
RetryLimit	byte	Number of incorrect PUK values (*in a sequence*), forcing the PUK object to permanently lock up. A zero value indicates that there is no limit but that the SKS will introduce an *internal* 1-10 second delay *before* acting on an unlock operation in order to thwart exhaustive attacks

Output

Name	Type	Comment
Status	byte	See Return Values
PUKPolicyHandle	int	Non-zero local handle to created PUK policy object

createPUKPolicy creates a PUK policy object that is meant to be referenced by the createPINPolicy method. The purpose of a PUK is to facilitate a master key for unlocking keys that have locked-up due to faulty PIN entries. See TBD.

createPINPolicy (6)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* provisioning session
ID	byte[]	ID string with a length of 1-32 bytes holding an *external* name of the PIN policy object. PIN IDs MUST be unique *within* a provisioning session
PUKPolicyHandle	int	Handle to a governing PUK policy object or zero
UserDefined	bool	True if PINs belonging to keys governed by the PIN policy are supposed to be set by the user or by the issuer. See PINValue
UserModifiable	bool	True if PINs can be changed by the user after provisioning
Format	byte	Format of PIN strings. See PIN and PUK Formats
RetryLimit	byte	Non-zero value holding the number of incorrect PIN values (*in a sequence*), forcing a key to lock up
Grouping	byte	See PIN Grouping Control
PatternRestrictions	byte	See PIN Pattern Restrictions
MinLength	byte	Minimum PIN length in *bytes*. See PIN and PUK Formats
MaxLength	byte	Maximum PIN length in *bytes*. See PIN and PUK Formats
InputMethod	byte	See PIN Input Methods

Output

Name	Type	Comment
Status	byte	See Return Values
PINPolicyHandle	int	Non-zero local handle to created PIN policy object

createPINPolicy creates a PIN policy object that is meant to be referenced by the createKeyPair method.

If **PUKPolicyHandle** is zero no PUK is associated with the PIN policy object.

A **PUKPolicyHandle** value of 0xFFFFFFFF presumes that the target SKS supports a “device PUK”, *otherwise an error is returned*. The characteristics of device PUKs are out of scope for the SKS specification.

createKeyPair (7)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* provisioning session
ID	byte[]	ID string with a length of 1-32 bytes holding an *external* name of the key. Key IDs MUST be unique *within* a provisioning session
PINPolicyHandle	int	Handle to a governing PIN policy object or zero
PINValue	byte[]	PIN value must depending on **PINPolicyHandle** either be of zero length, contain a plain-text PIN value defined by the user (see UserDefined), or constitute of an encrypted PIN set by the issuer (see Encrypted Data)
PrivateKeyBackup	bool	True if the generated private key is to be put in **EncryptedPrivateKey** for backup by the issuer
Migratable	bool	True if SKS should permit the key to be exported (in clear text) by the user
Updatable	bool	True if the key is subject to post provisioning updates. Note that this also requires the provisioning session's **Updatable** flag to be true, otherwise an error is returned
DeleteProtected	bool	True if the key needs a PUK in order to be deleted by a user. Note that this option requires that the key is associated with a PUK, otherwise an error is returned. A conforming SKS may ignore this flag since it does not introduce any vulnerabilities
EnablePINCaching	bool	True if middleware *may* cache PINs for this key
ImportPrivateKey	bool	True if restorePrivateKey is allowed for this key
KeyUsage	byte	See Key Usage
FriendlyName	byte[]	String of 0-100 bytes that will be associated with this key
AttestationAlgorithm	byte	Attestation algorithm in local representation. See TBD
KeyAlgorithmType	byte	Type of key to be generated: 0x00 = RSA, 0x01 = EC
The following elements are only defined for RSA keys		
KeySize	short	RSA key size in bits
Exponent	byte[]	Zero-length (use default) or a defined exponent
The following element is only defined for EC keys		
NamedCurve	byte[]	URI like **"urn:oid:1.2.840.10045.3.1.7"**

Output

Name	Type	Comment
Status	byte	See Return Values
GeneratedPublicKey	byte[]	Generated public key in X.509 DER representation
KeyAttestation	byte[]	Attestation of the authenticity of the generated public key and associated data. See SKS Attestations
EncryptedPrivateKey	byte[]	*Optional*. This element MUST only be created if **PrivateKeyBackup** is true. If present it contains the generated private key in PKCS #8 format but wrapped as described in Encrypted Data
KeyHandle	int	Local handle to created key-pair object

createKeyPair creates an asymmetric key-pair inside of the SKS according to the issuer's specification.

