# Subject Key Attestations in KeyGen2

For on-line (remote) provisioning of keys to Security Elements (SEs), like Smart Cards, there is a wish by issuers to be able to securely verify that the public key part they receive for inclusion in a certificate, indeed was (together with its private counterpart), generated *inside* of a “genuine” SE. In fact, *for compliance with security standards like FIPS 140-2, such a facility would be a prerequisite*. This document shows how key-attestations performed by an SE equipped with an *embedded private key and certificate for “credential bootstrapping”* have been integrated in the KeyGen2 protocol.

By “piggybacking” secret data on attested asymmetric key-pairs, the described key attestation mechanism becomes *equally applicable to downloadable symmetric keys*.

Below is a sample featuring a single key which is used for illustrating the key-attestation support.

### Request Phase

In the request the issuer declares requirements on the generated key(s) to the provisioning client.

```
<?xml version="1.0" encoding="UTF-8"?>
<KeyOperationRequest SubmitURL="https://ca.mybank.com/keycenter/deploy"
                     ID="R.11c6ffa38d96804bb04f9d79913"
                     ClientSessionID="S.11c6ffa3f23b544f7a3ae4b3409"
                     ServerTime="2009-03-03T21:03:04+01:00"
                     xmlns="http://xmlns.webpki.org/keygen2/beta/20090301#">

  <CreateObject>

     <KeyPair ID="Key.1" KeyUsage="authentication" Exportable="false">
①       <RSA KeySize="2048"/>
     </KeyPair>

  </CreateObject>

</KeyOperationRequest>
```

### Response Phase

In the subsequent response, the client returns generated keys including their associated key attestations.

```
<?xml version="1.0" encoding="UTF-8"?>
<KeyOperationResponse ClientTime="2009-03-03T21:03:17+01:00"

                   ②  ID="S.11c6ffa3f23b544f7a3ae4b3409"

                   ③  ServerSessionID="R.11c6ffa38d96804bb04f9d79913"

                      RequestURL="https://ca.mybank.com/keycenter/generate"
                      ServerCertificateFingerprint="AQQFAwYHCA … FAwYHCAM="
                      ServerTime="2009-03-03T21:03:04+01:00"
                      SubmitURL="https://ca.mybank.com/keycenter/deploy"
                      xmlns="http://xmlns.webpki.org/keygen2/beta/20090301#"
                      xmlns:ds="http://www.w3.org/2000/09/xmldsig#">

④  <GeneratedPublicKey ID="Key.1" KeyAttestation="Rvo9DnMcC … ElAiigypxb4=">
      <ds:KeyInfo>
         <ds:KeyValue>

            <ds:RSAKeyValue>
⑤              <ds:Modulus>AlyJ4QCz+0A … HRR1hOws8=</ds:Modulus>
               <ds:Exponent>AQAB</ds:Exponent>
            </ds:RSAKeyValue>

         </ds:KeyValue>
      </ds:KeyInfo>
    </GeneratedPublicKey>

⑥  <EndorsementKey KeyAttestationAlgorithm="http://xmlns.webpki.org/keygen2/1.0#algorithm.key-attestation-1">

      <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="#S.11c6ffa3f23b544f7a3ae4b3409">
            <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>97MVXXIMMm … locGhdROPik=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>

    ⑦  <ds:SignatureValue>gqVvmXw8dO … Sd/nurzR+Xw=</ds:SignatureValue>

        <ds:KeyInfo>
          <ds:X509Data>
            <ds:X509IssuerSerial>
              <ds:X509IssuerName>CN=Mobile Device Root CA,DC=webpki,DC=org</ds:X509IssuerName>
              <ds:X509SerialNumber>1221075403312</ds:X509SerialNumber>
            </ds:X509IssuerSerial>
            <!-- Signer DN: "CN=Device Type 1AK4, SerialNumber=75035, DC=webpki, DC=org" -->

        ⑧  <ds:X509Certificate>MIIC/jCCAea … ySVpfRGH0=</ds:X509Certificate>

          </ds:X509Data>
        </ds:KeyInfo>
      </ds:Signature>

    </EndorsementKey>

  </KeyOperationResponse>
```

