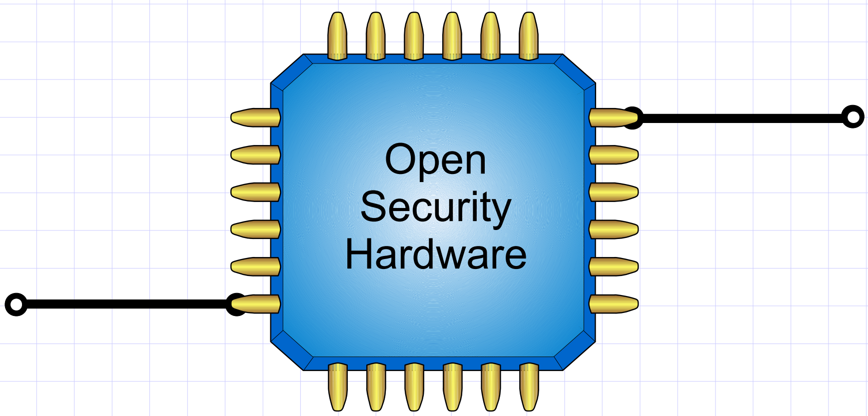
DRAFT

SKS (Secure Key Store)
API and Architecture



Disclaimer: This is a system in development. That is, the specification may change without notice.

Table of Contents

1 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 *secure*, *reliable*, and *user-friendly 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*.

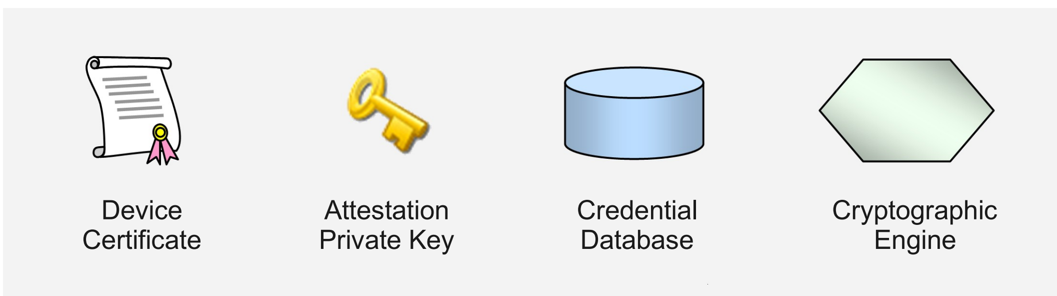
Could *existing* smart card users also benefit from an upgraded token technology? Yes, the new ways of working, like *virtual organization*s, doesn't make the current distribution scheme “Come and get your card” particularly useful.

For maintaining a link to the world of legacy authentication an SKS may also serve as a “Password Vault”.

2 Core Functionality

2.1 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 session mechanism that facilitates secure provisioning of keys, also when the provisioning middleware and network are non-secure.

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”.

2.2 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* which is more or less a prerequisite for *large-scale*, *secure* and *interoperable* ecosystems of cryptographic keys. Also see KeyGen2 Proxy.

One of the core features of the SKS Provisioning API is enabling independent issuers securely *sharing* a single “Key Ring”. The rationale for this is mainly to support mobile phones with embedded “Trusted Hardware”, but it appears that the already quite popular USB memory sticks augmented with SKS functionality would be a realistic product offering if they could deal with a potentially large chunk of a consumer's authentication hassles on the Internet.

2.3 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 smart card middleware has 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.

2.4 Security Model

Since the primary target for SKS is authentication to arbitrary service providers on the Internet, the security model is quite different to traditional multi-application card schemes like GlobalPlatform. In practical terms this means that it is the *user* who grants an issuer the right to create keys in the SKS. That is, there are no preconfigured “Security Domains”.

However, an issuer may during a provisioning session define a VSD (Virtual Security Domain) which enables *post provisioning (update) operations* by the issuer, while cryptographically shielding provisioned data from similar actions by *other* issuers.

When using KeyGen2 the grant operation is performed through a GUI dialog triggered by an issuer request, which in turn is the result of the user browsing to an issuer-related web address.

The SKS itself only trusts inbound data that can securely be derived from a session key created in the initial phase of a provisioning session. See createProvisioningSession.

The session key scheme is conceptually similar to GlobalPlatform's SCP (Secure Channel Protocol) but details differ because KeyGen2 uses an on-the-wire XML format requiring encoding/decoding by the middleware, rather than raw APDUs.

Regarding who trusts an SKS, this is effectively up to each issuer to decide and may be established anytime during an enrollment procedure. Trust in an SKS can be highly granular like only accepting requests from preregistered units or be fully open ended where any SKS complaint device is accepted. A potentially useful issuer policy would be specifying a set of endorsed SKS brands, presumably meeting some generally recognized certification level like EAL5.

Many smart card schemes depend on roles like SO (Security Officer) which squarely matches scenarios where users are associated with a *multitude of independent service providers*. By building on an E2ES (End To End Security) model, the *technical* part of the SO role, exclusively becomes an affair between the SKS and the *remote* issuers, *where each issuer is confined to their own virtual cards and SO policies*.

Also see Security Considerations and Privacy Enabled Provisioning.

2.5 Transaction Based Operation

An important characteristic for maintaining integrity and robustness is that provisioning and management operations either succeed or leave the system intact. This is accomplished by *deferring* the actual “commit” of container-modifying operations until the terminating closeProvisioningSession call.

Ideally an SKS container should be able dealing with power-failures regardless when they occur.

2.6 Privacy Enabled Provisioning

Note: Credential *provisioning* and credential *usage* (at least when the issuer is independent of the relying party)*, represent two entirely different scenarios from a privacy point of view*.

Although a one-size-fits-all approach would be nice, it seems that the span of Internet-related services motivates a design that supports on-line identity schemes where issuers have (often quite substantial) knowledge about users, as well as close to fully anonymous relationships.

The "Standard" E2ES (End To End Security) mode which exploits the SKS Device Certificate and Attestation Private Key in the provisioning API, is intended to suit the needs of banks, employers, governments, and high-security third-party identity providers.

The PEP (Privacy Enabled Provisioning) mode is identical to the E2ES mode, with the exception that the identity of the SKS is excluded. A valid question is if the PEP mode is equally secure as the E2ES mode. The simple answer to that is a clear "No", since the issuer neither learns the type (=quality, brand), nor the identity of the SKS.

However, from a *user's horizon* the PEP mode is as secure and trustworthy as the E2ES mode as long as the client platform is intact and the correct issuer enrollment URL is used. After provisioning there are no security differences whatsoever between the two modes.

From a purely technical perspective, Blind Signatures or elaborate schemes like TCG's DAA (Direct Anonymous Attestation) could also have been applied. Adoption considerations for a mode primarily intended replacing passwords were governing the decision keeping things simple.

The PEP mode is selected by the PrivacyEnabled parameter of createProvisioningSession.

2.7 Device ID

Since the exposed identity of the SKS container is dependent on the mode as described in the previous section, the affected provisioning methods refer to a "Device ID" which is the literal string **`"Anonymous"`** and the binary form of Device Certificate for the Privacy Enabled Provisioning and E2ES mode respectively.

2.8 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"

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

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*.

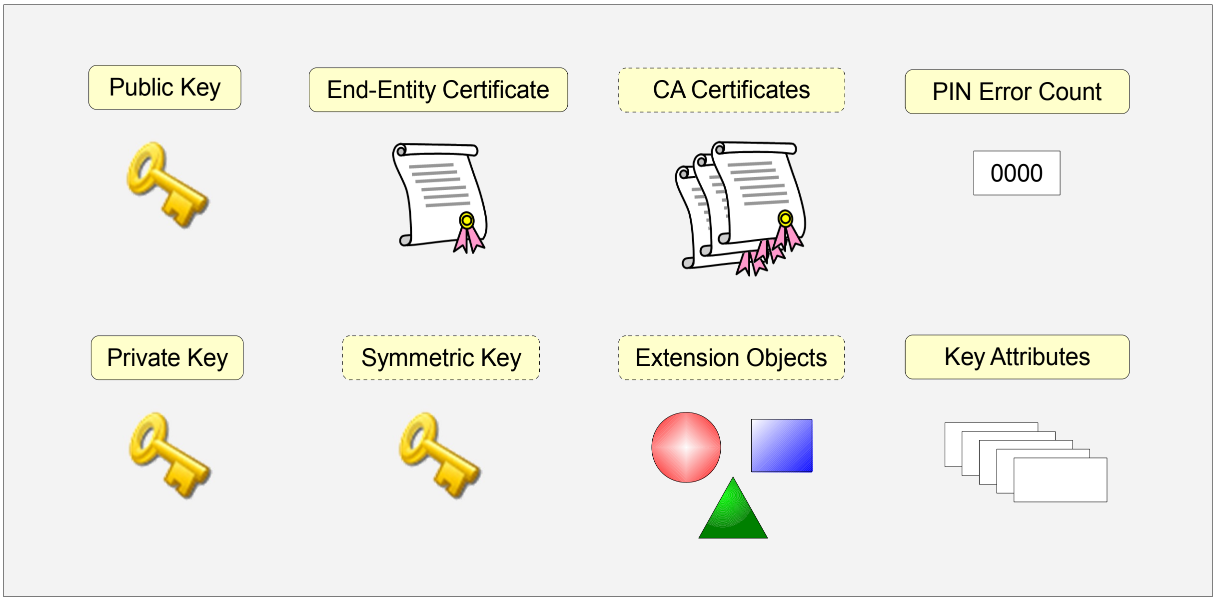
3 Objects

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, including symmetric dittos (see importSymmetricKey), are identified by X.509 certificates. 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.

3.1 Key Entries

The following picture shows the elements forming an SKS key entry:

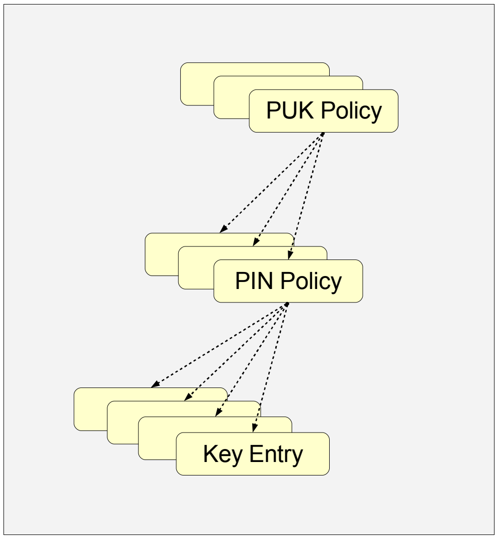


Element	Description
Public Key	Public part of the asymmetric key-pair created by createKeyEntry
Private Key	Private part of the asymmetric key-pair created by createKeyEntry
End-Entity Certificate	X.509 certificate set by the *mandatory* call to setCertificatePath
Symmetric Key	*Optional* symmetric key defined by calling importSymmetricKey
CA Certificates	*Optional* X.509 CA certificates defined during the call to setCertificatePath
Extension Objects	*Optional* extension objects defined by calling addExtension
PIN Error Count	Counter associated by keys protected by a PIN policy object. See createPINPolicy
Key Attributes	Attributes defined during the call to createKeyEntry

Note that key management operations always involve an entire key entry; *individual elements cannot be managed*.

3.2 Key Protection Objects

Keys can *optionally* be protected by PIN-codes ("passphrases"). PIN-protected keys maintain separate PIN error counters, but a single PIN policy object may govern multiple keys. A PIN policy and its associated keys can in turn be supplemented by an optional PUK (PIN 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:



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

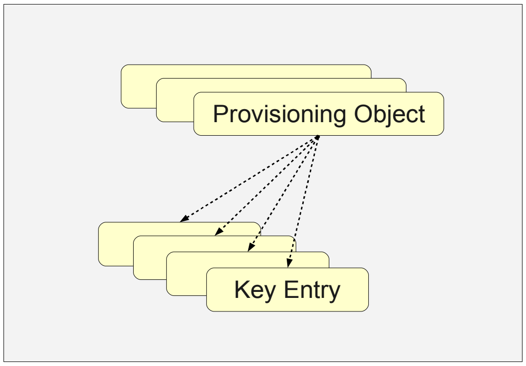
For an example how KeyGen2 deals with this structure, see KeyCreationRequest.

