

SKAE “light” – Invention Disclosure

Background

In some cryptographic systems, asymmetric key-pairs may be internally generated in Security Elements (SEs), like Smart Cards. For on-line (remote) provisioning of keys to SEs, there is a wish by CAs 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.

To support this requirement the Trusted Computing Group’s (TCG’s) Trusted Platform Module (TPM) V1.2 specification features a mechanism known as Subject Key Attestation Evidence (SKAE).

Prerequisite: The SE contains an embedded private key and a matching public key certificate identifying the SE in some way that makes sense for CAs. The private key is used for signing evidence attestations regarding generated key-pairs.

Problem: If the embedded private key of the SE is also to be usable for other cryptographic operations, a “hacked” software environment could exploit this by creating unprotected key-pairs externally and still be able providing CAs with genuine-looking attestation evidences. TCG’s solution to this problem is based on the use of *dedicated key attest keys, specific X.509 certificate extensions*, and potentially also relying on *multiple CA roots*. Although working, this scheme appears unnecessary complex for supporting a single-purpose key.

Solution

The following steps describe how a simpler form of SKAE, *purely based on cryptography*, could be implemented in an SE.

First you need to restrict the SE key-certifying private key from being able to perform unformatted (“raw”) RSA *decryption* operations through direct API calls to the SE.

You MAY still allow the key-certifying key creating standard PKCS #1 signatures which are useful for many purposes including authentication and message integrity. Encryption operations using the *public* key MAY also be supported.

To support SKAE, the proposal is to create *a unique variant of PKCS #1 signatures that must only be generated during key-generation*. Such a signature MAY also be created if the SE is explicitly invoked with a previously generated public key, unless that would lead to key-reuse vulnerabilities.

A unique variant of PKCS #1 could in its simplest form be implemented through the use of a non-standard padding pattern. The hash algorithms to use would still be the existing SHA-1 etc. A verifier should then be able to securely distinguish between standard signatures and SKAE signatures. A nonce option would also be suitable for inclusion in the signature packet. On the next pages, there is a sample program in Java implementing the proposed SKAE scheme.

Although this scheme only describes PKCS #1 (RSA) keys, the *principles* are presumable applicable to other asymmetric key types as well including ECDSA.

One might consider the proposed solution as a standards-defying “kludge”, but the fact is that all current SKAE schemes rely on very specific generation and validation code. In fact, “tagged signatures” are featured in TLS (RFC 4346).

The primary target for this invention are mobile phones which are likely to be fitted with secure hardware fairly soon, since this is more or less a requirement for other purposes as well, most notably for Operating System integrity protection.

Side Effect: PoP Becomes Redundant

It seems that a properly designed SKAE scheme makes Proof of Possession (PoP) signatures like in CRMF (RFC 4211) redundant which is an advantage because combining PoP signatures with PIN-code provisioning introduces fairly awkward state-handling in SEs *since the generated private key typically must be used before it is fully provisioned*.

Anders Rundgren, WebPKI.org, September 15, 2008

Author’s address:
Storbolsång 50
740 10 Almunge
Sweden

Email: anders.rundgren@telia.com

```

import javax.crypto.Cipher;

import java.security.GeneralSecurityException;
import java.security.MessageDigest;
import java.security.PublicKey;
import java.security.PrivateKey;
import java.security.KeyPair;
import java.security.KeyPairGenerator;

import java.security.interfaces.RSAKey;

/**
 * SKAE (Subject Key Attestation Evidence). The following J2SE compatible code is meant illustrate
 * the use of SKAE signatures.
 */
public class skae
{
    static final String SHA_1          = "SHA-1";

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

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

    static final byte[] DIGEST_INFO_SHA1 = new byte[] {(byte)0x30, (byte)0x21, (byte)0x30, (byte)0x09, (byte)0x06,
                                                       (byte)0x05, (byte)0x2b, (byte)0x0e, (byte)0x03, (byte)0x02,
                                                       (byte)0x1a, (byte)0x05, (byte)0x00, (byte)0x04, (byte)0x14};

    /**
     * Create an SKAE package for signing or verification
     * @param rsa_key The certifying (attesting) private or public key.
     * @param certified_public_key The certified (attested) key.
     * @param optional_nonce An optional "nonce" element.
     * @return The SKAE package.
     */
    public static byte[] createSKAEPackage (RSAKey rsa_key,
                                           PublicKey certified_public_key,
                                           byte[] optional_nonce)
        throws GeneralSecurityException
    {
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        // To make it feasible securely distinguishing standard RSASSA-PKCS1.5 signatures //
        // from SKAE signatures the latter are packaged in a different way which should //
        // create errors if processed by a crypto library that does not support SKAE. //
        // The following shows the packaging differences in detail. //
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        // EMSA-PKCS1-v1_5: EM = 0x00 || 0x01 || PS || 0x00 || T //
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        // EM-PKCS1-SKAE:   EM = 0x00 || 0x01 || PS || 0x00 || 'S' || 'K' || 'A' || 'E' || T //
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        byte[] modulus = rsa_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 (SHA_1);
        if (optional_nonce != null)
        {
            md.update (optional_nonce);
        }
        byte[] hash = md.digest (certified_public_key.getEncoded ());
        int i = k - 2 - PS_END_SEQUENCE.length - hash.length - DIGEST_INFO_SHA1.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_SHA1, 0, encoded_message, j, DIGEST_INFO_SHA1.length);
        System.arraycopy (hash, 0, encoded_message, j + DIGEST_INFO_SHA1.length, hash.length);
        return encoded_message;
    }
}