**Detailed Operation**

The `KeyAttestation` attribute ④ holds a key attestation signature for the binary object constituting of the concatenation of a `Nonce` object, an `Exportable` flag, a `KeyUsage` variable, and the generated public key ⑤ in ASN.1 DER encoded format like the following: Sign (`Nonce` || `Exportable` || `KeyUsage` || public key). `Nonce` is the SHA256 hash of the UTF-8 encoded string created by concatenating the content of the following XML attributes, where each attribute value has been appended by a trailing NULL (\0) character:

```
④ KeyOperationRequest/CreateObject/KeyPair/@ID
② KeyOperationRequest/@ClientSessionID
③ KeyOperationRequest/@ID
```

Note: Attribute order is significant! `Exportable` ① is a byte [0..1] telling if the generated key is exportable or not. This characteristic is defined during the request phase of a key-generation operation and can be checked by the issuer for compliance. For details on `KeyUsage` ① see section *Key Usage*.

The actual key attestation signature is created by applying the KeyGen2 `key-attestation-1` ⑥ signature algorithm using the private key of the SE “endorsement“ (device) key ⑧. KeyGen2 `key-attestation-1` signatures are technically identical to RSASSA-PKCS1-V1.5 / SHA256 signatures except for the padding format which is slightly different *to securely distinguish key attestation signatures* (which MUST only be creatable during key generation), from *other signatures* performed by *the same key*. The following shows the padding differences in detail:

```
EMSA-PKCS1-v1_5: EM = 0x00 || 0x01 || PS || 0x00 || T
EMDIAS-PKCS1:    EM = 0x00 || 0x01 || PS || 0x00 || 'D' || 'I' || 'A' || 'S' || T
```

DIAS is an acronym for Device Internal Attestation Signature. DIAS may be applied to other data which can be securely derived from the internals of an SE.

Note that the private key of the endorsement key ⑧ for *message integrity* purposes, also *signs the entire response*, using an enveloped XML signature with the *standard PKCS #1* format ⑦.

### Rationale for DIAS and Multiuse Device Keys

Among the advantages of using a single device key for multiple purposes as shown in the example we find:

- Creation of a unified device identity including device “fingerprint” (certificate hash)
- Simplified provisioning protocols; less device key identity information to transfer
- Reduced issuer code complexity; no need to “correlate” different device keys
- Enhanced cryptographic performance on the issuer side; only one certificate-path to validate
- Device fingerprints support improved “manual” enrolment and verification procedures
- Elimination of standardized certificate profiles for separating different device keys; any certificate will do

It is vital though that the SE does not permit direct API access to “raw” (unformatted) private-key operations using the SE device key, because that would enable fraudulent key-attestations by external “malware”!

### Key Usage

During the request phase of key generation (`KeyOperationRequest` in KeyGen2), the issuer indicates what cryptographic operations each requested key is supposed to support. The SE is obliged to honor these constraints and show that through matching key attestations. In case of doubt of what PKI-using applications actually need, the “universal” key usage constraint is recommended.

Note: the key usage described in this document is not to be confused with the X.509 key usage extension because the latter is primarily intended to be handled at the application-level (“filtering”) rather than in an SE.

The following table shows the currently defined key usage attributes and their function:

| Value | KeyGen2 “KeyUsage” Attribute | Private Key Constraints |
|---|---|---|
| 0 | `signature` | the key is only allowed to perform PKCS #1 signature operations |
| 1 | `authentication` | the key is only allowed to perform PKCS #1 signature and decryption operations |
| 2 | `encryption` | the key is only allowed to perform PKCS #1 decryption operations |
| 3 | `universal` | no restrictions apply |
| 4 | `transport` | the key is disabled |
| 5 | `piggybacked-symmetric-key` | the key is only allowed to decrypt “piggybacked” symmetric keys for SE-based storage or sealing *through dedicated SE API calls* |

### Symmetric Key Support

The purpose of the `piggybacked-symmetric-key` key usage constraint is to facilitate *secure transferal of symmetric keys where the keys are protected all the way from the issuing server to a final destination in authenticated secure storage*. Although SE device keys (or dedicated SE encryption keys), could be used for the same purpose this would limit the use of decryption to SE-internal operations only, otherwise keys could be exposed in clear outside of the SE. The use of instance-specific encryption keys enables a granular way of decrypting server data which may be necessary since not all kinds of data fits within an SE or even have an SE as the target.