Note that the set of keys bound to a particular PIN policy object "owns" the PIN policy object which means that when the *last* key of such a set has been deleted, the PIN policy object itself **must** be *automatically* deleted (by pp_deleteKey and deleteKey). The very same principle is also valid for PUK policy objects. Due to this there are no specific PIN or PUK delete methods.

An *embedded* SKS **may** also support a device (*system-wide*) PIN and PUK. See getDeviceInfo. *Usage and management of device PINs and PUKs is out of scope for the SKS API*.

3.3 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.

Note that when the *last* key owned by a provisioning object has been deleted, the provisioning object itself **must** be *automatically* deleted (by closeProvisioningSession and deleteKey).

If a KeyManagementKey is deployed during provisioning object creation (establishing a VSD), *post-provisioning operations* can also be performed. See pp_deleteKey, pp_unlockKey, pp_updateKey, and pp_cloneKeyProtection.

4 Algorithm Support

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

URI	Description

Symmetric Key Encryption	
http://www.w3.org/2001/04/xmlenc#aes128-cbc	See XML Encryption. Note that IV **must** be *internally generated* as well as *prepended* to encrypted data
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.cbc.pkcs5	See FIPS 197. Support for 128, 192, and 256-bit keys
http://xmlns.webpki.org/keygen2/1.0#algorithm.aes.ecb.nopad	

HMAC Operations	
http://www.w3.org/2000/09/xmldsig#hmac-sha1	See HMAC-SHA1
http://www.w3.org/2001/04/xmldsig-more#hmac-sha256	See HMAC-SHA256

Asymmetric Key Encryption		
http://www.w3.org/2001/04/xmlenc#rsa-1_5	See XML Encryption	*Decryption mode only*
http://xmlns.webpki.org/keygen2/1.0#algorithm.rsa.raw	Non-padded RSA operation	

Diffie-Hellman Key Agreement	
http://xmlns.webpki.org/keygen2/1.0#algorithm.ecdh.raw	See SP800-56A ECC CDH primitive (Section 5.7.1.2)

Asymmetric Key Signatures		
http://www.w3.org/2000/09/xmldsig#rsa-sha1	See XML Signature	*Signing mode only*
http://www.w3.org/2001/04/xmldsig-more#rsa-sha256		
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256		
http://xmlns.webpki.org/keygen2/1.0#algorithm.rsa.none	See signHashedData	
http://xmlns.webpki.org/keygen2/1.0#algorithm.ecdsa.none		

Elliptic Curves	
urn:oid:1.2.840.10045.3.1.7	Also known as “P-256”. See FIPS 186-3

Special Algorithms	
http://xmlns.webpki.org/keygen2/1.0#algorithm.sks.s1	See createProvisioningSession
http://xmlns.webpki.org/keygen2/1.0#algorithm.sks.k1	See createKeyEntry
http://xmlns.webpki.org/keygen2/1.0#algorithm.none	See createKeyEntry and importSymmetricKey

Supported algorithms can be acquired by calling getDeviceInfo which also lists RSA key generation capabilities which **must** include 1024 and 2048 bit keys. Note that RSA “multi-prime” keys are not supported by this specification.

5 Key Protection Attributes

The following section describes the attributes issuers need to set for defining suitable key protection policies. Also see getKeyProtectionInfo, KeyManagementKey, DevicePINProtection, and EnablePINCaching.

During provisioning of *user defined PINs*, the provisioning middleware **should** maintain the PIN policy and optionally ask the user to create another PIN if there is a policy mismatch because createKeyEntry **must** return an error and abort the entire session if fed with incorrect data. Also see KeyGen2 Proxy.

In addition to protection policies, a key **may** also be constrained with respect to algorithm usage. See EndorsedAlgorithms.

5.1 Key Export

The following table illustrates the use of the ExportProtection attribute:

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

Also see exportKey.

5.2 Key Delete

The following table illustrates the use of the DeleteProtection attribute:

KeyGen2 Name	Value	Description
none	0x00	No delete restrictions apply
pin	0x01	Correct PIN is required
puk	0x02	Correct PUK is required
non-deletable	0x03	The key **must not** be deleted

Also see deleteKey.

5.3 PIN Input Methods

The InputMethod policy attribute tells how PINs **should** be inputted to the SKS according to the following table:

KeyGen2 Name	Value	Description
`programmatic`	0x01	PINs **should** only be given 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
`any`	0x03	No restrictions

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

5.4 PIN Patterns

The PatternRestrictions policy attribute specifies how PINs **must** be designed according to the following table:

KeyGen2 Name	Value	Description
`two-in-a-row`	0x01	Flags 1124 as *invalid*
`three-in-a-row`	0x02	Flags 1114 as *invalid*
`sequence`	0x04	Flags 1234, 9876, etc as *invalid*
`repeated`	0x08	All PIN bytes **must** be *unique*
`missing-group`	0x10	The PIN format **must** be **`alphanumeric`** or **`string`** and contain a mix of *letters* and *digits*. The **`string`** format also requires *lowercase* letters and *non-alphanumeric* characters. See PIN and PUK Formats

Note that the PatternRestrictions byte actually holds 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.

5.5 PIN and PUK Formats

PINs and PUKs **must** adhere to one of formats described in the following table:

KeyGen2 Name	Value	Description
`numeric`	0x00	0 - 9
`alphanumeric`	0x01	0 - 9, A - Z
`string`	0x02	Any valid UTF-8 string
`binary`	0x03	Binary value, typically expressed as hexadecimal data

Note that format specifiers only deal with how PINs and PUKs are treated in GUIs; internally and in the SKS API, key protection data **must** always be handled as *decoded* strings of bytes. A conforming SKS **must** perform syntax validation during createKeyEntry on **`numeric`** and **`alphanumeric`** PIN data. Length of the clear-text binary value **must not** exceed 128 bytes. See **`Format`** attribute in createPINPolicy and createPUKPolicy.

5.6 PIN Grouping

A PIN policy object may govern multiple keys. The Grouping policy attribute (which is intimately linked to the Key Application Usage scheme), controls how PINs to the different keys relate to each other according to the following table:

KeyGen2 Name	Value	Description
none	0x00	No restrictions
shared	0x01	All keys share the *same* PIN (*synchronized*)
signature+standard	0x02	Keys with AppUsage = **signature** share a common PIN while all other keys share *another* PIN (*synchronized*)
unique	0x03	All four AppUsage types must have *different* PINs while keys with the same AppUsage share a common PIN (*synchronized*)

Note that keys having a **shared** PIN groping attribute **must** be treated as having a single virtual PIN error counter, while **signature+standard** implies two separate error counters. “Synchronized” means that a PIN *value* or *status* change **must** propagate to all keys sharing the PIN.

5.7 Key Application Usage

The AppUsage attribute specifies what *applications* keys are intended for according to the following table:

KeyGen2 Name	Value	Description
signature	0x00	The key **should** only be used in signature applications like S/MIME
authentication	0x01	The key **should** only be used in applications like TLS client certificate authentication and login to AD (Active Directory)
encryption	0x02	The key **should** only be used in encryption applications
universal	0x03	There are no restrictions on key usage

Enforcement of AppUsage is up to each *application* to perform.

Note that AppUsage **must not** constrain a key's *internal* use of cryptographic algorithms in any way, because for that purpose there is the EndorsedAlgorithm mechanism.

Although AppUsage could be be regarded as a duplication of the X.509 key usage and extended key usage attributes the latter have proved hard to use as “filters” to certificate selection GUIs. AppUsage is also applicable for other credentials like OTPs (One Time Passwords) and Information Cards.

However, an equally important target for AppUsage is that in conjunction with PIN Grouping provide the means for aiding users in PIN input GUIs in the case an issuer requires separate PINs for different keys and applications.

The following matrix shows the **recommended** interpretation of PIN GUI “hints”:

PIN Grouping	**signature**	**authentication**	**encryption**	**universal**
none	PIN	PIN	PIN	PIN
shared	PIN	PIN	PIN	PIN
signature+standard	Signature PIN	PIN	PIN	PIN
unique	Signature PIN	Authentication PIN	Encryption PIN	PIN

For this scheme to work a prerequisite is (of course) that the middleware is specifically adapted for SKS.

5.8 Biometric Protection

An SKS **may** also support using biometric data as an alternative to PINs. See getDeviceInfo. The following table shows the biometric protection options as defined by the BiometricProtection policy attribute:

KeyGen2 Name	Value	Description
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.

6 Session Security Mechanisms

After the SessionKey has been created the actual provisioning methods can be called. Depending on the specific method downloaded data may be confidential or need to be authenticated. For certain operations the SKS needs to prove for the issuer that sent data indeed stems from internal SKS operations which is referred to as attestations.

This section describes the security mechanisms used during a provisioning session. Also see SessionKeyLimit.

String literals like **`"Encryption Key"`** featured in the following definitions **must** be supplied "as is" *without* length indicators, while when being a part of MAC or attestation *Data* arguments **must** be treated as described in Data Types.

6.1 Encrypted Data

During provisioning encrypted data is occasionally exchanged between the issuer and the SKS. The encryption key is created by the following key derivation scheme:

`EncryptionKey` = HMAC-SHA256 (SessionKey, **`"Encryption Key"`**)

`EncryptionKey` **must** only be used with the AES256-CBC algorithm. Note that the IV (Initialization Vector) **must** always be *prepended* to the encrypted data as in XML Encryption.

6.2 MAC Operations

In order to verify the integrity of provisioned data, many of the provisioning methods mandate that the data-carrying arguments are included in a MAC (Message Authentication Code) operation as well. MAC operations use the following scheme:

`MAC` = HMAC-SHA256 (SessionKey || *MethodName* || MACSequenceCounter, *Data*)

MethodName is the string literal of the target method like **`"closeProvisioningSession"`**, while *Data* represents the arguments as specified for the actual method. Note that individual elements featured in *Data* **must** use the representation described in Data Types.

After each MAC operation, MACSequenceCounter **must** be incremented by one. Due the use of a sequence counter, the provisioning system **must** honor the order of objects as defined by the issuer.

6.3 Attestations

Attestations created by the SKS are identical to MAC Operations where *MethodName* is set to **`"Device Attestation"`**.

6.4 Target Key Reference

In order to perform post provisioning operations the issuer must provide evidence of ownership to keys. *Target Key Reference* denotes a key management authorization signature scheme using the KeyManagementKey associated with the "owning" provisioning object of the target key (see Provisioning Objects) according to the following:

`Authorization` = *Sign* (KeyManagementKey$_{target}$,
HMAC-SHA256 (SessionKey$_{current}$ || Device ID, End-Entity Certificate$_{target}$))

Notes:

- All elements **must** be represented "as is" in the HMAC operation, *excluding* length information
- *Sign* **must** use an PKCS #1 RSASSA signature for RSA keys and ECDSA for EC keys with the *private key* associated with **`KeyManagementKey`**, and utilizing SHA256 as hash function
- An SKS **must** verify that the signature validates with respect to the *public key* (**`KeyManagementKey`**) as well as checking that End-Entity Certificate matches **`TargetKeyHandle`**
- If a KeyManagementKey is not present in the target key's provisioning object, the key is considered "not updatable" and the provisioning session **must** be aborted
- The provisioning session **must** be aborted if the PrivacyEnabled flag differs between the original and the updating session.

7 Detailed Operation

This chapter describes the SKS API in detail.

7.1 Data Types

The table below shows the data types used by the SKS API. Note that multi-byte integers **must** be stored in big-endian fashion whenever they are *serialized* like in MAC operations. Also see Method List.

Type	Length	Description
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
id	2 + length	Special form of byte[] which **must** contain an 1-32 byte string according to the XML Schema data type *NCName* but restricted to: | a-z | A-Z | 0-9 | . | _ | - |
uri	2 + length	UTF-8 encoded byte[] array which **must not** exceed 1000 bytes

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

Variables and literals that represent textual data **must** be UTF-8 encoded and *not* include terminating null characters; they are in this specification considered equivalent to byte[].

