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

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

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

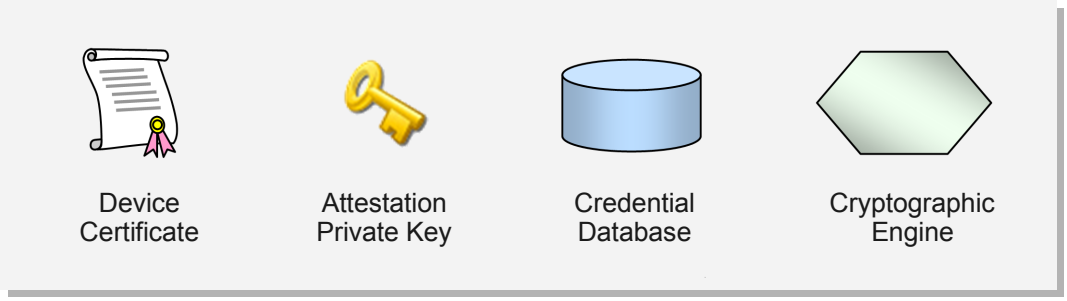
The primary objective with SKS and the related specifications is *establishing two-factor authentication as a viable alternative for any provider* by making the scheme a standard feature in the "Universal Client", the Internet browser.

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

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

Architecture

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



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

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

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

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

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

Provisioning API

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

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

Backward Compatibility

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

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

"On the Internet anybody can be an operator of something"

Many card schemes also depend on roles like SO (Security Officer) which squarely matches scenarios with users associated with a multitude of *independent* service providers. The typical workaround, upgrading the middleware to become a surrogate SO or making users implicit SOs is a pretty nasty hack *that violates the core idea behind smart cards*. Using SKS, the *technical* part of the SO role, exclusively becomes an affair between the card and the issuers, *where each issuer is confined to their own virtual card and SO policy*.

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

User API

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

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

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

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

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

Security Model

Since the primary target for SKS is authentication to arbitrary service providers on the Internet, the security model is quite different to for example that of Global Platform. In practical terms this means that it is the *user* who grants an issuer the right to create keys in the SKS.

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.

That is, there are no predefined "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.

The SKS itself only trusts inbound data that is securely derivable from the applicable 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 very granular like only accepting requests from preregistered units or be fully open ended where any SKS complaint device is accepted. A potentially useful issuer policy would be specifying a set of endorsed SKS brands, presumably meeting some generally recognized certification level like EAL5.

Also see Security Considerations.

Objects

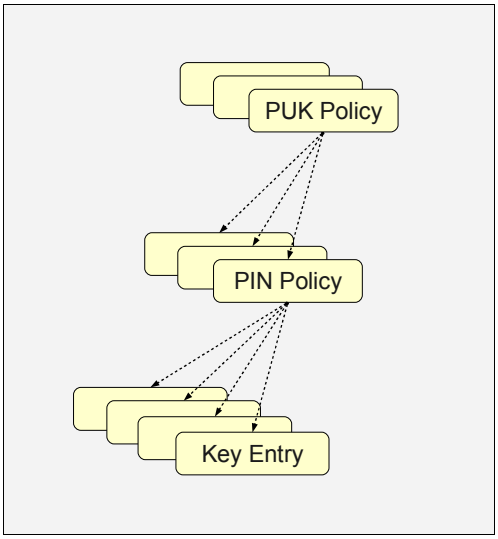
The SKS API (as well as its companion protocol KeyGen2), assumes that objects are arranged in a specific fashion in order to work. At the heart of the system there are the typical cryptographic keys intended for user authentication, signing etc., but also dedicated keys supporting life-cycle management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

