Efficient Provisioning of Complex Structures over Unsecured Channels

Background to the Invention

This invention is an intrinsic part of a scheme called SKS/KeyGen2 where SKS represent a comparatively simple (but secure) electronic device that can hold cryptographic keys (e.g. “smart card”) which can be use for authentication etc. KeyGen2 is a matching high-level protocol for provisioning an SKS device with such keys.

Since the provisioning may take place over the Internet an issuer of keys like a bank, government agency, employer etc., do usually not have a particularly good control over the security in the computer to which keys are to be provisioned to.

To make this process more trustworthy the SKS has been equipped with a built-in key which can be used for vouching the brand and even the individual serial number of the SKS to the issuer. This enables an issuer to at least determine if keys will be stored in a trustworthy key-container or in a container having unknown characteristics.

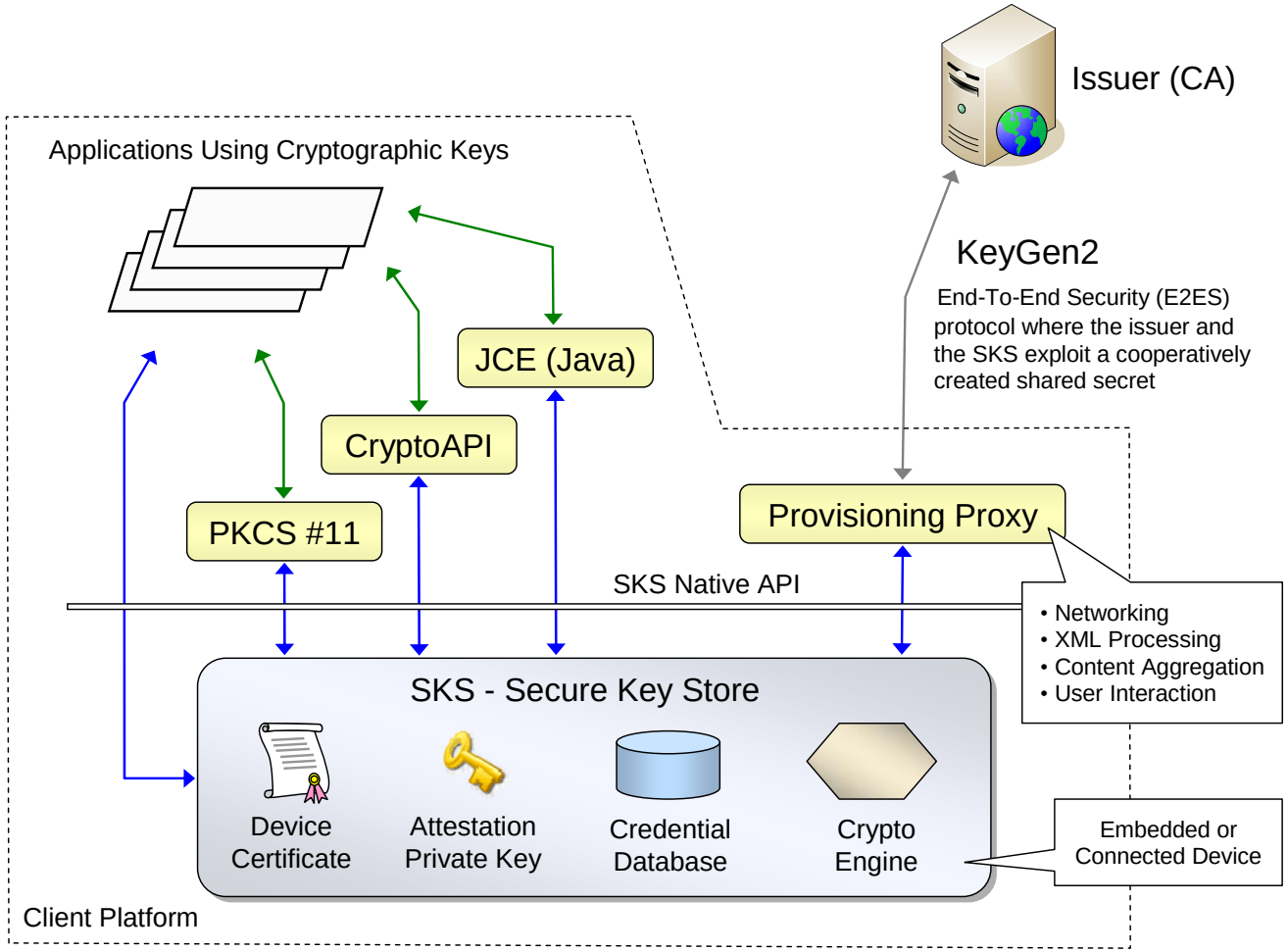
However, this is not enough because a provisioning session consists of multiple steps and an adversary (“man-in-the-middle”) could modify and/or redirect the information flow after the initial authentication. Although TLS (Transport Level Security) could be used to address this problem, TLS would greatly complicate both the design and administration of an SKS. In addition, the use of TLS would effectively disable the ability letting a user define a PIN-code associated with a key to be provisioned unless the SKS also have a built-in user-interface.