Note that length indicators are only applicable to *array objects* when included in MAC operations, or when they are serialized.

7.2 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 **must** be a second return value containing a UTF-8 encoded description in *English* to be used for logging and debugging purposes as shown below:

Name	Type	Description
`Status`	byte	Non-zero (error) value
`ErrorString`	byte[]	A human-readable error description

7.3 Error Codes

The following table shows the standard SKS error-codes:

Name	Value	Description
`ERROR_AUTHORIZATION`	0x01	Non-fatal error returned when there is something wrong with a supplied PIN or PUK code. See getKeyProtectionInfo
`ERROR_NOT_ALLOWED`	0x02	Operation is not allowed
`ERROR_STORAGE`	0x03	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_NO_KEY`	0x07	Key not found
`ERROR_ALGORITHM`	0x08	Unknown or non-matching algorithm
`ERROR_OPTION`	0x09	Invalid or unsupported option
`ERROR_INTERNAL`	0x0A	Internal error
`ERROR_EXTERNAL`	0x0B	External error like communication link failure
`ERROR_USER_ABORT`	0x0C	User aborted PIN input or similar
`ERROR_NOT_AVAILABLE`	0x0D	External error when a requested SKS is unavailable

7.4 Method List

This section provides a list of the SKS methods. The number in square brackets denotes the *decimal value* used to identify the method in a call. Method calls are formatted as strings of bytes where the first byte holds the method ID and the succeeding bytes the applicable argument data. User API methods have method IDs ≥ 100.

> *Note: The described API is adapted for an SKS using low-level byte-streams for communication. However, the SKS design is equally applicable to API schemes using high-level objects and exceptions. The only thing that **must** remain intact are the cryptographic operations including how objects are represented in MACs.*

Note that a `KeyHandle` in this specification always refers to a *key entry*. See Key Entries.

getDeviceInfo [1]

Input

Name	Type	Description
This method does not have any input arguments		

Output

Name	Type	Description
`Status`	byte	See Return Values
`APILevel`	short	100 (1.00) => Applies to *this* API specification
`DeviceType`	byte	Holds basic device data. See DeviceType
`UpdateURL`	uri	URL pointing to a firmware update service or a zero length array. See updateFirmware
`VendorName`	byte[]	1-128 byte string holding the name of the vendor
`VendorDescription`	byte[]	1-128 byte string holding a vendor description of the SKS device
`PathLength`	byte	Non-zero value holding the number of **`X509Certificate`** objects
`X509Certificate...`	byte[]	DER encoded X.509 certificate object *repeated* as defined by **`PathLength`**
`SupportedAlgorithms`	short	Non-zero value holding the number of **`SupportedAlgorithm`** objects
`SupportedAlgorithm...`	uri	Algorithm URI *repeated* as defined by **`SupportedAlgorithms`**. See Algorithm Support
`RSAExponentSupport`	bool	True if the issuer may specify an *explicit* exponent value. See createKeyEntry
`RSAKeySizes`	byte	Non-zero value holding the number of **`RSAKeySize`** objects
`RSAKeySize...`	short	Holds an RSA key size in *bits* and is *repeated* as defined by **`RSAKeySizes`**. See Algorithm Support
`CryptoDataSize`	int	Maximum number of bytes in the **`Data`** argument of cryptographic methods
`ExtensionDataSize`	int	Maximum size of ExtensionData objects
`DevicePINSupport`	bool	True if the SKS supports a device PIN. See createKeyEntry
`BiometricSupport`	bool	True if the SKS supports biometric authentication options. See Biometric Protection

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

The **`X509Certificate`** objects **must** form an *ordered* and *contiguous* certificate path so that the *first* object contains the actual SKS Device Certificate. The path does though not have to be complete (include all upper-level CAs).

`RSAKeySizes` **must** be specified in *ascending order*.

A compliant SKS **must** support ExtensionData objects with a size of at least 65536 bytes.

A compliant SKS **must** support a **`CryptoDataSize`** of at least 16384 bytes.

For EC key generation see Elliptic Curves.

Continued on the next page...

DeviceType contains a set of fields according to the following table:

Bit	Value	Label	Description
0-1	0x00	**LOCATION_EXTERNAL**	Connected device
	0x01	**LOCATION_EMBEDDED**	Embedded in the client platform
	0x02	**LOCATION_SOCKETED**	Mounted inside a socket
	0x03	**LOCATION_SIM**	SIM/USIM card
2-3	0x00	**TYPE_SOFTWARE**	Software implementation
	0x04	**TYPE_HARDWARE**	Unqualified hardware implementation
	0x08	**TYPE_HSM**	Hardware Security Module
	0x0C	**TYPE_CPU**	Implemented inside of the main CPU
4-7	-	-	-

createProvisioningSession [2]

Input

Name	Type	Description
`Algorithm`	uri	Session creation algorithm URI. See next page
`PrivacyEnabled`	bool	If true the PEP (Privacy Enabled Provisioning) mode **must** be honored
`ServerSessionID`	id	Server-created provisioning ID which **should** be unique for each session
`ServerEphemeralKey`	byte[]	Server-created ephemeral ECDH key. See ServerEphemeralKey
`IssuerURI`	uri	URI associated with the issuer. See IssuerURI
`KeyManagementKey`	byte[]	Key management key or zero length array. See KeyManagementKey
`ClientTime`	int	Locally acquired time in UNIX "epoch" format in *seconds. See* ClientTime
`SessionLifeTime`	int	Validity of the provisioning session in seconds. See SessionLifeTime
`SessionKeyLimit`	short	Upper limit of **`SessionKey`** operations. See SessionKeyLimit

Output

Name	Type	Description
`Status`	byte	See Return Values
`ClientSessionID`	id	SKS-created provisioning ID which **must** be unique for each session
`ClientEphemeralKey`	byte[]	SKS-created ephemeral ECDH key. See ClientEphemeralKey
`Attestation`	byte[]	Session creation attestation signature
`ProvisioningHandle`	int	Non-zero local handle to created provisioning session

`createProvisioningSession` establishes a *persistent session key* that is only known by the issuer and the SKS for usage in subsequent provisioning steps. In addition, the SKS is *optionally* authenticated by the issuer.

Continued on the next page...

Shown below is the mandatory to support SKS session key creation algorithm:

http://xmlns.webpki.org/keygen2/1.0#algorithm.sks.s1

• Generate a for this SKS *unique* **`ClientSessionID`**
• Output **`ClientSessionID`**
• Generate an *ephemeral* ECDH key-pair **`EKP`** using *the same named curve* as **`ServerEphemeralKey`**
• Output **`ClientEphemeralKey`** = **`EKP.PublicKey`**
• Apply the SP800-56A ECC CDH primitive on **`EKP.PrivateKey`** and **`ServerEphemeralKey`** creating a shared secret **`Z`**
• Define internal variable: byte[32] **`SessionKey`**
• Set **`SessionKey`** = HMAC-SHA256 (**`Z`**, **`ClientSessionID`** || **`ServerSessionID`** || **`IssuerURI`** || Device ID) // KDF (Key Derivation Function)
• Output **`Attestation`** = Sign (Attestation Private Key, HMAC-SHA256 (**`SessionKey`**, **`Algorithm`** || **`PrivacyEnabled`** || **`ServerEphemeralKey`** || **`EKP.PublicKey`** || **`KeyManagementKey`** || **`ClientTime`** || **`SessionLifeTime`** || **`SessionKeyLimit`**)) // See Architecture
• Define internal variable: short **`MACSequenceCounter`** and set it to *zero*
• Store **`SessionKey`**, **`Algorithm`**, **`PrivacyEnabled`**, **`MACSequenceCounter`**, **`ClientSessionID`**, **`ServerSessionID`**, **`IssuerURI`**, **`KeyManagementKey`**, **`ClientTime`**, **`SessionLifeTime`** and **`SessionKeyLimit`** in the Credential Database and return a handle to the database entry in **`ProvisioningHandle`**
• Output **`ProvisioningHandle`**
 	
Note that individual elements featured in the *argument* (e.g. **`ClientSessionID`**) to the HMAC operations **must** be represented as described in Data Types.

Creation of a session key is an *atomic* operation.

Continued on the next page...

Remarks

If any succeeding operation in the same provisioning session, is regarded as incorrect by the SKS, *the session **must** be terminated and removed from internal storage including all associated data created in the session.*

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.

`Algorithm` defines the creation of **`SessionKey`** but also the integrity, confidentiality, and attestation mechanisms used during the provisioning session. See Session Security Mechanisms.

Using KeyGen2 **`IssuerURI`** is the URL which *invoked* the **`createProvisioningSession`** method.

`ServerEphemeralKey` and **`ClientEphemeralKey`** **must** be in X.509 DER format and **must** match the elliptic curve capabilities given by getDeviceInfo.

In the E2ES mode the *Sign* function **must** use 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.

In the Privacy Enabled Provisioning mode the *Sign* function **must** only transfer the result of the HMAC-SHA256 operation.

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

When **`ClientTime`** is transferred through a protocol such as KeyGen2 it **must** always as a *minimum* have seconds resolution otherwise serious interoperability issues will occur. Possible milliseconds **must** though be *truncated* during the HMAC calculation. **`ClientTime`** **should** be interpreted as a *32-bit unsigned integer* to cope with the Y2038 problem.

It is **recommended** setting **`SessionLifeTime`** as low as possible to enable efficient automatic “cleanup” of possible aborted provisioning sessions.

The **`SessionKeyLimit`** attribute **must** be large enough to handle all **`SessionKey`** related operations required during the rest of the provisioning session, otherwise the session **must** be terminated. See Session Security Mechanisms. Note that methods like importSymmetricKey and pp_deleteKey actually use *two* **`SessionKey`** operations.

A **`KeyManagementKey`** **must** be supplied if provisioned objects should be *updatable in a future session* (see pp_deleteKey, pp_unlockKey, pp_updateKey, and pp_cloneKeyProtection), else this item **must** be a zero-length array.

A **`KeyManagementKey`** **must** either be an RSA or an ECDSA public key in X.509 DER format, compatible with the SKS Algorithm Support.

Continued on the next page...

On the server side the following **should** be performed:

Server Response Validation
• Decide if **DeviceCertificatePath** of ProvisioningInitializationResponse is to be accepted/trusted. • Run the the same SP800-56A procedure and KDF as for the SKS but now using **ClientEphemeralKey** and the saved private key of **ServerEphemeralKey** to obtain **SessionKey** • *VerifySignature* (Device Certificate**.PublicKey**, // Received **Attestation**, // Received HMAC-SHA256 (**SessionKey**, **Algorithm** || // Saved **PrivacyEnabled** || // Saved **ServerEphemeralKey** || // Saved **ClientEphemeralKey** || // Received **KeyManagementKey** || // Saved **ClientTime** || // Received **SessionLifeTime** || // Saved **SessionKeyLimit**)) // Saved

If all tests above succeed the issuer server may continue with the actual provisioning process.

Note that in the Privacy Enabled Provisioning mode the **DeviceCertificatePath** does not apply, and *VerifySignature* is replaced by a straight comparison between **Attestation** and the output from the HMAC-SHA256 operation.

Continued on the next page...

When using KeyGen2 the *input* to **`createProvisioningSession`** is expressed as shown in the fragment below:

```
<ProvisioningInitializationRequest ID="_0fa47ab3c00c … a67992b6ac61c"
                                   SubmitURL="https://ca.example.com/enroll"
                                   SessionLifeTime="50000"
                                   SessionKeyLimit="25"
                                   Algorithm="http://xmlns.webpki.org/keygen2/1.0#algorithm.sks.s1"  …  >

   <ServerEphemeralKey>
      <ds11:ECKeyValue>
         <ds11:NamedCurve URI="urn:oid:1.2.840.10045.3.1.7"/>
         <ds11:PublicKey>JK/mSALhBpUjjPAe/ … fXG8z17eZV3mVDZTBM</ds11:PublicKey>
      </ds11:ECKeyValue>
   </ServerEphemeralKey>
   <KeyManagementKey>
      <ds:RSAKeyValue>
         <ds:Modulus>ALhBpUjJK/mSjPAe/ … fXG8z1V3mVDZTBM7eZ</ds:Modulus>
         <ds:Exponent>AQAB</ds:Exponent>
      </ds:RSAKeyValue>
   </KeyManagementKey>

</ProvisioningInitializationRequest>
```

Notes:

The KeyManagementKey element is *optional*.

