SKS (Secure Key Store)
API and Architecture

Invention Disclosure

This document is a complete description of a version of a smart card, that can be “personalized” with keys directly from an Internet site, like a “cloud service”. It contains both standard technology and methods that are *potentially* novel. Rather than ripping out pieces for the novel approaches, I took the liberty of adding this preface with references to the interesting chapters.

Innovation #1: During each provisioning session a unique shared secret is established between the *issuer and the card* and *stored inside* of the card. Subsequent MAC-, encryption-, and attestation-operations (using the shared session key), combined with object bookkeeping enables multi-stage provisioning of keys and attributes while maintaining a *technical* security- and data-integrity level which is fully comparable to traditional smart card production in a “bunker”. This scheme is coined “air-tight” provisioning. See Architecture, createProvisioningSession, closeProvisioningSession, and createKeyPair.

Innovation #2: Creating an attested key-pair where not only the public key is attested, but all associated attributes including PIN and PUK data as well. See createKeyPair, createPINPolicy, and createPUKPolicy.

Innovation #3: Remote key lookup is a privacy-enabled cryptographic method for an issuer to remotely lookup a consumer's virtual wallet and only seeing the virtual “cards” that it has issued. See Remote Key Lookup.

innovation #4: Shared and persistent provisioning session key enables key life-cycle management through *post-provisioning operations* while maintaining *issuer isolation* and only using simple symmetric crypto. See Architecture, createProvisioningSession, closeProvisioningSession, createKeyPair, setCertificatePath, and postProvisioningDeleteKey.

Innovation #5: MAC (Message Authentication Code) operations that are “targeted” by including the method's literal name in the derived key. See MAC Operations. That is, such a MAC will only be valid for the intended method.

Innovation #6: Using an X.509 certificate as a universal key ID for PKI, symmetric keys, Information Cards etc.

This document has been filed at http://defensivepublications.org

Disclaimer: This is an early version of a system in development. That is, the specification may change without notice. However, it might give still you a fairly good idea about the “air-tight” provisioning concept. *Feedback is encouraged!*

Table of Contents

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* and *life-cycle management* of cryptographic keys and associated attributes.

In addition to PKI and symmetric keys (including OTP applications), SKS also supports recent additions to the credential family tree like Information Cards.

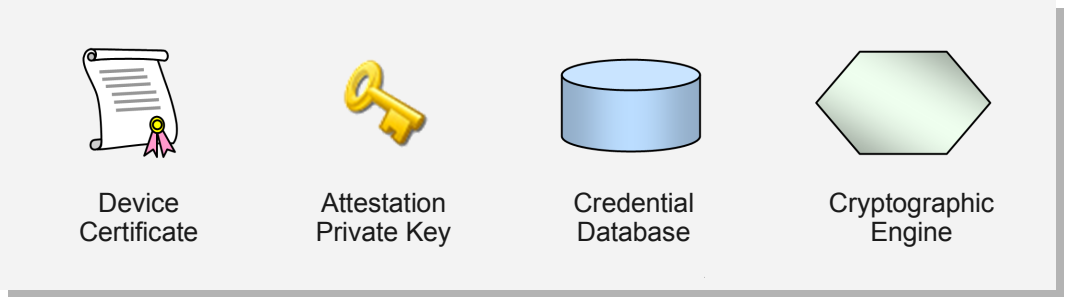
The primary objective with SKS and the related specifications is *establishing two-factor authentication as a viable alternative for any provider* by making the scheme a standard feature in the "Universal Client", the Internet browser.

An equally important means for reaching this undeniable bold goal, is that the API and protocols mandate full "on-the-wire" compliance in order to eliminate the current "Smart Card Middleware Hell"; *a single driver per platform should suffice*.

Could *existing* smart card users also benefit from an upgraded token technology? Yes, the new ways of working, like *virtual organization*s, doesn't make the current distribution scheme "come and get your card" particularly useful.

Architecture

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



The *Device Certificate* forms together with a matching *Attestation Private Key* 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 *Credential Database* holds keys and other data that is related to keys such as protection and extension objects. It also keeps the provisioning state.

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.

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 memory where programs are stored, but sealed by a CPU-resident "Master Key".

Provisioning API

Although SKS may be regarded as a "component", it actually comprises of three associated pieces: The KeyGen2 protocol, the SKS architecture, and the provisioning API described in this document. These items are *tightly matched* to form a *secure* and *interoperable* ecosystem for cryptographic keys.

One of the biggest challenges with the SKS Provisioning API was enabling independent issuers to securely *share* a single "key ring". The rationale for this was mainly to support mobile phones with built-in "trusted hardware", but it appears that USB memory sticks augmented with SKS functionality would be a slightly more realistic product offering if they could deal with a potentially large chunk of a consumer's authentication hassles on the Internet.

Backward Compatibility

A question that arises is of course how compatible the SKS Provisioning API is with respect to existing protocols, APIs, and smart cards. The answer is simply: NOT AT ALL due to the fact that current schemes do *generally* not support secure on-line provisioning and key life-cycle management directly towards end-users.

In fact, *smart cards are almost exclusively personalized by more or less proprietary software under the supervision of card administrators or performed in 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"

Although the lack of compatibility with the current state-of-the-art ("nothing"), may be regarded as a major short-coming, the good news is that SKS by separating key provisioning from actual usage, *does neither require applications nor cryptographic APIs to be rewritten*. See next section.

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 "connected" 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 the majority of current PKI-using applications rely on.

The standard User API does not utilize authenticated sessions like featured in TPM 1.2 because this is a *local security option*, which is independent of the *network centric* Provisioning API.

