SafeNet ProtectToolkit J
JCA/JCE API

Tutorial

gemalto
security to be free

Part Number 007-008399-007
Version 5.2

Trademarks

Gemalto Rebranding

In early 2015, Gemalto NV completed its acquisition of SafeNet, Inc. As part of the process of rationalizing the product portfolios between the two organizations, the HSM product portfolio has been streamlined under the SafeNet brand. As a result, the ProtectServer/ProtectToolkit product line has been rebranded as follows:

Old product name	New product name
Protect Server External 2 (PSE2)	SafeNet ProtectServer Network HSM
Protect Server Internal Express 2 (PSI-E2)	SafeNet ProtectServer PCIe HSM
ProtectToolkit	SafeNet ProtectToolkit

Disclaimer

Technical Support

If you encounter a problem while installing, registering or operating this product, please make sure that you have read the documentation. If you cannot resolve the issue, please contact your supplier or Gemalto support. Gemalto support operates 24 hours a day, 7 days a week. Your level of access to this service is governed by the support plan arrangements made between Gemalto and your organization. Please consult this support plan for further information about your entitlements, including the hours when telephone support is available to you.

Contact method	Contact	
Address	Gemalto NV 4690 Millennium Drive Belcamp, Maryland 21017 USA	
Phone	Global	+1 410-931-7520
	Australia	1800.020.183
	China	(86) 10 8851 9191
	France	0825 341000
	Germany	01803 7246269
	India	000.800.100.4290
	Netherlands	0800.022.2996
	New Zealand	0800.440.359
	Portugal	800.1302.029
	Singapore	800.863.499
	Spain	900.938.717
	Sweden	020.791.028
	Switzerland	0800.564.849
	United Kingdom	0800.056.3158
	United States	(800) 545-6608
Web	www.safenet-inc.com	
Support and Downloads	www.safenet-inc.com/support Provides access to the Gemalto Knowledge Base and quick downloads for various products.	
Technical Support Customer Portal	https://serviceportal.safenet-inc.com Existing customers with a Technical Support Customer Portal account can log in to manage incidents, get the latest software upgrades, and access the Gemalto Knowledge Base.	

Revision History

Revision	Date	Reason
A	14 March 2016	Release 5.2

TABLE OF CONTENTS

1.0 SCOPE **1**

2.0 INTRODUCTION **3**

3.0 PUBLIC KEY CRYPTOGRAPHY **5**

4.0 FILECRYPT APPLICATION **7**

4.1 File Encryption *7*
4.2 File Decryption *12*
4.3 Accessing Public Keys *16*
4.4 Putting it all Together *17*

THIS PAGE INTENTIONALLY LEFT BLANK

1.0 Scope

The purpose of this document is to introduce the reader to the Java API known as the Java Cryptography Extension (JCE) through the development of a simple application.

It is important to realise that this tutorial does not provide complete coverage of this API. The *JCE Application Programming Interface Overview* provides a good introduction to this API and the API specification documentation should serve as the detailed reference.

THIS PAGE INTENTIONALLY LEFT BLANK

2.0 Introduction

During this tutorial we will develop a JCE based application that allows for simple file encryption. This application will allow the user to encrypt and decrypt files.

The files are encrypted using a combination of public-key and secret-key cryptography. The encrypted files also include a Message Authentication Code (MAC) to ensure the integrity of their contents. Where possible, the standard API mechanisms will be used to achieve the desired functionality.

The code fragments included in this document are used to highlight the important sections of the application. The full source code for the application may be found in the Java source file FileCrypt.java.

THIS PAGE INTENTIONALLY LEFT BLANK

3.0 Public Key Cryptography

The sample application will encrypt a document using a secret-key cipher algorithm, for example DES or RC4, and a randomly generated key. This algorithm is known as the bulk cipher as it is used to perform the bulk of the encryption. The randomly generated key will be encrypted using a public-key cipher algorithm.

By combining public-key and secret-key encryption in this manner we retain the advantages of public-key cryptography (we don't have to share a secret key with them) while retaining the performance advantage of a secret-key cipher.

It is assumed that two public key pairs have been generated for this application, the first for the document sender and the second for the recipient.

THIS PAGE INTENTIONALLY LEFT BLANK

4.0 FileCrypt Application

The `FileCrypt` application enables files to be encrypted for a given recipient and then decrypted by that recipient. Since the encrypted file contains a MAC, the recipient of a document will also be able to verify that the encrypted file was not tampered with.

