

DRAFT

SKS (Secure Key Store) API and Architecture

Disclaimer: This is a system in development. That is, the specification may change without notice. However, it should still give you a good insight in the “Cloud” Token concept. *Feedback is encouraged!*

Table of Contents

Introduction.....	4
Architecture.....	4
Provisioning API.....	4
Backward Compatibility.....	5
User API.....	5
Security Model.....	5
Atomic Operation.....	6
Objects.....	6
Key Protection Objects.....	6
Key Entries.....	7
Provisioning Objects.....	7
Algorithm Support.....	8
Data Types.....	9
Return Values.....	9
Error Codes.....	9
Key Policy Attributes.....	10
Export Control.....	10
Delete Control.....	10
PIN Grouping Control.....	10
PIN Input Control.....	11
PIN Pattern Control.....	11
PIN and PUK Formats.....	11
Biometric Protection.....	12
Key Usage.....	12
Method List.....	13
getDeviceInfo (1).....	14
createProvisioningSession (2).....	15
closeProvisioningSession (3).....	20
enumerateProvisioningSessions (4).....	21
abortProvisioningSession (5).....	22
signProvisioningSessionData (6).....	23
createPUKPolicy (7).....	24
createPINPolicy (8).....	25
createKeyPair (9).....	26
getKeyHandle (10).....	29
setCertificatePath (11).....	30
setSymmetricKey (12).....	32
addExtension (13).....	33
restorePrivateKey (14).....	36
pp_deleteKey (50).....	37
pp_updateKey (51).....	38
pp_cloneKeyProtection (52).....	39
enumerateKeys (100).....	40
getKeyAttributes (101).....	41
getKeyProtectionInfo (102).....	42
getExtension (103).....	43

setProperty (104).....	44
deleteKey (105).....	45
unlockKey (106).....	46
exportKey (107).....	47
signHashedData (110).....	48
asymmetricKeyDecrypt (111).....	49
performHMAC (112).....	50
symmetricKeyEncrypt (113).....	51
Session Security Mechanisms.....	52
Encrypted Data.....	52
MAC Operations.....	52
Attestations.....	52
Post Provisioning MAC Operations.....	52
Sample Session.....	53
Reference Implementation.....	53
Remote Key Lookup.....	54
Security Considerations.....	55
Intellectual Property Rights.....	55
References.....	56
Acknowledgments.....	57
Author.....	57
To Do List.....	57

Introduction

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

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

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

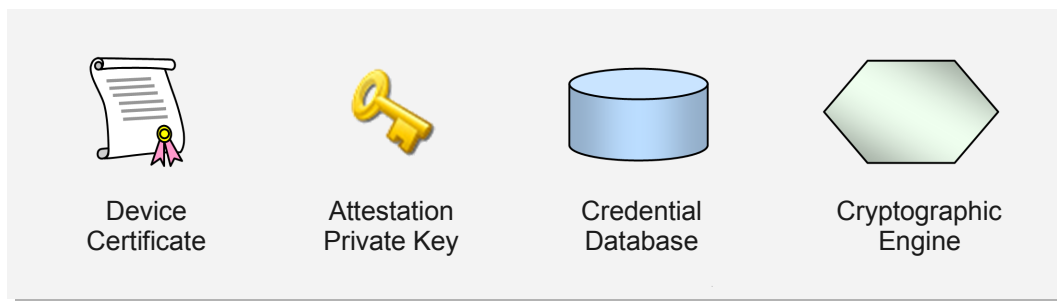
An equally important means for reaching this undeniable bold goal, is that the API and protocols mandate full “on-the-wire” compliance in order to eliminate the current “Smart Card Middleware Hell”; *a single driver per platform should suffice*.

Could *existing* smart card users also benefit from an upgraded token technology? Yes, the new ways of working, like *virtual 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”.

Architecture

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



The *Device Certificate* forms together with a matching *Attestation Private Key* the foundation for the mechanism that facilitates secure provisioning of keys, also when the surrounding middleware (for *self-contained* SKSes NB) and network are unsecured.

The *Credential Database* holds keys and other data that is related to keys such as protection and extension objects. It also keeps the provisioning state.

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

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

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

Provisioning API

Although SKS may be regarded as a “component”, it actually comprises of three associated pieces: The [KeyGen2](#) protocol, the SKS architecture, and the provisioning API described in this document. These items are *tightly matched* to form the foundation for *secure* and *interoperable* ecosystems for provisioning and managing cryptographic keys.

One of the core features of the SKS Provisioning API is enabling independent issuers to securely *share* 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.

Backward Compatibility

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

In fact, *smart cards are almost exclusively personalized by more or less proprietary software under the supervision of card administrators or performed in automated production facilities*. It is evident that (at least) mobile phones need a scheme that is more consistent with the on-line paradigm since SIM-cards due to operator-bindings do not scale particularly well.

“On the Internet anybody can be an operator of something”

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

User API

In this document “User API” refers to operations that are required by security applications like TLS client-certificate authentication, S/MIME, and Kerberos (PKINIT).

The User API is not a core SKS facility but its implementation is anyway **recommended**, particularly for SKSes that are featured in “connected” containers such as smart cards since smart card middleware has proved to be a major stumbling block for wide-spread adoption of PKI cards for consumers.

The described User API is fully mappable to the subset of [CryptoAPI](#), [PKCS #11](#), and [JCE](#) that the majority of current PKI-using applications rely on.

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

If another User API is used the only requirement is that the key objects created by the provisioning API, are compatible with the former.

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 [Global Platform](#). 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, after each successful provisioning session, an *implicit* Security Domain is created which shelters different issuers from each other both from a security and privacy point of view.

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 [Global Platform](#)'s SCP (Secure Channel Protocol) but details differ because [KeyGen2](#) uses an on-the-wire XML format requiring encoding/decoding by the middleware, rather than raw APDUs.

Regarding who trusts an SKS, this is effectively up to each issuer to decide and may be established anytime during an enrollment procedure. Trust in an SKS can be highly granular like only accepting requests from preregistered units or be fully open ended where any SKS compliant 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 with users 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](#).

Atomic Operation

An important aspect of SKS is that a provisioning session **should** either succeed or leave the system intact, AKA “Atomic” or “Transaction-Based” operation. To accomplish this, provisioning operations **must not** be committed until the terminating [closeProvisioningSession](#).

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

Objects

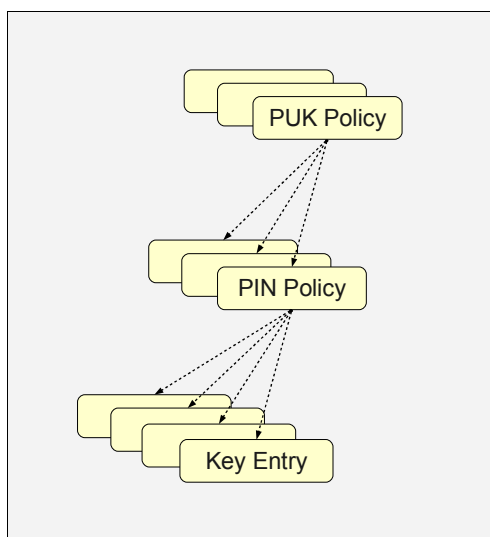
The SKS API (as well as its companion protocol [KeyGen2](#)), assumes that objects are arranged in a specific fashion in order to work. At the heart of the system there are the typical cryptographic keys intended for user authentication, signing etc., but also dedicated keys supporting life-cycle management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

All provisioned user keys, including symmetric dittos (see [setSymmetricKey](#)), 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](#).

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

Key Protection Objects

Keys can *optionally* be protected by PIN-codes (“passphrases”). PIN-protected keys maintain separate PIN error counters, but a single PIN policy object may govern multiple keys. A PIN policy and its associated keys can in turn be supplemented by an optional PUK (Personal Unlock Key) policy object that can be used to reset error-counters that have passed the limit as defined by the PIN policy. Below is an illustration of the SKS protection object hierarchy:



An SKS **may** also support a device (*system-wide*) PIN and PUK. See [getDeviceInfo](#).