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

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

Key Protection Objects

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



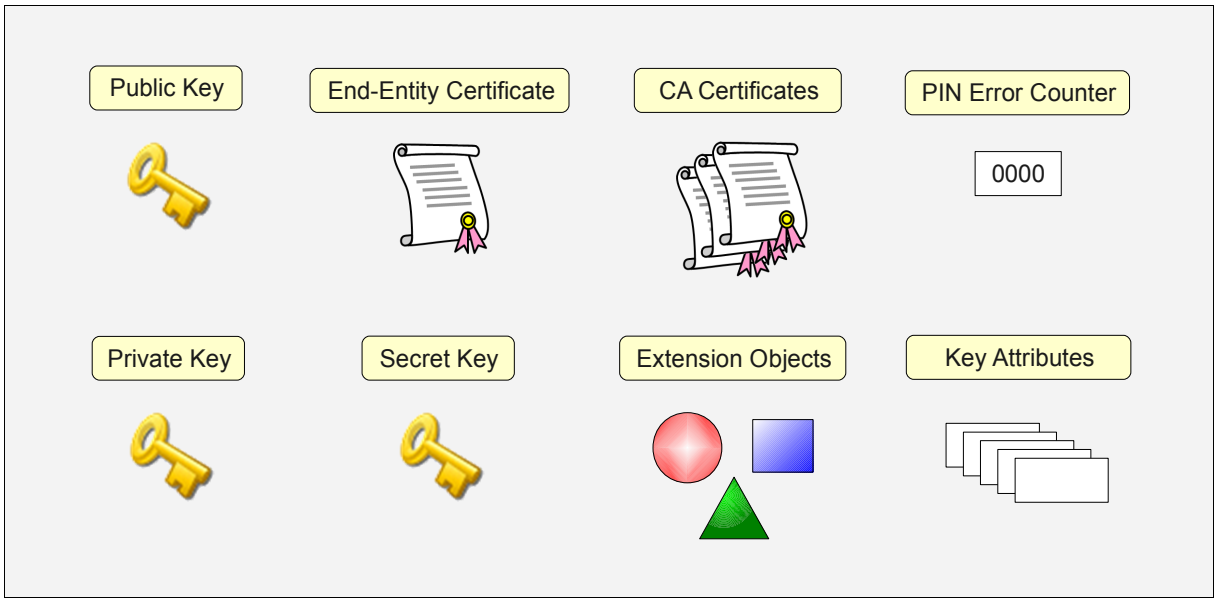
An SKS MAY also support a device (*system-wide*) PIN and PUK. See getDeviceInfo.

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

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

Key Entries

The following picture shows the components 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.

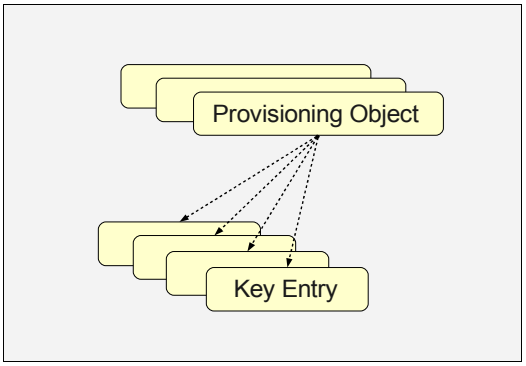
Extension Objects denote *optional* extension objects defined by calling addExtensionData.

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

Key Attributes denote the attributes defined during the call to createKeyPair and the *optional* call to setSymmetricKey.

Provisioning Objects

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



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

Algorithm Support

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

URI	Comment
Symmetric Key Encryption	
http://www.w3.org/2001/04/xmlenc#aes128-cbc	See XML Encryption
http://www.w3.org/2001/04/xmlenc#aes192-cbc	
http://www.w3.org/2001/04/xmlenc#aes256-cbc	
http://xmlns.webpki.org/keygen2/1.0#algorithm.aes.ecb.nopad	See FIPS 197. Support for 128, 192, and 256-bit keys
http://xmlns.webpki.org/keygen2/1.0#algorithm.aes.ecb.pkcs5	
HMAC Operations	
http://www.w3.org/2000/09/xmldsig#hmac-sha1	See XML Signature
http://www.w3.org/2001/04/xmldsig-more#hmac-sha256	
Asymmetric Key Encryption	
http://www.w3.org/2001/04/xmlenc#rsa-1_5	See XML Encryption
Asymmetric Key Signatures	
http://www.w3.org/2000/09/xmldsig#rsa-sha1	See XML Signature
http://www.w3.org/2001/04/xmldsig-more#rsa-sha256	
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256	
Session Keys	
http://xmlns.webpki.org/keygen2/1.0#algorithm.sk1	See createProvisioningSession
Key Attestations	
http://xmlns.webpki.org/keygen2/1.0#algorithm.ka1	See KeyAttestation and createKeyPair
Elliptic Curves	
urn:oid:1.2.840.10045.3.1.7	Also known as “P-256”. See FIPS 186-3

Data Types

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

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

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. String literals are in this specification considered equivalent to byte[].