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

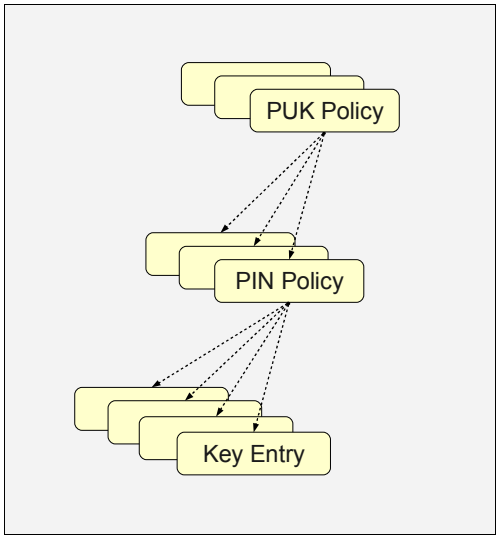
The SKS API (as well as its companion protocol KeyGen2), assumes that objects are arranged in a specific fashion in order to work. At the heart of the system there are the typical cryptographic keys intended for user authentication, signing etc., but also dedicated keys supporting life-cycle management and of user keys and attributes.

All provisioned user keys, included symmetric dittos (see setSymmetricKey), are identified and managed through an X.509 certificate. The reason for this somewhat unusual arrangement is that this enables *universal key IDs* as well as *secure remote object management by independent issuers*. See Remote Key Lookup.

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

Key Protection Objects

Keys may optionally be protected by PIN-codes (aka “passphrases”). 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. Below is an illustration of the SKS protection object hierarchy:



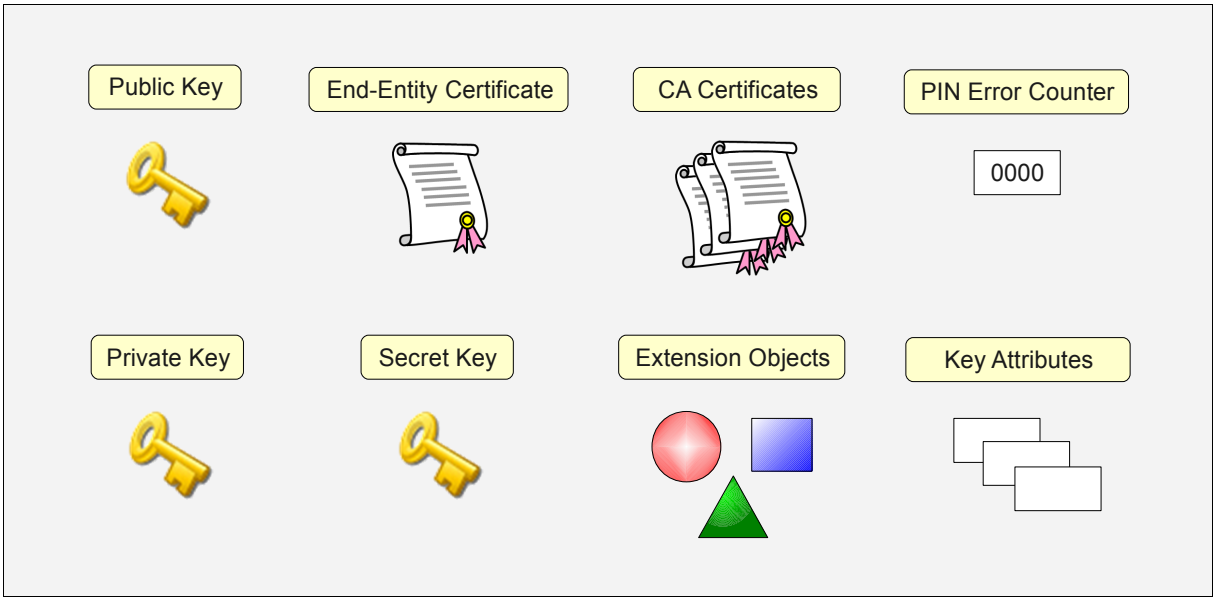
An SKS MAY also support a device (*system-wide*) PIN and PUK. See getDeviceInfo.

For the creation of protection objects, see createPUKPolicy, createPINPolicy and createKeyPair.

For an example how KeyGen2 deals with this structure, see KeyInitializationRequest.

Key Entries

The following picture shows the components inside of an SKS key entry:



Public Key denotes the public part of the key-pair created by createKeyPair.

Private Key denotes the private part of the key-pair created by createKeyPair.

End-Entity Certificate denotes the X.509 certificate set by the mandatory call to setCertificatePath.

Secret Key denotes an *optional* secret key defined by calling setSymmetricKey.

CA Certificates denote *optional* X.509 CA certificates defined during the call to setCertificatePath.

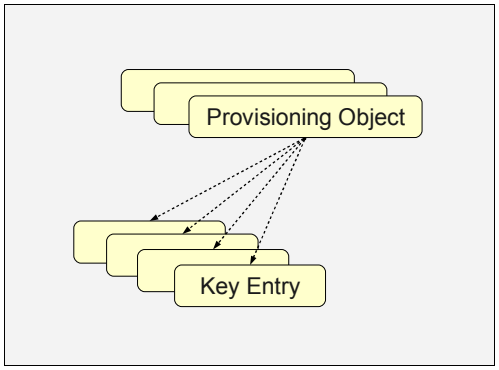
Extension Objects denote *optional* extension objects defined by calling addExtensionData.

PIN Error Counter denotes a counter associated by keys protected by a local PIN policy object. See createPINPolicy.

Key Attributes denote the attributes defined during the call to createKeyPair.

Provisioning Objects

The following picture shows how provisioning objects “own” the keys they have provisioned:



For detailed information concerning the contents of a provisioning object see createProvisioningSession.

