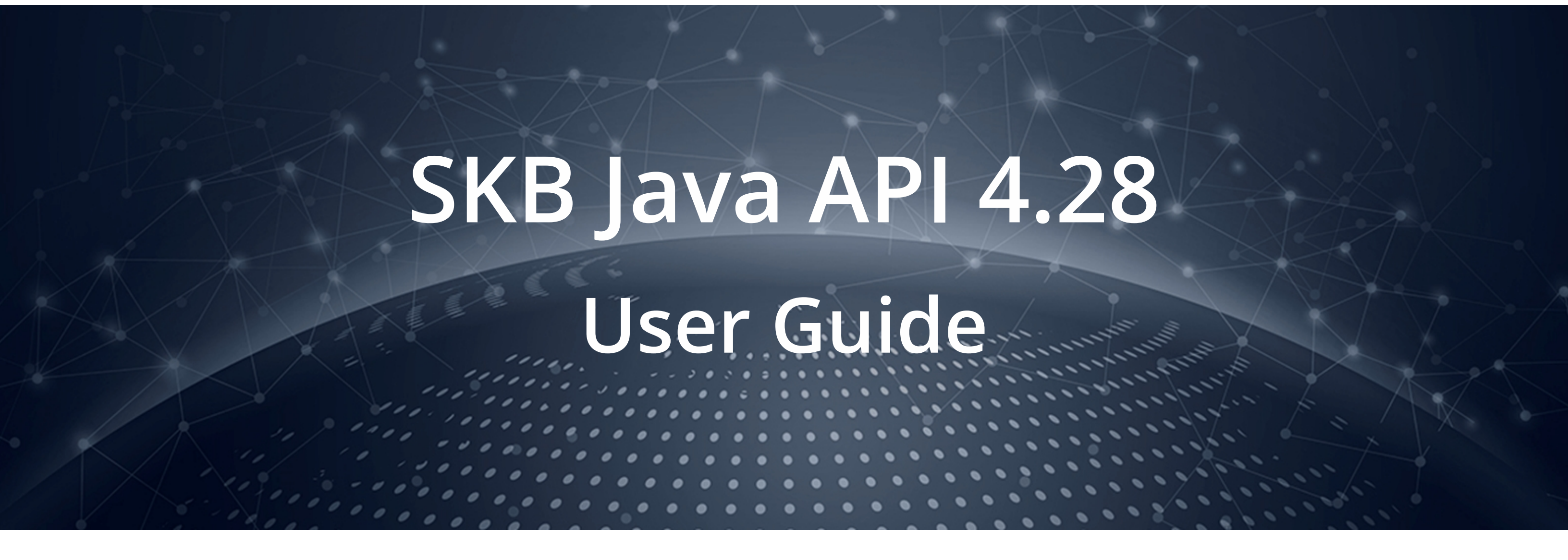
whiteCryption®



SKB Java API 4.28

User Guide

Copyright Information

Contact Information

whiteCryption Corporation, 920 Stewart Drive, Suite 100, Sunnyvale, California 94085, USA

contact@whitecryption.com

www.whitecryption.com

Table of Contents

1 Introduction

This document describes the Java API of Secure Key Box (SKB).

> ⚠ This document assumes that you are very well familiar with general concepts of SKB and its C API as described in *Secure Key Box User Guide*.

1.1 Overview of the SKB Java API

The purpose of the SKB Java API is to provide natural means of linking the native SKB library, which is written in C, with the Java programming language. In simple words, the SKB Java API is a Java wrapper around the native SKB classes and methods.

Figure 1.1 shows how linking with the native SKB library is realized.