Due to this, many existing key-provisioning systems rather rely on adding MAC (Message Authentication Code) checksums to each message interchange. Sensitive data is usually encrypted using a key derived from the same key that is used for MACs.

This document describes how you by combining MACs with a virtual name-space can create complex structures in an SKS not only in a secure manner but efficiently as well

SKS/KeyGen2 Architecture

The figure below shows the components involved in this description. The parts of SKS that deal with the actual usage of cryptographic keys (in the upper left corner of the picture) are not elaborated on here because they are more or less “standard”.



The *Device Certificate* and *Attestation Private Key* are built-in objects having a single task: vouching for the authenticity of the SKS device during provisioning sessions. The *Credential Database* stores keys and associated data while the *Crypto Engine* performs the cryptographic operations needed.

The SKS native API is *by design* primitive to keep the device cost down, while the KeyGen2 protocol is comparatively high-level. A proxy is therefore necessary as a “translator” between the two worlds. The proxy does most of the “heavy lifting”. It also provides user interaction support during the provisioning process. However, this scheme does not presume that the proxy is entirely “honest”, otherwise it wouldn’t be an end-to-end security solution. The true end-points still constitute of the Issuer and the SKS.

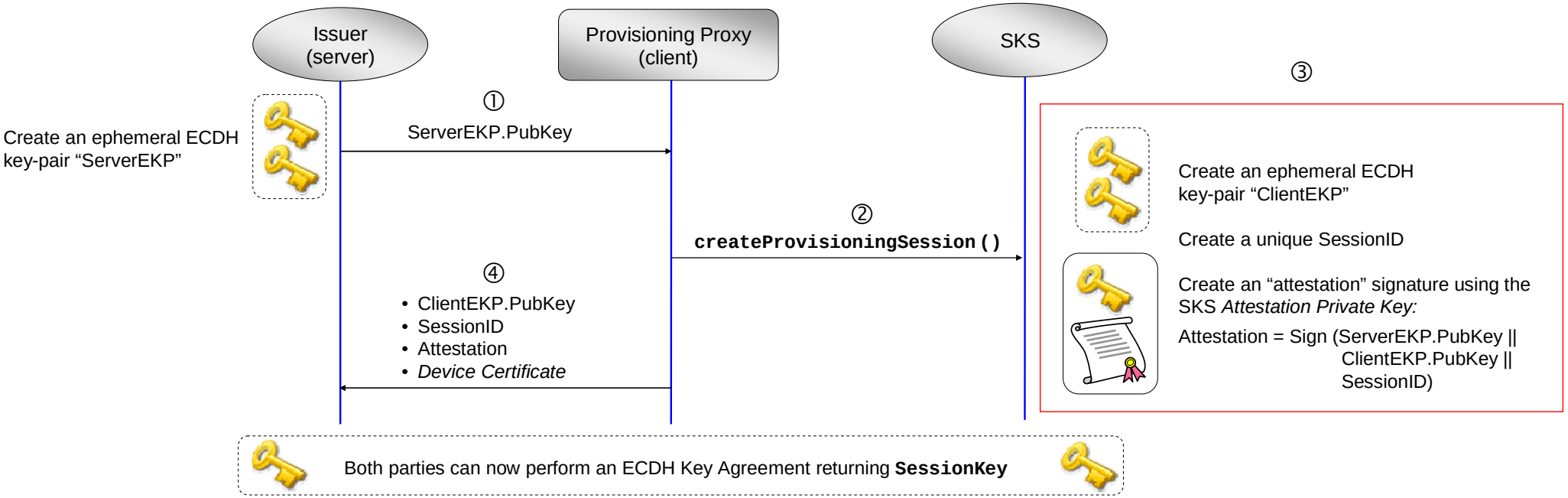
The rest of this document shows how the Issuer, Provisioning Proxy and SKS interact in order to maintain security and efficiency.