If the PrivacyEnabled attribute is missing the E2ES mode is assumed.

The table below illustrates argument mapping:

KeyGen2 Element	SKS Counterpart
`ProvisioningInitializationRequest@ID`	**`ServerSessionID`**
`ProvisioningInitializationRequest@SubmitURL`	**`IssuerURI`**
`ProvisioningInitializationRequest@SessionLifeTime`	**`SessionLifeTime`**
`ProvisioningInitializationRequest@SessionKeyLimit`	**`SessionKeyLimit`**
`ProvisioningInitializationRequest@Algorithm`	**`Algorithm`**
`ProvisioningInitializationRequest@PrivacyEnabled`	**`PrivacyEnabled`**
`ServerEphemeralKey/ECKeyValue`	**`ServerEphemeralKey`**
`KeyManagementKey`	**`KeyManagementKey`**
N/A - Gathered by the local provisioning middleware	**`ClientTime`**

Continued on the next page...

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


```
<ProvisioningInitializationResponse ID="_126992b6 … a8a6b484db8f"
                                    ServerSessionID="_0fa47ab3c00c … a67992b6ac61c"
                                    ClientTime="2010-03-18T11:23:29Z"
                                    ServerCertificateFingerprint="AvDWx6xbg3GDGzdz … yJk8Js0OuI+Ba/Xc8="
                                    Attestation="Ob7MvaXC/rNx/rkNZJEo … 8lch/6sngIszfpElrggQfl" …  >

  <ClientEphemeralKey>
    <ds11:ECKeyValue>
      <ds11:NamedCurve URI="urn:oid:1.2.840.10045.3.1.7"/>
      <ds11:PublicKey>PRZre90SQLp … 16m9FokKxV3F40Y=</ds11:PublicKey>
    </ds11:ECKeyValue>
  </ClientEphemeralKey>
  <DeviceCertificatePath>
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAg ... hugc53W4nNzggt2w==</ds:X509Certificate>
    </ds:X509Data>
  </DeviceCertificatePath>
  <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#hmac-sha256"/>
      <ds:Reference URI="#_126992b6 … a8a6b484db8f">
        <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>yLD0zNA48Xt9xXNHuBUlK0hL51zn0SYj2IfDXm42PLc=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>aRiSdmrn/KgtjqtTReF+6DOuIemRuw2xV9yuOPAlMj8=</ds:SignatureValue>
    <ds:KeyInfo>
      <ds:KeyName>derived-session-key</ds:KeyName>
    </ds:KeyInfo>
  </ds:Signature>

</ProvisioningInitializationResponse>
```


The table below illustrates mapping:

KeyGen2 Element	SKS Counterpart
`ProvisioningInitializationResponse@ID`	**`ClientSessionID`**
`ProvisioningInitializationResponse@ServerSessionID`	**`Input.ServerSessionID`**
`ProvisioningInitializationResponse@ClientTime`	**`Input.ClientTime`**
`ProvisioningInitializationResponse@Attestation`	**`Attestation`**
`ClientEphemeralKey/ECKeyValue`	**`ClientEphemeralKey`**
`DeviceCertificatePath/X509Data/X509Certificate`	*Optional*: Certificate *path* from getDeviceInfo
`Signature/SignatureValue`	Created with signProvisioningSessionData

In the standard E2ES mode the **`DeviceCertificatePath`** **must** be available for verification of the **`Attestation`** signature as well as for identification of the SKS container. The **`Signature`** element holds a standard enveloped XML Signature. Also see Security Considerations.

closeProvisioningSession [3]

Input

Name	Type	Description
`ProvisioningHandle`	int	Local handle to an *open* provisioning session
`Nonce`	byte[]	Server generated 1-32 byte nonce value
`MAC`	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Description
`Status`	byte	See Return Values
`Attestation`	byte[32]	Session termination attestation signature. 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`**
- The **`MAC`** computes correctly using the method described in MAC Operations where *Data* is arranged as follows:

  Data = ClientSessionID || ServerSessionID || IssuerURI || **`Nonce`**
- All generated keys are fully provisioned which means that matching public key certificates have been deployed and checked regarding disallowed duplicates. See setCertificatePath
- EndorsedAlgorithm URIs match the provisioned key material with respect to symmetric or asymmetric operations as well as to length. Asymmetric keys are also tested for RSA and EC algorithm compliance
- There are no unreferenced PIN or PUK policy objects. See createPUKPolicy and createPINPolicy
- The post provisioning operations succeed during the final *commit*. See Transaction Based Operation

If verification is successful, **`closeProvisioningSession`** **must** also *reassign 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. Also see Provisioning Objects.

If verification fails, *all* objects created in the session **must** be deleted and post provisioning operations **must** be rolled back.

When a provisioning session has been successfully closed by this method, it remains stored until all associated keys have been deleted.

Using KeyGen2 **`closeProvisioningSession`** is invoked as the last step of processing ProvisioningFinalizationRequest where the top element holds the associated **`MAC`** and **`Nonce`** attributes.

The **`Attestation`** object is created by attesting the following *Data*:

Data = **`Nonce`** || **`ProvisioningHandle`**.Algorithm

Also see SessionKeyLimit.

A *successful* KeyGen2 response would typically look like:

```
<ProvisioningFinalizationResponse ID="_126992b6 … a8a6b484db8f"
                                  ServerSessionID="_0fa47ab3c00c … a67992b6ac61c"
                                  Attestation="gEWzCCD ... 0w81L/ls2W0UYNE="  …  />
```

enumerateProvisioningSessions [4]

Input

Name	Type	Description
ProvisioningHandle	int	Input enumeration handle
ProvisioningState	bool	If true list only *open* provisioning sessions. If false list only *closed* dittos

Output

Name	Type	Description
Status	byte	See Return Values
ProvisioningHandle	int	Output enumeration handle
*The following elements **must** be set to zero if the output* **ProvisioningHandle** = 0		
Algorithm	uri	See createProvisioningSession
PrivacyEnabled	bool	
KeyManagementKey	byte[]	
ClientTime	int	
SessionLifeTime	int	
ServerSessionID	id	
ClientSessionID	id	
IssuerURI	uri	

enumerateProvisioningSessions is primarily 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 may also be useful for debugging and “cleaning” purposes.

The input **ProvisioningHandle** **must** initially be set to 0 to start an enumeration round.

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

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

abortProvisioningSession [5]

Input

Name	Type	Description
ProvisioningHandle	int	Local handle to an *open* provisioning session

Output

Name	Type	Description
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 **must** be removed from the SKS, otherwise an error **must** be returned.

signProvisioningSessionData [6]

Input

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

Output

Name	Type	Description
`Status`	byte	See Return Values
`Signature`	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 scheme is as follows:

> **`Signature`** = HMAC-SHA256 (SessionKey || **`"External Signature"`**, **`Data`**)

Note that **`Data`** **must** be used “as is” in the HMAC operation, *excluding* length information.

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

Also see SessionKeyLimit.

createPUKPolicy [7]

Input

Name	Type	Description
ProvisioningHandle	int	Local handle to an *open* provisioning session
ID	id	*External name* of the PUK policy object. See Object IDs
PUKValue	byte[]	Encrypted PUK value. See Encrypted Data
Format	byte	Format of PUK strings. See PIN and PUK Formats
RetryLimit	short	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
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

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

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

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

Data = **ID** || **PUKValue** || **Format** || **RetryLimit**

Note that **PUKValue** is MACed in encrypted form and *then* decrypted by the SKS before storing.

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	Description
ProvisioningHandle	int	Local handle to an *open* provisioning session
ID	id	*External name* of the PIN policy object. See Object IDs
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	short	Non-zero value holding the number of incorrect PIN values (*in a sequence*), forcing a key to lock up
Grouping	byte	See PIN Grouping
PatternRestrictions	byte	See PIN Patterns
MinLength	short	Minimum *decoded* PIN length in bytes. See PIN and PUK Formats
MaxLength	short	Maximum *decoded* PIN length in bytes. See PIN and PUK Formats
InputMethod	byte	See PIN Input Methods
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

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

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

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

Data = **ID** || *PUKReference* || **UserDefined** || **UserModifiable** || **Format** || **RetryLimit** ||
Grouping || **PatternRestrictions** || **MinLength** || **MaxLength** || **InputMethod**

PUKReference is set to **"#N/A"** if **PUKPolicyHandle** is zero, else it is set to the **ID** of the referenced PUK policy object.

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.

createKeyEntry [9]

Input

Name	Type	Description
`ProvisioningHandle`	int	Local handle to an *open* provisioning session
`ID`	id	*External name* of the key. See Object IDs
`Algorithm`	uri	Key generation and attestation algorithm URI. See next page
`ServerSeed`	byte[]	Server input to the random number generation process. See ServerSeed
`DevicePINProtection`	bool	True if the key is to be protected by a *device PIN*. See Key Protection Objects
`PINPolicyHandle`	int	Handle to a governing PIN policy object or zero. See createPINPolicy
`PINValue`	byte[]	See PINValue, PIN Patterns and PIN Grouping
`EnablePINCaching`	bool	True if middleware **may** cache PINs for this key. See EnablePINCaching
`BiometricProtection`	byte	See Biometric Protection
`ExportProtection`	byte	See Key Export
`DeleteProtection`	byte	See Key Delete
`AppUsage`	byte	See Key Application Usage
`FriendlyName`	byte[]	String of 0-128 bytes that will be associated with this key for use in GUIs
`KeySpecifier`	byte[]	Algorithm and length of key to be created. See KeySpecifier
`EndorsedAlgorithms`	byte	Value [0..255] holding the number of **`EndorsedAlgorithm`** URIs
`EndorsedAlgorithm...`	uri	Endorsed algorithm URI *repeated* as defined by **`EndorsedAlgorithms`**
`MAC`	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Description
`Status`	byte	See Return Values
`KeyHandle`	int	Non-zero local handle to created key entry
`PublicKey`	byte[]	Generated public key in X.509 DER representation
`Attestation`	byte[32]	See Attestation

`createKeyEntry` generates an asymmetric key-pair according to the issuer's specification. In addition, **`createKeyEntry`** creates a *key entry* (see Key Entries) in the Credential Database where the key-pair and its protection attributes are stored.

Continued on the next page...

The following operations match the mandatory to support key generation and attestation algorithm:

http://xmlns.webpki.org/keygen2/1.0#algorithm.sks.k1

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

Data = **ID** || **Algorithm** || **ServerSeed** || *PINPolicyReference* || *PINValueReference* || **EnablePINCaching** || **BiometricProtection** || **ExportProtection** || **DeleteProtection** || **AppUsage** || **FriendlyName** || **KeySpecifier** [|| **EndorsedAlgorithm...**]

PINPolicyReference is set to **"#Device PIN"** if **DevicePINProtection** is true, to **"#N/A"** if **PINPolicyHandle** is zero, else it is set to the **ID** of the referenced PIN policy object.

PINValueReference is set to **"#N/A"** if **PINPolicyHandle** is zero, or if **DevicePINProtection** is true, or if the PIN is UserDefined, else it is set to the *encrypted* **PINValue**.

KeySpecifier denotes a blob encoded as follows depending on the requested key algorithm:

RSA	
byte	Type of key to be generated: 0x00 = RSA
short	RSA key size in bits. See getDeviceInfo
int	Zero (use default) or a defined exponent. See getDeviceInfo

ECDSA/ECDH	
byte	Type of key to be generated: 0x01 = EC
uri	Curve name URI where the URI is stored *without* length indicator. See Elliptic Curves

Attestation vouches for that generated key-pairs actually reside in the SKS by attesting (see Attestations) keys according to the following *Data* scheme:

Data = **ID** || **PublicKey**

Remarks

KeyHandle must be *static*, *unique* and *never be reused*. Note that a **KeyHandle** returned by **createKeyEntry must not** be featured in User API operations until the associated provisioning session has been closed (see closeProvisioningSession).