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 XML Encryption
http://www.w3.org/2001/04/xmlenc#aes192-cbc	
http://www.w3.org/2001/04/xmlenc#aes256-cbc	
http://xmlns.webpki.org/keygen2/1.0#algorithm.aes.ecb.nopad	See FIPS 197. Support for 128, 192, and 256-bit keys
http://xmlns.webpki.org/keygen2/1.0#algorithm.aes.ecb.pkcs5	
HMAC Operations	
http://www.w3.org/2000/09/xmldsig#hmac-sha1	See XML Signature
http://www.w3.org/2001/04/xmldsig-more#hmac-sha256	
Asymmetric Key Encryption	
http://www.w3.org/2001/04/xmlenc#rsa-1_5	See XML Encryption
Asymmetric Key Signatures	
http://www.w3.org/2000/09/xmldsig#rsa-sha1	See XML Signature
http://www.w3.org/2001/04/xmldsig-more#rsa-sha256	
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256	
Session Keys	
http://xmlns.webpki.org/keygen2/1.0#algorithm.sk1	See createProvisioningSession
Key Attestations	
http://xmlns.webpki.org/keygen2/1.0#algorithm.ka1	See Key Attestation Data and createKeyPair
Elliptic Curves	
urn:oid:1.2.840.10045.3.1.7	Also known as “P-256”. See FIPS 186-3

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

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.

Note that all variables that represent textual data are supposed to be UTF-8 encoded.

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_AUTHORIZATION	0x01	This non-fatal error is returned when there is something wrong with a supplied PIN-code. See getKeyProtectionInfo
ERROR_NOT_ALLOWED	0x02	Operation is not allowed
ERROR_STORAGE	0x03	There is no persistent storage available for the operation
ERROR_MAC	0x04	MAC does not match supplied data
ERROR_CRYPTO	0x05	Various cryptographic errors
ERROR_NO_SESSION	0x06	Provisioning session not found
ERROR_SESSION_VERIFY	0x07	closeProvisioningSession failed to verify
ERROR_NO_KEY	0x08	Key not found
ERROR_ALGORITHM	0x09	Unknown or not fitting algorithm
ERROR_OPTION	0x0A	Invalid or unsupported option
ERROR_INTERNAL	0x0B	Internal error

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 mandate that the data-carrying arguments are included in a MAC (Message Authentication Code) operation as well. 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* represents 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.

Attestations

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

Post Provisioning MAC Operations

In order to update already provisioned objects, an issuer may perform *post provisioning* operations. To do that the issuer MUST provide a valid MAC (see MAC Operations) based on *the original session keys* where *MethodName* is set to **"Proof Of Issuance"** while *Data* holds the End-Entity Certificate of the *target key*. Note that post provisioning operations require that the target key's Updatable flag is set. Also see closeProvisioningSession.

PIN and PUK Formats

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

KeyGen2 Name	Value	Comment
numeric	0x00	0 - 9
alphanumeric	0x01	0 - 9, A - Z
string	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 **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:

KeyGen2 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:

KeyGen2 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:

KeyGen2 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.

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:

KeyGen2 Name	Value	Notes	Comment
signature	0x00	1	The key MUST only be used in signature applications like S/MIME
authentication	0x01	1	The key MUST only be used in authentication applications
encryption	0x02	1	The key MUST only be used for PKCS #1 or Diffie-Hellman encryption operations
universal	0x03	1	There are no restrictions on key usage
transport	0x04	1, 2, 3	The private key MUST NOT be available for the User API
symmetric-key	0x05	3	The key MUST include a “piggybacked” symmetric key during provisioning. The private key MUST be *disabled*. See setSymmetricKey

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.

Notes

1. The key MUST NOT be subject to a setSymmetricKey operation.
2. The key MUST NOT be exportable. See ExportPolicy.
3. The key MUST NOT have the ImportPrivateKey or PrivateKeyBackup attributes set to true.

Methods

This section provides a (*not yet 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. User API methods have method IDs ≥ 100.

createProvisioningSession (1)

Input

Name	Type	Comment
SessionKeyAlgorithm	byte	Algorithm in local representation. See description below and Session Keys
EncryptionAlgorithm	byte	Algorithm in local representation. See Asymmetric Key Encryption
ServerSessionID	byte[32]	Server nonce value
ClientSessionID	byte[32]	Client nonce value
IssuerURI	byte[]	URI identifying the issuer
IssuerPublicKey	byte[]	Issuer-supplied RSA key (in X.509 DER format), for encrypting the session key (**SK**). The size of the key MUST match getDeviceInfo
Updatable	bool	True if objects created in the session should support post provisioning updates
CurrentTime	int	The provisioning client's time in UNIX epoch format
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 a number of internal operations in an *atomic* fashion. Shown below is the mandatory to support SKS session key creation algorithm:

http://xmlns.webpki.org/keygen2/1.0#algorithm.sk1

- Generate a *random*, *secret* 32-byte **SK** (Session Key).
- Store **SK**, **ClientSessionID**, **ServerSessionID**, **IssuerURI**, **Updatable**, **ClientOperationLimit**, **CurrentTime**, and **SessionLifeTime** in the Credential Database and return a handle to the database entry in **ProvisioningHandle**.
- Set **EncryptedSessionKey** = ***Encrypt*** (**IssuerPublicKey**, **SK**)
- Set **SessionKeyAttestation** = ***Sign*** (*Attestation Private Key*, // See Architecture
  HMAC-SHA256 (**SK**, **ClientSessionID** ||
  ServerSessionID ||
  IssuerPublicKey ||
  IssuerURI ||
  Updatable ||
  ClientOperationLimit ||
  SessionLifeTime))

The purpose of **createProvisioningSession** is establishing a shared session key (**SK**) that is only known by the issuer and the SKS which is used in subsequent provisioning steps. In addition, the SKS is *optionally* authenticated by the issuer.

