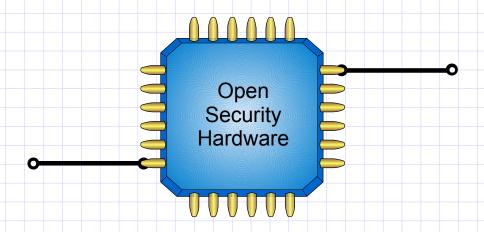


# SKS (Secure Key Store) API and Architecture



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

# Table of Contents

1	Int	troduction	4
2	Co	ore Functionality	4
	2.1	Architecture	.4
	2.2	Provisioning API	.4
	2.3	User API	. 5
	2.4	Security Model	5
	2.5	Transaction Based Operation	5
	2.6	Privacy Enabled Provisioning	6
	2.7	Device ID	6
	2.8	Backward Compatibility	6
3	Ob	ojects	. 7
	3.1	Key Entries	7
	3.2	PIN and PUK Objects	8
	3.3	Provisioning Objects	.8
4	Alg	gorithm Support	9
	4.1	Mandatory Algorithms	9
	4.2	Special Algorithms	10
	4.3	Optional Algorithms	10
5	Pro	otection Attributes	11
	5.1	Export Protection	11
	5.2	Delete Protection	11
	5.3	Biometric Protection	11
	5.4	PIN Input Methods	12
	5.5	PIN Patterns	12
	5.6	PIN and PUK Formats	12
	5.7	PIN Grouping	13
		Application Usage	
6	Se	ession Security Mechanisms	14
	6.1	Encrypted Data	14
	6.2	MAC Operations	14
	6.3	Attestations	14
		Target Key Reference	
7	De	etailed Operation	15
		Data Types	
	7.2	Return Values	16
		Error Codes	
	7.4	Method List	
		getDeviceInfo [1]	
		createProvisioningSession [2]	
		closeProvisioningSession [3]	25

	enumerateProvisioningSessions [4]	27
	abortProvisioningSession [5]	28
	signProvisioningSessionData [6]	29
	updateKeyManagementKey [7]	30
	createPukPolicy [8]	32
	createPinPolicy [9]	33
	createKeyEntry [10]	34
	getKeyHandle [11]	39
	setCertificatePath [12]	40
	importSymmetricKey [13]	42
	importPrivateKey [14]	45
	addExtension [15]	47
	postDeleteKey [50]	51
	postUnlockKey [51]	53
	postUpdateKey [52]	54
	postCloneKeyProtection [53]	56
	enumerateKeys [70]	58
	getKeyAttributes [71]	59
	getKeyProtectionInfo [72]	60
	getExtension [73]	
	setProperty [74]	63
	deleteKey [80]	
	exportKey [81]	
	unlockKey [82]	66
	changePin [83]	67
	setPin [84]	68
	updateFirmware [90]	
	signHashedData [100]	
	asymmetricKeyDecrypt [101]	
	keyAgreement [102]	
	performHmac [103]	
	symmetricKeyEncrypt [104]	
	A. KeyGen2 Proxy	
	B. Sample Session	
	C. Reference Implementation	
	D. Remote Key Lookup	
	E. Security Considerations	
	F. Intellectual Property Rights	
	G. References	
	H. Acknowledgments	
<b>Appendix</b>	I. Author	82

# 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.

In addition to PKI and symmetric keys (including OTP applications), SKS also supports arbitrary key attributes which for example can support novel schemes for identity federation and payment networks.

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.

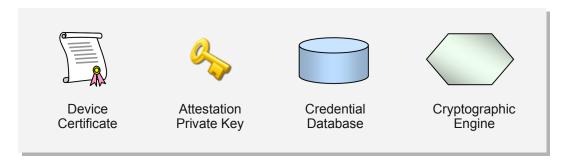
Could existing smart card users also benefit from an upgraded token technology? Yes, the new ways of working, like *virtual* organizations, doesn't make the current distribution scheme "Come and get your card" particularly useful.

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 2.0 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 JSON 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.

Due to the fact that the "Standard" mode potentially affects the user's privacy, it is **recommended** that such requests are equipped with an appropriate user alert notice in the GUI

#### 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" or the X.509 DER format of the 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 <u>Provisioning API</u> 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*.

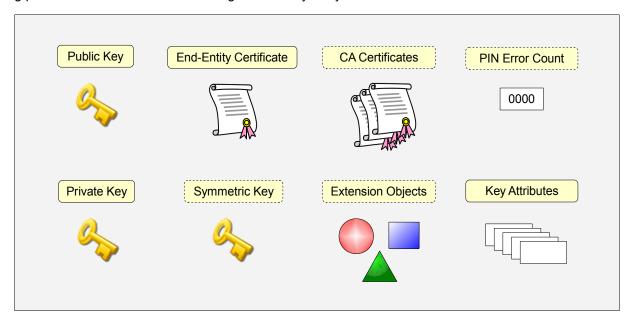
# 3 Objects

The SKS API (as well as its companion protocol KeyGen2), presumes that objects are arranged in a specific fashion. 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:

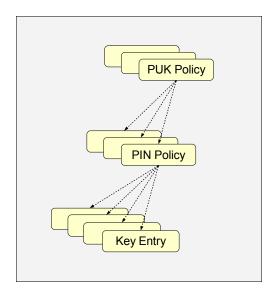


Element	Description	
Public Key	Public part of the asymmetric key-pair created by createKeyEntry	
Private Key Private part of the asymmetric key-pair created by createKeyEntry		
End-Entity Certificate	X.509 certificate set by the <i>mandatory</i> call to setCertificatePath	
Symmetric Key Optional symmetric key defined by calling importSymmetricKey		
CA Certificates Optional X.509 CA certificates defined during the call to setCertificateP		
Extension Objects	Optional extension objects defined by calling addExtension	
PIN Error Count Optional counter for keys protected by a PIN policy object. See createPir		
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 PIN and PUK 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.

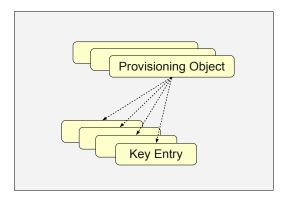
For an example how KeyGen2 deals with this structure, see KeyCreationRequest.

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 **postDeleteKey** 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 devicePinProtection. 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 postDeleteKey, postUnlockKey, postUpdateKey, and postCloneKeyProtection. Also see updateKeyManagementKey.

# 4 Algorithm Support

# 4.1 Mandatory Algorithms

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		
http://www.w3.org/2001/04/xmlenc#aes192-cbc	See XML Encryption. Note that IV <b>must</b> be <i>internally</i> generated as well as prepended to encrypted data	
http://www.w3.org/2001/04/xmlenc#aes256-cbc	gonorated as trem as proportion to strengths and	
http://xmlns.webpki.org/sks/algorithm#aes.cbc	See FIPS 197. Support for 128, 192, and 256-bit keys	
http://xmlns.webpki.org/sks/algorithm#aes.ecb.nopad	See FIF3 197. Support for 126, 192, and 256-bit keys	

HMAC Operations		
http://www.w3.org/2000/09/xmldsig#hmac-sha1	See HMAC-SHA1	
http://www.w3.org/2001/04/xmldsig-more#hmac-sha256		
http://www.w3.org/2001/04/xmldsig-more#hmac-sha384	See XML Signature	
http://www.w3.org/2001/04/xmldsig-more#hmac-sha512		

Asymmetric Key Encryption			
http://xmlns.webpki.org/sks/algorithm#rsa.es.pkcs1_5	See RFC 3447  MGF1: hash function = mgf1 function. No explicit argument	See RFC 3447	
http://xmlns.webpki.org/sks/algorithm#rsa.oaep.sha1.mgf1p		Decryption mode only	
http://xmlns.webpki.org/sks/algorithm#rsa.oaep.sha256.mgf1p			
http://xmlns.webpki.org/sks/algorithm#rsa.raw	Non-padded RSA operation		

Diffie-Hellman Key Agreement	
http://xmlns.webpki.org/sks/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		Signing mode only
http://www.w3.org/2001/04/xmldsig-more#rsa-sha256		
http://www.w3.org/2001/04/xmldsig-more#rsa-sha384		
http://www.w3.org/2001/04/xmldsig-more#rsa-sha512	See XML Signature	
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256		
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384		
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512		
http://xmlns.webpki.org/sks/algorithm#rsa.pkcs1.none	Coo signUpshedData	
http://xmlns.webpki.org/sks/algorithm#ecdsa.none	See signHashedData	

Note that the *binary* encoding of signature values **must** be in accordance with XML Signature which for ECDSA differs from for example OpenSSL and JCE.