Return Values

All methods return a single-byte status code. In case the status is <> 0 there is an error and any expected succeeding values MUST NOT be read as they are not supposed to be available. Instead there is a second return value containing 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
`ErrorString`	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_CRYPTO`	0x05	Various cryptographic errors
`ERROR_NO_SESSION`	0x06	Provisioning session not found
`ERROR_SESSION_VERIFY`	0x07	closeProvisioningSession failed to verify
`ERROR_NO_KEY`	0x08	Key not found
`ERROR_ALGORITHM`	0x09	Unknown or not fitting algorithm
`ERROR_OPTION`	0x0A	Invalid or unsupported option
`ERROR_INTERNAL`	0x0B	Internal error

Key Policy Attributes

The following section describes the attributes issuers need to set for defining suitable protection policies for keys. Also see getKeyProtectionInfo, PrivateKeyBackup, Updatable, and EnablePINCaching.

Export Control

The following table illustrates the use of the ExportPolicy attribute:

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

Also see exportKey.

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
any	0x00	No restrictions
programmatic	0x01	PINs SHOULD only be issued through the SKS User API
trusted-gui	0x02	Keys SHOULD only be used through a trusted GUI that does the actual PIN request and API invocation

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

PIN 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 *same* PIN (synchronized)
signature+standard	0x02	Keys with Key Usage = **signature** share one PIN while all other keys share *another* PIN
unique	0x03	All keys must have *different* PIN codes

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

Keys protected by a **shared** PIN MUST be treated as having a single PIN error counter.

Continued on the next page...

PIN Pattern Control

The PatternRestrictions policy attribute specifies how PIN codes MUST NOT be designed according to the following table:

KeyGen2 Name	Mask	Comment
two-in-a-row	0x01	Flags 1124
three-in-a-row	0x02	Flags 1114
sequence	0x04	Flags 1234, 9876, etc
repeated	0x08	All PIN bytes MUST be *unique*
missing-group	0x10	The PIN MUST be "alphanumeric" and contain a mix of *letters*, *digits* and *punctuation* characters. See PIN and PUK Formats

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

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

PIN and PUK Formats

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

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

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

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

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

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.

Continued on the next page...

Biometric Protection

SKS also supports options for 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 *or* by biometrics
`combined`	0x02	The key is protected by a PIN *and* by biometrics
`exclusive`	0x03	The key is *only* protected by biometrics

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

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

Key Usage

The KeyUsage policy attribute specifies how *asymmetric keys* may be used according to the following table:

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

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

Notes

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

Methods

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

> *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.*

createProvisioningSession (1)

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 *seconds. 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 a 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`**, **`ClientSessionID`** ||
  `ServerSessionID` ||
  `IssuerURI` ||
  `ServerEphemeralKey` ||
  `EKP.PublicKey` ||
  `Updatable` ||
  `ClientTime` ||
  `SessionLifeTime` ||
  `SessionKeyLimit`))
- Define a 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.

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

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 postProvisioningDeleteKey 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 be *ignored*.

Note that individual elements featured in the *argument* (not the key) to the HMAC operations MUST use the representation described in Data Types.

On the server side the following SHOULD be performed:

Server Response Validation
• Decide if **`DeviceCertificate`** 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 (**`DeviceCertificate.PublicKey`**, // Received **`SessionAttestation`**, // Received HMAC-SHA256 (**`SessionKey`**, **`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"
                             SessionAttestation="Ob7MvaXC/rNx/rkNZJEo … 8lch/6sngIszfpElrggQfl" … >