Step #1: Creating a Shared SessionKey

In order to perform multiple interactions using light-weight symmetric-key cryptography, the Issuer and the SKS need a shared key. Shared keys can be established in many ways. The scheme used in SKS/KeyGen2 is simply provided as a reference. The SKS/KeyGen2 concept builds on a specific trust model which is as follows:

The Issuer trusts/recognizes a set of SKS devices based on their *Device Certificates*. The SKS however, does not have any opinion about issuers and accepts provisioning requests from anybody. The actual granting of an Issuer is therefore supposed to be done by the *user* through a GUI which is a part of the Provisioning Proxy.

Not until the user hits OK, the actual provisioning process can start. When this happen the Provisioning Proxy sends an (not shown) acknowledge message to the Issuer which then responds with a set of Issuer-Proxy-SKS message exchanges where the first one establishes a mutual **`SessionKey`**. This process is described in detail on the next page.



The Issuer begins by creating an ephemeral ECDH (Elliptic Curve Diffie-Hellman) key-pair and sending it down to the Provisioning Proxy which in turn translates the request to the SKS API format and invokes a method called **`createProvisioningSession`**. This method creates a number of data objects in an “atomic” fashion:

```
ProvisioningSession session = createProvisioningSession (ServerEKP.PubKey)
```

The **`ProvisioningSession`** object contains the following elements:

```
PublicKey ClientEphemeralKey;   // ClientEKP.PubKey.  Public part of the ephemeral ECDH key-pair created by the SKS
String SessionID;               // SKS-created SessionID which is used externally for identifying a particular session
byte[] Attestation;             // Asymmetric signature created by the SKS Private Attestation Key
int handle;                     // Low-level “handle” to the created session which is only used locally by the Provisioning Proxy
```