Object IDs for createKeyEntry, createPINPolicy and createPUKPolicy *share a common namespace* but the namespace is entirely local to the *provisioning session*. Although only static identifiers are used in the examples, Object IDs *may be randomized* to increase entropy of MAC Operations.

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

ServerSeed must be a 0-32 byte binary string holding a *random number seed*. How **ServerSeed** is applied to the random number generation process is *unspecified*. The only requirement is that it **must not** be able *reducing* the entropy.

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

Continued on the next page...

EndorsedAlgorithm URIs **must** be *sorted in ascending alphabetical order* before calling **createKeyEntry**.

EndorsedAlgorithm URIs **must** be checked for compatibility with Algorithm Support.

EndorsedAlgorithm compliance **must** be *enforced* by the User API.

EndorsedAlgorithm URIs **must not** be checked against actual key material during **createKeyEntry**. This check **must** be *deferred* to closeProvisioningSession.

If no **EndorsedAlgorithm** URIs are specified, *the key is only constrained by the key material*.

With the special algorithm **http://xmlns.webpki.org/keygen2/1.0#algorithm.none** (which is only permitted as a single **EndorsedAlgorithm** item), keys **must** be *disabled* from executing cryptographic operations through the User API.

A set **DevicePINProtection** presumes that the target SKS supports a "device PUK/PIN", otherwise an *error* **must be** returned. The characteristics of device PINs are out of scope for the SKS specification. See getDeviceInfo.

DevicePINProtection must not be combined with local PIN policy objects.

EnablePINCaching must only be used with keys protected by local PIN policy objects having the InputMethod set to **"trusted-gui"**.

PINValue objects **must** be set by the *caller* as illustrated by the following pseudo code:

```
if (PINPolicyHandle == 0)    // No PIN or device PIN
 {
   PINValue = zero length array;
 }
else if (PINPolicyHandle.UsedDefined)     // see UserDefined
 {
   PINValue = user-defined clear text PIN value;   // taken from a local provisioning GUI
 }
else
 {
   PINValue = encrypted issuer-set PIN value;    // see Encrypted Data
 }
```

Continued on the next page...

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

```
<KeyCreationRequest Algorithm="http://xmlns.webpki.org/keygen2/1.0#algorithm.sks.k1"  ...  >

   <PUKPolicy ID="PUK.1" Format="numeric" RetryLimit="3" Value="mjRuO …1Oqw" MAC="uPqE ... HIF=">
      <PINPolicy ID="PIN.1" Format="numeric" Grouping="shared" MaxLength="8" MinLength="4"
                    PatternRestrictions="three-in-a-row sequence" RetryLimit="3" MAC="gz56 ... 2B7=">
         <KeyEntry ID="Key.1" AppUsage="authentication" MAC="nFRuPgz ... H2B429S=">
            <EC NamedCurve="urn:oid:1.2.840.10045.3.1.7"/>
         </KeyEntry>
         <KeyEntry ID="Key.2" AppUsage="encryption" MAC="gSg4chh ... H2BEE7=">
            <RSA KeySize="2048"/>
         </KeyEntry>
      </PINPolicy>
   </PUKPolicy>

</KeyCreationRequest>
```

This sequence should be interpreted as a request for an EC key and an RSA key where both keys are protected by a single (shared) user-defined (within the specified policy limits) PIN. The PIN is in turn governed by an issuer-defined, *protocol wise secret* PUK. Also see KeyGen2 Proxy.

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 **`createKeyEntry`** is translated as shown in the fragment below:

```
<KeyCreationResponse  ...  >

   <PublicKey ID="Key.1" Attestation="X2oMtrm8rRL … XyTvPuTbergHfnJw==">
      <ds11:ECKeyValue>
         <ds11:NamedCurve URI="urn:oid:1.2.840.10045.3.1.7"/>
         <ds11:PublicKey>BMUY+QiZdFJyzozMgQqA … ptPGNy/LcDVSXx7s/Y=</ds11:PublicKey>
      </ds11:ECKeyValue>
   </PublicKey>
   <PublicKey ID="Key.2" Attestation="TbergHftrm8rRL … wyTvPX2XoMunJ==">
      <ds:RSAKeyValue>
         <ds:Modulus>ALhBpUjJK/mSjPAe/ … fXG8z1V3mVDZTBM7eZ</ds:Modulus>
         <ds:Exponent>AQAB</ds:Exponent>
      </ds:RSAKeyValue>
   </PublicKey>

</KeyCreationResponse>
```

A conforming server **must** after receival of the response verify that the number and IDs of returned keys match the request. In addition, each returned key **must** be checked for correctness regarding attestation data and that the generated public key actually complies with that of the request.

getKeyHandle [10]

Input

Name	Type	Description
`ProvisioningHandle`	int	Local handle to an *open* provisioning session
`ID`	id	See createKeyEntry

Output

Name	Type	Description
`Status`	byte	See Return Values
`KeyHandle`	int	Local handle to a key belonging to an *open* provisioning session

`getKeyHandle` returns a **`KeyHandle`** based on the provisioning session specific key ID.

An invalid key **must** return an error and abort the provisioning session.

setCertificatePath [11]

Input

Name	Type	Description
KeyHandle	int	Local handle to a key-pair belonging to an *open* provisioning session
PathLength	byte	Non-zero value holding the number of **X509Certificate** objects in the call
X509Certificate...	byte[]	DER encoded X.509 certificate object *repeated* as defined by **PathLength**
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Description
Status	byte	See Return Values

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

The **X509Certificate** objects **must** form an *ordered* and *contiguous* certificate path so that the *first* object contains the End-Entity Certificate *usually* holding the public key of the target key-pair. The path does though not have to be complete (include all upper-level CAs). Path validity **should** be verified by the provisioning middleware before calling this method.

Individual **X509Certificate** objects **must not** exceed CryptoDataSize.

Note that an SKS **must not** not attempt to verify that the End-Entity Certificate and **KeyHandle.**PublicKey match because that would disable the restorePrivateKey method. It is the **MAC** operation that is facilitating a cryptographically verifiable binding between the certificate path and the designated key entry.

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

Data = **KeyHandle.**PublicKey || **KeyHandle.**ID || **X509Certificate...**

A compliant SKS **must not** accept multiple key entries being associated by the same End-Entity Certificate unless the conflicting key is subject to a pp_updateKey or pp_deleteKey operation.

The provisioning middleware **should** verify that the public key of the End-Entity Certificate matches the algorithmic capabilities of the SKS. See Algorithm Support.

Continued on the next page...

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

```
<ProvisioningFinalizationRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                                 ID="_0fa47ab3c00c … a67992b6ac61c"
                                 MAC="GcaqnRxW ... 8kabXDgWr=" Nonce="aqnPb6x0 ... kms34gfW2="  ... >

  <CertficatePath ID="Key.1" MAC="ngSgmRuPE ... HIFWrM421wY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
  </CertficatePath>

</ProvisioningFinalizationRequest>
```

The table below illustrates argument mapping:

KeyGen2 Element	SKS Counterpart
`CertficatePath@ID`	**`KeyHandle.ID`**
`CertficatePath@MAC`	**`MAC`**
`CertficatePath/X509Data/X509Certificate...`	**`X509Certificate...`**

The owning **`ProvisioningHandle`** and local **`KeyHandle`** can be retrieved by calling enumerateProvisioningSessions and getKeyHandle respectively. The request attributes **`ProvisioningFinalizationRequest@ClientSessionID`** and **`ProvisioningFinalizationRequest@ID`** hold the ClientSessionID and ServerSessionID of the provisioning session.

importSymmetricKey [12]

Input

Name	Type	Description
KeyHandle	int	Local handle to a key belonging to an *open* provisioning session
SymmetricKey	byte[]	“Piggybacked” symmetric key encrypted as described in Encrypted Data
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Description
Status	byte	See Return Values

importSymmetricKey imports and links a symmetric key 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 || **SymmetricKey**

Note that **SymmetricKey** objects **must be** MACed in *encrypted form* and *then* decrypted by the SKS before storing.

Symmetric keys **must not** exceed 128 bytes.

With the special EndorsedAlgorithm **http://xmlns.webpki.org/keygen2/1.0#algorithm.none** arbitrary static shared secrets can be specified. When used together with exportKey, a suitable PIN policy and a PropertyBag object holding site information, an SKS could then also serve as a *browser password store*.

After **importSymmetricKey** has been called the key entry is marked as “symmetric”. That is, *the private key is disabled* as well as all operations associated with it. See getKeyAttributes.

The KeyBackup.**SERVER** flag of the key **must** be set after execution of **importSymmetricKey**.

Continued on the next page...

The following KeyGen2 fragments show the "piggyback" arrangement. First the server issues a key-pair request:

```
<KeyCreationRequest Algorithm="http://xmlns.webpki.org/keygen2/1.0#algorithm.sks.k1" ... >

  <PUKPolicy ID="PUK.1" Format="numeric" RetryLimit="3" Value="mjRuO …1Oqw" MAC="uPqE ... HIF=">
    <PINPolicy ID="PIN.1" Format="numeric" MaxLength="8" MinLength="4"
               PatternRestrictions="three-in-a-row sequence" RetryLimit="3" MAC="gz56 ... 2B7=">
      <KeyEntry ID="Key.1" AppUsage="authentication"
                EndorsedAlgorithms="http://www.w3.org/2000/09/xmldsig#hmac-sha1"
                MAC="gSg4chh ... H2BEE7=">
        <RSA KeySize="1024"/>
      </KeyEntry>
    </PINPolicy>
  </PUKPolicy>

</KeyCreationRequest>
```

The request above is identical to requests for PKI except for the *optional* EndorsedAlgorithm declaration which in the sample limited symmetric key operations to HMAC-SHA1.

After the request the client generates a compatible key-pair response which is *identical* to that of PKI:

```
<KeyCreationResponse ... >

  <PublicKey ID="Key.1" Attestation="TbergHftrm8rRL … wyTvPX2XoMunJ==">
    <ds:RSAKeyValue>
      <ds:Modulus>ALhBpUjJK/mSjPAe/ … fXG8z1V3mVDZTBM7eZ</ds:Modulus>
      <ds:Exponent>AQAB</ds:Exponent>
    </ds:RSAKeyValue>
  </PublicKey>

</KeyCreationResponse>
```

The server then responds with the usual certificated public key plus an encrypted "piggybacked" symmetric key:

```
<ProvisioningFinalizationRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                                 ID="_0fa47ab3c00c … a67992b6ac61c"
                                 MAC="GcaqnRxW ... 8kabXDgWr=" Nonce="aqnPb6x0 ... kms34gfW2=" ... >

  <CertficatePath ID="Key.1" MAC="ngSgmRuPE ... HIFWrM421wY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <SymmetricKey MAC="je7KiznTIl … vInumZCjxSl1=">vInt09Esmg94v ... YU3tgIdhcNNby</SymmetricKey>
  </CertficatePath>

</ProvisioningFinalizationRequest>
```

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

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

addExtension [13]

Input

Name	Type	Description
KeyHandle	int	Local handle to a key belonging to an *open* provisioning session
Type	uri	Type URI. Holds a unique name identifying the extension type
SubType	byte	See table below
Qualifier	byte[]	See table below
ExtensionData	blob	Extension object. Regarding size constraints see getDeviceInfo
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Description
Status	byte	See Return Values

addExtension 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 || **Type** || **SubType** || **Qualifier** || **ExtensionData**

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

KeyGen2 Element	SubType	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	See PropertyBag canonicalization
Logotype	**0x03**	**MIMEType**	Binary image data extracted from Base64 encoded XML

Remarks

N/A = zero-length array.