Asymmetric Key Generation			
http://xmlns.webpki.org/sks/algorithm#rsa1024	RSA 1024-bit key	Implicit exponent: 65537	
http://xmlns.webpki.org/sks/algorithm#rsa2048	RSA 2048-bit key		
http://xmlns.webpki.org/sks/algorithm#ec.nist.p256	EC NIST "P-256"	See FIPS 186-4	
http://xmlns.webpki.org/sks/algorithm#ec.nist.p384	EC NIST "P-384"		
http://xmlns.webpki.org/sks/algorithm#ec.nist.p521	EC NIST "P-521"		
http://xmlns.webpki.org/sks/algorithm#ec.brainpool.p256r1	EC Brainpool "P256r1"	See RFC 5639	

Supported algorithms can be acquired by calling getDeviceInfo.

Note: RSA "multi-prime" keys are *not* supported by this specification.

# 4.2 Special Algorithms

Special algorithms are unique to SKS:

Special Algorithms		
http://xmlns.webpki.org/sks/algorithm#session.1	See createProvisioningSession	
http://xmlns.webpki.org/sks/algorithm#key.1	See createKeyEntry	
http://xmlns.webpki.org/sks/algorithm#none	See createKeyEntry and importSymmetricKey	

# 4.3 Optional Algorithms

The following algorithms are defined but are optional:

Asymmetric Key Generation			
http://xmlns.webpki.org/sks/algorithm#rsa3072	RSA 3072-bit key	Implicit exponent:	
http://xmlns.webpki.org/sks/algorithm#rsa4096	RSA 4096-bit key	65537	
http://xmlns.webpki.org/sks/algorithm#rsa1024.exp	RSA 1024-bit key		
http://xmlns.webpki.org/sks/algorithm#rsa2048.exp	RSA 2048-bit key	Variable exponent	
http://xmlns.webpki.org/sks/algorithm#rsa3072.exp	RSA 3072-bit key	See keyParameters	
http://xmlns.webpki.org/sks/algorithm#rsa4096.exp	RSA 4096-bit key	,	

Note that the KeyGen2 samples use JOSE algorithm-IDs when there is such a counterpart available.

# 5 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 <u>createKeyEntry</u> **must** return an error and abort the entire session if fed with incorrect data. Also see <u>KeyGen2 Proxy</u>.

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

# 5.1 Export Protection

The following table illustrates the use of the exportProtection attribute:

KeyGen2 Name	Value	Description	
none	0x00	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 Delete Protection

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 Biometric Protection

An SKS **may** also support using biometric data as an alternative or complement 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 <i>or</i> by biometrics	
combined	0x02	The key is protected by a PIN and by biometrics	
exclusive	0x03	The key is <i>only</i> 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.

## 5.4 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
any	0x00 No restrictions	
programmatic	0x01	PINs <b>should</b> only be given through the SKS User API
trusted-gui	0x02	Keys <b>should</b> only be used through a trusted GUI that does the actual PIN request and API invocation

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

#### 5.5 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 <i>invalid</i>	
repeated	0x08	All PIN bytes must be unique	
missing-group	0x10	The PIN format <b>must</b> be alphanumeric or string and contain a mix of <i>letters</i> and <i>digits</i> . The string format also requires <i>lowercase</i> letters and <i>non-alphanumeric</i> 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).

An alternative for organizations having strict requirements on PIN patterns, it is letting users define PINs during enrollment in a web application and then deploy issuer-set PIN codes during provisioning. See pinValue.

#### 5.6 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 encoded 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.

## 5.7 PIN Grouping

A PIN policy object may govern multiple keys. The grouping policy attribute (which is intimately linked to the 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 <b>must</b> share the same PIN
signature+standard 0x02 Keys with appUsage = signature must share a commo while all other keys must share a different PIN		Keys with appUsage = signature must share a common PIN while all other keys must share a different PIN
unique	0x03 All four appUsage types <b>must</b> have <i>different</i> PINs while keys the same appUsage <b>must</b> share a common PIN	

Note that keys having a shared PIN groping attribute **must** be treated as having a single "virtual" PIN object (holding PIN value and error counter), while signature+standard and unique imply two respectively four independent PIN objects.

Shared PINs require that a PIN value or status change must propagate to all keys sharing the particular PIN.

# 5.8 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 <b>should</b> only be used in signature applications like S/M	
authentication 0x01 The key <b>should</b> only be used in applications like TLS cli authentication and login to AD (Active Directory)		The key <b>should</b> only be used in applications like TLS client certificate authentication and login to AD (Active Directory)
encryption 0x02 The key <b>should</b> only be used in encryption applications		The key <b>should</b> only be used in encryption applications
universal 0x03 There are no restrictions on key usage		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).

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.

# 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 *default* security mechanisms used during a provisioning session (defined by the SKS properties sessionKeyAlgorithm and keyEntryAlgorithm). Also see sessionKeyLimit and signProvisioningSessionData.

Note that all elements featured in the following definitions must be supplied "as is" without length indicators.

# 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, "EncryptionKey")
```

*EncryptionKey* **must** only be used with the AES256-CBC algorithm. Note that the IV (Initialization Vector) **must** be *prepended* to the encrypted data as in XML Encryption as well as *freshly generated for each 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, that is, include applicable length-indicators.

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 "DeviceAttestation".

# 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
TargetKeyReference" || HMAC-SHA256 (sessionKey
current || Device ID,
End-Entity Certificate
target))
```

#### Notes:

- 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.

Туре	Length	Description
byte	1	Unsigned byte (0 - 0xFF)
bool	1	Byte containing 0x01 (true) or 0x00 (false)
ushort	2	Unsigned two-byte integer (0 - 0xFFFF)
uint	4	Unsigned four-byte integer (0 - 0xFFFFFFF)
byte[]	2 + length	Array of bytes with a leading "ushort" holding the length of the data
blob	4 + length	Long array of bytes with a leading "uint" holding the length of the data
id	2 + length	Special form of byte[] which <b>must</b> contain an 1-32 byte string with a character set restricted to printable ASCII (0x21 - 0x7E)
uri	2 + length	UTF-8 encoded URI which must not exceed 1000 bytes
string	2 + length	UTF-8 encoded string with arbitrary content

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, and when they are *serialized*.

#### 7.2 Return Values

All methods return a single-byte status code. In case return 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	Туре	Description	
status	byte	Non-zero (error) value	
ErrorString	String	A human-readable error-description with length <= 2000 bytes	

#### 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	Туре	Description
This method does not have any input arguments		

#### Output

Name	Type	Description
status	byte	See Return Values
apiLevel	ushort	100 (1.00) => Applies to this API specification
deviceType	byte	Holds basic device data. See deviceType
updateUrl	uri	HTTP or HTTPS URL pointing to a firmware update service or a zero length array. See updateFirmware
vendorName	string	String of 1-128 characters holding the name of the vendor
vendorDescription	string	String of 1-1000 characters holding a vendor description of the SKS device
pathLength	ushort	Non-zero value holding the number of certificate objects
certificate	byte[]	DER encoded X.509 certificate object repeated as defined by pathLength
supportedAlgorithms	ushort	Non-zero value holding the number of supportedAlgorithm objects
supportedAlgorithm	uri	Algorithm URI repeated as defined by supportedAlgorithms.  See Algorithm Support
cryptoDataSize	uint	Maximum number of bytes in the data argument of cryptographic methods
extensionDataSize	uint	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 certificate 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).

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.

The deviceType property holds a set of fields according to the following table:

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

# createProvisioningSession [2]

## Input

Name	Type	Description
sessionKeyAlgorithm	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 <b>should</b> 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	uint	Locally acquired time in UNIX "epoch" format in seconds. See clientTime
sessionLifeTime	uint	Validity of the provisioning session in seconds. See sessionLifeTime
sessionKeyLimit	ushort	Upper limit of sessionKey operations. See sessionKeyLimit

## Output

Name	Туре	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	uint	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.

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

#### http://xmlns.webpki.org/sks/algorithm#session.1

- 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 || // KDF (Key Derivation Function)
   serverSessionId ||
   issuerUri ||
   Device ID)

- Define internal variable: ushort macSequenceCounter and set it to zero
- Store sessionKey, sessionKeyAlgorithm, 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 *arguments* (e.g. clientSessionId) of the Sign and HMAC operations **must** be represented as described in Data Types.

Creation of a session key is an atomic operation.