  <ClientEphemeralKey>
    <ds11:ECKeyValue>
      <ds11:NamedCurve URI="urn:oid:1.2.840.10045.3.1.7"/>
      <ds11:PublicKey>PRZre90SQLp … 16m9FokKxV3F40Y=</ds11:PublicKey>
    </ds11:ECKeyValue>
  </ClientEphemeralKey>
  <DeviceCertificate>
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAg ... hugc53W4nNzggt2w==</ds:X509Certificate>
    </ds:X509Data>
  </DeviceCertificate>
  <ds:Signature>
    <ds:SignedInfo>
      <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
      <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#hmac-sha256"/>
      <ds:Reference URI="#_126992b6 … a8a6b484db8f">
        <ds:Transforms>
          <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
          <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        </ds:Transforms>
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
        <ds:DigestValue>yLD0zNA48Xt9xXNHuBUlK0hL51zn0SYj2IfDXm42PLc=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>aRiSdmrn/KgtjqtTReF+6DOuIemRuw2xV9yuOPAIMj8=</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
`ProvisioningSessionResponse@ID`	**`ClientSessionID`**
`ProvisioningSessionResponse@ServerSessionID`	**`Input.ServerSessionID`**
`ProvisioningSessionResponse@ClientTime`	**`Input.ClientTime`**
`ProvisioningSessionResponse@SessionAttestation`	**`SessionAttestation`**
`ClientEphemeralKey/ECKeyValue`	**`ClientEphemeralKey`**
`DeviceCertificate/X509Data/X509Certificate`	Certificate *path* from getDeviceInfo
`Signature/SignatureValue`	Created with signProvisioningSessionData

Note: A few non-SKS-related KeyGen2 elements were omitted for brevity.

Apart from needed for verifying the **`SessionAttestation`** signature the Device Certificate identifies the SKS. See Security Considerations.

closeProvisioningSession (2)

Input

Name	Type	Comment
ProvisioningHandle	int	Local handle to an *open* 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** || **ServerSessionID** || **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

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

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

Using KeyGen2 **closeProvisioningSession** is invoked as the last step of processing CredentialDeploymentRequest.

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

Data = **"Success"** || MACSequenceCounter

Note that MACSequenceCounter is *incremented* by the initial MAC operation.

Also see SessionKeyLimit.

enumerateProvisioningSessions (3)

Input

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

Output

Name	Type	Comment
Status	byte	See Return Values
ProvisioningHandle	int	Output enumeration handle
The following elements MUST only be emitted if **ProvisioningHandle** <> 0xFFFFFFFF		
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 (4)

Input

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

Output

Name	Type	Comment
Status	byte	See Return Values

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

signProvisioningSessionData (5)

Input

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

Output

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

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

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

The signature scheme is as follows:

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

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 *open* provisioning session
ID	id	*External name* of the PUK policy object. See Object IDs
PUKValue	byte[]	Encrypted PUK value. See Encrypted Data
Format	byte	Format of PUK strings. See PIN and PUK Formats
RetryLimit	short	Number of incorrect PUK values (*in a sequence*), forcing the PUK object to permanently lock up. A zero value indicates that there is no limit but that the SKS will introduce an *internal* 1-10 second delay *before* acting on an unlock operation in order to thwart exhaustive attacks
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

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

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

The **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 *open* provisioning session
ID	id	*External name* of the PIN policy object. See Object IDs
PUKPolicyHandle	int	Handle to a governing PUK policy object or zero
UserDefined	bool	True if PINs belonging to keys governed by the PIN policy are supposed to be set by the user or by the issuer. See PINValue
UserModifiable	bool	True if PINs can be changed by the user after provisioning
Format	byte	Format of PIN strings. See PIN and PUK Formats
RetryLimit	short	Non-zero value holding the number of incorrect PIN values (*in a sequence*), forcing a key to lock up
Grouping	byte	See PIN Grouping Control
PatternRestrictions	byte	See PIN Pattern Control
MinLength	byte	Minimum PIN length in *bytes*. See PIN and PUK Formats
MaxLength	byte	Maximum PIN length in *bytes*. See PIN and PUK Formats
InputMethod	byte	See PIN Input 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 locally defined PIN policy object

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

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 *open* provisioning session
ID	id	*External name* 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
PINValue	byte[]	Object which MUST depending on **PINPolicyHandle** either be of zero length, else depending on UserDefined contain a plain-text PIN value defined by the user or constitute of an encrypted PIN set by the issuer (see Encrypted Data)
BiometricProtection	byte	See Biometric Protection
PrivateKeyBackup	bool	True if the generated private key is to be outputted in **PrivateKey** for backup by the issuer
ExportPolicy	byte	See Export Control
Updatable	bool	True if the key is subject to post provisioning updates. Note that this also requires the provisioning session's **Updatable** flag to be true
DeletePolicy	byte	See Delete Control
EnablePINCaching	bool	True if middleware MAY cache PINs for this key
ImportPrivateKey	bool	True if restorePrivateKey is *expected* for this key
KeyUsage	byte	See Key Usage
FriendlyName	byte[]	String of 0-100 bytes that will be associated with this key for use in GUIs
KeyAlgorithmType	byte	Type of key to be generated: 0x00 = RSA, 0x01 = ECC
The following two elements MUST only be set for RSA keys		
RSAKeySize	short	RSA key size in bits. See getDeviceInfo
RSAExponent	int	Zero (use default) or a defined exponent. See getDeviceInfo
The following element MUST only be set for ECC keys		
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
GeneratedPublicKey	byte[]	Generated public key in X.509 DER representation
KeyAttestation	byte[]	Attestation of the authenticity of the generated public key. See Attestations
PrivateKey	byte[]	*Optional*. See PrivateKey
KeyHandle	int	Local handle to created key-pair object