Note the handling of the **EncryptedExtension**: **ExtensionData** which is encrypted as described in Encrypted Data **must** be MACed in *encrypted form* and *then* decrypted by the SKS before storing.

A compliant SKS **must not** allow a given key to be associated with multiple extensions of the same **Type**. *If multiple objects of the same type are needed, you must define a container type holding these*.

Type URIs *do not have to be recognized by the SKS*, since they are intended for interpretation by external applications.

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 extension **Type** URI **must** be featured in the SKS *supported algorithm list*. See getDeviceInfo and getExtension.

Qualifier objects **must not** exceed 128 bytes.

Continued on the next page...

The *simplified* XML schema extract below describes the KeyGen2 representation of **`PropertyBag`** objects:

PropertyBag XML Schema

```
<xs:element name="PropertyBag">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Property" maxOccurs="unbounded">
        <xs:complexType>

          <!-- The unique name of the property -->
          <xs:attribute name="Name" use="required"/>
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:minLength value="1"/>
                <xs:maxLength value="100"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>

          <!-- The value of the property -->
          <xs:attribute name="Value" type="xs:string" use="required"/>

          <!-- By default values are read-only but they may be  declared as read/writable as well -->
          <xs:attribute name="Writable" type="xs:boolean" use="optional"/>

        </xs:complexType>
      </xs:element>
    </xs:sequence>

    <!-- Extension type  -->
    <xs:attribute name="Type" type="xs:anyURI" use="required"/>

    <!-- MAC (Message Authentication Code) -->
    <xs:attribute name="MAC" type="xs:base64Binary" use="required"/>

  </xs:complexType>
</xs:element>
```

A **`PropertyBag`** **must** be converted to a *binary blob* before storage in SKS and MACing according to the following:

- Each **`Property`** is translated into a composite object consisting of the following attributes and transformed representation:

Name	Writable	Value
byte[]	bool	byte[]

  See Data Types

- The resulting **`Property`** objects are *concatenated* in the order they occur in the **`PropertyBag`**

Note that there are no delimiters added between attributes or objects. The assembled blob holds the actual ExtensionData.

Enforcement of **`Property`** name uniqueness **may** be delegated to the middleware layer. Also see setProperty.

Continued on the next page...

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

```
<ProvisioningFinalizationRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                                 ID="_0fa47ab3c00c … a67992b6ac61c"
                                 MAC="GcaqnRxW ... 8kabXDgWr=" Nonce="aqnPb6x0 ... kms34gfW2="  ...  >

  <CertficatePath ID="Key.1" MAC="ngSgmRuPE ... HlFWrM421wY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <Extension Type="http://schemas.xmlsoap.org/ws/2005/05/identity"
               MAC="UMRljeXFd … CrcqcGkI1=">IiBlbmHVy85cZS ...  B4bWxuczd3dy53My5vc</Extension>
  </CertficatePath>

</ProvisioningFinalizationRequest>
```

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

In the Information Card sample the primary authentication key (for authenticating to the IDP), would preferable be the PKI key associated by **`CertificatePath`**. That is, the issuance of a managed Information Card and its primary key *can be fully synchronized* making both usage and middleware design straightforward.

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

```
<ProvisioningFinalizationRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                                 ID="_0fa47ab3c00c … a67992b6ac61c"
                                 MAC="GcaqnRxW ... 8kabXDgWr=" Nonce="aqnPb6x0 ... kms34gfW2="  ...  >

  <CertficatePath ID="Key.1" MAC="ngSgmRuPE ... HlFWrM421wY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <SymmetricKey MAC="je7KiznTlI … vlnu7rcqcGkI=">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="+crSq5fvfx+f … ZmRnhxlj0d=">iAAABKCAIAAACD … tm/AAALjUlEQVRA=</Logotype>
  </CertficatePath>

</ProvisioningFinalizationRequest>
```

A HOTP application would preferably also make the corresponding KeyCreationRequest operation include an endorsement algorithm definition.

restorePrivateKey [14]

Input

Name	Type	Description
KeyHandle	int	Local handle to a key belonging to an *open* provisioning session
PrivateKey	byte[]	Private key in PKCS #8 format wrapped as described in Encrypted Data
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Description
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 || **PrivateKey**

Note that **PrivateKey** objects **must** be MACed in *encrypted form* and *then* decrypted by the SKS before storing.

The purpose of **restorePrivateKey** (preceded by setCertificatePath), is to install a certificate and private key that the issuer have generated or have a backup of.

The KeyBackup**.SERVER** flag of the key **must** be set after execution of **restorePrivateKey**.

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

```
<ProvisioningFinalizationRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                                 ID="_0fa47ab3c00c … a67992b6ac61c"
                                 MAC="GcaqnRxW ... 8kabXDgWr=" Nonce="aqnPb6x0 ... kms34gfW2="  ...  >

  <CertficatePath ID="Key.1" MAC="ngSgmRuPE ... HlFWrM421wY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <PrivateKey MAC="umZCjxSl1zX … CrcqcGkIv4Ki=">cVyZSIiBlbmH ...  uczpkcB4bWx3dy53</PrivateKey>
  </CertficatePath>

</ProvisioningFinalizationRequest>
```

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

If **restorePrivateKey** is executed over a networked protocol such as KeyGen2 (rather than locally), it is **recommended** alerting the user unless the key is having AppUsage = **encryption**

pp_deleteKey [50]

Input

<table>
<tr><th>Name</th><th>Type</th><th colspan="2">Description</th></tr>
<tr><td>ProvisioningHandle</td><td>int</td><td colspan="2">Local handle to an open provisioning session</td></tr>
<tr><td>TargetKeyHandle</td><td>int</td><td>Local handle to the target key</td><td rowspan="2">See Target Key Reference</td></tr>
<tr><td>Authorization</td><td>byte[]</td><td>Key management authorization signature</td></tr>
<tr><td>MAC</td><td>byte[32]</td><td colspan="2">Vouches for the integrity and authenticity of the operation</td></tr>
</table>

Output

Name	Type	Description
Status	byte	See Return Values

pp_deleteKey deletes a key created in an earlier provisioning session.

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

Data = **Authorization**

A conforming SKS **must** abort the provisioning session if **pp_deleteKey** is mixed with other post provisioning operations referring to the same **TargetKeyHandle**.

Note that the *execution* of this method **must** be *deferred* to closeProvisioningSession.

Regarding delete of PIN and PUK policy objects, see Key Protection Objects.

Continued on the next page...

The following fragment shows how **pp_deleteKey** operations have been integrated in the KeyGen2 protocol:

```
<ProvisioningFinalizationRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                                 ID="_0fa47ab3c00c … a67992b6ac61c"
                                 MAC="GcaqnRxW ... 8kabXDgWr=" Nonce="aqnPb6x0 ... kms34gfW2="  ...  >

   <CertficatePath … >
   </CertficatePath>

   <DeleteKey CertificateFingerprint="mNWEOLIAZrG ... dUg2aLA="
              ClientSessionID="_1292389cba4 ... 173d4334"
              ServerSessionID="_1292389cb8 ... c2ea36e"
              Authorization="ChPQRDT ... n/dJKrW3L"
              MAC="AvDWx6xbg3GDGzdz … yJk8Js0OuI+Ba/Xc8="/>

</ProvisioningFinalizationRequest>
```

Before invoking **pp_deleteKey** the provisioning middleware needs to perform a number of steps:

1. Find the the *old* provisioning session associated with the **ClientSessionID** and **ServerSessionID** attributes of the **DeleteKey** element by calling enumerateProvisioningSessions.
2. Find possible keys by calling enumerateKeys and ignoring all but those belonging to the provisioning session found in step #1.
3. For the set of keys found in step #2 call getKeyAttributes while looking for a key having an End-Entity Certificate matching the SHA256 **CertificateFingerprint**.
4. If step #3 is successful **TargetKeyHandle** is recovered and **pp_deleteKey** can be invoked.

If any of these steps fail the provisioning session **must** be aborted. Also see Remote Key Lookup.

pp_unlockKey [51]

Input

Name	Type	Description	
ProvisioningHandle	int	Local handle to an *open* provisioning session	
TargetKeyHandle	int	Local handle to the target key	See Target Key Reference
Authorization	byte[]	Key management authorization signature	
MAC	byte[32]	Vouches for the integrity and authenticity of the operation	

Output

Name	Type	Description
Status	byte	See Return Values

pp_unlockKey works like unlockKey except that authorization is derived from a Target Key Reference instead of a PUK.

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

Data = **Authorization**

Note that the *execution* of this method **must** be *deferred* to closeProvisioningSession and that it **must** (internally) be performed in *advance* of any pp_updateKey or pp_cloneKeyProtection calls referring to the same **TargetKeyHandle**.

If the target key is associated with a PUK object the PUK error count **must** be cleared as well.

The following fragment shows how **pp_unlockKey** operations have been integrated in the KeyGen2 protocol:

```
<ProvisioningFinalizationRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                                 ID="_0fa47ab3c00c … a67992b6ac61c"
                                 MAC="GcaqnRxW ... 8kabXDgWr=" Nonce="aqnPb6x0 ... kms34gfW2=" ... >

  <CertficatePath … >
  </CertficatePath>

  <UnlockKey CertificateFingerprint="mNWEOLIAZrG ... dUg2aLA="
             ClientSessionID="_1292389cba4 ... 173d4334"
             ServerSessionID="_1292389cb8 ... c2ea36e"
             Authorization="ChPQRDT ... n/dJKrW3L"
             MAC="AvDWx6xbg3GDGzdz … yJk8Js0OuI+Ba/Xc8="/>

</ProvisioningFinalizationRequest>
```

Before invoking **pp_unlockKey** the provisioning middleware needs to perform the same steps as for pp_deleteKey.

pp_updateKey [52]

Input

Name	Type	Description	
KeyHandle	int	Local handle to a *new* key belonging to an *open* provisioning session	
TargetKeyHandle	int	Local handle to the target key	See Target Key Reference
Authorization	byte[]	Key management authorization signature	
MAC	byte[32]	Vouches for the integrity and authenticity of the operation	

Output

Name	Type	Description
Status	byte	See Return Values

pp_updateKey updates (replaces) a key created in an earlier provisioning session.

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

Data = End-Entity Certificate || **Authorization**

The new key **must** be *fully provisioned* (fitted with a certificate and optional attributes), *before* this method is called. However, the new key **must not** be PIN-protected since it supposed to *inherit* the old key's PIN protection scheme (if there is one). Inheritance does not mean "copying" but *linking* the new key to an existing PIN object. See Key Protection Objects.

The target key and and the new key **must** have identical Key Application Usage.

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

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

A conforming SKS **must** allow a (single) **pp_updateKey** combined with an arbitrary number of pp_cloneKeyProtection calls referring to the same **TargetKeyHandle**.

Note that the *execution* of this method **must** be *deferred* to closeProvisioningSession.

The following fragment shows how **pp_updateKey** has been integrated in the KeyGen2 protocol:

```
<ProvisioningFinalizationRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                                 ID="_0fa47ab3c00c … a67992b6ac61c"
                                 MAC="GcaqnRxW ... 8kabXDgWr=" Nonce="aqnPb6x0 ... kms34gfW2="  …  >

  <CertficatePath ID="Key.1" MAC="ngSgmRuPE ... HIFWrM421wY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <UpdateKey CertificateFingerprint="mNWEOLIAZrG ... dUg2aLA="
               ClientSessionID="_1292389cba4 ... 173d4334"
               ServerSessionID="_1292389cb8 ... c2ea36e"
               Authorization="ChPQRDT ... n/dJKrW3L"
               MAC="AvDWx6xbg3GDGzdz … yJk8Js0OuI+Ba/Xc8="/>
  </CertficatePath>

</ProvisioningFinalizationRequest>
```

Before invoking **pp_updateKey** the provisioning middleware needs to perform the same steps as for pp_deleteKey.