#### 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.

sessionKeyAlgorithm 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 ProvisioningInitializationRequest.

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's attestationKey is the Attestation Private Key (see Architecture) and 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 use HMAC-SHA256 with sessionKey as the attestationKey.

provisioningHandle must be static, unique and never be reused.

The clientTime attribute is gathered by the local provisioning middleware and is typically derived from the operating system clock. 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 postDeleteKey actually use two sessionKey operations.

A keyManagementKey must be supplied if provisioned objects should be *updatable in a future session* (see postDeleteKey, postUnlockKey, postUpdateKey, and postCloneKeyProtection), 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.

When using KeyGen2 the *input* to createProvisioningSession is expressed as shown (in the E2ES mode) below:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningInitializationRequest",
  "serverSessionId": "14153858604BE5OTXkwbax23nslxS3gh",
  "submitUrl": "https://issuer.example.com/provsess",
  "serverTime": "2016-01-08T10:00:17Z",
  "sessionKeyAlgorithm": "http://xmlns.webpki.org/sks/algorithm#session.1",
  "sessionKeyLimit": 50,
  "sessionLifeTime": 10000,
  "serverEphemeralKey": {
     "publicKey": {
       "type": "EC",
       "curve": "P-256",
       "x": "INxNvAUEE8t7DSQBft93LVSXxKCiVjhbWWfyg023FCk",
       "y": "LmTIQxXB3LgZrNLmhOfMaCnDizczC RfQ6Kx8iNwfFA"
    }
  },
"keyManagementKey": {
     "publicKey": {
       "type": "RSA",
       "n": "jvct15zkH0lw2OwFCn ... vPFX7K1GqLdnumNHNrY1YQ",
       "e": "AQAB"
  }
}
```

#### Notes:

The keyManagementKey object is optional. Also see updateKeyManagementKey.

serverTime is simply a reference and possible "sanity control" for the client.

submitUrl holds the web-address where the ProvisioningInitializationResponse is to be POSTed.

When using KeyGen2 the output from createProvisioningSession is translated as shown in the example below:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningInitializationResponse",
  "serverSessionId": "14153858604BE5OTXkwbax23nslxS3gh",
  "clientSessionId": "QqTIcUH Md7 i2dP4S5VKYmmYsbUbzGL",
  "serverTime": "2014-12-08T10:00:17Z",
  "clientTime": "2014-12-08T12:00:19+02:00",
  "clientEphemeralKey": {
    "publicKey": {
       "type": "EC"
       "curve": "P-256".
       "x": "INxNvAUEE8t7DSQBft93LVSXxKCiVjhbWWfyg023FCk",
       "y": "LmTIQxXB3LgZrNLmhOfMaCnDizczC RfQ6Kx8iNwfFA"
    }
  "deviceId": {
    "certificatePath": [
       "MIICIzCCAX-gAwlBAgI ... uk9W/uNIHdoyQn19w",
       "MIIDZjCCAk6gAwlBAgl ... xOmZyH10xvpsnmokg",
       "MIIDZjCCAk6gAwlBAgI ... ObXiOInYgeKdK-Dw"
    ]
  "attestation": "Tgzvnr k266LMXinVm ... 7pkJnYipIf9xjOuUJD6OYs",
  "serverCertificateFingerPrint": "HwKCofkgkTFXRmyyb CnWhAcTbQF7w8rl1OgCwyM4TM",
  "signature": {
    "algorithm": "HS256",
    "value": "wMC28Biv-QuYSMCB27AUz8hqwyHoqT6lob0Wk0nuRFk"
  }
}
```

#### Notes:

In the E2ES mode the deviceId must be available for verification of the attestation signature as well as for identification of the SKS container. The deviceId must supply the full Device Certificate path as provided by getDeviceInfo.

In the Privacy Enabled Provisioning mode the deviceId must not be emitted.

serverTime must contain a verbatim copy of the same attribute received in the ProvisioningInitializationRequest.

**serverCertificateFingerPrint** which constitutes of a SHA256 over the server's certificate **must** be created for KeyGen2 protocol invocations using HTTPS. Also see Security Considerations.

To also bind external (non-SKS) parameters to the response, the entire response **must** be signed by the use of an *enveloped* signature object based on the JCS (JSON Cleartext Signature) scheme. The signature operation utilizes a derived version of sessionKey indirectly provided through the signProvisioningSessionData method.

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

#### Server Response Validation

- Decide if the received deviceId featured in the ProvisioningInitializationResponse message 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

```
    Perform a Verify (Device Certificate.publicKey,

                                                            // Received
                  attestation.
                                                            // Received (holds a signature)
                                                            // Received
                  clientSessionId ||
                  serverSessionId |
                                                            // Saved
                                                            // Saved
                  issuerUri |
                                                            // Saved
                  Device ID ||
                  sessionKeyAlgorithm ||
                                                            // Saved
                                                            // Saved
                  privacyEnabled |
                  serverEphemeralKey |
                                                           // Saved
                  clientEphemeralKey |
                                                           // Received
                  keyManagementKey ||
                                                            // Saved
                  clientTime |
                                                            // Received
                  sessionLifeTime |
                                                            // Saved
                                                            // Saved
                  sessionKeyLimit))
```

- Verify that the received enveloped signature validates since this is a proof of that the sessionKey is correctly calculated
- Verify the the optional serverCertificateFingerPrint match the server's certificate

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

Note that in the Privacy Enabled Provisioning mode the deviceId does not apply, and the asymmetric key *Verify* operation is replaced by a comparison between attestation and the output from the HMAC-SHA256.

# closeProvisioningSession [3]

#### Input

Name	Туре	Description
provisioningHandle	uint	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	Туре	Description
status	byte	See Return Values
attestation	byte[32]	Session terminate success attestation signature. See attestation

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 ProvisioningFinalizationRequest processing, where two outermost-level properties hold the associated mac and nonce attributes:

```
{
    "@context": "http://xmlns.webpki.org/keygen2/1.0",
    "@qualifier": "ProvisioningFinalizationRequest",
    "serverSessionId": "1417fa0e508YzrfxGeX-w2ByTAKDSy8v",
    "clientSessionId": "fXQrec8rlgUz5XxQkSZKimbiPbb7eM3f",
    "submitUrl": "https://issuer.example.com/finalize",

    Other Message Payload

"nonce": "NajebxXBmgs1oNj81KzrQBNiAMts-I90kCMJ41QdZhI",
    "mac": "DVhtwgO7fnasR-gouyiReoFGDH2w4Sj6RWZ9SIWJeDQ"
}
```

The attestation object is created by attesting (see Attestations) the following Data:

Data = nonce

Also see sessionKeyLimit.

A successful KeyGen2 response would only contain the following:

```
{
    "@context": "http://xmlns.webpki.org/keygen2/1.0",
    "@qualifier": "ProvisioningFinalizationResponse",
    "serverSessionId": "1417fa0e508YzrfxGeX-w2ByTAKDSy8v",
    "clientSessionId": "fXQrec8rlgUz5XxQkSZKimbiPbb7eM3f",
    "attestation": "acpN8bVJwKZJadlaOsZ-b-7Ky2WRoltP9pFXFD3Nrlo"
}
```

# enumerateProvisioningSessions [4]

#### Input

Name	Туре	Description
provisioningHandle	uint	Input enumeration handle
provisioningState	bool	If true list only open provisioning sessions. If false list only closed dittos

#### Output

Name	Туре	Description
status	byte	See Return Values
provisioningHandle	uint	Output enumeration handle
The following elements must not be present if the returned provisioningHandle = 0		
sessionKeyAlgorithm	uri	
privacyEnabled	bool	
keyManagementKey	byte[]	
clientTime	uint	See create Provisioning Session
sessionLifeTime	uint	See createProvisioningSession
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 is also useful for debugging and for "cleaning-up" after failed provisioning sessions.

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	uint	Local handle to an open provisioning session

## Output

Name	Туре	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	Туре	Description
provisioningHandle	uint	Local handle to an open provisioning session
data	byte[]	Data to be signed

#### Output

Name	Туре	Description
status	byte	See Return Values
result	byte[32]	Signed data

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

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

The signature scheme is as follows:

result = HMAC-SHA256 (sessionKey || "ExternalSignature", data)

Note that all element **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 and ProvisioningInitializationResponse.