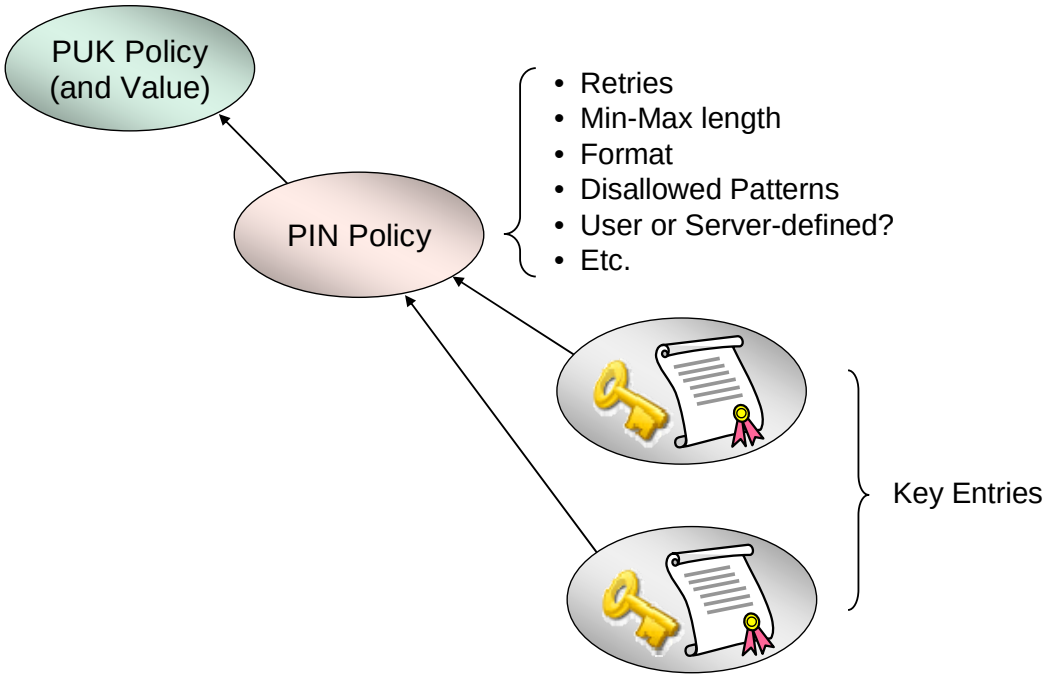
After the method call has returned, the Provisioning Proxy sends back the data shown in the figure. By running the attestation signature over both the server and client public keys, the Issuer (server) is assured that both ECDH keys where participating in the DH operation which is necessary for securing the **`SessionKey`**. The SKS *Device Certificate* is acquired by the Provisioning Proxy through another method since it is static. Each created **`SessionKey`** lives inside of the SKS until the session is explicitly closed or an error occurs. As can be seen in the diagram the Provisioning Proxy does neither handle private keys nor perform cryptographic operations. The exact format of the message exchange above between the Issuer and the Provisioning Proxy is not important but is included in the appendix of this document for completeness.

By using ephemeral keys in both ends, entropy of **`SessionKey`** is not dependent on one party only.

For brevity the above is slightly simplified; the actual implementation adds a set of security-enhancing elements including a Client time-stamp, Server-side SessionID, Provisioning algorithm ID, Session expiration time, Invoking URL, Session usage-limit, and an ECDH KDF but these are not central for this specification

Sample Object Structure

Although the scheme presented in this document may certainly have other applications, KeyGen2 is a dedicated protocol for provisioning cryptographic keys and associated PIN and PUK codes. The sample structure that is used throughout this document consists of two keys with certificates that are managed by a single PIN code which in turn is managed by a PUK.



Since we now have a **`SessionKey`** we are finally ready for creating the actual data! See next page.

Step #2: Creating the Sample Object Structure in an End-to-End Secured Fashion

KeyGen2 Request Sent by the Issuer	Corresponding SKS Operations Issued by the Provisioning Proxy

<KeyCreationRequest **SessionID**="_56af4735bc9af0d8e" **...** >
 	
<PUKPolicy ID=**"PUK.1"** Format="numeric" RetryLimit="3"
Value="mjRuO …1Oqw" **MAC**="uPqE ... HIF=">
 	
<PINPolicy ID=**"PIN.1"** Format="numeric" Grouping="shared"
MaxLength="8" MinLength="4"
PatternRestrictions="three-in-a-row sequence"
RetryLimit="3" **MAC**="gz56 ... 2B7=">
 	
<KeyEntry ID=**"Key.1"** AppUsage="authentication" **MAC**="nFRuPgz ... H2B429S=">
<**EC NamedCurve**="urn:oid:1.2.840.10045.3.1.7"/>
</KeyEntry>
 	