Remarks

If any succeeding operation in the same provisioning session, 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.

Using KeyGen2 **`IssuerURI`** is the URL to which the result of this method is POSTed. The string MUST NOT exceed 1024 bytes.

The ***Encrypt*** function MUST use the algorithm specified in the **`EncryptionAlgorithm`** parameter.

IssuerPublicKey SHOULD NOT be authenticated because a compliant SKS is supposed to be *agnostic* with respect to issuers. In fact, it is RECOMMENDED using an *ephemeral* IssuerPublicKey.

The ***Sign*** function MUST use DIAS or PKCS #1 RSASSA signatures for RSA keys and ECDSA for EC keys with SHA256 as the hash function. The distinction between RSA and ECDSA keys is performed through the Device Certificate (see getDeviceInfo) which in KeyGen2 is supplied as well as a part of the response to the issuer, while a DIAS signature also requires the DIAS policy OID to be present in the Device Certificate.

`ProvisioningHandle` MUST be *static*, *unique* and never be reused.

The **`SessionKeyAlgorithm`** does not only define the creation of **`SK`**, but also the integrity, confidentiality, and attestation mechanisms used during the provisioning session. See MAC Operations, Encrypted Data, and Attestations.

closeProvisioningSession (2)

Input

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

Output

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

closeProvisioningSession terminates a provisioning session and returns a proof of successful operation to the issuer. However, success status MUST 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 in declaration order using MAC Operations
- 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
- There are no unreferenced PIN or PUK policy objects

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.

If the verification is successful, **closeProvisioningSession** MUST also reassign the provisioning session ownership to the current (closing) session for *all* objects belonging to sessions that have been subject to a post provisioning operation. The original session objects MUST subsequently be deleted since they have no mission anymore.

getProvisioningSession (3)

Input

Name	Type	Comment
ServerSessionID	byte[32]	See createProvisioningSession
ClientSessionID	byte[32]	
IssuerURI	byte[]	

Output

Name	Type	Comment
Status	byte	See Return Values
ProvisioningHandle	int	Local handle to created provisioning session

getProvisioningSession is intended to be used by provisioning middleware for retrieving handles to *open* provisioning sessions in sessions that are interrupted due to a certification process or similar.

In addition, users of portable SKSes (like smart cards), may carry out provisioning steps on *different* computers through this method.

enumerateProvisioningSessions (4)

Input

Name	Type	Comment
ProvisioningHandle	int	Input enumeration handle

Output

Name	Type	Comment
Status	byte	See Return Values
ProvisioningHandle	int	Output enumeration handle
The following elements are only available if **ProvisioningHandle** <> 0xFFFFFFFF		
IsOpen	bool	True if the session is open
ServerSessionID	byte[32]	See createProvisioningSession
ClientSessionID	byte[32]	
IssuerURI	byte[]	

enumerateProvisioningSessions is used for enumerating provisioning session data which is primarily of interest for debugging and “cleaning” purposes.

The input **ProvisioningHandle** is initially set to 0xFFFFFFFF to start an enumeration round.

Succeeding calls should use the output **ProvisioningHandle** as input to the next call.

When **enumerateProvisioningSessions** returns with a **ProvisioningHandle** = 0xFFFFFFFF there are no more provisioning objects to read.

abortProvisioningSession (5)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* 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.

signProvisioningSessionData (6)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* provisioning session
Data	byte[]	Data to be signed

Output

Name	Type	Comment
Status	byte	See Return Values
Result	byte[32]	Signed data

signProvisioningSessionData signs *arbitrary data* that is supplied *by the provisioning middleware.*

The purpose of **signProvisioningSessionData** is adding data integrity to provisioning messages from clients to issuers.

The signature algorithm used is the same as for MAC Operations, with *MethodName* set to **"External Signature"**.

A *relying party* MUST distinguish between such signatures and Attestations since only the latter are actually vouched for by the SKS.

Also see ClientOperationLimit.

createPUKPolicy (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 PUK policy object. PUK IDs MUST be unique *within* a provisioning session
PUKValue	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 handle to locally defined PUK policy object

createPUKPolicy creates a local PUK policy object in the Credential Database to be referenced by subsequent calls to 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 unlockKey.

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

createPINPolicy (8)

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 handle to locally defined PIN policy object

createPINPolicy creates a local PIN policy object in the Credential Database to be referenced by subsequent calls to the createKeyPair method.

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

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

createKeyPair (9)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* provisioning session
AttestationAlgorithm	byte	Attestation algorithm in local representation. See Key Attestations
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[]	Object which MUST depending on **PINPolicyHandle** either be of zero length, else depending on UserDefined contain a plain-text PIN value defined by the user or constitute of an encrypted PIN set by the issuer (see Encrypted Data)
BiometricProtection	byte	See Biometric Protection Options
PrivateKeyBackup	bool	True if the generated private key is to be outputted in **PrivateKey** for backup by the issuer
ExportPolicy	byte	See ExportPolicy
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
DeleteProtection	byte	See DeleteProtection
EnablePINCaching	bool	True if middleware MAY cache PINs for this key
ImportPrivateKey	bool	True if restorePrivateKey is *expected* for this key
KeyUsage	byte	See Key Usage
FriendlyName	byte[]	String of 0-100 bytes that will be associated with this key for use in GUIs
KeyAlgorithmType	byte	Type of key to be generated: 0x00 = RSA, 0x01 = ECC
The following two elements are only defined for RSA keys		
RSAKeySize	short	RSA key size in bits. See getDeviceInfo
RSAExponent	int	Zero (use default) or a defined exponent. See getDeviceInfo
The following element is only defined for ECC keys		
NamedCurve	byte	URI in local representation. See Elliptic Curves

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 Attestations
PrivateKey	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 generates an asymmetric key-pair in the Credential Database according to the issuer's specification.