# updateKeyManagementKey [7]

#### Input

Name	Туре	Description
provisioningHandle	uint	Local handle to an existing ( <i>closed</i> ) provisioning session object holding a keyManagementKey that needs to be updated to support post-operations using a new keyManagementKey. See Provisioning Objects
keyManagementKey	byte[]	The new keyManagementKey
authorization	byte[]	Authorization signature performed by the <i>old</i> keyManagementKey

#### Output

Name	Туре	Description
status	byte	See Return Values

updateKeyManagementKey associates an existing provisioning session object with an updated keyManagementKey. The update must be cryptographically secured by the authorization signature which is created as follows:

For details on allowed signature algorithms and data representation, see Target Key Reference.

The operation **must** be aborted if the **authorization** signature does not verify or if the target provisioning object lacks a **keyManagementKey**.

Also see enumerateProvisioningSessions.

The following request shows how updateKeyManagementKey is integrated in KeyGen2:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningInitializationRequest",
  "serverSessionId": "14182a80df8 4YcBFZmNkVUnAw9losHa",
  "submitUrl": "https://issuer.example.com/provsess",
  "serverTime": "2014-12-08T10:49:13Z",
  "sessionKeyAlgorithm": "http://xmlns.webpki.org/sks/algorithm#session.1",
  "sessionKeyLimit": 50,
  "sessionLifeTime": 10000,
  "serverEphemeralKey": {
     "publicKey": {
       "type": "EC"
       "curve": "P-256",
       "x": "chrt0S6C3eLbKzbV4R8n1-kKNKHoggbAi4FH3fsDiaQ",
       "y": "WcW6PlkSj3-1GYNu--cdlljTjYtjuhlGEOk6/vv1kTc"
    }
  },
"keyManagementKey": {
     "publicKey": {
       "type": "EC"
       "curve": "P-256",
       "x": "INxNvAUEE8t7DSQBft93LVSXxKCiVjhbWWfyg023FCk",
       "y": "LmTIQxXB3LgZrNLmhOfMaCnDizczC RfQ6Kx8iNwfFA"
     "updatableKeyManagementKeys": [{
       "publicKey": {
          "type": "RSA",
          "n": "kCNcOpatALB21jHrPIv1BgXIUJ . . . pqNo75jsAZIucG9w",
         "e": "AQAB"
       },
       "authorization": "Xjzloz0muM8AMjFafySIR . . . 3sLm1Bfkm4XbbdbrvJw"
    }]
  }
}
```

updatableKeyManagementKeys holds an array of old keyManagementKeys which can be upgraded to the heading (current) keyManagementKey if a matching key is found through calls to enumerateProvisioningSessions.

The updatableKeyManagementKeys array can in turn (recursively) also hold an updatableKeyManagementKeys array making it possible to have any number of keyManagementKey generations deployed. To make this feasible, updates must be performed in steps, starting at the oldest level (leaf updatableKeyManagementKeys array).

**keyManagementKey** updates **must** be done *before* calling createProvisioningSession since open sessions cannot be updated.

# createPukPolicy [8]

#### Input

Name	Type	Description
provisioningHandle	uint	Local handle to an open provisioning session
id	id	External name of the PUK policy object. See Object IDs
encryptedPuk	byte[]	Encrypted PUK value. See Encrypted Data
format	byte	Format of PUK strings. See PIN and PUK Formats
retryLimit	ushort	Value [010000] holding the number of incorrect PUK values ( <i>in a sequence</i> ), forcing the PUK object to permanently lock up. A zero value indicates that there is no limit but that the SKS will introduce an <i>internal</i> 1-10 second delay <i>before</i> 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	Туре	Description
status	byte	See Return Values
pukPolicyHandle	uint	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 relies on the method described in MAC Operations where Data is arranged as follows:

Data = id || encryptedPuk || format || retryLimit

Note that encryptedPuk 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 <a href="mailto:getKeyProtectionInfo">getKeyProtectionInfo</a>.

# createPinPolicy [9]

#### Input

Name	Туре	Description
provisioningHandle	uint	Local handle to an open provisioning session
id	id	External name of the PIN policy object. See Object IDs
pukPolicyHandle	uint	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	ushort	Value [110000] 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	ushort	Minimum decoded PIN length in bytes. See PIN and PUK Formats
maxLength	ushort	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	uint	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 relies on 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 "" 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 <a href="mailto:getKeyProtectionInfo">getKeyProtectionInfo</a>.

# createKeyEntry [10]

## Input

Name	Туре	Description
provisioningHandle	uint	Local handle to an open provisioning session
id	id	External name of the key. See Object IDs
keyEntryAlgorithm	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 <i>device PIN</i> . See PIN and PUK Objects
pinPolicyHandle	uint	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 Export Protection
deleteProtection	byte	See Delete Protection
appUsage	byte	See Application Usage
friendlyName	string	String of 0-100 characters that will be associated with this key for use in GUIs
keyAlgorithm	uri	Algorithm of the key to be created. See Asymmetric Key Generation
keyParameters	byte[]	Optional parameter data needed for some algorithms. See keyParameters
endorsedAlgorithms	ushort	Value [0255] 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	Туре	Description
status	byte	See Return Values
keyHandle	uint	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.

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

#### http://xmlns.webpki.org/sks/algorithm#key.1

The mac relies on the method described in MAC Operations where Data is arranged as follows:

PINPolicyReference is set to "" if pinPolicyHandle is zero, else it is set to the id of the referenced PIN policy object.

PINValueReference is set to "" if pinPolicyHandle is zero, or if the PIN is userDefined, else it is set to the encrypted pinValue.

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.

serverSeed must be a 0-64 byte binary string holding a random number seed. How serverSeed is applied to the random number generation process is unspecified with the exception that a zero-byte input string must not affect the SKS internal random number generation.

For RSA keys with *variable* exponent **keyParameters must** be 1-8 bytes holding a positive big-endian integer, else **keyParameters must** be of zero length.

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

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/sks/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
}
```

The following JSON object shows a typical key generation (initialization) request in KeyGen2:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "KeyCreationRequest",
  "serverSessionId": "1417fa0bedb7rjEFGS-BL3RnJoDyh5UZ",
  "clientSessionId": "PpZRTVg2wa-TLvsFJE7GZPASEeEgk4Yz",
  "submitUrl": "https://issuer.example.com/keyinit",
  "keyEntryAlgorithm": "http://xmlns.webpki.org/sks/algorithm#key.1",
  "pukPolicySpecifiers": [{
     "id": "PUK.1",
     "retryLimit": 3,
     "format": "numeric",
     "encryptedPuk": "xkELvWmx-nHdemfJltY-KmcArGNTsusM7jATLHKHC5U",
     "mac": "oNTuaVBPqgOGJE7xs1tNtlCuzviE2wskcoW1kiuZlKg",
     "pinPolicySpecifiers": [{
       "id": "PIN.1",
       "minLength": 6,
       "maxLength": 8,
       "retryLimit": 3,
       "grouping": "shared",
       "format": "numeric",
       "patternRestrictions": ["three-in-a-row", "sequence"],
       "mac": "Z3IMErjv6varAj5Ww31AAj8e 0QZjkYgFdtquDSf4G0",
       "keyEntrySpecifiers": [
            "id": "Kev.1".
            "appUsage": "authentication",
            "keyAlgorithm": "http://xmlns.webpki.org/sks/algorithm#rsa2048",
            "mac": "ksg1ZwSfGrUjWPWpbK6wrhOKRH7TlwMc V9N51GhFCc"
            "id": "Key.2",
            "appUsage": "signature",
            "keyAlgorithm": "http://xmlns.webpki.org/sks/algorithm#ec.nist.p256",
            "mac": "dC--5J1yQ1SnP4WyRQv4sZJG9gPlq29wO4E2nnX5sFk"
    }]
  }]
}
```

This sequence should be interpreted as a request for an RSA key and an EC 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.

Note that the actual linkage of PUK, PIN and key-specifiers is accomplished through *object embedding* in the protocol which the KeyGen2 Proxy **must** be honoring.

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 below:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "KeyCreationResponse",
  "serverSessionId": "1417fa0bedb7rjEFGS-BL3RnJoDyh5UZ",
  "clientSessionId": "PpZRTVq2wa-TLvsFJE7GZPASEeEqk4Yz",
  "generatedKeys": [
    {
       "id": "Key.1",
       "publicKey": {
         "type": "RSA",
         "n": "sol7DCkNaGZtMP8QLMCu . . . TzTPWM6qFKWLzR45-3DWcPw",
         "e": "AQAB"
       },
"attestation": "bYNI0YTCnVXvuNUM1Im_grDC9U2c63nRbqchnpaoUVg"
     },
{
       "id": "Key.2",
       "publicKey": {
         "type": "EC",
         "curve": "P-256",
         "x": "nGlEGlaJp0aSJzD3aNsqt1QC3CCSGDgPTVG_2pFLQ6w",
         "y": "XOa0-BbXVqqcvwBBOMvV1fs5BzbC9rLdBnXigWNy97o"
       },
"attestation": "TtScC3wolB_hGt3SmSvpgglB2Z33S87vSl94hCFFsSE"
  ]
}
```

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 [11]

### Input

Name	Туре	Description	
provisioningHandle	uint	Local handle to an open provisioning session	
id	id	See createKeyEntry	

### Output

Name	Туре	Description	
status	byte	See Return Values	
keyHandle	uint	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 id must return an error and abort the provisioning session.