These encrypted files will be stored in a custom format which is as follows:

Field	Length (bytes)
KeyLength	4
KeyBytes	As specified by KeyLength
AlgParamsLength	4
AlgParams	As specified by AlgParamsLength
MacLength	4
Mac	As specified by MacLength
Encrypted Data	Remainder of file

4.1 File Encryption

In order to encrypt a file, we need to know the public key of its recipient - that is the party who can decrypt the file. These arguments are passed to the `encryptFile()` method.

The `encryptFile()` method will:

1. generate a random session key
2. encrypt the session key with the recipients public key
3. initialise the bulk cipher with the session key
4. encode the bulk cipher's algorithm parameters
5. initialise the MAC algorithm
6. process the input file
7. create the output from the various components

4.1.1 Generating a Random Session Key

To achieve acceptable performance during file encryption and decryption we need to use a symmetric-key cipher. This symmetric key, which we will call the session key, will then be encrypted (using the recipient's public key) and then stored with the encrypted file. Rather than simply using the same key for each file, we need to generate a random key for each encryption.

The `KeyGenerator` mechanism is used to create random `SecretKey` key objects. A provider based instance is created using the `KeyGenerator.getInstance()` method.

This instance can then be initialised using one of the `KeyGenerator.init()` methods. In the simplest case, no initialisation is required, in which case the provider's default initialisation is used. Alternatively, initialisation can request a key of the given key size, or other key parameters by using a `java.security.AlgorithmParameterSpec` class.

The following method will create a new random `SecretKey` for the given algorithm and provider using the default initialisation;

```
SecretKey generateSecretKey(String algorithm, String
                             provider)
{
   KeyGenerator keyGen = KeyGenerator.getInstance(
      algorithm, provider);

   return keyGen.generateKey();
}
```

4.1.2 Encrypting the Session Key

Once we have generated the session key, we need to encrypt it using the recipient's public key. In this way we can safely transmit the session key such that only the recipient can recover the actual key. The SafeNet "SAFENET" provider includes a special interface to its `KeyStore` to provide session key encryption.

The `au.com.safenet.crypto.WrappingKeyStore` class extends the standard `KeyStore` mechanism to provide "key wrapping" which enables a session key to be generated in the hardware, then encrypted on the hardware and exported in an encrypted form. This means that the session key is never visible outside the hardware. (For more information on the `WrappingKeyStore` interface please consult the SAFENET Provider Reference manual.)

The `WrappingKeyStore.wrapKey()` method accepts three arguments; two keys and a transformation string. The first `Key` is the RSA `PublicKey` used to perform the encryption, the second `Key` is the DES key we wish to encrypt. The final parameter, the transformation string, describes the encryption method that should be used to encrypt the key. Currently, this string may be `RSA/ECB/PKCS1Padding` or `RSA/ECB/NoPadding`.

```
static final String PROVIDER = "SAFENET";
static final String WRAP_KEYSTORE = "CRYPTOKI";
static final String WRAP_TRANSFORM =
                        "RSA/ECB/PKCS1Padding";

byte[] encryptKey(PublicKey wrapKey, SecretKey key)
{
   WrappingKeyStore keyStore;
   keyStore = WrappingKeyStore.getInstance(WRAP_KEYSTORE,
                                           PROVIDER);
   keyStore.load(null, null);
   return keyStore.wrapKey(wrapKey, WRAP_TRANSFORM, key);
}
```

4.1.3 Create and initialise the Bulk Cipher

This application will simply use the default `AlgorithmParameters` for the bulk encryption algorithm. Therefore, the initialisation of our `Cipher` is quite simple:

```
static final String PROVIDER = "SAFENET";
static final String BULK_ALGORITHM = "DES";

   Cipher bulkCipher = Cipher.getInstance(BULK_ALGORITHM,
                                          PROVIDER);
   bulkCipher.init(Cipher.ENCRYPT_MODE, secretKey);
```

4.1.4 Encoded Algorithm Parameters

The only algorithm parameter supported by the SafeNet "SAFENET" provider is an initialisation vector. An initialisation vector is used in a block cipher when it is operating in a feedback mode: DES in CBC mode for example. During encryption the initialisation vector is used to prime the cipher, however unlike the key its value is not secret.

The cipher used to decrypt the data stream must be initialised with the same initialisation vector for the decryption to succeed.