Remarks

`KeyHandle` MUST be *static*, *unique* and never be reused.

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

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

A compliant SKS SHOULD use 65537 as the default RSA exponent value.

A non-zero **`BiometricProtection`** value presumes that the target SKS supports Biometric Protection Options, *otherwise an error is returned*. See getDeviceInfo.

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

Continued on the next page...

```
////////////////////////////////////////////////////////////////////////////////
// Key Attestation Data for: http://xmlns.webpki.org/keygen2/1.0#algorithm.ka1  //
////////////////////////////////////////////////////////////////////////////////

addData ("PUK Policy=");
if (PINPolicyHandle == 0 || PINPolicyHandle.PUKPolicyHandle == 0)
  {
    addData ("N/A");
  }
else if (PINPolicyHandle == 0xFFFFFFFF)  // Device PIN implies device PUK
  {
    addData ("Device");
  }
else  // Standard PUK
  {
    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 ("N/A");
  }
else if (PINPolicyHandle == 0xFFFFFFFF)  // The key is protected by a device PIN
  {
    addData ("Device");
  }
else  // Standard PIN protection
  {
    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 (BiometricProtection);
addData (ExportPolicy);
addData (Updatable);
addData (DeleteProtection);
addData (EnablePINCaching);
addData (ImportPrivateKey);
addData (KeyUsage);
addData (FriendlyName);
```

Continued on the next page...

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

```
<KeyInitializationRequest  ... >

  <CreateObject>
    <PUKPolicy ID="PUK.1" Format="numeric" RetryLimit="3" Value="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>

</KeyInitializationRequest>
```

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

In the sample KeyGen2 *default values* have been utilized which is why there are few *visible* key generation attributes.

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

```
<KeyInitializationResponse KeyAttestationAlgorithm="http://xmlns.webpki.org/keygen2/1.0#algorithm.ka1"  ... >

  <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>
  <GeneratedPublicKey ID="Key.2" KeyAttestation="TbergHftrm8rRL … wyTvPX2XoMunJ==">
    <ds:KeyInfo>
      <ds:KeyValue>
        <ds:RSAKeyValue>
          <ds:Modulus>ADZTBMLhBpUjJKe/ … fXG8z17eZV/mSjPA3mV</ds:Modulus>
          <ds:Exponent>AQAB</ds:Exponent>
        </ds:RSAKeyValue>
      </ds:KeyValue>
    </ds:KeyInfo>
  </GeneratedPublicKey>

</KeyInitializationResponse>
```

setCertificatePath (10)

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[32]	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 where *Data* is arranged as follows:

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

Continued on the next page...

The following KeyGen2 fragment shows its interaction with **`setCertificatePath`**:

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

  <CertifiedPublicKey ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

The table below 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 getProvisioningSession and enumerateKeys respectively. Note that in an *interactive* provisioning session, the various handles and IDs involved are preferably cached by the provisioning middleware, eliminating the need for enumerating keys etc.

setSymmetricKey (11)

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
SymmetricKey	byte[]	“Piggybacked” symmetric key encrypted as described in Encrypted Data
EndorsedAlgorithms	byte[]	Array holding granted symmetric key algorithms in local representation
MAC	byte[32]	Vouches for integrity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

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

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

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

DecryptedKey holds the decrypted version of **SymmetricKey**, while *EndorsedAlgorithms* denote the actual actual algorithm URIs sorted in alphabetical order.

There MUST only be a single symmetric key defined for a given 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>
    <SymmetricKey EndorsedAlgorithms="http://www.w3.org/2000/09/xmldsig#hmac-sha1"
                  MAC="je7KiznTlQXFdUMRl … vInumZCjxSl1CrcqcGkI=">vInt09Esmg94v ...
                                                                 YU3tgIdhcNNby</SymmetricKey>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

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

Note that the X.509 certificate serves as the key ID. That is, *SKS treats asymmetric and symmetric keys close to identically*.

addExtensionData (12)

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
BasicType	byte	See table below
Qualifier	byte[]	See table below
ExtensionData	blob	Extension object. Regarding size constraints see getDeviceInfo
ExternalType	byte[]	KeyGen2 "Type" URI
MAC	byte[32]	Vouches for integrity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

addExtensionData adds attribute (extension) data to an already created key-pair and certificate.

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

Data = End-Entity Certificate || *Order* || **BasicType** || **Qualifier** || **ExtensionData** || **ExternalType**

Order holds a single byte with the declaration order (starting at 0x00) of the extension for the *target key* and is intended to assure issuers that extensions are not replayed, replaced, or forgotten. *This arrangement requires that the request order of extensions is respected throughout the provisioning process*.

The following table shows **BasicType**, **Qualifier** and **ExtensionData** mapping using KeyGen2:

KeyGen2 Element	BasicType	Qualifier	ExtensionData
Extension	**0x00**	N/A	Binary data extracted from Base64-encoded XML
EncryptedExtension	**0x01**	N/A	Encrypted binary data extracted from Base64-encoded XML
PropertyBag	**0x02**	N/A	XML data *canonicalized* using the XML Signature http://www.w3.org/2001/10/xml-exc-c14n# algorithm
Logotype	**0x03**	Mime Type	Binary data extracted from Base64-encoded XML

Remarks

N/A = zero-length array.