## setCertificatePath [12]

#### Input

Name	Туре	Description	
keyHandle	uint	Local handle to a key-pair belonging to an open provisioning session	
pathLength	ushort	Non-zero value holding the number of certificate objects in the call	
certificate	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	Туре	Description	
status	byte	See Return Values	

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

The certificate 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 certificate 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 importPrivateKey method. It is the MAC operation that is facilitating a cryptographically verifiable binding between the certificate path and the designated key entry.

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

Data = keyHandle.publicKey || keyHandle.id || certificate...

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 postUpdateKey or postDeleteKey operation.

A compliant SKS **must** verify that the public key of the End-Entity Certificate matches the Asymmetric Key Generation capabilities of the SKS.

The following KeyGen2 object shows its interaction with setCertificatePath:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningFinalizationRequest",
  "serverSessionId": "1417fa0ad90cEhH32g3fqhY 6EbeenIK",
  "clientSessionId": "jPGg77Uqp_A59u7Yo4laSRBZMxmoeLay",
  "submitUrl": "https://issuer.example.com/finalize",
  "IssuedCredentials": [{
    "id": "Key.1",
    "certificatePath": [
       "MIIDbDCCAISgAwlBAgIGAUF_oLFEMA0GCS . . . LNTAajQcWBwAmvX5dvlzg",
       "MIIDYTCCAkmgAwlBAglGAUGCqAG . . . qqN3fG5GMaTCZNuJfRQJyU"
    ],
"mac": "b3hr4Rc6pHo-MuJYGvvAzdV3knV6tVLphdzDUTEfa9w"
  }],
  "nonce": "DGfjSX3JaLVeWd2Q-PS7pKvKwlbOvlqZR0hlu2GSVIs",
  "mac": "6RYr-Lech-bMdttEWJP_cyPNPt0Iw_YXVqx3UuCouNE"
}
```

The certificatePath array must hold a sorted certificate path.

The owning provisioningHandle and local keyHandle can be retrieved by calling enumerateProvisioningSessions and getKeyHandle using the clientSessionId, serverSessionId and id attributes respectively.

## importSymmetricKey [13]

#### Input

Name	Туре	Description	
keyHandle	uint	Local handle to a key belonging to an open provisioning session	
encryptedKey	byte[]	Raw symmetric key encrypted as described in Encrypted Data	
mac	byte[32]	Vouches for the integrity and authenticity of the operation	

#### Output

Name	Туре	Description	
status	byte	See Return Values	

importSymmetricKey imports and links a symmetric key to an already created key-pair and certificate.

The MAC relies on the method described in MAC Operations where Data is arranged as follows:

Data = End-Entity Certificate || encryptedKey

Note that encryptedKey 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/sks/algorithm#none arbitrary static shared secrets can be specified. When used together with exportKey, a suitable PIN policy and a propertyBags 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.IMPORTED flag of the key must be set after execution of importSymmetricKey.

The following KeyGen2 steps show how symmetric keys are provisioned. First the server issues a key-pair request:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "KeyCreationRequest",
  "serverSessionId": "1417fa0c061hwoiSTca BwhHjI7tm5yj",
  "clientSessionId": "yCW200bErAF8DFFmzWWOlphYa2GuFHis",
  "submitUrl": "https://issuer.example.com/keyinit",
  "keyEntryAlgorithm": "http://xmlns.webpki.org/sks/algorithm#key.1",
  "pinPolicySpecifiers": [{
     "id": "PIN.1",
     "minLength": 4,
     "maxLength": 8,
     "retryLimit": 3,
     "format": "numeric",
     "mac": "OvfnCQy7y0v3C234ESYu3KE0iQ1We9JWAipQ-1J0A64",
     "keyEntrySpecifiers": [{
        "id": "Key.1",
        "appUsage": "authentication",
        "keyAlgorithm": "http://xmlns.webpki.org/sks/algorithm#rsa2048",
        "endorsedAlgorithms": ["http://www.w3.org/2000/09/xmldsig#hmac-sha1"],
        "mac": "5s7dC3SX-jZxjPN7Gg3ssvfX-gOYjcsMEWUn8P3dU7g"
     }]
  }]
}
```

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

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

```
{
    "@context": "http://xmlns.webpki.org/keygen2/1.0",
    "@qualifier": "KeyCreationResponse",
    "serverSessionId": "1417fa0c061hwoiSTca_BwhHjI7tm5yj",
    "clientSessionId": "yCW200bErAF8DFFmzWWOIphYa2GuFHis",
    "generatedKeys": [{
        "id": "Key.1",
        "publicKey": {
            "type": "RSA",
            "n": "u6peYjs2LQjo3EiaYK4XIvRdMxLMA7 . . . VCsOAgDVfo8vf3RNmWH53Fw",
            "e": "AQAB"
        },
        "attestation": "grWmZzeyah1OjlvT8KJ3-hOZHx599fnKH4RtbEysiKI"
    }]
}
```

The server then responds with a PKI-compliant certified public key including an encrypted "piggybacked" symmetric key:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningFinalizationRequest",
  "serverSessionId": "1417fa0c061hwoiSTca_BwhHjI7tm5yj",
  "clientSessionId": "yCW200bErAF8DFFmzWWOlphYa2GuFHis",
  "submitUrl": "https://issuer.example.com/finalize",
  "IssuedCredentials": [{
    "id": "Key.1",
    "certificatePath": [
       "MIIDFjCCAf6gAwlBAgIGAUF_oMFSMA0G . . . EJwsqSLO88IVL5jpwW036AVtW3BhILP_Q"
    ],
"mac": "go5cioJmIzyNROKfrA0jGZEmoq_6w15YeLdz8QYq8ns",
    "ImportSymmetricKey": {
      "encryptedKey": "oh1J luDY0jfQYVokvhRvSMw3nfOxiGAVu x9qAg3RJtwt6uhLtNNmukVb4gqx6a",
       "mac": "y0T2uVwaJrUQVPna9CtpgdNxzPdvjRYr kdx8uaDyTc"
    }
  }],
  "nonce": "R7sXoLU2vYoETzmeO6cTNiWJADILyUso-2dZhzhqDBM",
  "mac": "Ks9BCNsBQ407Bv1wa4pAx7WqWXeyttbLEyzARZ7sOH4"
}
```

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/KeyGen2 treats asymmetric and symmetric keys close to identically for provisioning, management and user-selection operations

## importPrivateKey [14]

#### Input

Name	Туре	Description	
keyHandle	uint	Local handle to a key belonging to an open provisioning session	
encryptedKey	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	Туре	Description	
status	byte	See Return Values	

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

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

The mac relies on the method described in MAC Operations where Data is arranged as follows:

Data = End-Entity Certificate || encryptedKey

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

A compliant SKS **must** verify that the imported private key matches the Asymmetric Key Generation capabilities of the SKS.

The keyBackup.IMPORTED flag of the key must be set after execution of importPrivateKey.

If importPrivateKey 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

The following KeyGen2 object shows how a private key is "piggybacked" to a credential to be restored:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningFinalizationRequest",
  "serverSessionId": "1417fa0dcd8PY8 OldKNfCrGh-PPdsXG",
  "clientSessionId": "m5BeY94pU9hqBOh MgQl69lTIYD06eRg",
  "submitUrl": "https://issuer.example.com/finalize",
  "IssuedCredentials": [{
    "id": "Key.1",
    "certificatePath": [
       "MIIC5DCCAcygAwlBAgIGAUF_oN3/MA0G . . . T71wQ5pkQ67eZwqcfGjwmS9H0vVU"
    "mac": "vg5TluFnxqyyVILcEqwRdjA_y_eBOh-s1R3hkQ5_mE8",
    "ImportPrivateKey": {
    "encryptedKey": "uyplo2qEvSzxjkkjtygEhM3e3o . . . clfyK9jyvxhDpUuxKO1PRXR44maaU",
       "mac": "-iu-iigjqZAyQRvYA0oq3aN r87SVzImD3HQwIB0 el"
  }],
  "nonce": "nUEdz6aKN5e8ggLmlp631Lr1gizXe57kE0MM2H05XEE",
  "mac": "I1KxKKns4-9GUnKp6pcTYdK6YxLFncHsSKY7D9cnb2U"
}
```

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

## addExtension [15]

#### Input

Name	Туре	Description	
keyHandle	uint	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	string	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	Туре	Description	
status	byte	See Return Values	

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

The mac relies on 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:

Property Name (Array of)	SubType (Implicit)	Qualifier	ExtensionData	
extensions	0x00	N/A	Binary data extracted from Base64URL encoded strings	
encryptedExtensions	0x01	N/A	Encrypted binary data extracted from Base64URL encoded strings	
propertyBags	0x02	N/A	See propertyBags data normalization	
logotypes	0x03	mimeType	Binary image data extracted from Base64URL encoded strings	

#### Remarks

N/A = zero-length string.