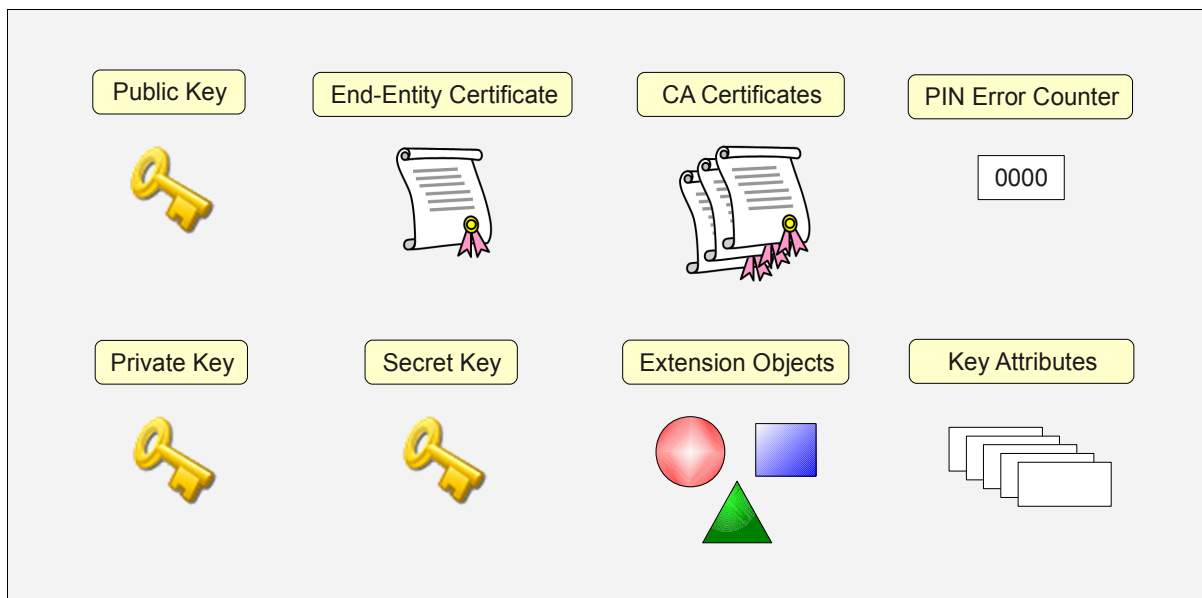
For the creation of protection objects, see [createPUKPolicy](#), [createPINPolicy](#) and [createKeyPair](#).

For an example how [KeyGen2](#) deals with this structure, see [KeyInitializationRequest](#).

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

Key Entries

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



Public Key denotes the public part of the key-pair created by [createKeyPair](#).

Private Key denotes the private part of the key-pair created by [createKeyPair](#).

End-Entity Certificate denotes the [X.509](#) certificate set by the *mandatory* call to [setCertificatePath](#).

Secret Key denotes an *optional* secret key defined by calling [setSymmetricKey](#).

CA Certificates denote *optional* [X.509](#) CA certificates defined during the call to [setCertificatePath](#).

Extension Objects denote *optional* extension objects defined by calling [addExtension](#).

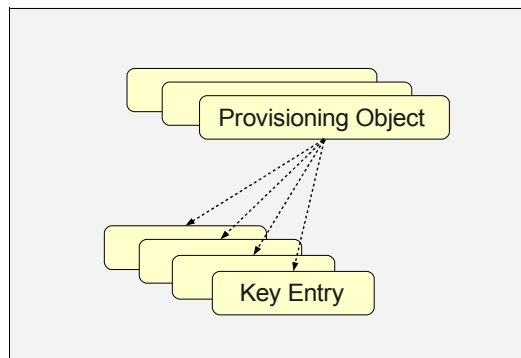
PIN Error Counter denotes a counter associated by keys protected by a PIN policy object. See [createPINPolicy](#).

Key Attributes denote the attributes defined during the call to [createKeyPair](#) and the *optional* call to [setSymmetricKey](#).

Note that the key management methods always involve an entire key entry; *individual components cannot be managed*.

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](#)).

Algorithm Support

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

URI	Comment
Symmetric Key Encryption	
http://www.w3.org/2001/04/xmlenc#aes128-cbc	See XML Encryption
http://www.w3.org/2001/04/xmlenc#aes192-cbc	
http://www.w3.org/2001/04/xmlenc#aes256-cbc	
http://xmlns.webpki.org/keygen2/1.0#algorithm.aes.ecb.nopad	See FIPS 197 . Support for 128, 192, and 256-bit keys
http://xmlns.webpki.org/keygen2/1.0#algorithm.aes.ecb.pkcs5	
HMAC Operations	
http://www.w3.org/2000/09/xmlsig#hmac-sha1	See XML Signature
http://www.w3.org/2001/04/xmlsig-more#hmac-sha256	
Asymmetric Key Encryption	
http://www.w3.org/2001/04/xmlenc#rsa-1_5	See XML Encryption
Diffie-Hellman Key Agreement	
http://xmlns.webpki.org/keygen2/1.0#algorithm.ecdh	See SP800-56A ECC CDH primitive (Section 5.7.1.2)
Asymmetric Key Signatures	
http://www.w3.org/2000/09/xmlsig#rsa-sha1	See XML Signature
http://www.w3.org/2001/04/xmlsig-more#rsa-sha256	
http://www.w3.org/2001/04/xmlsig-more#ecdsa-sha256	
http://xmlns.webpki.org/keygen2/1.0#algorithm.rsa.none	See signHashedData
http://xmlns.webpki.org/keygen2/1.0#algorithm.ecdsa.none	
Session Keys	
http://xmlns.webpki.org/keygen2/1.0#algorithm.sk1	See createProvisioningSession
Key Attestations	
http://xmlns.webpki.org/keygen2/1.0#algorithm.ka1	See KeyAttestation and createKeyPair
Elliptic Curves	
urn:oid:1.2.840.10045.3.1.7	Also known as “P-256”. See FIPS 186-3

Supported algorithms can be acquired by calling [getDeviceInfo](#) which also lists RSA key generation capabilities which **must** include 1024 and 2048 bit keys.

Data Types

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

Type	Length	Comment
byte	1	Unsigned byte (0 - 0xFF)
bool	1	Byte containing 0x01 (true) or 0x00 (false)
short	2	Unsigned two-byte integer (0 - 0xFFFF)
int	4	Unsigned four-byte integer (0 - 0xFFFFFFFF)
byte[]	2 + length	Array of bytes with a leading “short” holding the length of the data
blob	4 + length	Long array of bytes with a leading “int” holding the length of the data
id	2 + length	Special form of byte[] which must contain an 1-32 byte string according to the XML Schema data type <i>NCName</i> but restricted to: a-z A-Z 0-9 . _ -

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[].

Return Values

All methods return a single-byte status code. In case the status is $\neq 0$ there is an error and any expected succeeding values **must not** be read as they are not supposed to be available. Instead there **must** be a second return value containing a [UTF-8](#) encoded description in English to be used for logging and debugging purposes as shown below:

Name	Type	Comment
Status	byte	Non-zero (error) value
ErrorMessage	byte[]	A human-readable error description

Error Codes

The following table shows the standard SKS error-codes:

Name	Value	Comment
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_CRYPT	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

Key Policy Attributes

The following section describes the attributes issuers need to set for defining suitable key protection policies. Also see [getKeyProtectionInfo](#), [PrivateKeyBackup](#), [Updatable](#), 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 [createKeyPair](#) **must** return an error and abort the entire session if fed with incorrect data.

Export Control

The following table illustrates the use of the [ExportPolicy](#) attribute:

KeyGen2 Name	Value	Comment
none	0x00	No authorization needed for exporting the key
pin	0x01	Correct PIN is required
puk	0x02	Correct PUK is required
non-exportable	0x04	The key must not be exported

Also see [exportKey](#).

Delete Control

The following table illustrates the use of the [DeletePolicy](#) attribute:

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

Also see [deleteKey](#).