<KeyEntry ID=**"Key.2"** AppUsage="encryption" **MAC**="gSg4chh ... H2BEE7=">
<**RSA KeySize**="2048"/>
</KeyEntry>
 	
</PINPolicy>
 	
</PUKPolicy>
 	
</KeyCreationRequest>
	// Create PUKPolicy object
	int puk_handle = createPUKPolicy (session.handle, ”PUK.1”, *OtherArguments*, MAC);
	
	// Create PINPolicy object
	int pin_handle = createPINPolicy (session.handle, ”PIN.1”, puk_handle, *OtherArguments*, MAC);
	
	// Create key-pair object
	KeyData key_1 = createKeyEntry (session.handle, ”Key.1”, pin_handle, *OtherArguments*, MAC);
	
	// Create key-pair object
	KeyData key_2 = createKeyEntry (session.handle, ”Key.2”, pin_handle, *OtherArguments*, MAC);

Explanation: The object structure is in KeyGen2 expressed through *embedding*: The PUKPolicy object embeds a PINPolicy object which in turn embeds two KeyEntry objects. However, SKS does not understand XML or object embedding, it only deals with simple stuff like object handles. In addition an SKS can only process one object at a time. Due to that the Provisioning Proxy rearranges the request so that it becomes a set of low-level SKS calls. Since this could be done in an incorrect way (deliberately or not), the Issuer assigns a MAC to each XML construct. This MAC is read and verified by the SKS. The MACs used for the KeyCreationRequest all follow the pattern:

MAC = HMACSHA256 (**`SessionKey`** || *MethodName* || **`MACSessionCount`**, *Data*)

- *MethodName* holds the name of the expected SKS method like “**`createPUKPolicy`**”. This binds the method with the *Data*.
- **`MACSessionCount`** holds a session-internal counter which is incremented for each call. This thwarts call sequence changes or exclusions.
- *Data* holds a concatenated byte-array consisting of the input arguments where the arguments have been converted to simple types like 1-,2-, and 4-byte integers and byte-arrays preceded by 2-byte length indicators. *An Issuer must (of course) use exactly the same conventions as specified by the SKS when creating KeyGen2 objects*.

Note: The locally created object handles **`puk_handle`** and **`pin_handle`** are not included in the MAC calculations because the Issuer can have no idea what these values could be. This is where the virtual name-space comes in. The following shows the actual MAC for the **`createPINPolicy`** call:

HMACSHA256 (**`SessionKey`** || “**`createPINPolicy`**” || **`MACSessionCount`**, “**`PIN.1`**” || “**`PUK.1`**” || *OtherArguments*)

That is, the **`createPINPolicy`** method uses the **`puk_handle`** argument to locate an *internally stored* object that must be of the type PUKPolicy. When the local MAC is created by the SKS the identity of the located PUKPolicy object is included which this case is “**`PUK.1`**”. Any deviation from what the Issuer “ordered” will thus be detected by the SKS. A MAC or other argument error should in a proper SKS implementation result in a terminated provisioning session and full “rollback” of created data.

Because object names like “PIN.1” are meant to be entirely local to the provisioning session (“virtual”), issuers do never have to bother about local object handles or worry about possible name collisions with existing objects

User-defined PINs

Although not shown here, user-defined PIN-codes are also inputted during the key request phase, supervised by the Provisioning Proxy. The proxy interprets the PIN policy and only returns when it is fulfilled. The actual value is supplied as an argument to **`createKeyEntry`**. To maintain true end-to-end security and data integrity, the SKS also checks PIN syntax and aborts the session on invalid PIN data. User-defined PIN-codes are excluded from the MAC operations; their values are replaced by the literal string "**`N/A`**".