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 strings must not exceed 128 bytes.

Using KeyGen2 an optional propertyBags array holds typed collections of name-value pairs which are referred to as Properties. The following BNF-like definitions outline the syntax:

Notes:

A name must not exceed 255 bytes.

If writable is absent false is assumed.

A properties name-value collection **must** be converted to a *binary blob* before storage in SKS and MACing according to the following:

• Each name-value pair 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 objects are concatenated in the order they occur in the collection.

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

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

The following KeyGen2 sample shows how properties and logotypes can be added to a symmetric key for usage by a HOTP (RFC 4226) application:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningFinalizationRequest",
  "serverSessionId": "14182a7f9f7u8bTUUFaTJVLo29TxtUpG",
  "clientSessionId": "1SJaeriZ6sdL PT3a8qcZ66d2gyW0QpU",
  "submitUrl": "https://issuer.example.com/finalize",
  "IssuedCredentials": [{
    "id": "Key.1",
    "certificatePath": [
       "MIIDYDCCAkigAwlBAgIGAUGCp_w4MA0GCS . . BR0UoFDeHc4NH8ZmJgd_drnyw"
    ],
"mac": "UX1urB8mPPeO5rFwVGL5Sm0zO2zeXnZJtumCSOn7KjU",
    "ImportSymmetricKey": {
       "encryptedKey": "Kx6TU7TwRF65a4ufQdz48fmrABt7ZByc6uK6mkoj6HeY9fdU0axZDf06MqHH",
       "mac": "63icILm4SP393yHTNpYW4sqxy7TPXe96uffH_NzvTvs"
    "propertyBags": [{
       "type": "urn:ietf:rfc:4226",
       "properties": [
         {
            "name": "Counter",
            "value": "0",
            "writable": true
         },
            "name": "Digits",
            "value": "8"
         }
       ],
"mac": "C0bNbjOePsFdYRcvlc3LKISskYKPwW2Ce4ql3egOqhE"
     }],
     "logotypes": [{
       "type": "http://xmlns.webpki.org/keygen2/logotype#application",
       "mimeType": "image/png",
       "extensionData": "P3k0jz0ZilZf9U5Ag1I... Mq1mW1XUF KrhPxs8Aoe3Irrx",
       "mac": "Ir70oKOdGBYa9iISp2QC14V5YznFmfne2o0-5DWHmSo"
     }]
  "nonce": "OX4VP9NrelBDs4YvF6aBuPUJdUtkgm6G1DMnwKKNZJU",
  "mac": "5aS0-MmYweTUAu1dZxYyPZifrZyP9062ELv--labH5Q"
}
```

For HOTP the corresponding KeyCreationRequest operation would preferably include an endorsement algorithm definition as well.

Below is a KeyGen2 sample showing an Extension object holding a Base64URL encoded object containing attributes that presumably have a function to play together with the deployed key:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningFinalizationRequest",
  "serverSessionId": "14182a7f517qhCEyqav1suQZTmKPLF1V",
  "clientSessionId": "oWPA9nCj1_uWy0Ax41tsloVDA_L4cAE0",
  "submitUrl": "https://issuer.example.com/finalize",
  "IssuedCredentials": [{
    "id": "Key.1",
    "certificatePath": [
       "MIICnjCCAYagAwlBAgIGAUGCp VGMA0 . . . w4q16pugWr7CFW4fu3bP4KI"
    ],
"mac": "pr_dgwUNZXBe2v1DKz7m5WUITihosyR2sG_9MKuWuFs",
    "extensions": [{
       "type": "http://xmlns.example.com/payment-credential",
       "extensionData": "IiBIbmHVy85cZS . . . B4bWxuczd3dy53My5vc",
       "mac": "dl3 3anZBaPQcW4ZofhTlgO9WRpEF9HbBcmbFwbMYAE"
    }]
  }],
  "nonce": "0 DxfSgk4uuA3HRBI87zr0RiWQQLLIXeNc-0-ox1VpY",
  "mac": "yltV0QfOg-ZjDEz3YvvEjWlbg0t1vjLFYQXGUVMwXjY"
}
```

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

# postDeleteKey [50]

### Input

Name	Туре	Description		
provisioningHandle	uint	Local handle to an open provisioning session		
targetKeyHandle	uint	Local handle to the target key	Con Target Vey Deference	
authorization	byte[]	Key management authorization signature	See Target Key Reference	
mac	byte[32]	Vouches for the integrity and authenticity of the operation		

#### Output

Name	Туре	Description
status	byte	See Return Values

postDeleteKey deletes a key created in an earlier provisioning session.

The mac relies on the method described in MAC Operations where Data is arranged as follows:

Data = authorization

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

This method is *independent* of Delete Protection settings.

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

The following request shows how postDeleteKey operations are integrated in the KeyGen2 protocol:

```
{
    "@context": "http://xmlns.webpki.org/keygen2/1.0",
    "@qualifier": "ProvisioningFinalizationRequest",
    "serverSessionId": "14186f4ce39zKRaGUE0trW6DrhGgZ58L",
    "clientSessionId": "cxV1TPgFdmTnvFRXhDX6_6a7FAD9Z8fJ",
    "submitUrl": "https://issuer.example.com/finalize",

    Other Message Payload

"deleteKeys": [{
        "fingerPrint": "M_7NT9IYHtcClty2eBqZiddvsoxmQzZ0kzmVcg6llPs",
        "serverSessionId": "14186f4cbd8JwNfYUivrkFyrU5asnmkg",
        "clientSessionId": "u1tVxuCw-ux2TyZlkkq1Rdq732GbpZiV",
        "authorization": "LsWkDWhwcmSXVkuoqeNj0mQ-Vdpb . . . bch7Lr5J22rdtciAFRLHGxZxUK6gZhqw",
        "mac": "pZb5fXDp0hYVOKVXqzW0oP6g11i6Ckw54Wzz0NRVkJo"
}],
        "nonce": "d8AsJSmNPTGf2iV9Hikl6nVIY8Cqkt-AyCHCTAGOqts",
        "mac": "xG5Q9tDNc1V_nO7lQZAkQQaAKDL1wdoyP1uoSRBiwp0"
}
```

Before invoking postDeleteKey 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 each deleteKeys object by calling enumerateProvisioningSessions.
- 2. Find possible keys by calling enumerateKeys and ignoring all but those belonging to the provisioning session found in step #1.
- For the set of keys found in step #2 call getKeyAttributes while looking for a key having an End-Entity Certificate
  matching the SHA256 fingerPrint.
- 4. If step #3 is successful targetKeyHandle is recovered and postDeleteKey can be invoked.