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

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`** || **`Updatable`** || **`DeletePolicy`** || **`EnablePINCaching`** || **`ImportPrivateKey`** || **`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, 0xFFFFFFF, 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.

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 only in effect *within* a provisioning session.

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

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

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

How **`ServerSeed`** is applied to the generation of random numbers 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.

`KeyAttestation` vouches for that the generated key-pair actually resides in the SKS by attesting (see Attestations) the public key according to the following *Data* scheme:

Data = `ID` || **`GeneratedPublicKey`** || **`PrivateKey`**

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

Continued on the next page...

The following XML extract shows a typical key generation (provisioning) 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="encryption" MAC="gSg4chh ... H2BEE7=">
        <RSA KeySize="2048"/>
      </KeyPair>
      <KeyPair ID="Key.2" KeyUsage="authentication" MAC="nFRuPgz ... H2B429S=">
        <EC NamedCurve="urn:oid:1.2.840.10045.3.1.7"/>
      </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 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  ...  >

  <GeneratedPublicKey ID="Key.1" KeyAttestation="X2oMtrm8rRL … XyTvPuTbergHfnJw==">
    <ds:RSAKeyValue>
      <ds:Modulus>ALhBpUjJK/mSjPAe/ … fXG8z1V3mVDZTBM7eZ</ds:Modulus>
      <ds:Exponent>AQAB</ds:Exponent>
    </ds:RSAKeyValue>
  </GeneratedPublicKey>
  <GeneratedPublicKey ID="Key.2" KeyAttestation="TbergHftrm8rRL … wyTvPX2XoMunJ==">
    <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>
  </GeneratedPublicKey>

</KeyInitializationResponse>
```

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

setCertificatePath (10)

Input

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

Output

Name	Type	Comment
Status	byte	See Return Values

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

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

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

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

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

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

Continued on the next page...

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

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

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

</CredentialDeploymentRequest>
```

The table below illustrates argument mapping:

KeyGen2 Element	SKS Counterpart
`CertifiedPublicKey@ID`	**`KeyHandle.ID`**
`CertifiedPublicKey@MAC`	**`MAC`**
`CertifiedPublicKey/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 are preferably cached by the provisioning middleware, eliminating the need for enumerating keys etc.

setSymmetricKey (11)

Input

Name	Type	Comment
KeyHandle	int	Local handle to a key belonging to an *open* 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 *repeated* 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 associates a symmetric key with an already created key-pair and certificate.

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

Data = End-Entity Certificate || **SymmetricKey** || **EndorsedAlgorithm...**

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

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

EndorsedAlgorithm URIs MUST be compatible with Algorithm Support for *symmetric* keys.

There MUST only be a single symmetric key defined for a given key-pair. See Key Usage.

The following KeyGen2 fragment shows the “piggyback” arrangement:

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

  <CertifiedPublicKey ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <SymmetricKey EndorsedAlgorithms="http://www.w3.org/2000/09/xmldsig#hmac-sha1"
                  MAC="je7KiznTlQXFdUMRl … vInumZCjxSl1CrcqcGkI=">vInt09Esmg94v ...
                                                                 YU3tgIdhcNNby</SymmetricKey>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

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

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

addExtensionData (12)

Input

Name	Type	Comment
KeyHandle	int	Local handle to a key belonging to an *open* 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

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

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

Data = End-Entity Certificate || **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.

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

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

Continued on the next page...