If **`PINPolicyHandle`** is zero the key is not PIN-protected.

A **`PINPolicyHandle`** value of 0xFFFFFFFF presumes that the target SKS supports a “device PIN”, *otherwise an error is returned*. The characteristics of device PINs are out of scope for the SKS specification.

When using KeyGen2 the output from **`createKeyPair`** is translated as shown in the fragment below:

```
<KeyOperationResponse KeyAttestationAlgorithm="http://xmlns.webpki.org/keygen2/1.0#algorithm.key-
attestation-1-pkcs1"  ... >

  <GeneratedPublicKey ID="Key.1" KeyAttestation="X2oMtrm8rRL … XyTvPuTbergHfnJw==">
    <ds:KeyInfo>
      <ds:KeyValue>
        <ds:RSAKeyValue>
          <ds:Modulus>ALhBpUjJK/mSjPAe/ … fXG8z1V3mVDZTBM7eZ</ds:Modulus>
          <ds:Exponent>AQAB</ds:Exponent>
        </ds:RSAKeyValue>
      </ds:KeyValue>
    </ds:KeyInfo>
  </GeneratedPublicKey>

</KeyOperationResponse>
```

To assure the issuer that the generated key-pair actually resides in the SKS, the public key, together with attributes and protection objects are signed (attested) by the SKS according to the *Data* scheme on the next page:

```
addData ("PUK Policy=");
if (PINPolicyHandle == 0 || PINPolicyHandle == 0xFFFFFFFF ||
    PINPolicyHandle.PUKPolicyHandle == 0)
  {
    addData ("No PUK");
  }
else if (PINPolicyHandle.PUKPolicyHandle == 0xFFFFFFFF)  // Device PUK
  {
    addData ("Device PUK");
  }
else  // Standard PUK
  {
    addData ("Standard");
    addData (PINPolicyHandle.PUKPolicyHandle.ID);
    addData (PINPolicyHandle.PUKPolicyHandle.RetryLimit);
    addData (PINPolicyHandle.PUKPolicyHandle.clearTextPUKValue ());
    addData (PINPolicyHandle.PUKPolicyHandle.Format);
  }
addData ("PIN Policy=");
if (PINPolicyHandle == 0)  // The key is not PIN protected
  {
    addData ("No PIN");
  }
else if (PINPolicyHandle == 0xFFFFFFFF)  // The key is protected by a device PIN
  {
    addData ("Device PIN");
  }
else  // Standard PIN protection
  {
    addData ("Standard");
    addData (PINPolicyHandle.ID);
    addData (PINPolicyHandle.UserDefined);
    if (!PINPolicyHandle.UserDefined)
      {
        addData (clearTextPINValue ());
      }
    addData (PINPolicyHandle.UserModifiable);
    addData (PINPolicyHandle.Format);
    addData (PINPolicyHandle.RetryLimit);
    addData (PINPolicyHandle.Grouping);
    addData (PINPolicyHandle.PatternRestrictions);
    addData (PINPolicyHandle.MinLength);
    addData (PINPolicyHandle.MaxLength);
    addData (PINPolicyHandle.InputMethod);
  }
addData ("Key=");
addData (ID);
addData (GeneratedPublicKey);
addData (PrivateKeyBackup);
addData (Migratable);
addData (Updatable);
addData (DeleteProtected);
addData (EnablePINCaching);
addData (ImportPrivateKey);
addData (KeyUsage);
addData (FriendlyName);
```

setCertificatePath (8)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* provisioning session
KeyHandle	int	Local handle to a key-pair created in the provisioning session
PathLength	byte	Non-zero value holding the number of **X509Certificate** objects in the call
X509Certificate...	byte[]	DER-encoded X.509 certificate object which is *repeated* as defined by **PathLength**
MAC	byte[]	Vouches for integrity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

setCertificatePath attaches an X.509 certificate path to an already created key-pair. See createKeyPair.