The following method will return the algorithm parameters encoded into a byte array. For now, we just return the IV directly as this is the only supported algorithm parameter.

```
byte[] encodeParameters(Cipher cipher)
{
   byte[] iv = cipher.getIV();
   return iv;
}
```

4.1.5 Initialise the MAC Algorithm

In this example we will use a MAC algorithm instead of a signature algorithm. The significant difference here is that the MAC will only tell us if the encrypted document has been tampered with, it will not authenticate the sender.

```
static final String PROVIDER = "SAFENET";
static final String MAC_ALGORITHM = "DESMac";

Mac mac = Mac.getInstance(MAC_ALGORITHM, PROVIDER);
mac.init(secretKey);
```

4.1.6 Process the Input File

We are now ready to process the input file to generate the encrypted output and the MAC. The following method will accept the initialised `Cipher`, `Mac` and input/output streams. The data on the `InputStream` will be read in blocks (of some arbitrary size), then processed by the `Mac` instance and then encrypted with the `Cipher` instance.

The encrypted data will then be written to the `OutputStream`. This method will return the MAC as a byte array.

```
static final int READ_BUFFER = 50;

byte[] encrypt(Cipher cipher, Mac mac, InputStream in,
               OutputStream out)
{
   byte[] block = new byte[READ_BUFFER];
   int len;
   while ((len = in.read(block)) != -1)
   {
      /*
       * update our MAC value
       */
      mac.update(block, 0, len);
```

```
            /*
             * encrypt the data
             */
            byte[] enc = cipher.update(block, 0, len);
            if (enc != null)
            {
               /*
                * output the encrypted data
                */
               out.write(enc);
            }
         }

         /*
          * output the final block if required
          */
         byte[] finalBlock = cipher.doFinal();
         if (finalBlock != null)
         {
            out.write(finalBlock);
         }

         return mac.doFinal();
      }
```

4.1.7 Create the encrypted Output

Now that we have written the various building blocks, we can construct the final `encryptFile()` method:

```
static final String PROVIDER = "SAFENET";
static final String BULK_ALGORITHM = "DES";
static final String BULK_TRANSFORM =
                       "DES/CBC/PKCS5Padding";
static final String MAC_ALGORITHM = "DESMac";

void encryptFile(InputStream in, OutputStream out,
                 PublicKey publicKey)
{
   /*
    * Create a random SecretKey and encrypt it using
    * the recipient's PublicKey
    */
   SecretKey secretKey = generateSecretKey(BULK_ALGORITHM,
                                           PROVIDER);
   byte[] wrappedKey = encryptKey(publicKey, secretKey);

   /*
    * Create and initialise the Cipher used to encrypt the
      document
    */
   Cipher bulkCipher =
             Cipher.getInstance(BULK_TRANSFORM,PROVIDER);
   bulkCipher.init(Cipher.ENCRYPT_MODE, secretKey);

   /*
    * Encode the algorithm parameters for the Cipher
    */
   byte[] algParams = encodeParameters(bulkCipher);
```

```
      /*
       * Create the Mac instance and initialise it with our
       * session key
       */
      Mac mac = Mac.getInstance(MAC_ALGORITHM, PROVIDER);
      mac.init(secretKey);

      /*
       * Encrypt the document to an internal buffer and
       * calculate the MAC value of the plain text
       */
      ByteArrayOutputStream bOut =
                                 new ByteArrayOutputStream();
      byte[] macValue = encrypt(bulkCipher, mac, in, bOut);

      /*
       * Encode the output file
       */
      DataOutputStream dOut = new DataOutputStream(out);

      /*
       * Write out the key
       */
      dOut.writeInt(wrappedKey.length);
      dOut.write(wrappedKey);

      /*
       * Write out the parameters, note these may be null
       */
      if (algParams != null)
      {
         dOut.writeInt(algParams.length);
         dOut.write(algParams);
      }
      else
      {
         dOut.writeInt(0);
      }

      /*
       * Write out the MAC
       */
      dOut.writeInt(macValue.length);
      dOut.write(macValue);

      /*
       * And finally the encrypted document
       */
      bOut.writeTo(dOut);
   }
```

4.2 File Decryption

To decrypt an encrypted file we simply need to reverse the encryption process. However, rather than using the recipient's public key, we need to use the private key in order to recover the session key.

The `decryptFile()` method will:

1. decode the input from the various components
2. decipher the session key with the recipient's private key
3. initialise the bulk cipher with the session key and algorithm parameters
4. initialise the MAC algorithm
5. process the encrypted input
6. verify the calculated MAC with the MAC from the document
7. write out the decrypted result

4.2.1 Decryption of the session key

```
static final String PROVIDER = "SAFENET";
static final String WRAP_KEYSTORE = "CRYPTOKI";
static final String WRAP_TRANSFORM =
"RSA/ECB/PKCS1Padding";
static final String BULK_ALGORITHM = "DES";

Key decryptKey(PrivateKey wrapKey, byte[] wrappedKey)
{
   WrappingKeyStore keyStore;
   keyStore = WrappingKeyStore.getInstance(WRAP_KEYSTORE,
                                           PROVIDER);

   return keyStore.unwrapKey(wrapKey, WRAP_TRANSFORM,
                             wrappedKey, BULK_ALGORITHM);
}
```