The next section, *Deployment Phase* briefly outlines PKI and symmetric key deployment options.

**Deployment Phase**

After the issuer has received one or more attested key-pairs, credentials are created and sent back to the client. The following KeyGen2 message represents a "plain-vanilla" PKI deployment operation:

```
<?xml version="1.0" encoding="UTF-8"?>
<CredentialDeploymentRequest SubmitURL="https://ca.mybank.com/keycenter/finish"
                             ID="R.11c6ffa38d96804bb04f9d79913"
                             ClientSessionID="S.11c6ffa3f23b544f7a3ae4b3409"
                             ServerTime="2009-03-03T21:03:04+01:00"
                             xmlns="http://xmlns.webpki.org/keygen2/beta/20090301#"
                             xmlns:ds="http://www.w3.org/2000/09/xmldsig#">

  <CertifiedPublicKey ID="Key.1">
    <ds:X509Data>
      <ds:X509Certificate>MIIDajCCAlKgA … nVRYf78Vo=</ds:X509Certificate>
    </ds:X509Data>
  </CertifiedPublicKey>

</CredentialDeploymentRequest>
```

After receival the client simply verifies session data and that the certified public key is matching the previously generated key.

Deployment of symmetric keys is slightly more complex due to the piggybacking scheme. The following XML fragment shows how symmetric keys are packaged:

```
<CertifiedPublicKey ID="Key.2">
  <ds:X509Data>
    <ds:X509Certificate>MIIC5jCCAc6 … dlS60o4F+0=</ds:X509Certificate>
  </ds:X509Data>
  <PiggybackedSymmetricKey EndorsedAlgorithms="http://www.w3.org/2000/09/xmldsig#hmac-sha1"
                           MAC="14z1RfdoVeDqYfSviPWZD4c2AL4=">
    <xenc:EncryptedKey>
      <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
      <xenc:CipherData>
        <xenc:CipherValue>cymS8WW+a92n/ … WcqAxD+ndOPo=</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedKey>
  </PiggybackedSymmetricKey>
  <PropertyBag Type="http://xmlns.webpki.org/keygen2/1.0#provider.ietf-hotp">
    <Property Name="Counter" Value="0" Writable="true"/>
    <Property Name="Digits" Value="8"/>
  </PropertyBag>
</CertifiedPublicKey>
```

A symmetric key is encrypted by a generated public key and decrypted by the corresponding SE-based private key. To enable key integrity verification, the issuer creates a MAC by performing an HMAC-SHA256 over a sorted list of the `EndorsedAlgorithms` URIs in UTF-8 encoding, where each URI has been appended by a NULL (\0) character, finally followed by the unencrypted key, and using a nonce created exactly like for `key-attestation-1` signatures as the HMAC input key. The MAC is supposed to be checked by the SE during the final (*fully atomic NB*) provisioning stage. The SE is also required to honor the `EndorsedAlgorithms` specification in order to prevent possible symmetric key "misuse".

The optional `PropertyBag` element contains key attributes that the key-using application may need. The `Type` URI indicates that the sample key is intended for HOTP (RFC 4226) operation.

Note: *keys are always certified by the issuer regardless if they are symmetric or asymmetric, making certificates serve as universal key GUIDs*.

## Security Considerations

Because the DIAS scheme defies a “best practice” rule in the cryptography community; namely not reusing a key for different purposes like for encryption and signatures, the following section is intended to provide a reasonable explanation why key reuse is not considered an issue for the special-purpose application described in this document.

The only publicly known key reuse attacks including Bleichenbacher's "million message attack", that actually have been *verified* were all based on incorrectly implemented algorithms or *servers* (DIAS is a *client* scheme) acting as “oracles”, not on breakthroughs in public key mathematics. It is likely that a broken implementation exposes *any* cryptosystem to potential attacks.