PIN Grouping Control

A PIN policy object may govern multiple keys. The [Grouping](#) policy attribute controls how PIN codes to the different keys may relate to each other according to the following table:

KeyGen2 Name	Value	Comment
none	0x00	No restrictions
shared	0x01	All keys share the <i>same</i> PIN (synchronized)
signature+standard	0x02	Keys with Key Usage = signature share one PIN while all other keys share <i>another</i> PIN (synchronized)
unique	0x03	All keys must have <i>different</i> PIN codes

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

PIN Input Control

The [InputMethod](#) policy attribute tells how PIN codes **should** be inputted to the SKS according to the following table:

KeyGen2 Name	Value	Comment
programmatic	0x01	PINs should only be issued through the SKS User API
trusted-gui	0x02	Keys should only be used through a trusted GUI that does the actual PIN request and API invocation
any	0x03	No restrictions

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

PIN Pattern Control

The [PatternRestrictions](#) policy attribute specifies how PIN codes **must** be designed according to the following table:

KeyGen2 Name	Mask	Comment
two-in-a-row	0x01	Flags 1124 as <i>invalid</i>
three-in-a-row	0x02	Flags 1114 as <i>invalid</i>
sequence	0x04	Flags 1234, 9876, etc as <i>invalid</i>
repeated	0x08	All PIN bytes must be <i>unique</i>
missing-group	0x10	The PIN format must 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](#)).

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

PIN and PUK Formats

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

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

Note that format specifiers only deal with how PINs and PUKs are treated in GUIs; internally 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 [createKeyPair](#) on **numeric** and **alphanumeric** PIN data. Length of the clear-text binary value **must not** exceed 100 bytes. See **Format** attribute in [createPINPolicy](#) and [createPUKPolicy](#).

Biometric Protection

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

KeyGen2 Name	Value	Comment
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 <i>and</i> 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](#).

Key Usage

The [KeyUsage](#) policy attribute specifies how *asymmetric keys* may be used according to the following table:

KeyGen2 Name	Value	Notes	Comment
signature	0x01	1	The key must only be used in signature applications like S/MIME
authentication	0x02	1	The key must only be used in authentication applications
encryption	0x04	1	The key must only be used for PKCS #1 or Diffie-Hellman encryption operations for RSA and ECC respectively
universal	0x08	1	There are no restrictions on key usage
transport	0x10	1, 2, 3	The private key must not be available for the User API
symmetric-key	0x20	3	The key entry must include a “piggybacked” symmetric key while the private key must be <i>disabled</i> . See setSymmetricKey

An additional purpose of the Key Usage attribute is supporting GUI middleware (and user) when requesting PIN codes for a [User API](#) operation. Also see [PIN Grouping Control](#).

Notes

1. The key **must not** be subject to a [setSymmetricKey](#) operation.
2. The key **must not** be exportable. See [Export Control](#).
3. The key **must not** have the [PrivateKeyBackup](#) attribute set or be subject to [restorePrivateKey](#) operations.

Method List

This section provides a (*not yet complete...*) list of the SKS methods. The number in parenthesis holds the *decimal* value used to identify the method in a call. Method calls are formatted as strings of bytes where the first byte is the method ID and the succeeding bytes the applicable argument data. [User API](#) methods have method IDs ≥ 100 .

*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.*

getDeviceInfo (1)

Input

Name	Type	Comment
<i>This method does not have any input arguments</i>		

Output

Name	Type	Comment
Status	byte	See Return Values
APILevel	short	0x0001 => Applies to <i>this</i> API specification
UpdateURL	byte[]	URL pointing to a firmware update center or if not applicable a zero length array
VendorName	byte[]	1-100 byte string holding the name of the vendor
VendorDescription	byte[]	1-100 byte string holding a vendor description of the SKS device
PathLength	byte	Non-zero value holding the number of X509Certificate objects
X509Certificate...	byte[]	DER-encoded X.509 certificate object <i>repeated</i> as defined by PathLength
Algorithms	short	Non-zero value holding the number of Algorithm objects
Algorithm...	byte[]	Algorithm URI <i>repeated</i> as defined by Algorithms . See Algorithm Support
RSAXponentSupport	bool	True if the issuer may specify an <i>explicit</i> exponent value
RSAKeySizes	byte	Number of supported RSA key sizes
RSAKeySize...	short	Holds an RSA key size in <i>bits</i> and is <i>repeated</i> as defined by RSAKeySizes . See Algorithm Support
CryptoDataSize	int	Maximum number of bytes in the Data argument of cryptographic methods
ExtensionDataSize	int	Maximum size of ExtensionData objects
DevicePINSupport	bool	True if the SKS supports a device PIN. See createKeyPair
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](#).

Note that **X509Certificate** objects **must** form an *ordered* certificate path so that the *first* object contains the actual SKS [Device Certificate](#).

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

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

A compliant SKS **must** support [ExtensionData](#) objects with a size of at least 65536 bytes.

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

For ECC key generation see [Elliptic Curves](#).

createProvisioningSession (2)

Input

Name	Type	Comment
SessionKeyAlgorithm	byte[]	Session creation algorithm. See next page and Session Keys
ServerSessionID	id	Server-created provisioning ID which should be unique for the server
ServerEphemeralKey	byte[]	Server-created ephemeral ECDH key. See ServerEphemeralKey
IssuerURI	byte[]	URI associated with the issuer. See IssuerURI
Updatable	bool	True if objects created in the session should support post provisioning updates
ClientTime	int	Locally acquired time in UNIX “epoch” format in <i>seconds</i> . See ClientTime
SessionLifeTime	int	Validity of the provisioning session in seconds
SessionKeyLimit	short	Upper limit of SessionKey operations. See SessionKeyLimit

Output

Name	Type	Comment
Status	byte	See Return Values
ClientSessionID	id	SKS-created provisioning ID which must be unique
ClientEphemeralKey	byte[]	SKS-created ephemeral ECDH key which must be in X.509 DER format
SessionAttestation	byte[]	Session attestation signature
ProvisioningHandle	int	Local handle to created provisioning session

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

Continued on the next page...

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

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

- 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](#) C(2, 0, ECC CDH) algorithm 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 Certificate)
- Output **SessionAttestation** = **Sign** (*Attestation Private Key*, // See [Architecture](#)
HMAC-SHA256 (**SessionKey**, **SessionKeyAlgorithm** ||
ClientSessionID ||
ServerSessionID ||
IssuerURI ||
ServerEphemeralKey ||
EKP.PublicKey ||
Updatable ||
ClientTime ||
SessionLifeTime ||
SessionKeyLimit))
- Define internal variable: short **MACSequenceCounter** and set it to zero
- Store **SessionKey**, **MACSequenceCounter**, **ClientSessionID**, **ServerSessionID**, **IssuerURI**,
Updatable, **ClientTime**, **SessionLifeTime** and **SessionKeyLimit** in the [Credential Database](#)
and return a handle to the database entry in **ProvisioningHandle**
- Output **ProvisioningHandle**

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

Continued on the next page...

Remarks

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

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

A provisioning session **should not** be terminated due to power down of an SKS.

The **SessionKeyAlgorithm** defines the creation of **SessionKey** but also the integrity, confidentiality, and attestation-mechanisms used during the provisioning session. See [MAC Operations](#), [Encrypted Data](#), and [Attestations](#).

Using [KeyGen2](#) `IssuerURI` is the URL to which the result of this method is POSTed. The string **must not** exceed 1024 bytes.

ServerEphemeralKey **must** be in [X.509](#) DER format and **must** match the [Elliptic Curves](#) capabilities given by [getDeviceInfo](#).

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

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

The `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 `setSymmetricKey` actually uses *two* `SessionKey` operations, while `pp_deleteKey` also depends on the `SessionKeyLimit` of a *previous* session.

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.

Note that individual elements featured in the *argument* (e.g. `ClientSessionID`) to the HMAC operations **must** use the representation described in [Data Types](#).

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

- Decide if **DeviceCertificatePath** is to be accepted/trusted. See [ProvisioningSessionResponse](#)
- Run the the same [SP800-56A](#) procedure and KDF as for the SKS but now using **ClientEphemeralKey** and the saved private key of **ServerEphemeralKey** to obtain **SessionKey**
- VerifySignature (**Device Certificate.PublicKey**, // Received
SessionAttestation, // Received
HMAC-SHA256 (**SessionKey**, **SessionKeyAlgorithm** || // Saved
ClientSessionID || // Received
ServerSessionID || // Saved
IssuerURI || // Saved
ServerEphemeralKey || // Saved
ClientEphemeralKey || // Received
Updatable || // Saved
ClientTime || // Received
SessionLifeTime || // Saved
SessionKeyLimit)) // Saved

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

Continued on the next page...