The SKS does not verify that the certificate path and the public key match for keys having the **ImportPrivateKey** flag set because that would disable the restorePrivateKey method. For other keys, the SKS MAY perform such a test although it is redundant since the **MAC** is assumed to cater for the binding between certificate path and the generated public key. That is, a conforming SKS MAY always treat certificate path data as “an array of blobs”.

Note that **X509Certificate** objects MUST form an *ordered* certificate path so that the first object contains the end-entity certificate holding the public key of the target key-pair.

The certificate path MUST NOT contain any “holes” but does not have to be complete (include all CAs).

The **MAC** uses the method described in MAC Operations while *Data* is arranged as follows:

Data = **KeyHandle.GeneratedPublicKey** || **X509Certificate...**

On the next page there is a KeyGen2 fragment showing how it interacts with **setCertificatePath**:

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                             ID="_0fa47ab3c00c … a67992b6ac61c"
                             IssuerURI="https://ca.example.com/enroll"  …  >

  <CertifiedPublicKey ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEElFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

The following table illustrates argument mapping:

KeyGen2 Element	SKS Counterpart
`CredentialDeploymentRequest@ClientSessionID`	**`ProvisioningHandle.ClientSessionID`**
`CredentialDeploymentRequest@ID`	**`ProvisioningHandle.ServerSessionID`**
`CredentialDeploymentRequest@IssuerURI`	**`ProvisioningHandle.IssuerURI`**
`CertifiedPublicKey@ID`	**`KeyHandle.ID`**
`CertifiedPublicKey@MAC`	**`MAC`**
`X509Certificate...`	**`X509Certificate...`**

The actual **`ProvisioningHandle`** and **`KeyHandle`** can be retrieved by calling TBD and TBD respectively.

setPiggybackedSymmetricKey (19)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* provisioning session
KeyHandle	int	Local handle to a key-pair created in the provisioning session
EncryptedKey	byte[]	Encrypted piggybacked symmetric key
EndorsedAlgorithms	byte[]	Array holding granted symmetric key algorithms in local representation
MAC	byte[]	Vouches for integrity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

setPiggybackedSymmetricKey associates a symmetric key with an already created key-pair and certificate.

The **MAC** uses the method described in MAC Operations while *Data* is arranged as follows:

Data = End-Entity-Certificate || DecryptedKey || EndorsedAlgorithms

Where *End-Entity-Certificate* is the lowest member of the certificate path (see setCertificatePath), *DecryptedKey* is the decrypted (see Encrypted Data) **EncryptedKey**, while *EndorsedAlgorithms* denote the actual actual algorithms URIs sorted in alphabetical order.

Note that there can only be one symmetric key associated with a key-pair. See Key Usage.

The following KeyGen2 fragment shows the piggyback arrangement:

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                             ID="_0fa47ab3c00c … a67992b6ac61c"
                             IssuerURI="https://ca.example.com/enroll"  … >

  <CertifiedPublicKey ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEElFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <PiggybackedSymmetricKey EndorsedAlgorithms="http://www.w3.org/2000/09/xmldsig#hmac-sha1"
                             EncryptedKey="vInt09Esmg94vcG … YU3tgIdhcNNbyE9U2QKsNB7UA="
                             MAC="je7KiznTlQXFdUMRl … vInumZCjxSl1CrcqcGkI="/>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

For details on how to map keys and sessions, see setCertificatePath.