Below are the two main objections to key reuse and how these objections have been dealt with in KeyGen2. Also see section *Rationale for DIAS and Multiuse Device Keys* regarding SE device private key constraints*.*

*“You shouldn’t encrypt using a signature key”*

Since anybody receiving a signature public key can perform an arbitrary number of encryptions with it, *there are no technical means protecting signature public keys from “algorithm misuse”*. A reason why it sometimes still makes sense separating encryption and signature keys is the fact that encryption private keys for *personal use*, needs to be backed-up in order to restore encrypted data when/if keys are lost. However, private key backup does not apply to *embedded SE device keys* which are exclusively designated for credential bootstrapping and device attestations, rather than for securing user data like e-mail.

*“You shouldn’t sign using an encryption key”*

The rationale for this is presumably that encrypted data could be revealed by correlating signature output with cipher-text. The reasons why this does not apply to key provisioning as performed by KeyGen2 include:

- Since the data in question is always *padded* using different padding schemes for encryption, signatures and DIAS, a successful attack effectively requires that signature hashes are “unwrapped” as well as mathematical advances in prime number factorization *which probably would be disastrous for the Internet as a whole*
- A key-provisioning scheme like KeyGen2 only deals with extremely high-entropy, random data, making any kind of data correlation *for practical purposes* infeasible
- A client-oriented scheme is not susceptible to massive *direct* attacks because it simply doesn’t have the processing power needed to break even relatively “simple” crypto like 768-bit RSA keys

Although maybe somewhat less significant, it is worth mentioning that KeyGen2 is intended to be used together with HTTPS, further limiting the attack-space.

In addition to these precautions, SE encryption using the device key is only used (in KeyGen2 NB), for wrapping random 256-bit AES key-encrypting-keys for protecting preset PINs and PUKs, never for handling cryptographic user-keys. For details on the latter, please consult section *Deployment Phase*.

*The actual security issues in the client-end seem to be of a more mundane nature such as careless users and not entirely trustworthy operating systems.*