When using [KeyGen2](#) the *input* to `createProvisioningSession` is expressed as shown in the fragment below:

```
<ProvisioningSessionRequest ID="_0fa47ab3c00c ... a67992b6ac61c"
    SubmitURL="https://ca.example.com/enroll"
    Updatable="true"
    SessionLifeTime="50000"
    SessionKeyLimit="25"
    SessionKeyAlgorithm="http://xmlns.webpki.org/keygen2/1.0#algorithm.sk1" ... >

  <ServerEphemeralKey>
    <ds11:ECKeyValue>
      <ds11:NamedCurve URI="urn:oid:1.2.840.10045.3.1.7"/>
      <ds11:PublicKey>JK/mSALhBpUjjPAe/ ... fXG8z17eZV3mVDZTBM</ds11:PublicKey>
    </ds11:ECKeyValue>
  </ServerEphemeralKey>

</ProvisioningSessionRequest>
```

The table below illustrates argument mapping:

KeyGen2 Element	SKS Counterpart
ProvisioningSessionRequest@ID	ServerSessionID
ProvisioningSessionRequest@SubmitURL	IssuerURI
ProvisioningSessionRequest@Updatable	Updatable
ProvisioningSessionRequest@SessionLifeTime	SessionLifeTime
ProvisioningSessionRequest@SessionKeyLimit	SessionKeyLimit
ProvisioningSessionRequest@SessionKeyAlgorithm	SessionKeyAlgorithm
ServerEphemeralKey/ECKeyValue	ServerEphemeralKey
Gathered by the local provisioning middleware	ClientTime

Note: A few non-SKS-related [KeyGen2](#) elements were omitted for brevity.

Continued on the next page...

When using [KeyGen2](#) the *output* from `createProvisioningSession` is translated as shown in the fragment below:

```
<ProvisioningSessionResponse ID="_126992b6 ... a8a6b484db8f"
    ServerSessionID="_0fa47ab3c00c ... a67992b6ac61c"
    ClientTime="2010-03-18T11:23:29Z"
    ServerCertificateFingerprint="AvDWx6xbg3GDGzdz ... yJk8Js0Oul+Ba/Xc8="
    SessionAttestation="Ob7MvaXC/rNx/rkNZJEo ... 8lch/6snglszfpElrggQfl" ... >

  <ClientEphemeralKey>
    <ds11:ECKeyValue>
      <ds11:NamedCurve URI="urn:oid:1.2.840.10045.3.1.7"/>
      <ds11:PublicKey>PRZre90SQLp ... 16m9FokKxV3F40Y=</ds11:PublicKey>
    </ds11:ECKeyValue>
  </ClientEphemeralKey>
  <DeviceCertificatePath>
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwlBAg ... hugc53W4nNzgg2w==</ds:X509Certificate>
    </ds:X509Data>
  </DeviceCertificatePath>
  <ds:Signature>
    <ds:SignedInfo>
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
      <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#hmac-sha256" />
      <ds:Reference URI="#_126992b6 ... a8a6b484db8f">
        <ds:Transforms>
          <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
          <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
        </ds:Transforms>
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#sha256" />
        <ds:DigestValue>yLD0zNA48Xt9xXNHuBUIK0hL51zn0SYj2lfDXm42PLc=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>aRiSdmrn/KgtjqtTReF+6DOulemRuw2xV9yuOPAlMj8=</ds:SignatureValue>
    <ds:KeyInfo>
      <ds:KeyName>derived-session-key</ds:KeyName>
    </ds:KeyInfo>
  </ds:Signature>

</ProvisioningSessionResponse>
```

The table below illustrates mapping:

KeyGen2 Element	SKS Counterpart
<code>ProvisioningSessionResponse@ID</code>	<code>ClientSessionID</code>
<code>ProvisioningSessionResponse@ServerSessionID</code>	<code>Input.ServerSessionID</code>
<code>ProvisioningSessionResponse@ClientTime</code>	<code>Input.ClientTime</code>
<code>ProvisioningSessionResponse@SessionAttestation</code>	<code>SessionAttestation</code>
<code>ClientEphemeralKey/ECKeyValue</code>	<code>ClientEphemeralKey</code>
<code>DeviceCertificatePath/X509Data/X509Certificate</code>	Certificate <i>path</i> from getDeviceInfo
<code>Signature/SignatureValue</code>	Created with signProvisioningSessionData

Note: A few non-SKS-related [KeyGen2](#) elements were omitted for brevity.

Apart from needed for verifying the `SessionAttestation` signature the `DeviceCertificatePath` identifies the SKS. See [Security Considerations](#).

closeProvisioningSession (3)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an <i>open</i> provisioning session
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values
AttestedResponse	byte[32]	Attestation of the provisioning result. See Attestations

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

- There is an open provisioning session associated with **ProvisioningHandle**
- The **MAC** computes correctly using the method described in [MAC Operations](#) where *Data* is arranged as follows:
$$Data = ClientSessionID \parallel ServerSessionID \parallel IssuerURI$$
- All generated keys are fully provisioned which means that matching public key certificates have been deployed. See [setCertificatePath](#)
- There are no unreferenced PIN or PUK policy objects. See [createPUKPolicy](#) and [createPINPolicy](#)
- The post provisioning operations succeed during the final *commit*. See [Atomic Operation](#)
- There are no keys lacking a [setSymmetricKey](#) operation

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

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

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

Using [KeyGen2](#) **closeProvisioningSession** is invoked as the last step of processing [CredentialDeploymentRequest](#).

The **AttestedResponse** is the result of attesting:

$$Data = "Success" \parallel MACSequenceCounter$$

Note that [MACSequenceCounter](#) is *incremented* by the initial MAC operation.

Also see [SessionKeyLimit](#).

enumerateProvisioningSessions (4)

Input

Name	Type	Comment
ProvisioningHandle	int	Input enumeration handle
ProvisioningState	bool	If true list only <i>open</i> provisioning sessions. If false list only <i>closed</i> dittos

Output

Name	Type	Comment
Status	byte	See Return Values
ProvisioningHandle	int	Output enumeration handle
<i>The following elements must only be emitted if ProvisioningHandle <> 0xFFFFFFFF</i>		
ClientTime	int	See createProvisioningSession
SessionLifeTime	int	
ServerSessionID	id	
ClientSessionID	id	
IssuerURI	byte[]	

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

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

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

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

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

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

abortProvisioningSession (5)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an <i>open</i> provisioning session

Output

Name	Type	Comment
Status	byte	See Return Values

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

signProvisioningSessionData (6)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an <i>open</i> provisioning session
Data	byte[]	Data to be signed

Output

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

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

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

The signature scheme is as follows:

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

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

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

Also see [SessionKeyLimit](#).

createPUKPolicy (7)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an <i>open</i> provisioning session
ID	id	<i>External name</i> of the PUK policy object. See Object IDs
PUKValue	byte[]	Encrypted PUK value. See Encrypted Data
Format	byte	Format of PUK strings. See PIN and PUK Formats
RetryLimit	short	Number of incorrect PUK values (<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	Type	Comment
Status	byte	See Return Values
PUKPolicyHandle	int	Non-zero handle to created PUK policy object

createPUKPolicy creates a PUK policy object in the [Credential Database](#) to be referenced by subsequent calls to the [createPINPolicy](#) method.

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

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

Note that **PUKValue** is MACed “as is” and *then* decrypted by the SKS before storing.

The purpose of a PUK is to facilitate a master key for unlocking keys that have locked-up due to faulty PIN entries. See [unlockKey](#).

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

createPINPolicy (8)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an <i>open</i> provisioning session
ID	id	<i>External name</i> of the PIN policy object. See Object IDs
PUKPolicyHandle	int	Handle to a governing PUK policy object or zero
UserDefined	bool	True if PINs belonging to keys governed by the PIN policy are supposed to be set by the user or by the issuer. See PINValue
UserModifiable	bool	True if PINs can be changed by the user after provisioning
Format	byte	Format of PIN strings. See PIN and PUK Formats
RetryLimit	short	Non-zero value holding the number of incorrect PIN values (<i>in a sequence</i>), forcing a key to lock up
Grouping	byte	See PIN Grouping Control
PatternRestrictions	byte	See PIN Pattern Control
MinLength	byte	Minimum <i>decoded</i> PIN length in bytes. See PIN and PUK Formats
MaxLength	byte	Maximum <i>decoded</i> PIN length in bytes. See PIN and PUK Formats
InputMethod	byte	See PIN Input Control
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

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

createPINPolicy creates a PIN policy object in the [Credential Database](#) to be referenced by subsequent calls to the [createKeyPair](#) method.