KeyHandle is the handle associated with **CertficatePath**.

pp_cloneKeyProtection [53]

Input

Name	Type	Description	
KeyHandle	int	Local handle to a *new* key belonging to an *open* provisioning session	
TargetKeyHandle	int	Local handle to the target key	See Target Key Reference
Authorization	byte[]	Key management authorization signature	
MAC	byte[32]	Vouches for the integrity and authenticity of the operation	

Output

Name	Type	Description
Status	byte	See Return Values

pp_cloneKeyProtection clones the *protection scheme* of a key created in an earlier provisioning session and applies it to a newly created key.

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

Data = End-Entity Certificate || **Authorization**

The new key **must** be *fully provisioned* (fitted with a certificate and optional attributes), *before* this method is called. However, the new key **must not** be PIN-protected since it supposed to *inherit* the old key's PIN protection scheme (if there is one). Inheritance does not mean “copying” but *linking* the new key to an existing PIN object. See Key Protection Objects.

The target key and and the new key **must** have identical Key Application Usage.

Note that an SKS **must not** allow inheritance of a custom PIN protection scheme with grouping **none** (see PIN Grouping).

A conforming SKS **must** allow multiple **pp_cloneKeyProtection** calls referring to the same **TargetKeyHandle**.

Note that the *execution* of this method **must** be *deferred* to closeProvisioningSession.

The following fragment shows how **pp_cloneKeyProtection** has been integrated in the KeyGen2 protocol:

```
<ProvisioningFinalizationRequest ClientSessionID="_126992b6 … a8a6b484db8f"
                                 ID="_0fa47ab3c00c … a67992b6ac61c"
                                 MAC="GcaqnRxW ... 8kabXDgWr=" Nonce="aqnPb6x0 ... kms34gfW2=" … >

  <CertficatePath ID="Key.1" MAC="ngSgmRuPE ... HIFWrM421wY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <CloneKeyProtection CertificateFingerprint="mNWEOLIAZrG ... dUg2aLA="
                        ClientSessionID="_1292389cba4 ... 173d4334"
                        ServerSessionID="_1292389cb8 ... c2ea36e"
                        Authorization="ChPQRDT ... n/dJKrW3L"
                        MAC="AvDWx6xbg3GDGzdz … yJk8Js0OuI+Ba/Xc8="/>
  </CertficatePath>

</ProvisioningFinalizationRequest>
```

Before invoking **pp_cloneKeyProtection** the provisioning middleware needs to perform the same steps as for pp_deleteKey.

KeyHandle is the handle associated with **CertficatePath**.

enumerateKeys [70]

Input

Name	Type	Description
`KeyHandle`	int	Input enumeration handle

Output

Name	Type	Description
`Status`	byte	See Return Values
`KeyHandle`	int	Output enumeration handle
*The following element **must** be set to zero if the output* **`KeyHandle`** = 0		
`ProvisioningHandle`	int	Handle to the associated provisioning session object

`enumerateKeys` enumerate keys for *closed* provisioning sessions. Closed provisioning session means that the key is ready for usage by *applications*.

The input **`KeyHandle`** **must** initially be set to 0 to start an enumeration round.

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

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

getKeyAttributes [71]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key

Output

<table>
<tr><th>Name</th><th>Type</th><th colspan="2">Description</th></tr>
<tr><td>Status</td><td>byte</td><td colspan="2">See Return Values</td></tr>
<tr><td>IsSymmetricKey</td><td>bool</td><td colspan="2">True if the active key is symmetric. See importSymmetricKey</td></tr>
<tr><td>PathLength</td><td>byte</td><td colspan="2" rowspan="2">See setCertificatePath</td></tr>
<tr><td>X509Certificate...</td><td>byte[]</td></tr>
<tr><td>AppUsage</td><td>byte</td><td colspan="2" rowspan="4">See createKeyEntry</td></tr>
<tr><td>FriendlyName</td><td>byte[]</td></tr>
<tr><td>EndorsedAlgorithms</td><td>byte</td></tr>
<tr><td>EndorsedAlgorithm...</td><td>uri</td></tr>
<tr><td>Extensions</td><td>short</td><td>Number of Type URIs</td><td rowspan="2">See addExtension</td></tr>
<tr><td>Type...</td><td>uri</td><td>Extension Type URI. Repeated object</td></tr>
</table>

getKeyAttributes returns attribute data for provisioned keys.

For asymmetric keys the public key of the End-Entity Certificate signifies RSA or EC algorithm.

Also see getKeyProtectionInfo.

getKeyProtectionInfo [72]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key

Output

Name	Type	Description
Status	byte	See Return Values
ProtectionStatus	byte	See ProtectionStatus table on the next page
PUKFormat	byte	Copy of **Format** defined by createPUKPolicy [1]
PUKRetryLimit	short	Copy of **RetryLimit** defined by createPUKPolicy [1]
PUKErrorCount	short	Current PUK error count for keys protected by a local PUK policy object [1]
UserDefined	bool	Copies of the corresponding createPINPolicy parameters for keys protected by a local PIN policy object [1]
UserModifiable	bool	
Format	byte	
RetryLimit	short	
Grouping	byte	
PatternRestrictions	byte	
MinLength	short	
MaxLength	short	
InputMethod	byte	
PINErrorCount	short	Current PIN error count for keys protected by a local PIN policy object [1] See ProtectionStatus table on the next page
EnablePINCaching	bool	Exact copies of the corresponding createKeyEntry parameters
BiometricProtection	byte	
ExportProtection	byte	
DeleteProtection	byte	
KeyBackup	byte	Tells if there exists a *copy* of the key. See KeyBackup table on the next page

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

Note 1: Fields **must** be set to zero if they do not apply to the key in question.

Continued on the next page...

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

Name	Value	Description
PIN_PROTECTED	0x01	The key is protected by a local PIN policy object
PUK_PROTECTED	0x02	The key is protected by a local PUK policy object. This bit **must** be *combined* with bit **PIN_PROTECTED**
PIN_BLOCKED	0x04	The key has locked-up due to PIN errors. This bit **must** be *combined* with bit **PIN_PROTECTED**
PUK_BLOCKED	0x08	The key has locked-up due to PUK errors. This bit **must** be *combined* with bit **PUK_PROTECTED**
DEVICE_PIN	0x10	The key is protected by a device PIN. Information about device PINs is out of scope for the SKS API. This bit **must** be the only active bit if applicable

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

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

Name	Value	Description
SERVER	0x01	The SERVER bit **must** be set if the key has been supplied through restorePrivateKey or importSymmetricKey
LOCAL	0x02	The LOCAL bit **must** be set if the key has been subject to an exportKey operation

getExtension [73]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key
Type	uri	Type URI. See addExtension

Output

Name	Type	Description
Status	byte	See Return Values
SubType	byte	Exact copies of the corresponding addExtension parameters
Qualifier	byte[]	
ExtensionData	blob	

getExtension returns a typed 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** **must** be returned as a zero-length array.

setProperty [74]

Input

Name	Type	Description
`KeyHandle`	int	Local handle to the target key
`Type`	uri	Type URI which **must** identify a **`PropertyBag`** extension. See addExtension
`Name`	byte[]	**`Property`** name
`Value`	byte[]	**`Property`** value

Output

Name	Type	Description
`Status`	byte	See Return Values

`setProperty` sets a **`Property`** in a **`PropertyBag`** linked to a key.

If the named **`Property`** does not exist or is not *writable*, an error **must** be returned.

deleteKey [80]

Input

Name	Type	Description
`KeyHandle`	int	Local handle to the target key
`Authorization`	byte[]	Zero-length array, PIN, or PUK depending on Key Delete

Output

Name	Type	Description
`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.

Invalid **`Authorization`** data to the key **must** return ERROR_AUTHORIZATION status.

A conforming SKS **may** introduce physical presence methods like GPIO-based buttons, *circumventing* key DeleteProtection settings.

Regarding delete of PIN and PUK policy objects, see Key Protection Objects.

exportKey [81]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key
Authorization	byte[]	Zero-length array, PIN, or PUK depending on Key Export

Output

Name	Type	Description
Status	byte	See Return Values
Key	byte[]	Unencrypted key. For type information see getKeyAttributes

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

Invalid **Authorization** data to the key **must** return ERROR_AUTHORIZATION status.

Private (asymmetric) keys **must** be exported in PKCS #8 format.

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

Note that the KeyBackup.**LOCAL** flag of the key **must** be set after execution of **exportKey**.

unlockKey [82]

Input

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

Output

Name	Type	Description
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.

Invalid **Authorization** data (PUK) to the key **must** return ERROR_AUTHORIZATION status.

If **unlockKey** succeeds all keys sharing the PIN object will be unlocked. See PIN Grouping.

changePIN [83]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key
Authorization	byte[]	Original PIN
NewPIN	byte[]	The requested new PIN

Output

Name	Type	Description
Status	byte	See Return Values

changePIN modifies a PIN for a key.

Note that the key **must** be protected by a local PIN policy object having the UserModifiable attribute set.

Invalid **Authorization** data (PIN) to the key **must** return ERROR_AUTHORIZATION status.

If **changePIN** succeeds all keys sharing the PIN object will be updated. See PIN Grouping.

setPIN [84]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key
Authorization	byte[]	PUK string
NewPIN	byte[]	The requested new PIN

Output

Name	Type	Description
Status	byte	See Return Values

setPIN sets a PIN for a key *regardless of PIN block status* since it uses a PUK as authorization.

Note that the key **must** be protected by local PUK and PIN policy objects where the latter have the UserModifiable attribute set.

Invalid **Authorization** data (PUK) **must** return ERROR_AUTHORIZATION status.

If **setPIN** succeeds all keys sharing the PIN object will be updated and *unlocked*. See PIN Grouping.

signHashedData [100]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key
Algorithm	uri	Signature algorithm URI. See Asymmetric Key Signatures
Parameters	byte[]	Parameters needed by some signature algorithms
Authorization	byte[]	Holds a PIN or is of zero length if no PIN is supplied
Data	byte[]	Hashed data to be signed. Also see CryptoDataSize

Output

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

signHashedData performs an asymmetric key signature operation on the input **Data** object.

Data must be hashed *as required by the signature algorithm*.

The **Parameters** object **must** be of zero length for signature algorithms not needing additional input.

Invalid **Authorization** data (PIN) to the key **must** return ERROR_AUTHORIZATION status.

The length of **Data must** match the hash algorithm. Note that signature algorithms that do not define a specific hash algorithm impose no tests on **Data** length. The **http://xmlns.webpki.org/keygen2/1.0#algorithm.rsa.none** signature algorithm **must** format the signature packet according to PKCS #1 but without hash algorithm identifiers:

EMSA = **0x00** || **0x01** || **PS** || **0x00** || **Data**

asymmetricKeyDecrypt [101]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key
Algorithm	uri	Encryption algorithm URI. See Asymmetric Key Encryption
Parameters	byte[]	Parameters needed by some encryption algorithms
Authorization	byte[]	Holds a PIN or is of zero length if no PIN is supplied
Data	byte[]	Encrypted data

Output

Name	Type	Description
Status	byte	See Return Values
Result	byte[]	Decrypted data

asymmetricKeyDecrypt performs an asymmetric key decryption operation on the input **Data** object.

Data must be padded *as required by the encryption algorithm* like PKCS #1 for **http://www.w3.org/2001/04/xmlenc#rsa-1_5**.

The **Parameters** object **must** be of zero length for encryption algorithms not needing additional input.

Invalid **Authorization** data (PIN) to the key **must** return ERROR_AUTHORIZATION status.

keyAgreement [102]

Input

Name	Type	Description
`KeyHandle`	int	Local handle to the target key
`Algorithm`	uri	Key agreement algorithm URI. See Diffie-Hellman Key Agreement
`Parameters`	byte[]	Parameters needed by some key agreement algorithms
`Authorization`	byte[]	Holds a PIN or is of zero length if no PIN is supplied
`PublicKey`	byte[]	The other party's public key

Output

Name	Type	Description
`Status`	byte	See Return Values
`Result`	byte[]	Shared secret

`keyAgreement` performs an asymmetric key agreement operation resulting in a shared secret.

`PublicKey` **must** be an EC public key in X.509 DER format using the same curve as **`KeyHandle`**. See Elliptic Curves.