## Attestation Signature Validation Code Sample

On the following pages there is authentic Java code showing how KeyGen2 key attestation signatures can be validated.

```
package org.webpki.keygen2;

import java.io.IOException;

import javax.crypto.Cipher;

import java.security.GeneralSecurityException;
import java.security.MessageDigest;
import java.security.PublicKey;
import java.security.interfaces.RSAKey;


public abstract class KeyAttestationUtil
  {
    static final String SHA256             = "SHA-256";

    static final String UNFORMATTED_RSA    = "RSA/ECB/NoPadding";

    static final byte[] PS_END_SEQUENCE    = new byte[] {(byte)0x00, (byte)'D',  (byte)'I',  (byte)'A',  (byte)'S'};

    static final byte[] DIGEST_INFO_SHA256 = new byte[] {(byte)0x30, (byte)0x31, (byte)0x30, (byte)0x0d, (byte)0x06,
                                                         (byte)0x09, (byte)0x60, (byte)0x86, (byte)0x48, (byte)0x01,
                                                         (byte)0x65, (byte)0x03, (byte)0x04, (byte)0x02, (byte)0x01,
                                                         (byte)0x05, (byte)0x00, (byte)0x04, (byte)0x20};

    /**
     * Create a KA1 package for signing or verification.
     * @param attesting_key The attesting (certifying) private or public key.
     * @param attested_public_key The attested (certified) key.
     * @param exportable True if exportable.
     * @param key_usage Tells how the key can be used.
     * @param nonce A "nonce" element.
     * @return The KA1 package.
     */
    public static byte[] createKA1Package (RSAKey attesting_key,
                                           PublicKey attested_public_key,
                                           boolean exportable,
                                           byte key_usage,
                                           byte[] nonce)
    throws GeneralSecurityException
      {
        ////////////////////////////////////////////////////////////////////////////////////////
        // To make it feasible securely distinguishing standard RSASSA-PKCS1.5 / SHA256       //
        // signatures from KA1 signatures the latter are packaged in a different way which    //
        // should create errors if processed by a crypto library that does not support KA1.   //
        // The following shows the packaging differences in detail.                           //
        //                                                                                    //
        // EMSA-PKCS1-v1_5: EM = 0x00 || 0x01 || PS || 0x00 || T                              //
        //                                                                                    //
        // EMDIAS-PKCS1:    EM = 0x00 || 0x01 || PS || 0x00 || 'D' || 'I' || 'A' || 'S' || T  //
        ////////////////////////////////////////////////////////////////////////////////////////
        byte[] modulus = attesting_key.getModulus ().toByteArray ();
        int k = modulus.length;
        if (modulus[0] == 0) k--;
        byte[] encoded_message = new byte [k];
        encoded_message[0] = (byte)0;
        encoded_message[1] = (byte)1;
        MessageDigest md = MessageDigest.getInstance (SHA256);
        md.update (nonce);
        md.update (exportable ? (byte)1 : (byte)0);
        md.update (key_usage);
        byte[] hash = md.digest (attested_public_key.getEncoded ());
        int i = k - 2 - PS_END_SEQUENCE.length - hash.length - DIGEST_INFO_SHA256.length;
        int j = 2;
        while (i-- > 0)
          {
            encoded_message[j++] = (byte)0xff;
          }
        i = 0;
        while (i < PS_END_SEQUENCE.length)
          {
            encoded_message[j++] = PS_END_SEQUENCE[i++];
          }
        System.arraycopy (DIGEST_INFO_SHA256, 0, encoded_message, j, DIGEST_INFO_SHA256.length);
        System.arraycopy (hash, 0, encoded_message, j + DIGEST_INFO_SHA256.length, hash.length);
        return encoded_message;
      }
  /**
   * Verify a KeyGen2 KA1 signature
   * @param attestation_signature The signature to be verified.
   * @param attesting_public_key The attesting (certifying) public key.
   * @param attested_public_key The attested (certified) key.
   * @param exportable True if exportable.
   * @param key_usage Tells how the key can be used.
   * @param nonce A "nonce" element.
   */
  public static void verifyKA1Signature (byte[] attestation_signature,
                                         PublicKey attesting_public_key,
                                         PublicKey attested_public_key,
                                         boolean exportable,
                                         byte key_usage,
                                         byte[] nonce)
  throws GeneralSecurityException
    {
      Cipher cipher = Cipher.getInstance (UNFORMATTED_RSA);
      cipher.init (Cipher.DECRYPT_MODE, attesting_public_key);
      byte[] received_signature_package = cipher.doFinal (attestation_signature);
      if ((received_signature_package.length & 1) != 0)
        {
          // BouncyCastle fix
          byte[] add_leading_zero = new byte[received_signature_package.length + 1];
          System.arraycopy (received_signature_package, 0, add_leading_zero, 1, received_signature_package.length);
          add_leading_zero[0] = (byte)0;
          received_signature_package = add_leading_zero;
        }
      byte[] reference_signature_package = createKA1Package ((RSAKey)attesting_public_key,
                                                              attested_public_key,
                                                              exportable,
                                                              key_usage,
                                                              nonce);
      if (reference_signature_package.length != received_signature_package.length)
        {
          throw new GeneralSecurityException ("Signature package length error");
        }
      for (int i = 0; i < received_signature_package.length ; i++)
        {
          if (received_signature_package[i] != reference_signature_package[i])
            {
              // A more comprehensive diagnostic would be preferable...
              throw new GeneralSecurityException ("Signature package content error");
            }
        }
    }


  /**
   * Create a KeyGen2 KA1 nonce value
   * @param key_id The ID element for the particular key.
   * @param client_session_id The client-side provisioning session-ID.
   * @param server_session_id The server-side provisioning session-ID.
   * @return The nonce.
   */
  public static byte[] createKA1Nonce (String key_id, String client_session_id, String server_session_id)
  throws GeneralSecurityException, IOException
    {
      MessageDigest md = MessageDigest.getInstance (SHA256);
      return md.digest (new StringBuffer (key_id).append ('\0').
                            append (client_session_id).append ('\0').
                            append (server_session_id).append ('\0').toString ().getBytes ("UTF-8"));
    }

}
```

IPR Declaration

*This specification is herby put in the public domain. It does to the author's knowledge not infringe on any existing patent.*