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

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

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

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

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

createKeyPair (9)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an <i>open</i> provisioning session
ID	id	<i>External name</i> of the key. See Object IDs
AttestationAlgorithm	byte[]	Attestation algorithm. See Key Attestations
ServerSeed	byte[32]	Server input to the random number generation process. See ServerSeed
PINPolicyHandle	int	Handle to a governing PIN policy object or zero. See PINPolicyHandle
PINValue	byte[]	See PINValue , PIN Pattern Control and PIN Grouping Control
BiometricProtection	byte	See Biometric Protection
PrivateKeyBackup	bool	True if the generated private key is to be outputted in PrivateKey for backup by the issuer
ExportPolicy	byte	See Export Control
DeletePolicy	byte	See Delete Control
EnablePINCaching	bool	True if middleware may cache PINs for this key
KeyUsage	byte	See Key Usage
FriendlyName	byte[]	String of 0-100 bytes that will be associated with this key for use in GUIs
KeyAlgorithmType	byte	Type of key to be generated: 0x00 = RSA, 0x01 = ECC
<i>The following two elements must only be set for RSA keys</i>		
RSASize	short	RSA key size in bits. See getDeviceInfo
RSASExponent	int	Zero (use default) or a defined exponent. See getDeviceInfo
<i>The following element must only be set for ECC keys</i>		
NamedCurve	byte[]	Curve name URI. See Elliptic Curves
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values
KeyHandle	int	Local handle to created key-pair object
PublicKey	byte[]	Generated public key in X.509 DER representation
KeyAttestation	byte[]	See KeyAttestation
PrivateKey	byte[]	<i>Optional generated private key.</i> See PrivateKey

createKeyPair generates an asymmetric key-pair in the [Credential Database](#) according to the issuer's specification.

Continued on the next page...

The following operations match the mandatory to support key generation and attestation algorithm: <http://xmlns.webpki.org/keygen2/1.0#algorithm.ka1>

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

```
Data = ID || AttestationAlgorithm || ServerSeed || PINPolicyReference || PINValueReference ||  

BiometricProtection || PrivateKeyBackup || ExportPolicy || DeletePolicy ||  

EnablePINCaching || KeyUsage || FriendlyName || KeySpecifier
```

PINPolicyReference is set to "#N/A" if **PINPolicyHandle** is zero, to "#Device PIN" if **PINPolicyHandle** is equal to 0xFFFFFFFF, else it is set to the ID of the referenced PIN policy object.

PINValueReference is set to "#N/A" if **PINPolicyHandle** is zero, 0xFFFFFFFF, or if the PIN is [UserDefined](#), else it is set to the encrypted **PINValue**.

KeySpecifier denotes **KeyAlgorithmType** and its related parameters in the same order and format as in the API.

KeyAttestation 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 || PrivateKey
```

Note that the encrypted **PrivateKey** **must** only be used in the attestation *Data* if actually emitted.

Remarks

KeyHandle **must** be *static, unique* and never be reused.

Object IDs for [createKeyPair](#), [createPINPolicy](#) and [createPUKPolicy](#) share a common namespace but the namespace is entirely local to the *provisioning session*.

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

How **ServerSeed** is applied to the random number generation process is *unspecified*. The only requirement is that it **must not** be able to *reduce* the entropy.

A non-zero **BiometricProtection** value presumes that the target SKS supports [Biometric Protection](#), *otherwise an error is returned*. See [getDeviceInfo](#).

The **PrivateKey** element **must** only be created if [PrivateKeyBackup](#) is true. If **PrivateKey** is present it **must** hold the generated private key in [PKCS #8](#) format but *wrapped* as described in [Encrypted Data](#).

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

A **PINPolicyHandle** value of 0xFFFFFFFF presumes that the target SKS supports a "device PUK/PIN", *otherwise an error is returned*. The characteristics of device PINs are out of scope for the SKS specification. See [getDeviceInfo](#).

The **PINValue** object **must** be set depending on if the key is PIN-protected or not according to the following formula:

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

Continued on the next page...

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

```
<KeyInitializationRequest KeyAttestationAlgorithm="http://xmlns.webpki.org/keygen2/1.0#algorithm.ka1" ... >
  <PUKPolicy ID="PUK.1" Format="numeric" RetryLimit="3" Value="mjRuO ... 1Oqw" MAC="uPqE ... HIF=">
    <PINPolicy ID="PIN.1" Format="numeric" Grouping="shared" MaxLength="8" MinLength="4"
      PatternRestrictions="three-in-a-row sequence" RetryLimit="3" MAC="gz56 ... 2B7=">
      <KeyPair ID="Key.1" KeyUsage="authentication" MAC="nFRuPgZ ... H2B429S=">
        <EC NamedCurve="urn:oid:1.2.840.10045.3.1.7"/>
      </KeyPair>
      <KeyPair ID="Key.2" KeyUsage="encryption" PrivateKeyBackup="true" MAC="gSg4chh ... H2BEE7=">
        <RSA KeySize="2048"/>
      </KeyPair>
    </PINPolicy>
  </PUKPolicy>
</KeyInitializationRequest>
```

This sequence should be interpreted as a request for one RSA key and one ECC 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. In addition, private key backup was requested for the RSA key.

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

When using [KeyGen2](#) the *output* from `createKeyPair` is translated as shown in the fragment below:

```
<KeyInitializationResponse ... >
  <PublicKey ID="Key.1" KeyAttestation="X2oMtrm8rRL ... XyTvPuTbergHfnJw==">
    <ds11:ECKeyValue>
      <ds11:NamedCurve URI="urn:oid:1.2.840.10045.3.1.7"/>
      <ds11:PublicKey>BMUY+QiZdFJyzozMgQqA ... ptPGNy/LcDVSXx7s/Y=</ds11:PublicKey>
    </ds11:ECKeyValue>
  </PublicKey>
  <PublicKey ID="Key.2" KeyAttestation="TbergHftrm8rRL ... wyTvPX2XoMunJ==">
    <ds:RSAKeyValue>
      <ds:Modulus>ALhBpUjJK/mSjPAe/ ... fXG8z1V3mVDZTBM7eZ</ds:Modulus>
      <ds:Exponent>AQAB</ds:Exponent>
    </ds:RSAKeyValue>
    <PrivateKey>lysZRM+dYS+bFn5 ... fvJLE6lJq7yshOIZTp</PrivateKey>
  </PublicKey>
</KeyInitializationResponse>
```

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

getKeyHandle (10)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an <i>open</i> provisioning session
ID	id	See createKeyPair

Output

Name	Type	Comment
Status	byte	See Return Values
KeyHandle	int	Local handle to a key belonging to an <i>open</i> provisioning session

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

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

setCertificatePath (11)

Input

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

Output

Name	Type	Comment
Status	byte	See Return Values

setCertificatePath attaches an [X.509](#) certificate path to an already created key-pair. See [createKeyPair](#).

The **X509Certificate** objects **must** form an *ordered* certificate path so that the *first* object contains the [End-Entity Certificate](#) *usually* holding the public key of the target key-pair.

Note that an SKS **must not** attempt to verify that the certificate path and the public key match because that would disable the [restorePrivateKey](#) method. The **MAC** is catering for the binding between certificate path and the generated public key.

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

Certificate path validity **must** be verified by the provisioning middleware.

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

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

Continued on the next page...

The following [KeyGen2](#) fragment shows its interaction with `setCertificatePath`:

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 ... a8a6b484db8f"
                             ID="_0fa47ab3c00c ... a67992b6ac61c"
                             CloseSessionMAC="GcaqnPqb6RxW ... 8kabXDgfMhEMWr=" ... >

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

</CredentialDeploymentRequest>
```

The table below illustrates argument mapping:

KeyGen2 Element	SKS Counterpart
CertificatePath@ID	KeyHandle.ID
CertificatePath@MAC	MAC
CertificatePath/X509Data/X509Certificate...	X509Certificate...