Note that the **EncryptedExtension** is handled slightly different: **ExtensionData** (which is encrypted as described in Encrypted Data) is first *decrypted* (inside the SKS) before storing and MACing.

All **ExternalType** attributes associated with a given key MUST be unique.

Although not a part of the current SKS specification, an extension could be created for consumption by the SKS only, like downloaded JavaCard code. In that case the associated **ExternalType** MUST be featured in the SKS *supported algorithm list*. See getDeviceInfo and getExtensionObject.

Continued on the next page...

Below is a KeyGen2 fragment showing an **`Extension`** object holding a Base64-encoded Information Card:

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

  <CertifiedPublicKey ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <Extension Type="http://schemas.xmlsoap.org/ws/2005/05/identity"
               MAC="UMRlje7KiznTlQXFd … CrcqcGkIvInumZCjxSl1=">PD94bWwgdmVyc2lvbj0iMS4w
                                                              IiBlbmHVyZS ...  B4bWxuczpkc
                                                              d3dy53My5vc</Extension>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

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

The following is a KeyGen2 sample showing the **`PropertyBag`** and **`Logotype`** objects added to a symmetric key for usage by an HOTP application:

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

  <CertifiedPublicKey ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <SymmetricKey EndorsedAlgorithms="http://www.w3.org/2000/09/xmldsig#hmac-sha1"
                  MAC="je7KiznTlQXFdUMRl … vInumZCjxSl1CrcqcGkI=">vInt09Esmg94v ...
                                                                 YU3tgIdhcNNby</SymmetricKey>
    <PropertyBag Type="http://xmlns.webpki.org/keygen2/1.0#provider.ietf-hotp"
                 MAC="jIOHDgwI4dO7Kzs … uEH8MtykIS46JfiJ3N=">
      <Property Name="Counter" Value="0" Writable="true"/>
      <Property Name="Digits" Value="8"/>
    </PropertyBag>
    <Logotype MimeType="image/png"
              Type="http://xmlns.webpki.org/keygen2/1.0#logotype.application"
              MAC="+crSq5fv+7z+fx+f … ZmRnhxljO0bh0d=">iVBORw0KGgoAAAANSUhEUgAAALo
                                                      AAABKCAIAAACD … /tm/AAALjUlEQVR42u
                                                      2d6W8UyRXA+=</Logotype>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

restorePrivateKey (20)

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
PrivateKey	byte[]	Private key in PKCS #8 format wrapped as described in Encrypted Data
MAC	byte[32]	Vouches for integrity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

restorePrivateKey replaces a generated private key with a key supplied by the issuer.

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

Data = End-Entity Certificate || *DecrypedKey*

DecryptedKey holds the decrypted version of **PrivateKey**.

The purpose of **restorePrivateKey** (preceded by setCertificatePath), is to install a certificate and private key that the issuer has kept a backup of although the certificate may have been renewed (while using the same key). It may also be used to deploy an entirely issuer-generated credential.

Prerequisite: see the ImportPrivateKey attribute.

A conforming SKS SHOULD NOT accept multiple restores of the same key within a provisioning session.