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

## postUnlockKey [51]

#### Input

Name	Туре	Description		
provisioningHandle	uint	Local handle to an open provisioning session		
targetKeyHandle	uint	Local handle to the target key	See Target Vey Deference	
authorization	byte[]	Key management authorization signature	See Target Key Reference	
mac	byte[32]	Vouches for the integrity and authenticity of the operation		

#### Output

Name	Туре	Description
status	byte	See Return Values

postUnlockKey works like unlockKey except that authorization is derived from a Target Key Reference instead of a PUK. The mac relies on the method described in MAC Operations where *Data* is arranged as follows:

Data = authorization

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

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

The following request shows how postUnlockKey operations are integrated in the KeyGen2 protocol:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningFinalizationRequest",
  "serverSessionId": "14186f4d4ccdaW-Z_IHEFw3xVLJ6kpKV",
  "clientSessionId": "qP5ioSdpeGxnJFmo6rE9G9pAUUfnc1cO",
  "submitUrl": "https://issuer.example.com/finalize",
      Other Message Payload
  "unlockKeys": [{
    "fingerPrint": "E0zdgsaxi7GOyBQxdaMeOZKKp4Gv90TLfgNwt7Z9Btw",
    "serverSessionId": "14186f4d44aEEI KtcKAnyLQpnVt3dVa",
    "clientSessionId": "KHdZHnyod54nd9TMixTWDnOtfUVpZW1A",
    "authorization": "f4xmvzt30boYtKpNA4nP . . . rslfnrEen5PJrq0DQPiZNa1Fo8Y6A",
    "mac": "nCTL88llkkr2a_gHtiUP3yBuDQZ7HB15T5yzixmzBYA"
  "nonce": "-EZE7S11okxMhgCNxpZBQ2WCmRPdDMNdjoYnkYmR5M0",
  "mac": "AUy8CWlog32dioIOhHYGpZNRxWJoN kZ6G G7QZSmW4"
}
```

Before invoking postUnlockKey the provisioning middleware must perform the same steps as for postDeleteKey.

# postUpdateKey [52]

#### Input

Name	Туре	Description		
keyHandle	uint	Local handle to a <i>new</i> key belonging to an <i>open</i> provisioning session		
targetKeyHandle	uint	Local handle to the target key	See Target Vey Deference	
authorization	byte[]	Key management authorization signature	See Target Key Reference	
mac	byte[32]	Vouches for the integrity and authenticity of the operation		

#### Output

Name	Туре	Description
status	byte	See Return Values

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

The mac relies on 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 PIN and PUK Objects.

The target key and and the new key **must** have identical 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) **postUpdateKey** combined with an arbitrary number of **postCloneKeyProtection** calls referring to the same **targetKeyHandle**.

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

The following request shows how postUpdateKey is integrated in the KeyGen2 protocol:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningFinalizationRequest",
  "serverSessionId": "14186f4c622d2ixzQBPpRoUe9PR7jC3D",
  "clientSessionId": "YME9J37aH1Xo7tQifFpa9nkiyyMcGESQ",
  "submitUrl": "https://issuer.example.com/finalize",
  "IssuedCredentials": [{
    "id": "Key.1",
    "certificatePath": [
      "MIIDYTCCAkmgAwlBAqIGAUGG9McmMA0GCSq . . . Iz9C0sc5Ak1jNYzvd8GpS4X6C6J3Uys"
    ],
"mac": "J_RnFJtv7SJp5ZPudqVW6wQnqGmKZ66bWBJqCoESgKk",
    "updateKey": {
      "fingerPrint": "PqCoZBJfCvRgikF1oqHa MOJ ZTXrlMFn6RvXCgGwps",
      "serverSessionId": "14186f4c405V9Z4dm6knbREoEA8EhQV8",
      "clientSessionId": "ALHIRvpj39AuDCag1qXj8TQOWc9i3Bor",
      "authorization": "dqJAh-ScttwndPN2Tu3Xy7m4zgmC . . . 0Qe92GoDHr0pes4prWn2rKUrsgw",
      "mac": "EZ0L4kaemzFtHSvSIFatYIC9rU4oXVKowQVTuRBMwNA"
    }
  }],
  "nonce": "nv1TZ1Z-BZfCjgmLzCZB-y1qSiAM8Ch0P93kPLTpHNQ",
  "mac": "Y luaPsb9JncrLKYmeEPKSqwADluXEAy9Yf6oZnDJQU"
}
```

Before invoking postUpdateKey the provisioning middleware must perform the same steps as for postDeleteKey. keyHandle is the handle associated with the issued credential embedding the updateKey operation.

# postCloneKeyProtection [53]

#### Input

Name	Type	Description		
keyHandle	uint	Local handle to a <i>new</i> key belonging to an <i>open</i> provisioning session		
targetKeyHandle	uint	Local handle to the target key	See Target Vey Reference	
authorization	byte[]	Key management authorization signature	See Target Key Reference	
mac	byte[32]	Vouches for the integrity and authenticity of the operation		

#### Output

Name	Туре	Description
status	byte	See Return Values

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

The mac relies on 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 PIN and PUK Objects.

An inherited custom PIN protection scheme must have its grouping attribute set to shared (see PIN Grouping).

A conforming SKS must allow multiple postCloneKeyProtection calls referring to the same targetKeyHandle.