Encrypted Data from the Issuer to the SKS

PUK-codes (the "Value" attribute in the PUKPolicy XML fragment) as well as Issuer-defined PINs are encrypted using a key derived from **`SessionKey`**. When featured in MAC operations the encrypted value is used. Note that the Provisioning Proxy (unlike the SKS) cannot decipher Issuer-encrypted data since the proxy does neither have direct nor indirect access to **`SessionKey`**. The encryption method used in the SKS/KeyGen2 scheme is as follows:

EncryptedData = AES256-CBC (HMACSHA256 (**`SessionKey`**, "**`Encryption Key`**"), *UnencryptedData*)

Summary of Benefits

Compared to the naive approach (making a separate networked call for each object and using the returned object handle as argument to the next call), the object aggregation and linking scheme described on the previous page reduces the number of network "roundtrips" from 4 to 1. This is a core feature of this invention

Step #3: Returning Generated Data to the Issuer

After processing the KeyCreationRequest the Provisioning Proxy is supposed to return the public keys associated with the created key entries. The public keys are supplied in the **KeyData** objects created by the calls to **createKeyEntry**. A **KeyData** object contains the following:

```
PublicKey GeneratedPublicKey;   // Generated public key in X.509 encoding
byte[] Attestation;             // Symmetric signature (MAC) created by the SessionKey
int handle;                     // Low-level "handle" to the created key entry which is only used locally by the Provisioning Proxy
```

The PublicKey and Attestation elements are XML-encoded before sent to the Issuer:

```
<KeyCreationResponse SessionID="_56af4735bc9af0d8e"  ... >

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

</KeyCreationResponse>
```

The **Attestation** element is an SKS-created MAC using the formula:

Attestation = HMACSHA256 (**SessionKey** || "**Device Attestation**" || **MACSessionCount**, **ID** || **GeneratedPublicKey**)

ID is the virtual name assigned to the actual key entry like "**Key.1**".

After receival the Issuer should verify that the number of keys matches that of the request, that keys are of the requested type, and that the **Attestation** signatures match.

The **Attestation** signature scheme effectively replaces the PoP (Proof of Possession) method commonly featured in PKI enrollment schemes. In fact, an **Attestation** is considerably stronger security-wise since it binds a generated public key to a *specific container*. This scheme can also be used for secure automated renewals. As a contrast, the currently established methods provide no cryptographic assurance regarding the origin of the *new* key-pair!

Step #4: Adding Certificates and Session Termination

After the Issuer has verified the authenticity of the received public keys, it creates matching certificates and sends them back to the Provisioning Proxy,

```
<ProvisioningFinalizationRequest SessionID="_56af4735bc9af0d8e"
                                 MAC="GcaqnRxW ... 8kabXDgWr="
                                 Nonce="aqnPb6x0 ... kms34gfW2="  ... >

  <CertficatePath ID="Key.1" MAC="Su74dw4 ... kUm643f=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCSAc6Aw ... 7hyrdrrw==</ds:X509Certificate>
    </ds:X509Data>
  </CertficatePath>
  <CertficatePath ID="Key.2" MAC="ngSgmRuPE ... HlFWrM421wY=">
    <ds:X509Data>
      <ds:X509Certificate>MIIC2TCCAcGgAwS ... NRecyALgeT0Dw==</ds:X509Certificate>
    </ds:X509Data>
  </CertficatePath>

</ProvisioningFinalizationRequest>
```

Each KeyGen2 CertificatePath element corresponds to an SKS call:

setCertificatePath (**KeyData.handle, CertificatePath, MAC**)

KeyData.handle holds the matching handle created during the KeyCreationRequest. **X509Certificate[] CertificatePath** holds the issued end-entity certificate plus optional upper certificates belonging to the same certificate path. The exact MAC operation is as follows:

MAC = HMACSHA256 (**SessionKey** || "**setCertificatePath**" || **MACSessionCount**, **ID** || **GeneratedPublicKey** || **CertificatePath)**

ID is the virtual key name like "**Key.1**". **GeneratedPublicKey** is the generated public key associated with the key entry.