The **`Parameters`** object **must** be of zero length for key agreement algorithms not needing additional input.

Invalid **`Authorization`** data (PIN) to the key **must** return ERROR_AUTHORIZATION status.

performHMAC [103]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key
Algorithm	uri	HMAC algorithm URI. See HMAC Operations
Authorization	byte[]	Holds a PIN or is of zero length if no PIN is supplied
Data	blob	Data to be HMACed. Also see CryptoDataSize

Output

Name	Type	Description
Status	byte	See Return Values
Result	byte[]	HMACed data

performHMAC performs a symmetric key HMAC operation on the input **Data** object.

Invalid **Authorization** data (PIN) to the key **must** return ERROR_AUTHORIZATION status.

symmetricKeyEncrypt [104]

Input

Name	Type	Description
KeyHandle	int	Local handle to the target key
Algorithm	uri	Encryption algorithm URI. See Symmetric Key Encryption
Mode	bool	True for encryption, false for decryption
IV	byte[]	Initialization Vector. Zero length for the XML Encryption algorithms
Authorization	byte[]	Holds a PIN or is of zero length if no PIN is supplied
Data	blob	Data to be encrypted or decrypted. Also see CryptoDataSize

Output

Name	Type	Description
Status	byte	See Return Values
Result	blob	Encrypted or decrypted data

symmetricKeyEncrypt performs a symmetric key encryption or decryption operation on the input **Data** object.

Note that if an **IV** (Initialization Vector) is not required by the encryption algorithm, **IV must** be of zero length.

Invalid **Authorization** data (PIN) to the key **must** return ERROR_AUTHORIZATION status.

updateFirmware [110]

Input

Name	Type	Description
Chunk	blob	Firmware code chunk

Output

Name	Type	Description
Status	byte	See Return Values
NextURL	uri	Next URL or zero-length string

updateFirmware is an *optional* method that performs a firmware update operation. The method is only available if the UpdateURL is non-zero. To perform an update, the SKS management system issues an HTTP GET operation to the service pointed out by UpdateURL. If the service returns a content of zero length, the SKS device is assumed to be up-to-date, else **updateFirmware** should be called with the content in **Chunk**. The return value from the call is either a new URL to be used analogous to UpdateURL, or a zero-length string indicating that the update is ready.

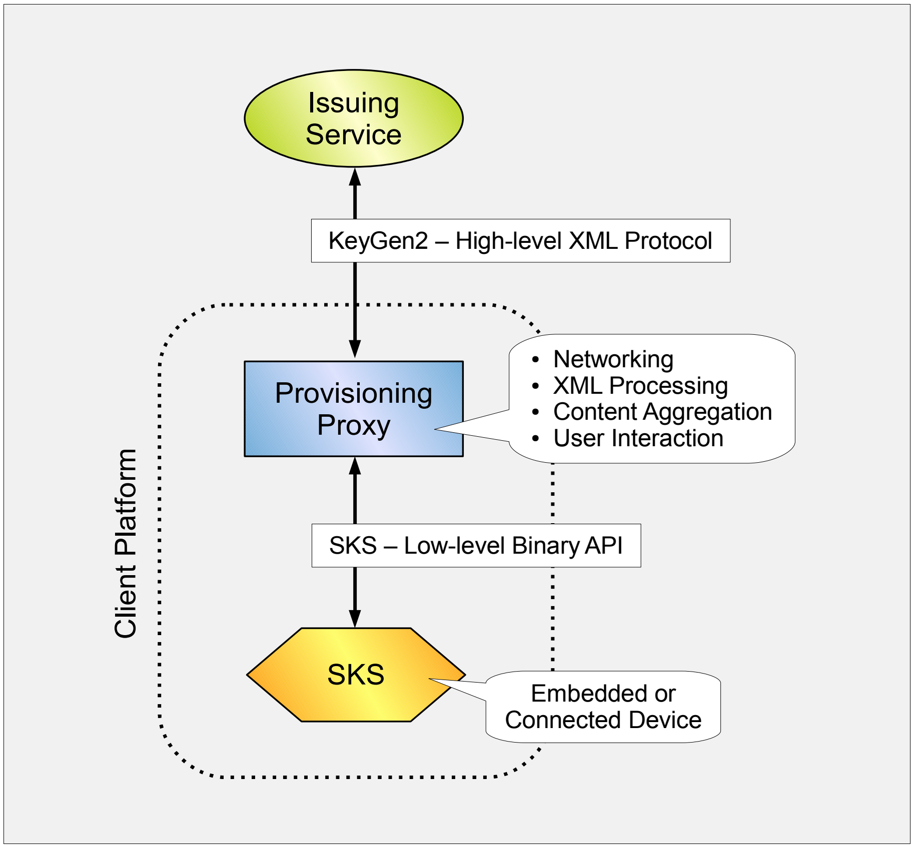
A conforming update service **must** use the MIME-type **application/octet-stream**.

> The **updateFirmware** method **must** be implemented in such a way that the SKS container cannot be made inoperable due to network errors or aborted update operations. In addition, the SKS container **must** be able to *securely authenticate* the update service's **Chunk** data

.

Appendix A. KeyGen2 Proxy

SKS departs from most other SE (Security Element) designs by relying on a "Semi-Trusted Proxy" for the provisioning and management of keys. Introducing a proxy in a scheme which is claimed supporting *true end-to-security* may sound like a contradiction. However, any alterations to the data flowing between the two end-points (the issuing service and the SKS) will be detected by one of them due to the use of *stateful sessions*, *sequence counters* and *MAC operations*. The picture below shows the SKS/KeyGen2 provisioning architecture:



Since SKS methods *by design* are low-level, most of the comparatively high-level provisioning operations result in multiple SKS calls. In addition, there is a need for referencing objects created by preceeding calls. As it would be quite inefficient if every call forced a network "roundtrip", a core proxy task is *aggregating and linking SKS calls and return data*. This is facilitated through the SKS virtual namespace concept which relieves issuers from ever dealing with raw (and device-dependent) object handles or worrying about name collisions. See Object IDs. The following graph outlines content aggregation and linking when applied to the KeyGen2 example on page 34:

Request	SKS Calls & Object References
	createPUKPolicy (, *"PUK.1"*, ...)
	createPINPolicy (, *"PIN.1"*, ...)
	createKeyEntry (, *"Key.1"*, ...)
	createKeyEntry (, *"Key.2"*, ...)

Response	Returned Data
	Attested public key *"Key.1"*
	Attested public key *"Key.2"*

Another provisioning activity orchestrated by the proxy is requesting (and validating according to the issuer's policy), user-defined PINs, because SKS depends on that all initial PIN values are set during key entry creation.

Appendix B. 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**, ... = createKeyEntry (**ProvisioningHandle**, … , **PINPolicyHandle**, …)
>
> *External certification of the generated public key happens here...*
>
> setCertificatePath (**KeyHandle**, ...)
>
> closeProvisioningSession (**ProvisioningHandle**, ...)

Note that **Handle** variables are only used by local middleware, while (not shown) variables like **SessionKey**, **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 local software emulation of an issuer.

Appendix C. Reference Implementation

To further guide implementers, an open source SKS reference implementation in java® is available including a JUnit suite.

URL: http://code.google.com/p/openkeystore

Appendix D. Remote Key Lookup

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

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:

```xml
<CredentialDiscoveryRequest ... >

  <LookupSpecifier ID="Lookup.1" Nonce="nSgmg4cznqE ... WrH2421w9SYA=">
    <SearchFilter Email="john.doe@example.com"/>
    <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:RSAKeyValue>
          <ds:Modulus>ALhBpUjJK/mSjPAe/ … fXG8z1V3mVDZTBM7eZ</ds:Modulus>
          <ds:Exponent>AQAB</ds:Exponent>
        </ds:RSAKeyValue>
      </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**). See createProvisioningSession and Data Types.
3. Enumerate all sessions having a KeyManagementKey matching the public key of the **Signature**. This serves as an *Issuer Filter.* See enumerateProvisioningSessions.
4. From step #3 enumerate all matching SKS keys and related certificates. See enumerateKeys and getKeyAttributes.
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 and session IDs.

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.

Appendix E. Security Considerations

Note: The following section only *partially* applies to the Privacy Enabled Provisioning mode.

This document does not cover the *physical* security of the key-store since SKS does not differ from other schemes in this respect.

However, the provisioning concept has some specific security characteristics. One of the most critical operations in SKS is the creation of a shared SessionKey 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 the SessionKey depends on issuer-generated and SKS-generated ephemeral public keys. It is pretty obvious that malicious middleware could replace such a key with one it has the private key to and the issuer wouldn't notice the difference. This is where the attestation signature comes in because it is computationally infeasible creating a matching signature since the both of the ephemeral public keys are enclosed as a part of the signed attestation object. That is, the issuer can when receiving the response to the provisioning session request, easily detect if it has been manipulated and *cease the rest of the operation*.

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

Missing or repeated objects are indirectly monitored by the use of MACSequenceCounter, while the SKS "book-keeping" functions will detect other 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 SessionKeyLimit attribute defined in createProvisioningSession is another security measure which aims to limit exhaustive attacks on the SessionKey.

For algorithms that are considered as vulnerable to brute-force key searches, a simple workaround is adding a short *initial delay* to the applicable User API method. Since SKS is exclusively intended for user authentication a 1-100 ms delay imposes a (from the user's point of view), *hardly noticeable* impact on the performance.

By using the EndorsedAlgorithm option, issuers can specify exactly which algorithms that are permitted for a given key.

A significant feature of SKS is that it 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 as a specific unit. The latter also 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*.

That any issuer (after the user's consent), can provision keys may appear a bit scary but *keys do not constitute of executable code* making it less interesting in tricking users accepting "bad" issuers. In addition, the provisioning middleware is also able to validate incoming data for "sanity" and even abort unreasonable requests, such as asking for 10 keys or more to be created.

Although not a part of SKS, KeyGen2 puts a signature derived from the SessionKey over the provisioning session response. The latter holds an HTTPS ServerCertificateFingerprint giving the issuer an opportunity verifying that there actually is a "straight line" between the client and server.

One might suspect that the VSD scheme by relying on a static, *potentially issuance-wide* KeyManagementKey could introduce client-side vulnerabilities but that is unlikely to be the case: If a key management signature is intercepted by an attacker, the inclusion of a high entropy SessionKey and the Device Certificate renders it useless in another session or device. It is also worth noting that the post provisioning operations *by design* do not expose secret or private key data.

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

SKS does not have any built-in policy, it is up to the individual *issuer* deciding about suitable key protections options, key sizes, and private key imports.

Appendix F. 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.

Some of the core concepts have been submitted to http://defensivepublications.org and subsequently been published in IP.COM's *prior art database*.

Appendix G. References

KeyGen2	TBD
PKCS #1	TDB
PKCS #8	TBD
ECDSA	TBD
AES256-CBC	TBD
HMAC-SHA1	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 Card	TBD
Base64	TBD
HOTP	TBD
JavaCard	TBD
JCE	TBD
CryptoAPI	TBD
PKCS #11	TBD
GlobalPlatform	TBD
TLS	TBD
XML Schema	TBD
SP800-56A	TBD
Kerberos	TBD

Blind Signatures	TBD
DAA	TBD

Appendix H. Acknowledgments

SKS and KeyGen2 heavily build on schemes pioneered by other individuals and organizations, most notably:

- *CT-KIP by RSA Security*: KeyGen2 format and basic operation
- *ObC by Nokia*: Key management through dynamic deployment of issuer-specific symmetric keys (VSD), and support for keys bound to downloaded data (in ObC code)
- *SCP80 by GlobalPlatform*: Secure messaging including “rolling MACs”
- *CertEnroll by Microsoft*: Processes

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

KeyGen2 is an “homage” to Netscape Communications Corp. who created the first on-line provisioning system called KeyGen.

Appendix I. Author

Anders Rundgren
anders.rundgren@telia.com

Appendix J. To Do List

Although it would be nice to say “it is 100% ready” there are still a few things missing:

- Investigating “physical presence” GPIO options
- Language check
- Filling in the references