4.2.2 Bulk Cipher Initialisation

Next, we need to create and initialise the `Cipher` instance we will use to decrypt the document. It is important here to ensure that our `Cipher` instance that will be used to perform the decryption is initialised with the same parameters that were generated by the encryption `Cipher`. In the case of the SafeNet "SAFENET" provider, the only parameter type is the `IvParameterSpec`, so we convert our serialised parameters directly.

```
static final String PROVIDER = "SAFENET";
static final String BULK_ALGORITHM = "DES";

Cipher bulkCipher = Cipher.getInstance(BULK_TRANSFORM,
                                       PROVIDER);
if (algParams != null)
{
   AlgorithmParameterSpec params;
   params = new IvParameterSpec(algParams);

   bulkCipher.init(Cipher.DECRYPT_MODE, secretKey,
                   params);
}
else
{
   bulkCipher.init(Cipher.DECRYPT_MODE, secretKey);
}
```

4.2.3 Initialise the MAC Algorithm

Initialisation of the MAC during decryption is identical to that during encryption:

```
static final String PROVIDER = "SAFENET";
static final String MAC_ALGORITHM = "DESMac";

   Mac mac = Mac.getInstance(MAC_ALGORITHM, PROVIDER);
   mac.init(secretKey);
```

4.2.4 Process the Encrypted input

Next we need to recover the plaintext from the ciphertext and calculate a new MAC. This process is nearly identical to the `encrypt()` method, however, since the MAC is calculated on the plaintext, we update the `Mac` with the output from the `Cipher`.

```
static final int READ_BUFFER = 50;

byte[] decrypt(Cipher cipher, Mac mac, InputStream in,
OutputStream out)
{
   /*
    * read the input in chunks and process each chunk
    */
   byte[] block = new byte[READ_BUFFER];
   int len;
   while ((len = in.read(block)) != -1)
   {
      /*
       * decipher the data
       */
      byte[] plain = cipher.update(block, 0, len);
      if (plain != null)
      {
         /*
          * update our MAC value
          */
         mac.update(plain);

         /*
          * output the deciphered data
          */
         out.write(plain);
      }
   }

   /*
    * output the final block if required
    */
   byte[] finalBlock = cipher.doFinal();
   if (finalBlock != null)
   {
      /*
       * update our MAC value
       */
      mac.update(finalBlock);

      /*
       * output the deciphered data
       */
      out.write(finalBlock);
   }
```

```
    return mac.doFinal();
}
```

4.2.5 Verify the MAC

To verify the MAC, we simply compare the MAC bytes we previously extracted with the value just calculated.

```
if (!Arrays.equals(fileMac, calculatedMac))

{
    throw new GeneralSecurityException("File has been
    tampered with.");
}
```

4.2.6 Write out the Decrypted result

Now that we have verified that the file is not corrupted we can output the contents to the destination.

```
static final String PROVIDER = "SAFENET";
static final String BULK_ALGORITHM = "DES";
static final String BULK_TRANSFORM =
"DES/CBC/PKCS5Padding";
static final String MAC_ALGORITHM = "DESMac";

void decryptFile(InputStream in, OutputStream
out,PrivateKey privateKey)
{
    /*
     * Decode the input file
     */
    DataInputStream dIn = new DataInputStream(in);

    /*
     * recover the encrypted Key data
     */
    int keyLen = dIn.readInt();
    byte[] keyBytes = new byte[keyLen];
    dIn.readFully(keyBytes);

    /*
     * recover the algorithm parameters
     */
    int algLen = dIn.readInt();
    byte[] algBytes = null;
    if (algLen > 0)
    {
        algBytes = new byte[algLen];
        dIn.readFully(algBytes);
    }

    /*
     * recover the stored MAC value
     */
    int macLen = dIn.readInt();
    byte[] fileMac = new byte[macLen];
    dIn.readFully(fileMac);

    /*
     * recreate the session key
```

```
         */
        Key secretKey = decryptKey(privateKey, keyBytes);

        /*
         * Create our Cipher and initialise it with our key
         * and algorithm parameters.
         */
        Cipher bulkCipher =
                  Cipher.getInstance(BULK_TRANSFORM,PROVIDER);
        if (algBytes != null)
        {
           AlgorithmParameterSpec params;
           params = new IvParameterSpec(algBytes);

           bulkCipher.init(Cipher.DECRYPT_MODE, secretKey,
                           params);
        }
        else
        {
           bulkCipher.init(Cipher.DECRYPT_MODE, secretKey);
        }

        /*
         * Initialise the Mac we use to verify the file
           integrity
         */
        Mac mac = Mac.getInstance(MAC_ALGORITHM, PROVIDER);
        mac.init(secretKey);

        /*
         * Decrypt the file to a temporary buffer
         */
        ByteArrayOutputStream bOut =
                              new ByteArrayOutputStream();
        byte[] calculatedMac = decrypt(bulkCipher, mac, in,
                                      bOut);

        /*
         * verify the stored MAC value with the calculated
           value
         */
        if (!Arrays.equals(fileMac, calculatedMac))
        {
           throw new GeneralSecurityException(
              "File has been tampered with.");
        }
        else
        {
           /*
            * save the decrypted output to the outputstream
            */
           bOut.writeTo(out);
        }
     }
```