Due to the fact that the SKS has no predefined idea of when a session is ready, the Issuer must as a final measure send a session termination request. To avoid unnecessary network roundtrips, the MAC and Nonce attributes in the request header hold the input arguments to the following call which is executed immediately after the body of the request has been processed:

byte[] Attestation = **closeProvisioningSession** (**session.handle**, **Nonce, MAC**)

This method "commits" all operations to the SKS container after first verifying the consistency of the provisioned data. The MAC is calculated as follows:

MAC = HMACSHA256 (**SessionKey** || "**closeProvisioningSession**" || **MACSessionCount**, **SessionID** || **Nonce**)

If **closeProvisioningSession** succeeds a result in the form of "receipt" is created by:

HMACSHA256 (**SessionKey** || "**Device Attestation**" || **MACSessionCount**, **Nonce**)

Only if the Issuer receives a valid receipt it can be assured that the provisioning process has been carried out as planned. The receipt also serves as a way to gather information about the quality of the Issuer operations. The actual message sent to the Issuer holding the receipt only contains the **SessionID** and the **Attestation**:

```
<ProvisioningFinalizationResponse SessionID="_56af4735bc9af0d8e" Attestation="gEWzCCD ... 0w81L/ls2W0UYNE="/>
```

Secure Extensions through “Piggybacking”

The SKS/KeyGen2 scheme allows you to securely “piggyback” other data to a key entry in step #4 by object embedding on the XML level and binding with MACs to the associated end-entity certificate on the SKS level. Each object type has its own SKS method. SymmetricKey for example is a key import-method where the value is encrypted as described for PUK and PIN codes.

```
<ProvisioningFinalizationRequest SessionID="_56af4735bc9af0d8e"
                                 MAC="GcaqnRxW ... 8kabXDgWr="
                                 Nonce="aqnPb6x0 ... kms34gfW2="  ... >

 <CertficatePath ID="Key.1" MAC="ngSgmRuPE ... HlFWrM421wY=">
   <ds:X509Data>
     <ds:X509Certificate>MIIC2TCCAcGgAgS ... NRVokJJsgeT0Dw==</ds:X509Certificate>
   </ds:X509Data>

   <SymmetricKey MAC="je76Ki ... vInuGkI=">vInt09gf4v ... YdhcNNgby</SymmetricKey>

   <PropertyBag Type= "http://xmlns.webpki.org/keygen2/1.0#provider.ietf-hotp"
                MAC="jIOHDgwI4dO7Kzs ... uEH8MtykIS46JfiJ3N=">
     <Property Name="Counter" Value="0" Writable="true"/>
     <Property Name="Digits" Value="8"/>
   </PropertyBag>

   <Logotype MIMEType="image/png"
             Type="http://xmlns.webpki.org/keygen2/1.0#logotype.application"
             MAC="+crSq5ff ... ZmRnhx0d=">iAAABKCAD ... tmAALRA=</Logotype>
 </CertficatePath>

</ProvisioningFinalizationRequest>
```

SKS Counterpart:

Method (**`KeyData.handle`**,
Arguments,
HMACSHA256 (**`SessionKey`** || “*Method*” || **`MACSessionCount`**,
`EECertificate` || *Arguments*))

Notes:

Method is the actual name of the extension method

`EECertificate` is the first certificate in a **`CertificatePath`**

Appendix

The following are the two messages in XML format exchanged between the Issuer and the Provisioning Proxy detailed on page 4. The circled number represents the message’s place is the timing diagram.

①

```
<ProvisioningInitializationRequest  ... >
 <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>
</ProvisioningInitializationRequest>
```

④

```
<ProvisioningInitializationResponse Attestation="Ob7MvaXCZJEo ... 8lch/6sngIszfpElfl"
                                    SessionID="_56af4735bc9af0d8e"  ... >
 <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>MIIC2TCCAcGgg ... hugc53Wt2w==</ds:X509Certificate>
  </ds:X509Data>
 </DeviceCertificatePath>
</ProvisioningInitializationResponse>
```