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

PropertyBag XML Schema

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

          <!-- The 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>
```

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.

Continued on the next page...

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

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

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

</CredentialDeploymentRequest>
```

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

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

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

  <CertifiedPublicKey ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <SymmetricKey EndorsedAlgorithms="http://www.w3.org/2000/09/xmldsig#hmac-sha1"
                  MAC="je7KiznTlQXFdUMRl … vInumZCjxSl1CrcqcGkI=">vInt09Esmg94v ...
                                                                 YU3tgIdhcNNby</SymmetricKey>
    <PropertyBag Type="http://xmlns.webpki.org/keygen2/1.0#provider.ietf-hotp"
                 MAC="jIOHDgwl4dO7Kzs … uEH8MtykIS46JfiJ3N=">
      <Property Name="Counter" Value="0" Writable="true"/>
      <Property Name="Digits" Value="8"/>
    </PropertyBag>
    <Logotype MimeType="image/png"
              Type="http://xmlns.webpki.org/keygen2/1.0#logotype.application"
              MAC="+crSq5fv+7z+fx+f … ZmRnhxljO0bh0d=">iVBORw0KGgoAAAANSUhEUgAAALo
                                                      AAABKCAIAAACD … /tm/AAALjUlEQVR42u
                                                      2d6W8UyRXA+=</Logotype>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

getKeyHandle (19)

Input

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

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

restorePrivateKey (20)

Input

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

Output

Name	Type	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.

Prerequisite: see the ImportPrivateKey attribute.

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

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

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

  <CertifiedPublicKey ID="Key.1" MAC="ngSgm4cYeJnFRuPgznqE ... H2BEEIFWrM421w9SYAbY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwIBAgS … NRT+VokJJsBecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
    <PrivateKey MAC="umZCjxSl1znTlQX … CrcqcGkIvInFdUMRlje7Ki=">c2lvbj0iMSwgdmVy4wPD94bW
                                                              VyZSIiBlbmH ...  uczpkcB4bWx
                                                              My5vcd3dy53</PrivateKey>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

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

getDeviceInfo (22)

Input

Name	Type	Comment
This method does not have any input arguments		

Output

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

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

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

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

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

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

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

For ECC key generation see Elliptic Curves.

postProvisioningDeleteKey (50)

Input

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

Output

Name	Type	Comment
Status	byte	See Return Values

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

Data = Post Provisioning MAC

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

postProvisioningUpdateKey (51)

Input

Name	Type	Comment
KeyHandle	int	Local handle to a new key belonging to an *open* provisioning session
KeyHandleOriginal	int	Local handle to the old (=target) key
MAC	byte[32]	Vouches for the integrity and authenticity of the operation

Output

Name	Type	Comment
Status	byte	See Return Values

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

Data = End-Entity Certificate || Post Provisioning MAC

The new (update) 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 PINs as the only exception.

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

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
The following element MUST only be emitted if **KeyHandle** <> 0xFFFFFFFF		
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 addExtensionData
ExtensionType...	byte[]	KeyGen2 **Type** URI. *Repeated* object	

getKeyAttributes returns key attributes for provisioned keys..

Key Usage determines if the 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
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	
Updatable	bool	
DeletePolicy	byte	
EnablePINCaching	bool	
ImportPrivateKey	bool	

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

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

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

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

getExtensionObject (103)

Input

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

Output

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

getExtensionObject 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.

deleteKey (104)

Input

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

unlockKey (105)

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 (106)

Input

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

Output

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

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
PIN	byte[]	Holds a PIN value or is of zero length indicating that no PIN is supplied
HashedData	byte[]	Hashed data to be signed. Length MUST match the hash algorithm

Output

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

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

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

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 a MAC or attestation argument 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.

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 Issuance"`**, 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 target key's Updatable flag is set.

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

Missing Methods

Symmetric key encryption and hmac, asymmetric decryption, and some utility methods

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.

Remote Key Lookup

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

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

```
<CredentialDiscoveryRequest ... >

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

</CredentialDiscoveryRequest>
```

The example works as follows:

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

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

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

Security Considerations

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

However, SKS introduces a concept sometimes referred to as “air-tight” provisioning which has some specific security characteristics. One of the most critical operations in SKS is the creation of 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 will when receiving the response to the provisioning session request, detect if it has been manipulated and *cease the rest of the operation*.

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

Replay attacks are thwarted by the 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.

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

That any issuer (after the user's grant), 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 adds a signature for the provisioning session response containing the fingerprint of the HTTPS server-cert allowing the issuer to verify 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 notion of policy, it is up to the *issuer* to decide 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.

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

References

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

Acknowledgments

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

Author

Anders Rundgren
anders.rundgren@telia.com