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

The following request shows how postCloneKeyProtection is integrated in the KeyGen2 protocol:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "ProvisioningFinalizationRequest",
  "serverSessionId": "14186f4c20a3ly83wJZoJMA3x hZ2gKo",
  "clientSessionId": "j3CcN3e8UI5XKN1exKqcF19dBi8eGD78",
  "submitUrl": "https://issuer.example.com/finalize",
  "IssuedCredentials": [{
    "id": "Key.1",
    "certificatePath": [
       "MIIDajCCAlKgAwIBAgIGAUGG9MPKM . . . KUtYzmixtncrPb6NveG0x9yrothzHd9k"
    ],
"mac": "zwGCYuuKoiLR5n_OyufcS1Z9sABX4W4dl2dRmyBd8gE",
    "cloneKeyProtection": {
       "fingerPrint": "cnEQwI7hGtfqNgtXeCqG dSN1KOkW1amRx2t6RcPQY0",
      "serverSessionId": "14186f4bfeeibYVPx01l0VbbqspZ0NAY",
      "clientSessionId": "uENhOyeLZjhXo9CT5dqdTC0H4LtEEDqm",
       "authorization": "MEYCIQC5BTwVz8VbrwPo7ujLx . . . HJzsDemjamO6r9yyR15Cw241w",
       "mac": "yViSzGjcqnVpAvkLzkxs5QwoccX-3lVr3 2lbdWJjOg"
    }
  }],
  "nonce": "o3iWxmuLyGNGhMHxEP22At0R5QhvRm2bGK4kzc btJQ",
  "mac": "KGcta9GWH gCnZzcz dUwqxt8YVBq2 lwUJEX/dDTxk"
}
```

Before invoking **postCloneKeyProtection** the provisioning middleware must perform the same steps as for **postDeleteKey**.

keyHandle is the handle associated with the issued credential embedding the cloneKeyProtection operation.

# enumerateKeys [70]

### Input

Name	Туре	Description
keyHandle	uint	Input enumeration handle

#### Output

Name	Туре	Description	
status	byte	See Return Values	
keyHandle	uint	Output enumeration handle	
The following element must not be present if the returned keyHandle = 0			
provisioningHandle	uint	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	Туре	Description
keyHandle	uint	Local handle to the target key

### Output

Name	Туре	Description	
status	byte	See Return Values	
symmetricKeyLength	ushort	Length of symmetric key in <i>bytes</i> . If <b>symmetricKeyLength</b> > 0 the active key is symmetric. See importSymmetricKey	
pathLength	ushort	See setCertificatePath	
certificate	byte[]		
appUsage	byte	See createKeyEntry	
friendlyName	string		
endorsedAlgorithms	ushort		
endorsedAlgorithm	uri		
extensions	ushort	Number of type URIs	Can addEvtancian
type	uri	Extension type URI. Repeated object	See addExtension

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	Туре	Description
keyHandle	uint	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	ushort	Copy of retryLimit defined by createPukPolicy [1]
pukErrorCount	ushort	Current PUK error count for keys protected by a local PUK policy object [1]
userDefined	bool	
userModifiable	bool	
format	byte	
retryLimit	ushort	
grouping	byte	Copies of the corresponding createPinPolicy parameters for keys protected by a local PIN policy object [1]
patternRestrictions	byte	
minLength	ushort	
maxLength	ushort	
inputMethod	byte	
pinErrorCount	ushort	Current PIN error count for keys protected by a local PIN policy object [1] See protectionStatus table on the next page
enablePinCaching	bool	
biometricProtection	byte	Exact copies of the corresponding createKeyEntry parameters
exportProtection	byte	Exact copies of the corresponding createney Entry parameters
deleteProtection	byte	
keyBackup	byte	Tells if there exists a <i>copy</i> 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.

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 <b>must</b> be combined with bit PIN_PROTECTED
PIN_BLOCKED	0x04	The key has locked-up due to PIN errors. This bit <b>must</b> be combined with bit PIN_PROTECTED
PUK_BLOCKED	0x08	The key has locked-up due to PUK errors. This bit <b>must</b> 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 <b>must</b> 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
IMPORTED	0x01	The IMPORTED bit <b>must</b> be set if the key has been supplied through importPrivateKey or importSymmetricKey
EXPORTED	0x02	The EXPORTED bit <b>must</b> be set if the key has been subject to an exportKey operation

# getExtension [73]

### Input

Name	Туре	Description
keyHandle	uint	Local handle to the target key
type	uri	Type URI. See addExtension

### Output

Name	Туре	Description
status	byte	See Return Values
subType	byte	
qualifier	string	Exact copies of the corresponding addExtension parameters
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.

If the requested extension type doesn't exist, the status ERROR\_OPTION must be returned.

# setProperty [74]

### Input

Name	Туре	Description
keyHandle	uint	Local handle to the target key
type	uri	Type URI which <b>must</b> identify a properties extension. See propertyBags
name	string	Property name. String of 1-255 characters
value	string	Property value. Note extensionData size limit

### Output

Name	Туре	Description
status	byte	See Return Values

setProperty sets a named property value in a properties collection linked to a key.

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

# deleteKey [80]

#### Input

Name	Туре	Description
keyHandle	uint	Local handle to the target key
authorization	byte[]	Zero-length array, PIN, or PUK depending on Delete Protection

### Output

Name	Туре	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* Delete Protection settings.

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

# exportKey [81]

#### Input

Name	Туре	Description
keyHandle	uint	Local handle to the target key
authorization	byte[]	Zero-length array, PIN, or PUK depending on Export Protection

### Output

Name	Туре	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. EXPORTED flag of the key must be set after execution of exportKey.

# unlockKey [82]

### Input

Name	Туре	Description
keyHandle	uint	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	Туре	Description
keyHandle	uint	Local handle to the target key
authorization	byte[]	Original PIN
newPin	byte[]	The requested new PIN

### Output

Name	Туре	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	Туре	Description
keyHandle	uint	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.

## updateFirmware [90]

#### Input

Name	Туре	Description
chunk	blob	Firmware code chunk

#### Output

Name	Туре	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

# signHashedData [100]

#### Input

Name	Туре	Description
keyHandle	uint	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	Туре	Description
status	byte	See Return Values
result	byte[]	Signature in algorithm-specific encoding. See Asymmetric Key Signatures

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 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 must verify that the length of data is within the limits for the particular key type.

The http://xmlns.webpki.org/sks/algorithm#rsa.pkcs1.none signature algorithm must encode the signature value according to PKCS #1 but without hash algorithm identifiers:

EMSA = 0x00 || 0x01 || PS || 0x00 || data

# asymmetricKeyDecrypt [101]

#### Input

Name	Туре	Description
keyHandle	uint	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	Туре	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://xmlns.webpki.org/sks/algorithm#rsa.es.pkcs1\_5.

The parameters object must be of zero length for algorithms not needing additional input.

# keyAgreement [102]

### Input

Name	Type	Description
keyHandle	uint	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. publicKey must match the elliptic curve capabilities given by getDeviceInfo.

The parameters object must be of zero length for algorithms not needing additional input.

# performHmac [103]

### Input

Name	Туре	Description
keyHandle	uint	Local handle to the target key
algorithm	uri	HMAC algorithm URI. See HMAC Operations
parameters	byte[]	Parameters needed by some HMAC algorithms
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	Туре	Description
status	byte	See Return Values
result	byte[]	HMACed data

performHmac performs a symmetric key HMAC operation on the input data object.

The parameters object must be of zero length for algorithms not needing additional input.

# symmetricKeyEncrypt [104]

### Input

Name	Туре	Description
keyHandle	uint	Local handle to the target key
algorithm	uri	Encryption algorithm URI. See Symmetric Key Encryption
mode	bool	True for encryption, false for decryption
parameters	byte[]	Parameters needed by some 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	Туре	Description
status	byte	See Return Values
result	blob	Encrypted or decrypted data

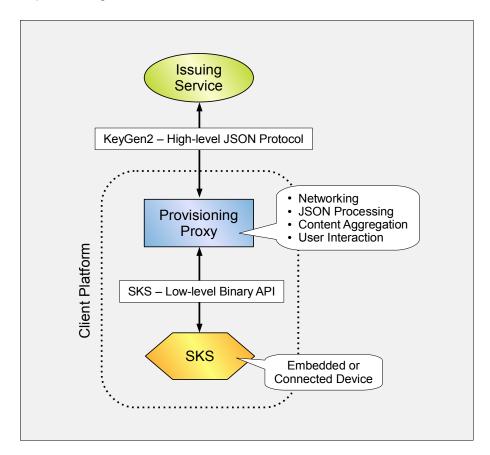
symmetricKeyEncrypt performs a symmetric key encryption or decryption operation on the input data object.

Note that if an IV (Initialization Vector) is required by the encryption algorithm it **must** be supplied in **parameters** unless it is supposed to supplied as a part of data like for XML Encryption.

The parameters object must be of zero length for algorithms not needing additional input.

## 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 preceding calls. As it would be quite inefficient if every call forced a network "round-trip", 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 37:

Request	SKS Calls & Object Referen	ices
	createPukPolicy (, "PUK.1",)	7
	createPinPolicy (, "PIN.1",)	
	createKeyEntry (, "Key.1",)	)
	createKeyEntry (, "Key.2",)	

Response	Returned Data
<b>†</b>	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 postDeleteKey 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 message below:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "CredentialDiscoveryRequest",
  "serverSessionId": "14184c1f09eqCkPtjqY54Ehalc2_EjFN",
  "clientSessionId": "Qn7o4xCRp1sewDrpqMJjEDieZHp2hego",
  "submitUrl": "https://issuer.example.com/credisc",
  "lookupSpecifiers": [{
     "id": "Lookup.1",
     "nonce": "eG3XgquTRh6ASFpcUpEe0gc1qnlL_I2CoPx8xqJTvQ0",
     "searchFilter": {
       "emailRegEx": "\\Qjohn.doe@example.com\\E"
    },
"signature": {
       "algorithm": "ES256".
       "publicKey": {
         "type": "EC"
         "curve": "P-256",
         "x": "INxNvAUEE8t7DSQBft93LVSXxKCiVjhbWWfyg023FCk",
         "y": "LmTIQxXB3LgZrNLmhOfMaCnDizczC RfQ6Kx8iNwfFA"
       },
"value": "MEUCIHWCPcDl6kea9DMy . . . Av7Px3bfwvagWcQY4kVrdeT38clzhiKnpiluigY"
    }
  }]
}
```

For each object in the lookupSpecifiers array, perform the following steps:

- 1. Verify that the signature is technically valid while the origin of the signing key is ignored since the KeyGen2 Proxy has no opinion about those.
- 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 object. 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 all *unique* keys from step #4 having matching search conditions. In the sample that is having an e-mail address "john.doe@example.com" in the End-Entity Certificate.

The result of each is sent back to the issuer in the form of a list of End-Entity Certificate paths and session IDs:

```
{
  "@context": "http://xmlns.webpki.org/keygen2/1.0",
  "@qualifier": "CredentialDiscoveryResponse".
  "serverSessionId": "14184c1f09eqCkPtjqY54Ehalc2 EjFN",
  "clientSessionId": "Qn7o4xCRp1sewDrpqMJjEDieZHp2hego",
  "lookupResults": [{
    "id": "Lookup.1",
     "matchingCredentials": [{
        "serverSessionId": "14184c1f0438OwdjLnmGgIx2c8245rDH",
        "clientSessionId": "wmdVVHWjI666GvHnwmIALFRJQ-GC3Scr",
        "certificatePath": [
          "MIICIjCCAX6gAwIBAgIGAUGEwfB4MA0GCSq . . . rGnyW8pnGcQ1U2cIYD6vWN28GEup"
       ],
"locked": true
     }]
  }]
}
```

#### Notes:

Remote key lookups are performed at the *middleware level* since they are passive, JSON-centric, 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.

If a matching credential is locked (presumably due to user authorization failures), this information will also be available as shown in sample.

## 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 <u>createProvisioningSession</u> you will note that the <u>sessionKey</u> 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  $\odot$ 

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 <a href="http://defensivepublications.org">http://defensivepublications.org</a> 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 2.0 TBD

Diffie-Hellman TBD

S/MIME TBD

UTF-8 TBD

XML Encryption TBD

RFC 3447 TBD

RFC 5639 TBD

XML Signature TBD

FIPS 197 TBD

FIPS 186-4 TBD

Base64URL 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

URI TBD

JCS JSON Cleartext Signature

JOSE 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.net@gmail.com

## 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 proofing
- · Filling in the references