Note that the purpose of the X.509 part is only to serve as a universal immutable public key ID which can be securely linked to a specific issuer.

restorePrivateKey (20)

TBD

Sample Session

The following provisioning sample session shows the *sequence* for creating an X.509 certificate with a matching PIN-protected private key:

ProvisioningHandle, ... = createProvisioningSession (…)

PUKPolicyHandle = createPUKPolicy (**ProvisioningHandle**, ...)

PINPolicyHandle = createPINPolicy (**ProvisioningHandle**, , **PUKPolicyHandle**, ...)

KeyHandle, ... = createKeyPair (**ProvisioningHandle**, , **PINPolicyHandle**, ...)

setCertificatePath (**ProvisioningHandle**, **KeyHandle**, ...)

closeProvisioningSession (**ProvisioningHandle**, ...)

Note that **Handle** variables are only used by local middleware, while (not shown) variables like **SK**, **MAC**, **ID**, etc. are exclusively used between the issuer and the SKS.

If keys are to be created entirely locally, this requires a local software emulation of an issuer.

Security Considerations

This document does not cover the security of the actual key-store since SKS does not differ from other systems like smart cards in this respect.

However, SKS introduces a concept sometimes referred to as “air-tight” provisioning which has some specific security characteristics. One of the most critical operations in SKS is the creation of a random shared session key (**SK**) because if such a key is intercepted or guessed by an attacker, the integrity of the entire session is potentially jeopardized.

If you take a peek at createProvisioningSession you will note that **SK** is encrypted by an issuer-supplied public key. It is pretty obvious that a malicious middleware could replace this key with one it has the private key to and an SKS wouldn't notice any difference. This is where the attestation signature comes in because it is computationally infeasible creating a matching signature since the public key is a part of the signed object. That is, the issuer will when receiving the response to the provisioning session request, detect if it has been manipulated and cease the rest of the operation in that case.

As earlier noted, the randomness of **SK** is crucial for all provisioning operations.

Replay attacks are indeed feasible since there is no general rolling nonce scheme, but the SKS “book-keeping” functions will detect any irregularities during closeProvisioningSession. This means that an issuer SHOULD NOT consider issued credentials as valid unless it has received a successful response from closeProvisioningSession.

The **ClientOperationLimit** in createProvisioningSession is another security measure which aims to limit exhaustive attacks on **SK**. In most provisioning sessions only a handful of SK-related operations are actually needed.

One of the most important features in SKS is the fact that the device is identified by a digital certificate, preferably issued by a presumably known vendor of trusted hardware. This enables the issuer to securely identify the key-container both from a cryptographic point of view (brand, type etc) and and as a specific unit. The latter makes it possible to communicate the container identity as an SHA1 fingerprint of the Device Certificate which facilitates novel and secure enrollment procedures, typically eliminating the traditional sign-up password.

There is no protection against DoS (Denial of Service) attacks on SKS storage space due to malicious middleware.

SKS does not have any notion of policy, it is up to the issuer to decide what a suitable key size is and which private keys that should be backed-up. Provisioning middleware MAY also enforce certain policies by rejecting “bad” requests.

Intellectual Property Rights

This document contains several constructs that *could* be patentable but the author has no such interests and therefore puts the entire design in “public domain” allowing anybody to use all or parts of it at their discretion. In case you adopt something you found useful in this specification, feel free mentioning where you got it from ☺

Note: it is possible that there are pieces that already are patented by *other parties* but the author is currently unaware of any IPR encumbrances.

References

KeyGen2	TBD
DIAS	TBD
PKCS #1	TDB
PKCS #8	TBD
ECDSA	TBD
AES256-CBC	TBD
HMAC-SHA256	TBD
X.509	TBD
SHA256	TBD
TPM 1.2	TBD
Diffie-Hellman	TBD
S/MIME	TBD

Acknowledgments

There is a bunch of organizations, mailing-lists, and individuals that have been instrumental for the creation of SKS. I need to check who would accept to be mentioned :-)

Author

Anders Rundgren
anders.rundgren@telia.com