The owning `ProvisioningHandle` and local `KeyHandle` can be retrieved by calling [enumerateProvisioningSessions](#) and [getKeyHandle](#) respectively. Note that in an *interactive* provisioning session, the various handles and IDs involved may be cached by the provisioning middleware, eliminating the need for enumerating keys etc.

setSymmetricKey (12)

Input

Name	Type	Comment
KeyHandle	int	Local handle to a key belonging to an <i>open</i> provisioning session
SymmetricKey	byte[]	“Piggybacked” symmetric key encrypted as described in Encrypted Data
Algorithms	byte	Non-zero value holding the number of EndorsedAlgorithm URIs
EndorsedAlgorithm...	byte[]	Endorsed symmetric key algorithm URI <i>repeated</i> as defined by Algorithms
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

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

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

Data = [End-Entity Certificate](#) || **SymmetricKey** || **EndorsedAlgorithm...**

Note that **SymmetricKey** is MACed “as is” and *then* decrypted by the SKS before storing.

The decrypted **SymmetricKey** **must not** exceed 64 bytes.

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

EndorsedAlgorithm URIs **must** be compatible with [Algorithm Support](#) for *symmetric* keys.

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

See [Key Usage](#) for additional constraints on symmetric keys .

The following [KeyGen2](#) fragment shows the “piggyback” arrangement:

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 ... a8a6b484db8f"
                             ID="_0fa47ab3c00c ... a67992b6ac61c"
                             CloseSessionMAC="GcaqnPqb6RxW ... 8kabXDgfMhEMWr=" ... >

  <CertificatePath ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS ... NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <SymmetricKey EndorsedAlgorithms="http://www.w3.org/2000/09/xmldsig#hmac-sha1"
                  MAC="je7KiznTIQXFdUMRI ... vlnumZCjxSl1CrcqcGkl=">vInt09Esmg94v ...
                                                                YU3tgldhcNNby</SymmetricKey>
  </CertificatePath>

</CredentialDeploymentRequest>
```

For details on how to map keys and sessions, see [setCertificatePath](#).

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

addExtension (13)

Input

Name	Type	Comment
KeyHandle	int	Local handle to a key belonging to an <i>open</i> provisioning session
BaseType	byte	See table below
Qualifier	byte[]	See table below
ExtensionType	byte[]	KeyGen2 Type URI
ExtensionData	blob	Extension object. Regarding size constraints see getDeviceInfo
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

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

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

Data = [End-Entity Certificate](#) || **BaseType** || **Qualifier** || **ExtensionType** || **ExtensionData**

The following table shows **BaseType**, **Qualifier** and **ExtensionData** mapping using [KeyGen2](#):

KeyGen2 Element	BaseType	Qualifier	ExtensionData
Extension	0x00	N/A	Binary data extracted from Base64 -encoded XML
EncryptedExtension	0x01	N/A	Encrypted binary data extracted from Base64 -encoded XML
PropertyBag	0x02	N/A	See PropertyBag canonicalization
Logotype	0x03	MimeType	Binary image data extracted from Base64 -encoded XML

Remarks

N/A = zero-length array.

Note the handling of the **EncryptedExtension**: **ExtensionData** which is encrypted as described in [Encrypted Data](#) is MACed “as is” and *then* decrypted by the SKS before storing.

ExtensionType attributes associated with a given key **must** be *unique*.

ExtensionType attributes *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 **ExtensionType** **must** be featured in the SKS *supported algorithm list*. See [getDeviceInfo](#) and [getExtension](#).

Continued on the next page...

The *simplified* XML schema extract below describes the [KeyGen2](#) representation of **PropertyBag** objects:

PropertyBag XML Schema
<pre><xs:element name="PropertyBag"> <xs:complexType> <xs:sequence> <xs:element name="Property" maxOccurs="unbounded"> <xs:complexType> <!-- The unique name of the property --> <xs:attribute name="Name" use="required"/> <xs:simpleType> <xs:restriction base="xs:string"> <xs:minLength value="1"/> <xs:maxLength value="100"/> </xs:restriction> </xs:simpleType> </xs:attribute> <!-- The value of the property --> <xs:attribute name="Value" type="xs:string" use="required"/> <!-- By default values are read-only but they may be declared as read/writable as well --> <xs:attribute name="Writable" type="xs:boolean" use="optional"/> </xs:complexType> </xs:element> </xs:sequence> <!-- ExtensionType in SKS terms --> <xs:attribute name="Type" type="xs:anyURI" use="required"/> <!-- MAC (Message Authentication Code) --> <xs:attribute name="MAC" type="xs:base64Binary" use="required"/> </xs:complexType> </xs:element></pre>

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

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

Name	Writable	Value
byte[]	bool	byte[]

See [Data Types](#)

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

Note that there are no delimiters added between attributes or objects. The assembled blob holds the actual [ExtensionData](#).

Enforcement of **Property** name uniqueness **may** be delegated to the middleware layer. Also see [setProperty](#).

Continued on the next page...

Below is a [KeyGen2](#) fragment showing an **Extension** object holding a [Base64](#)-encoded [Information Card](#):

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 ... a8a6b484db8f"
                               ID="_0fa47ab3c00c ... a67992b6ac61c"
                               CloseSessionMAC="GcaqnPqb6RxW ... 8kabXDgfMhEMWr=" ... >

  <CertificatePath ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS ... NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <Extension Type="http://schemas.xmlsoap.org/ws/2005/05/identity"
               MAC="UMRIje7KiznTIQXFd ... CrcqcGklvInumZCjxSI1=">PD94bWwgdmVyc2lvbj0iMS4w
               liBlbmHVyZS ... B4bWxuczpkc
               d3dy53My5vc</Extension>

  </CertificatePath>

</CredentialDeploymentRequest>
```

For details on how to map keys and sessions, see [setCertificatePath](#).

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

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 ... a8a6b484db8f"
                               ID="_0fa47ab3c00c ... a67992b6ac61c"
                               CloseSessionMAC="GcaqnPqb6RxW ... 8kabXDgfMhEMWr=" ... >

  <CertificatePath ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS ... NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <SymmetricKey EndorsedAlgorithms="http://www.w3.org/2000/09/xmldsig#hmac-sha1"
                  MAC="je7KiznTIQXFdUMRI ... vInumZCjxSI1CrcqcGkl=">vInt09Esmg94v ...
                  YU3tgdldhcNNby</SymmetricKey>

    <PropertyBag Type="http://xmlns.webpki.org/keygen2/1.0#provider.ietf-hotp"
                 MAC="jI0HDgwl4dO7Kzs ... uEH8MtykIS46JfiJ3N=">
      <Property Name="Counter" Value="0" Writable="true"/>
      <Property Name="Digits" Value="8"/>
    </PropertyBag>
    <Logotype MimeType="image/png"
              Type="http://xmlns.webpki.org/keygen2/1.0#logotype.application"
              MAC="+crSq5fv+7z+fx+f ... ZmRnhxIjO0bh0d=">iVBORw0KGgoAAAANSUHEUgAAALo
              AAABKCAIAAACD ... /tm/AAALjUIEQVR42u
              2d6W8UyRXA+=</Logotype>

  </CertificatePath>

</CredentialDeploymentRequest>
```

restorePrivateKey (14)

Input

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

Output

Name	Type	Comment
Status	byte	See Return Values

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

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

Data = [End-Entity Certificate](#) || **PrivateKey**

Note that **PrivateKey** is MACed “as is” and *then* decrypted by the SKS before storing.

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

See [Key Usage](#) for additional constraints.

The following [KeyGen2](#) fragment shows how credentials that are to be restored should be formatted:

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 ... a8a6b484db8f"
                               ID="_0fa47ab3c00c ... a67992b6ac61c"
                               CloseSessionMAC="GcaqnPqb6RxW ... 8kabXDgfMhEMWr=" ... >

  <CertificatePath ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS ... NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <PrivateKey MAC="umZCjxSI1znTIQX ... CrcqcGklvlnFdUMRIje7Ki=">c2lvbj0iMSwgdMvy4wPD94bW
      VyZSliBlbmH ... ucZpkcB4bWx
      My5vcd3dy53</PrivateKey>
  </CertificatePath>

</CredentialDeploymentRequest>
```

For details on how to map keys and sessions, see [setCertificatePath](#).

pp_deleteKey (50)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an <i>open</i> provisioning session
TargetKeyHandle	int	Local handle to the target key
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

pp_deleteKey deletes a key created in an earlier provisioning session.