The following KeyGen2 fragment shows how credentials that are to be restored should be formatted:

```
<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>
    <PrivateKey MAC="umZCjxSI1znTIQX … CrcqcGkIvInFdUMRlje7Ki=">c2lvbj0iMSwgdmVy4wPD94bW
                                                              VyZSIiBlbmH ...  uczpkcB4bWx
                                                              My5vcd3dy53</PrivateKey>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

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

getDeviceInfo (31)

Input

Name	Type	Comment
This method does not have any input arguments		

Output

Name	Type	Comment
Status	byte	See Return Values
APILevel	short	0x0001 => Applies to *this* API specification
VendorName	byte[]	1-100 byte string holding the name of the vendor
VendorDescription	byte[]	1-100 byte string holding a vendor description of the SKS device
DeviceCertPathLength	byte	Non-zero value holding the number of **X509Certificate** objects
X509Certificate...	byte[]	DER-encoded X.509 certificate object which is *repeated* as defined by **DeviceCertPathLength**
Algorithms	byte	Number of supported algorithms
The following *two* elements are repeated as defined by **Algorithms**		
Algorithm	byte[]	The algorithm URI. See Algorithm Support
LocalAlgorithmID	byte	Holds a *local representation* of the algorithm for usage in SKS methods
RSAExponentSupport	bool	True if the issuer may specify an *explicit* exponent value
RSAKeySizes	byte	Number of supported RSA key sizes
RSAKeySize...	short	Holds an RSA key size in *bits* and is *repeated* as defined by **RSAKeySizes**
ExtensionDataSize	int	Maximum size of ExtensionData objects
DevicePINSupport	bool	True if the SKS supports a device PIN. See createKeyPair
BiometricSupport	bool	True if the SKS supports biometric authentication options. See Biometric Protection Options

getDeviceData lists the core characteristics of an SKS which is used by provisioning schemes like KeyGen2.

Note that **X509Certificate** objects MUST form an *ordered* certificate path so that the *first* object contains the actual SKS Device Certificate.

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

RSAKeySizes MUST be *ordered* so that the smallest key size is *first* in the list.

A compliant SKS MUST support at least 1024-bit and 2048-bit RSA keys.

A compliant SKS SHOULD support ExtensionData objects with sizes of at least 65536 bytes.

For ECC key generation see Algorithm Support.

enumerateKeys (32)

Input

Name	Type	Comment
KeyHandle	int	Input enumeration handle

Output

Name	Type	Comment
Status	byte	See Return Values
KeyHandle	int	Output enumeration handle
The following elements are only available if **KeyHandle** <> 0xFFFFFFFF		
ProvisioningHandle	int	Handle to the associated provisioning session object
ID	byte[]	See createKeyPair
KeyUsage	byte	
FriendlyName	byte[]	
PathLength	byte	See setCertificatePath
X509Certificate...	byte[]	

enumerateKeys is used for enumerating keys both for open and closed provisioning sessions. If an enumerated key has not yet been fitted with a certificate, **PathLength** is zero.

Key Usage determines if the key is asymmetric or symmetric.

For asymmetric keys End-Entity Certificate signifies RSA or ECC.

The input **KeyHandle** is initially set to 0xFFFFFFFF to start an enumeration round.

Succeeding calls should use the output **KeyHandle** as input to the next call.

When **enumerateKeys** returns with a **KeyHandle** = 0xFFFFFFFF there are no more key objects to read.

postProvisioningDeleteKey (50)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* provisioning session
KeyHandle	int	Local handle to the target key
MAC	byte[32]	Vouches for integrity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

postProvisioningDeleteKey deletes a key created in an earlier provisioning session. In order to perform this operation the issuer MUST supply a matching MAC according to MAC Operations where *Data* is arranged as follows:

Data = Post Provisioning MAC

The key to be deleted MUST be present, otherwise the provisioning session will be aborted. See Remote Key Lookup.

postProvisioningUpdateKey (51)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* provisioning session
KeyHandle	int	Local handle to a new key associated by End-Entity Certificate
KeyHandleOriginal	int	Local handle to the old (=target) key
MAC	byte[32]	Vouches for integrity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

postProvisioningUpdateKey updates a key created in an earlier provisioning session. In order to perform this operation the issuer MUST supply a matching MAC according to MAC Operations where *Data* is arranged as follows:

Data = Post Provisioning MAC || End-Entity Certificate

The new (update) key MUST be fully provisioned before this method is called. In addition, the new key MUST NOT be PIN-protected since it supposed to *inherit* the old key's PIN protection scheme (if there is one).

Note that updating a key involves *all related data* (see Key Entries), with PINs as the only exception.

The **KeyHandle** of the updated key MUST after a successful update be set equal to **KeyHandleOriginal**.

signHashedData (100)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key
SignatureAlgorithm	byte[]	Algorithm in local representation. See Asymmetric Key Signatures
PIN	byte[]	Holds a PIN value or is of zero length indicating that no PIN is supplied
HashedData	byte[]	Hashed data to be signed. Length MUST match the hash algorithm

Output

Name	Type	Comment
Status	byte	See Return Values
Result	byte[]	Signed data including algorithm-specific padding

signHashedData performs an asymmetric key signature where the data MUST be hashed *as required by the signature algorithm*.

In addition to possible internal errors that do not have any obvious handling in user programs, invalid authorization data (PIN) to the key MUST return a non-fatal ERROR_AUTHORIZATION status.

getKeyProtectionInfo (101)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key

Output

Name	Type	Comment
Status	byte	See Return Values
ProtectionStatus	byte	See table below
PUKRetryLimit	byte	Copy of **RetryLimit** defined by createPUKPolicy or zero if not defined
PUKErrorCount	byte	Current PUK error count for keys protected by a local PUK policy object
UserModifiable	bool	Exact copies of the corresponding createPINPolicy parameters if the key is protected by a local PIN policy object, otherwise these elements contain zeros
Format	byte	
RetryLimit	byte	
Grouping	byte	
PatternRestrictions	byte	
MinLength	byte	
MaxLength	byte	
InputMethod	byte	
ErrorCount	byte	Current PIN error count for keys protected by a local PIN policy object
BiometricProtection	byte	Exact copies of the corresponding createKeyPair parameters
PrivateKeyBackup	bool	
ExportPolicy	byte	
Updatable	bool	
DeleteProtection	byte	
EnablePINCaching	bool	
ImportPrivateKey	bool	

getKeyProtectionInfo returns information about the protection scheme for a key including PIN-codes and possible biometric options. In addition, the call retrieves the current protection status for the key.

The following table illustrates how the **ProtectionStatus** bit field should be interpreted:

Bit	Comment
0	The key is protected by a local PIN policy object
1	The key is protected by a local PUK policy object. MUST be combined with bit 0
2	The key has locked-up due to PIN errors. MUST be combined with bit 0
3	The key has locked-up due to PUK errors. MUST be combined with bit 1
4	The key is protected by a device PIN. Information about device PINs is out of scope for the SKS API

If all bits are zero the key is not PIN protected.

getExtensionObject (102)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key
Index	byte	Extension order 0..255. See addExtensionData

Output

Name	Type	Comment
Status	byte	See Return Values
HasData	bool	True if there is an extension with order = **Index**
The following elements are only available if **HasData** is true		
BasicType	byte	Exact copies of the corresponding addExtensionData parameters
Qualifier	byte[]	
ExtensionData	blob	
ExternalType	byte[]	

getExtensionObject returns an extension object associated with a key.

Note that encrypted extensions are decrypted during provisioning.

If the extension is intended to be consumed by the SKS, **ExtensionData** will be a zero-length array.

deleteKey (103)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key
OptionalAuthorization	byte[]	Zero-length array or a PIN or PUK string depending on DeleteProtection

Output

Name	Type	Comment
Status	byte	See Return Values

deleteKey removes a key from the Credential Database.

If the key is the last belonging to a provisioning session, the session data objects are removed as well.

In addition to possible internal errors that do not have any obvious handling in user programs, invalid authorization data to the key MUST return a non-fatal ERROR_AUTHORIZATION status.

The following table illustrates the use of the **DeleteProtection** attribute:

KeyGen2 Name	Value	Comment
none	0x00	No delete restrictions apply
pin	0x01	Correct PIN is required
puk	0x02	Correct PUK is required

A conforming SKS MAY introduce physical presence methods like GPIO-based buttons, *circumventing* key delete protection attributes.

unlockKey (104)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key
Authorization	byte[]	PUK string

Output

Name	Type	Comment
Status	byte	See Return Values

unlockKey re-enables a key that has been locked due to erroneous PIN entries.

Note that this method only applies to keys that are protected by local PIN and PUK policy objects. Device PINs and their possible unlocking is out of scope for the SKS API.

In addition to possible internal errors that do not have any obvious handling in user programs, invalid authorization data to the key MUST return a non-fatal ERROR_AUTHORIZATION status.

exportKey (105)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key
OptionalAuthorization	byte[]	Zero-length array or a PIN or PUK string depending on ExportPolicy

Output

Name	Type	Comment
Status	byte	See Return Values
RawKey	byte[]	Unencrypted raw key. For type information see enumerateKeys

exportKey exports a private or secret key from the Credential Database.

In addition to possible internal errors that do not have any obvious handling in user programs, invalid authorization data to the key MUST return a non-fatal ERROR_AUTHORIZATION status.

The following table illustrates the use of the **ExportPolicy** attribute:

KeyGen2 Name	Value	Comment
non-exportable	0x00	The key MUST NOT be exported
pin	0x01	Correct PIN is required
puk	0x02	Correct PUK is required
none	0x03	No authorization needed for exporting the key

If a **non-exportable** key is referred to, **exportKey** MUST return ERROR_NOT_ALLOWED status.

Biometric Protection Options

SKS also supports options for using biometric data as an alternative to PINs which is defined in the createKeyPair method. The following table shows the biometric protection options:

KeyGen2 Name	Value	Comment
`none`	0x00	No biometric protection
`alternative`	0x01	The key may be authorized with a PIN *or* by biometrics
`combined`	0x02	The key is protected by a PIN *and* by biometrics
`exclusive`	0x03	The key is *only* protected by biometrics

Note that there is no API support for biometric authentication, such information is typically provided through GPIO (General Purpose Input Output) ports between the biometric sensor and the SKS. The type of biometrics used is outside the scope of SKS and is usually established during enrollment.

The biometric protection option is only intended to be applied to User API methods like signHashedData.

Sample Session

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

`ProvisioningHandle`, ... = createProvisioningSession (…)

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

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

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

External certification of the generated public key happens here...

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 primarily used in the communication between an issuer and the SKS.

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

Remote Key Lookup

In order to update keys and related data, SKS supports post provisioning operations like postProvisioningDeleteKey where issuers are securely shielded from each other by the use of a Post Provisioning MAC.

However, depending on the use-case, an issuer may need to get a list of applicable keys, *before* launching post provisioning operations. Such a facility is available in KeyGen2 as illustrated by the XML fragment below:

```
<CredentialDiscoveryRequest ... >

  <LookupSpecifier ID="Lookup.1" Email="john.doe@example.com" Nonce="nSgmg4cznqE ... WrH2421w9SYA=">
    <ds:Signature>
      <ds:SignedInfo>
        <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
        <ds:Reference URI="#Lookup.1">
          <ds:Transforms>
            <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
            <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </ds:Transforms>
          <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
          <ds:DigestValue>JBfoi8iBKRyWxXYITTU1cdyybMTyJr+WDW+qCJdxoGE=</ds:DigestValue>
        </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue>mSMaH6wChPQRDT... JKrW3n/dL7seGbg==</ds:SignatureValue>
      <ds:KeyInfo>
        <ds:X509Data>
          <ds:X509IssuerSerial>
            <ds:X509IssuerName>CN=Root CA,O=example.com,C=us</ds:X509IssuerName>
            <ds:X509SerialNumber>2</ds:X509SerialNumber>
          </ds:X509IssuerSerial>
          <!-- The Issuer's Certificate: -->
          <ds:X509Certificate>MIIDbzCCAlegAw … gtzO/rITZcbKHyCZvQ==</ds:X509Certificate>
        </ds:X509Data>
      </ds:KeyInfo>
    </ds:Signature>
  </LookupSpecifier>

</CredentialDiscoveryRequest>
```

The example works as follows:

1. Verify that the **`Signature`** is *technically* valid. Note that the actual issuer is *ignored* since an SKS has no opinion about what issuers are trustworthy or not
2. Verify that the freshness **`Nonce`** matches SHA256 (**`ClientSessionID`** || **`ServerSessionID`** || **`IssuerURI`**). See createProvisioningSession.
3. Enumerate all SKS keys and related certificates. See enumerateKeys
4. Find all keys from step #3 having an End-Entity Certificate forming a valid 2-level certificate path with the **`X509Certificate`** element which (together with the **`Signature`**), serves as an *Issuer Filter*
5. Collect the keys from step #4 that also feature the e-mail addresss **`"john.doe@example.com"`** in the End-Entity Certificate.

The result is sent back to the issuer in the form of a list of SHA256 (End-Entity Certificate) fingerprints.

Remote key lookups are performed at the *middleware level* since they are passive, XML intensive, and do not access private or secret keys. The primary purpose with credential lookups is *improving provisioning robustness*, while the *Issuer Filter protects user privacy* by constraining lookup data to the party to where it belongs.

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 the SKS wouldn't notice the 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*.

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 possible 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 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.

A predecessor of this document has been submitted to http://defensivepublications.org.

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
UTF-8	TBD
XML Encryption	TBD
XML Signature	TBD
FIPS 197	TBD
FIPS 186-3	TBD
Information Cards	TBD
Base64	TBD
HOTP	TBD
JavaCard	TBD
JCE	TBD
CryptoAPI	TBD
PKCS #11	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