```

```

/**
 * Verify an SKAE signature
 * @param skae_signature The signature to be verified.
 * @param certifying_public_key The certifying (attesting) public key.
 * @param certified_public_key The certified (attested) key.
 * @param optional_nonce An optional "nonce" element.
 * @throws GeneralSecurityException if the signature is invalid or indata is incorrect.
 */
public static void verifySKAESignature (byte[] skae_signature,
                                       PublicKey certifying_public_key,
                                       PublicKey certified_public_key,
                                       byte[] optional_nonce)
throws GeneralSecurityException
{
    Cipher cipher = Cipher.getInstance (UNFORMATTED_RSA);
    cipher.init (Cipher.ENCRYPT_MODE, certifying_public_key);
    byte[] received_signature_package = cipher.doFinal (skae_signature);
    byte[] reference_signature_package = createSKAEPackage ((RSAKey)certifying_public_key,
                                                         certified_public_key,
                                                         optional_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");
        }
    }
}

public static class GeneratedKey
{
    PublicKey certified_public_key;
    PublicKey certifying_public_key;
    byte[] skae_signature;
}

public static class SecurityElement
{
    PublicKey certifying_public_key;

    private PrivateKey certifying_private_key;

    public SecurityElement () throws GeneralSecurityException
    {
        ////////////////////////////////////////////////////////////////////
        // Key-certifying keys are typically created once during      //
        // device manufacturing. The public key part is also most     //
        // likely distributed in an X.509 certificate issued by a CA   //
        // setup specifically for certifying crypto hardware.         //
        // That is, the following lines are just for showing the     //
        // cryptography, without any infrastructural considerations.  //
        ////////////////////////////////////////////////////////////////////
        KeyPairGenerator certifier = KeyPairGenerator.getInstance ("RSA");
        certifier.initialize (2048);
        KeyPair certifying_key_pair = certifier.generateKeyPair ();
        certifying_public_key = certifying_key_pair.getPublic ();
        certifying_private_key = certifying_key_pair.getPrivate ();
    }
}

/**
 * Create a certified key-pair.
 * @param size The size of the RSA key.
 * @param optional_nonce An optional "nonce" element.
 * @return A container with a generated public key and attesting signature.
 */
public GeneratedKey generateCertifiedKeyPair (int size, byte[] optional_nonce)
throws GeneralSecurityException
{
    ////////////////////////////////////////////////////////////////////
    // Generate a new key-pair in the Security Element. The         //
    // private key is presumably stored securely in hardware and    //
    // never leave its container, unless "sealed" by the latter.    //
    ////////////////////////////////////////////////////////////////////
    KeyPairGenerator kpg = KeyPairGenerator.getInstance ("RSA");
    kpg.initialize (size);
    KeyPair new_key_pair = kpg.generateKeyPair ();
}

```

```

////////////////////////////////////
// Now let the Security Element attest that the new key-pair //
// actually was created inside of the Security Element. //
// //
// NOTE 1: The Security Element MUST NOT expose an API that //
// allows unformatted RSA decryptions like used below to be //
// performed with the key-certifying key, otherwise "malware" //
// could easily create fake attestations for any externally //
// supplied key-pair! //
// //
// NOTE 2: Due to the fact that SKAE signatures are only to //
// be created for generated keys, the key-certifying key MAY //
// also be used for creating ordinary PKCS1.5 signatures for //
// things like authentications and securing message integrity //
////////////////////////////////////
GeneratedKey gk = new GeneratedKey ();
gk.certified_public_key = new_key_pair.getPublic ();
Cipher cipher = Cipher.getInstance (UNFORMATTED_RSA);
cipher.init (Cipher.DECRYPT_MODE, certifying_private_key);
gk.skae_signature = cipher.doFinal (createSKAEPackage ((RSAKey)certifying_private_key,
                                                    gk.certified_public_key,
                                                    optional_nonce));

gk.certifying_public_key = certifying_public_key;
return gk;
}
}

public static void main (String[] args) throws Exception
{
    //////////////////////////////////////
    // //
    // CLIENT Operations //
    // //
    // It is assumed that the critical operations are performed //
    // inside of the Security Element, otherwise attestations //
    // would not be more trustworthy than the environment where //
    // the Security Element is actually running in! //
    //////////////////////////////////////
    SecurityElement se = new SecurityElement ();

    //////////////////////////////////////
    // Generate a new key-pair in the Security Element. The //
    // private key is presumably stored securely in hardware and //
    // never leave its container, unless "sealed" by the latter. //
    //////////////////////////////////////
    byte[] nonce = null; // Didn't use a nonce in the sample run
    GeneratedKey gk = se.generateCertifiedKeyPair (1024, nonce);

    //////////////////////////////////////
    // //
    // VERIFIER Operations //
    // //
    // The certifying public key is supposed to be transferred to //
    // the verifier by some kind of protocol, together with the //
    // SKAE-signature, certified public key, and the optional //
    // nonce. A nonce (if used) would preferably be created by //
    // the verifier during an earlier (not shown) protocol phase. //
    //////////////////////////////////////
    verifySKAESignature (gk.skae_signature,
                        gk.certifying_public_key,
                        gk.certified_public_key,
                        nonce);

    System.out.println ("The SKAE signature appears to be valid!");
}
}

```