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

Data = [Post Provisioning MAC](#)

If the key to be deleted is not present the provisioning session **must** be aborted. See [Remote Key Lookup](#).

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

Note that the *execution* of this method **must** be *deferred* to [closeProvisioningSession](#). Also see [Atomic Operation](#).

Regarding delete of PIN and PUK policy objects, see [Key Protection Objects](#).

The following fragment shows how **pp_deleteKey** operations have been integrated in the [KeyGen2](#) protocol:

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 ... a8a6b484db8f"
    ID="_0fa47ab3c00c ... a67992b6ac61c"
    CloseSessionMAC="GcaqnPqb6RxW ... 8kabXDgfMhEMWr=" ... >

  <CertificatePath ... >
  </CertificatePath>

  <DeleteKey CertificateFingerprint="mNWEOLIAZrG ... dUg2aLA="
    ClientSessionID="_1292389cba4 ... 173d4334"
    ServerSessionID="_1292389cb8 ... c2ea36e"
    MAC="AvDWx6xbg3GDGzdz ... yJk8Js0Oul+Ba/Xc8="/>

</CredentialDeploymentRequest>
```

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

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

pp_updateKey (51)

Input

Name	Type	Comment
KeyHandle	int	Local handle to a <i>new</i> key belonging to an <i>open</i> provisioning session
TargetKeyHandle	int	Local handle to the key to be updated
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

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

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

Data = [End-Entity Certificate](#) || [Post Provisioning MAC](#)

The new key **must** have been fitted with a certificate before this method is called. In addition, the new key **must NOT** be PIN-protected since it supposed to *inherit* the old key's PIN protection scheme (if there is one).

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

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

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

Note that the *execution* of this method **must** be *deferred* to [closeProvisioningSession](#). Also see [Atomic Operation](#).

The following fragment shows how **pp_updateKey** has been integrated in the [KeyGen2](#) protocol:

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 ... a8a6b484db8f"
                               ID="_0fa47ab3c00c ... a67992b6ac61c"
                               CloseSessionMAC="GcaqnPqb6RxW ... 8kabXDgfMhEMWr=" ... >

  <CertificatePath ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS ... NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <UpdateKey CertificateFingerprint="mNWEOLIAZrG ... dUg2aLA="
               ClientSessionID="_1292389cba4 ... 173d4334"
               ServerSessionID="_1292389cb8 ... c2ea36e"
               MAC="AvDWx6xbg3GDGzdz ... yJk8Js0Oul+Ba/Xc8="/>
  </CertificatePath>

</CredentialDeploymentRequest>
```

Before invoking **pp_updateKey** the provisioning middleware needs to perform the same steps as for [pp_deleteKey](#).

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

pp_cloneKeyProtection (52)

Input

Name	Type	Comment
KeyHandle	int	Local handle to a <i>new</i> key belonging to an <i>open</i> provisioning session
TargetKeyHandle	int	Local handle to a previously provisioned key
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

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

The **MAC** uses the method described in [MAC Operations](#) where *Data* is arranged as follows:

Data = [End-Entity Certificate](#) || [Post Provisioning MAC](#)

The new key **must** have been fitted with a certificate before this method is called. In addition, the new key **must not** be PIN-protected since it supposed to *inherit* the old key's PIN protection scheme (if there is one). Inheritance does not mean "copying" but *linking* the new key to an existing PIN object. See [Key Protection Objects](#).

Note that an SKS **must not** allow inheritance of PIN protection schemes except device PIN (see [createKeyPair](#)) or a custom protection scheme with grouping = **shared** (see [PIN Grouping Control](#)).

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

Note that the *execution* of this method **must** be *deferred* to [closeProvisioningSession](#). Also see [Atomic Operation](#).

The following fragment shows how **pp_cloneKeyProtection** has been integrated in the [KeyGen2](#) protocol:

```
<CredentialDeploymentRequest ClientSessionID="_126992b6 ... a8a6b484db8f"
                               ID="_0fa47ab3c00c ... a67992b6ac61c"
                               CloseSessionMAC="GcaqnPqb6RxW ... 8kabXDgfMhEMWr=" ... >

  <CertificatePath ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS ... NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <CloneKeyProtection CertificateFingerprint="mNWEOLIAZrG ... dUg2aLA="
                        ClientSessionID="_1292389cba4 ... 173d4334"
                        ServerSessionID="_1292389cb8 ... c2ea36e"
                        MAC="AvDWx6xbg3GDGzdz ... yJk8Js0Oul+Ba/Xc8="/>

  </CertificatePath>

</CredentialDeploymentRequest>
```

Before invoking **pp_cloneKeyProtection** the provisioning middleware needs to perform the same steps as for [pp_deleteKey](#).

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

enumerateKeys (100)

Input

Name	Type	Comment
KeyHandle	int	Input enumeration handle

Output

Name	Type	Comment
Status	byte	See Return Values
KeyHandle	int	Output enumeration handle
<i>The following element must only be emitted if KeyHandle <> 0xFFFFFFFF</i>		
ProvisioningHandle	int	Handle to the associated provisioning session object

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

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

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

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

getKeyAttributes (101)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key

Output

Name	Type	Comment	
Status	byte	See Return Values	
KeyUsage	byte	See createKeyPair	
FriendlyName	byte[]		
PathLength	byte	See setCertificatePath	
X509Certificate...	byte[]		
Algorithms	byte	See setSymmetricKey	
EndorsedAlgorithm...	byte[]		
Extensions	short	Number of ExtensionType URIs	See addExtension
ExtensionType...	byte[]	KeyGen2 Type URI. <i>Repeated</i> object	

getKeyAttributes returns attribute data for provisioned keys..

[Key Usage](#) determines if the secret key is asymmetric or symmetric.

Asymmetric keys **must** always return a zero in **Algorithms**.

For asymmetric keys [End-Entity Certificate](#) signifies RSA or ECC.

getKeyProtectionInfo (102)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key

Output

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

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

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

Bit	Comment
0	The key is protected by a local PIN policy object
1	The key is protected by a local PUK policy object. This bit must be <i>combined</i> with bit 0
2	The key has locked-up due to PIN errors. This bit must be <i>combined</i> with bit 0
3	The key has locked-up due to PUK errors. This bit must be <i>combined</i> with bit 1
4	The key is protected by a device PIN. Information about device PINs is out of scope for the SKS API. This bit must be the only active bit if applicable

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

getExtension (103)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key
ExtensionType	byte[]	KeyGen2 Type . See addExtension

Output

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

getExtension returns a typed extension object associated with a key.

Note that encrypted extensions are decrypted during provisioning.

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

setProperty (104)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key
ExtensionType	byte[]	must identify a PropertyBag extension. See addExtension
Name	byte[]	Property name
Value	byte[]	Property value

Output

Name	Type	Comment
Status	byte	See Return Values

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

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

deleteKey (105)

Input

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

Output

Name	Type	Comment
Status	byte	See Return Values

deleteKey removes a key from the [Credential Database](#).

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

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

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

Regarding delete of PIN and PUK policy objects, see [Key Protection Objects](#).

unlockKey (106)

Input

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

Output

Name	Type	Comment
Status	byte	See Return Values

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

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

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

exportKey (107)

Input

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

Output

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

exportKey exports a private or secret key from the [Credential Database](#).