4.3 Accessing Public Keys

A Java `java.security.KeyStore` implementation is used to store the public keys for this application. The SafeNet "SAFENET" provider implementation of the `KeyStore` is known as `"CRYPTOKI"` and enables access to the keys stored on the hardware. At present, this `KeyStore` only supports storage of `Key` objects and does not provide any support for the storage of `Certificate` objects. Additionally, this `KeyStore` will ignore the password parameter supplied to the `getKey()` method.

4.3.1 Creating the KeyStore

Creating a `KeyStore` instance and populating it is generally a two step process. Firstly, we create the instance and then use the `KeyStore.load()` method to initialise it with the key data. The `load()` method accepts an `InputStream` instance which allows for keys to be stored on an arbitrary data source. The `"CRYPTOKI" KeyStore`, however, accesses key storage on the hardware directly and so ignores the `load()` method completely.

```
static final String PROVIDER = "SAFENET";
static final String KS_NAME = "CRYPTOKI";

KeyStore loadKeyStore()
{
   KeyStore ks = KeyStore.getInstance(KS_NAME, PROVIDER);
   ks.load(null, null);

   return ks;
}
```

4.3.2 Retrieving the Public Key

Our application needs to determine the recipient's public key in order to encrypt the file. The standard mechanism for accessing public keys is to extract the `Certificate` for the recipient by using the `KeyStore.getCertificate()` method and then use the `Certificate.getPublicKey` method to recover the key. However with the `"CRYPTOKI" KeyStore` we will simply use the `KeyStore.getKey()` method.

```
PublicKey publicKey = (PublicKey)ks.getKey(recipientAlias,
                       null);
```

4.3.3 Retrieving the Private Key

To decrypt the file we need to look up the private key. To access private keys stored in a `KeyStore` use the `KeyStore.getKey()` method.

```
PrivateKey privateKey = (PrivateKey)ks.getKey(myAlias,
                         null);
```

4.4 Putting it all Together

Now that we have all the required building blocks, the last remaining step is to put it all together. We need to process command line arguments and call the appropriate methods. We also need to add exception handling.

The following `main()` method is responsible for determining if we are encrypting or decrypting the file and the names of the keys to use:

```
public static void main(String[] args)
{
   boolean encrypt = false;
   boolean decrypt = false;

   String keyName = null;

   /*
    * examine all the command line arguments
    */
   for (int i = 0; i < args.length; i++)
   {
      if (args[i].equals("-encrypt"))
      {
         encrypt = true;
      }
      else if (args[i].equals("-decrypt"))
      {
         decrypt = true;
      }
      else if (args[i].equals("-key"))
      {
         keyName = args[++i];
      }
   }

   /*
    * validate the arguments
    */
   if (encrypt == decrypt)
   {
      if (encrypt)
      {
         System.err.println("Cannot encrypt and decrypt
         file!");
      }
      else
      {
         System.err.println("Must specify -encrypt or -
         decrypt.");
      }
      System.exit(1);
   }

   if (keyName == null)
   {
      System.err.println("Missing key name.");
      System.exit(1);
   }
```

```
        FileCrypt fileCrypt = new FileCrypt();
        KeyStore ks = fileCrypt.loadKeyStore();

        if (encrypt)
        {
            PublicKey publicKey = (PublicKey)ks.getKey(keyName,
                                                                 null);

            fileCrypt.encryptFile(System.in, System.out, publicKey);
        }
        else
        {
            PrivateKey privateKey = (PrivateKey)ks.getKey(keyName,
            null);
            fileCrypt.decryptFile(System.in, System.out, privateKey);
        }
    }
```

END OF DOCUMENT