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

If a **non-exportable** key is referred to, **exportKey** **must** return [ERROR_NOT_ALLOWED](#) status.

signHashedData (110)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key
SignatureAlgorithm	byte[]	Algorithm URI. See Asymmetric Key Signatures
Authorization	byte[]	Holds a PIN value or is of zero length indicating that no PIN is supplied
Data	byte[]	Hashed data to be signed. Also see CryptoDataSize

Output

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

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

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

The length of **Data** **must** match the hash algorithm. Note that signature algorithms that do not specify a hash algorithm impose no tests on **Data** length. For RSA signatures without hash algorithm the signature packet **must** be formatted according to [PKCS #1](#) but without hash algorithm identifiers:

$$\text{EMSA} = 0x00 \parallel 0x01 \parallel \text{PS} \parallel 0x00 \parallel \text{Data}$$

asymmetricKeyDecrypt (111)

Input

Name	Type	Comment
KeyHandle	int	Local handle to the target key
Parameters	byte[]	Parameters needed by some encryption algorithms
EncryptionAlgorithm	byte[]	Algorithm URI. See Asymmetric Key Encryption
Authorization	byte[]	Holds a PIN value or is of zero length indicating that no PIN is supplied
Data	byte[]	Encrypted data. Also see CryptoDataSize

Output

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

asymmetricKeyDecrypt performs an asymmetric key decryption where the input data **must** be formatted *as required by the encryption algorithm* like [PKCS #1](#) for RSA.

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

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

performHMAC (112)

Input

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

Output

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

performHMAC performs a symmetric key HMAC operation on the input data.

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

symmetricKeyEncrypt (113)

Input

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

Output

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

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

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

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

Session Security Mechanisms

After the [SessionKey](#) has been created the actual provisioning methods can be called. Depending on the specific method downloaded data may be confidential or need to be authenticated. For certain operations the SKS needs to prove for the issuer that sent data indeed stems from internal SKS operations which is referred to as attestations. This section describes the security mechanisms used during a provisioning session. Also see [SessionKeyLimit](#).

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

Encrypted Data

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

EncryptionKey = [HMAC-SHA256](#) ([SessionKey](#), "**Encryption Key**")

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

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.

Individual elements featured in *Data* **must** use the representation described in [Data Types](#).

After each MAC operation, [MACSequenceCounter](#) **must** be incremented by one.

Due the use of a sequence counter, the provisioning system **must** honor the order of objects as defined by the issuer.

Attestations

Except for the [createProvisioningSession](#) call, SKS attestations during provisioning sessions use symmetric key signatures according to the following scheme:

Attestation = [HMAC-SHA256](#) ([SessionKey](#) || "**Device Attestation**", *Data*)

Data represents the data to be attested. Also see [signProvisioningSessionData](#).

Individual elements featured in *Data* **must** use the representation described in [Data Types](#).

Post Provisioning MAC Operations

In order to update already provisioned objects, an issuer may perform *post provisioning* operations. To do that the issuer **must** provide a valid MAC according to the following scheme:

MAC = [HMAC-SHA256](#) (*OriginalSessionKey* || "**Proof Of Ownership**", [End-Entity Certificate](#))

OriginalSessionKey is the [SessionKey](#) of the *original* provisioning session while [End-Entity Certificate](#) points to the *target* key. Note that post provisioning operations require that the [Updatable](#) flag of the target key's provisioning session is set.

[End-Entity Certificate](#) **must** be represented as byte[]. See [Data Types](#).

If [SessionKeyLimit](#) for the *OriginalSessionKey* is exceeded (use count is incremented for each post provisioning MAC), the post provisioning operation **must** be aborted.

Sample Session

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

```
ProvisioningHandle, ... = createProvisioningSession (...)  
PUKPolicyHandle = createPUKPolicy (ProvisioningHandle, ...)  
PINPolicyHandle = createPINPolicy (ProvisioningHandle, , PUKPolicyHandle, ...)  
KeyHandle, ... = createKeyPair (ProvisioningHandle, , PINPolicyHandle, ...)  
  
    External certification of the generated public key happens here...  
  
setCertificatePath (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.

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>

Remote Key Lookup

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

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

```
<CredentialDiscoveryRequest ... >

  <LookupSpecifier ID="Lookup.1" Nonce="nSgmg4cznqE ... WrH2421w9SYA=">
    <SearchFilter Email="john.doe@example.com"/>
    <ds:Signature>
      <ds:SignedInfo>
        <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
        <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256" />
        <ds:Reference URI="#Lookup.1">
          <ds:Transforms>
            <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
            <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
          </ds:Transforms>
          <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256" />
          <ds:DigestValue>JBfoi8iBKRYWxXYITTU1cdyybMTyJr+WDW+qCJdxoGE=</ds:DigestValue>
        </ds:Reference>
      </ds:SignedInfo>
      <ds:SignatureValue>mSMaH6wChPQRDT... JKrW3n/dL7seGbg==</ds:SignatureValue>
    </ds:Signature>
    <ds:KeyInfo>
      <ds:X509Data>
        <ds:X509IssuerSerial>
          <ds:X509IssuerName>CN=Root CA,O=example.com,C=us</ds:X509IssuerName>
          <ds:X509SerialNumber>2</ds:X509SerialNumber>
        </ds:X509IssuerSerial>
        <!-- The Issuer's Certificate: -->
        <ds:X509Certificate>MIIDbzCCAlegAw ... gtzO/rITZcbKHyzCZvQ==</ds:X509Certificate>
      </ds:X509Data>
    </ds:KeyInfo>
  </ds:Signature>
</LookupSpecifier>

</CredentialDiscoveryRequest>
```

The example works as follows:

1. Verify that the **Signature** is *technically* valid. Note that the actual issuer is *ignored* since an SKS has no opinion about what issuers are trustworthy or not
2. Verify that the freshness **Nonce** matches [SHA256](#) (`ClientSessionID || ServerSessionID || IssuerURI`). See [createProvisioningSession](#).
3. Enumerate all SKS keys and related certificates. See [enumerateKeys](#)
4. Find all keys from step #3 having an [End-Entity Certificate](#) forming a valid 2-level certificate path with the `x509Certificate` element which (together with the **Signature**), serves as an *Issuer Filter*
5. Collect the keys from step #4 that also feature the e-mail addresss "`john.doe@example.com`" in the [End-Entity Certificate](#).

The result is sent back to the issuer in the form of a list of [SHA256](#) ([End-Entity Certificate](#)) fingerprints and session IDs.

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

Security Considerations

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

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

If you take a peek at [createProvisioningSession](#) you will note that [SessionKey](#) depends on issuer-generated and SKS-generated ephemeral public keys. It is pretty obvious that malicious middleware could replace such a key with one it has the private key to and the issuer wouldn't notice the difference. This is where the attestation signature comes in because it is computationally infeasible creating a matching signature since the both of the ephemeral public keys are enclosed as a part of the signed attestation object. That is, the issuer can when receiving the response to the provisioning session request, easily detect if it has been manipulated and *cease the rest of the operation*.

As earlier noted, the randomness of [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 in [createProvisioningSession](#) is another security measure which aims to limit exhaustive attacks on [SessionKey](#).

For algorithms that are considered as vulnerable to exhaustive 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.

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

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 [SessionKey](#)-derived signature over the provisioning session response holding an HTTPS [ServerCertificateFingerprint](#) giving the issuer an opportunity verifying that there actually is a “straight line” between the client and server.

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 notion of policy, it is up to the individual *issuer* deciding about suitable key protections options, key sizes, and private key backups.

Intellectual Property Rights

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

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

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

References

KeyGen2	TBD
DIAS	TBD
PKCS #1	TDB
PKCS #8	TBD
ECDSA	TBD
AES256-CBC	TBD
HMAC-SHA256	TBD
X.509	TBD
SHA256	TBD
TPM 1.2	TBD
Diffie-Hellman	TBD
S/MIME	TBD
UTF-8	TBD
XML Encryption	TBD
XML Signature	TBD
FIPS 197	TBD
FIPS 186-3	TBD
Information Card	TBD
Base64	TBD
HOTP	TBD
JavaCard	TBD
JCE	TBD
CryptoAPI	TBD
PKCS #11	TBD
Global Platform	TBD
TLS	TBD
XML Schema	TBD
SP800-56A	TBD

Acknowledgments

SKS and KeyGen2 heavily build on schemes pioneered by others, most notably:

- CT-KIP by RSA Security (KeyGen2 format)
- ObC by Nokia (Key management through issuer-specific symmetric keys, and support for keys bound to downloaded data including code)
- SCP80 by GlobalPlatform (Secure messaging)
- 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.

Author

Anders Rundgren
anders.rundgren@telia.com

To Do List

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

- Diffie-Hellman API
- Firmware upgrade API
- Investigating possible token “wipe” options
- Language check
- Filling in the references