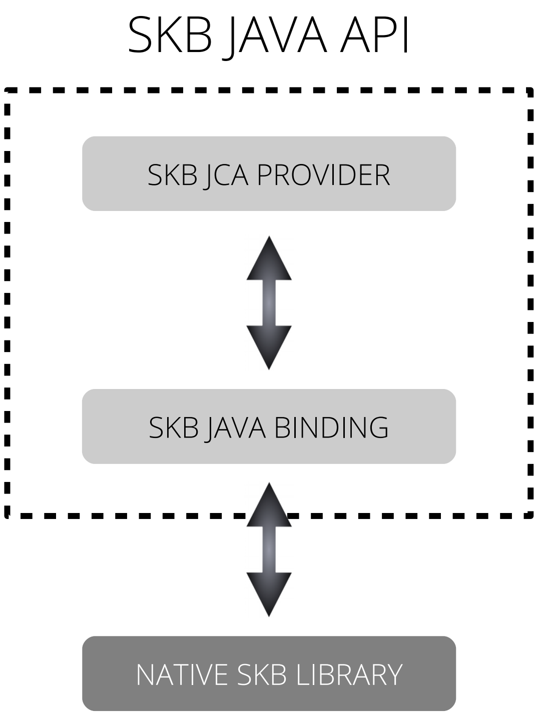


Figure 1.1: SKB Java API overview

The following are the elements in Figure 1.1:

SKB JCA Provider

The SKB JCA Provider is an implementation of the Java Cryptography Architecture (JCA) API, which is a popular standard for cryptographic libraries on Java platforms.

> ⚠ The SKB JCA Provider is the primary means by which you should use SKB features from Java.

For more information on the SKB JCA Provider implementation, see §2.

SKB Java Binding

The SKB Java Binding is a low-level API that provides an almost one-to-one mapping between native SKB elements and their Java counterparts.

For more information on the SKB Java Binding, see §3.

Native SKB library

The actual native SKB library that is being invoked by the SKB Java Binding and, by extension, also by the SKB JCA Provider.

Different target platforms will require a different edition of the native SKB library.

In this document, the SKB JCA Provider and SKB Java Binding are collectively called the SKB Java API.

⚠ Although you have the option to access SKB functionality directly via the SKB Java Binding API, we strongly recommend using the SKB JCA Provider, because backward compatibility of the SKB Java Binding API in future releases is not guaranteed.

1.2 Package Contents

SKB Java API is delivered as part of the native SKB package. The following are the contents of the SKB package that are specific to the SKB Java API:

`Java/Examples/`

Contains source code for SKB Java API examples.

`Java/Libraries/`

Contains the native SKB library for every target platform.

`Java/cryptanium-skb-«version».jar`

Contains all classes and files of the SKB Java API.

1.3 Installing the SKB Java API

To use the SKB Java API in your application, you must include the following main files into your application:

`cryptanium-skb-«version».jar`

This JAR file contains all classes and files of the SKB Java API. The classes are located in the `com.cryptanium.skb` package.

`libSecureKeyBoxJava.so`

The actual native SKB library that is called by the SKB Java API.

Both the `.jar` file and the native SKB library must be available on the class path. Before you start using the SKB Java API, the native SKB library must be specifically loaded in your Java application using the following command:

```
System.loadLibrary("SecureKeyBoxJava");
```

1.4 Limitations

In this release, SKB Java API has the following limitations when compared to the native SKB API:

- Only the following target platforms are supported:
  - GNU/Linux (x86, x86_64)
  - Android (x86, x86_64, ARM (32-bit and 64-bit), MIPS (32-bit and 64-bit))
- The SKB JCA Provider does not support the following algorithms:
  - loading plain RSA public keys
  - wrapping raw bytes using an RSA public key
  - XOR-based wrapping and unwrapping
  - ECDH using a static private ECC key (`SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC`)
  - wrapping plain data using 128-bit, 192-bit, and 256-bit AES
  - all key derivation algorithms
  - device binding
  - PDF decryption
- Multi-threading is not supported.

2 SKB Java Provider

This chapter provides implementation details of the SKB JCA Provider.

2.1 JCA Overview

Java Cryptography Architecture (JCA) is an accepted standard of APIs for encryption, digital signatures, digests, certificates, key generation and management, secure random number generation, and other cryptographic algorithms. For details on JCA, see the following web page:

http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html

⚠ This document assumes that you are very well familiar with general JCA concepts and API.

2.2 Implementation Details

whiteCryption offers a specific white-box implementation of a JCA Provider that provides the same level of security for Java applications as it does for the native applications written in C/C++.

Here is general information that you need to know about the SKB-specific implementation:

- The SKB JCA Provider implementation supports Java versions 6 and 7.
- The SKB JCA Provider code is located in the `com.cryptanium.skb.provider` package.
- SKB-specific implementation classes are prefixed with "`Skb`".

  For example, `SkbExportedKeySpec` is an SKB-specific implementation of the `java.security.spec.KeySpec` interface used to securely export SKB-protected keys for offline storage.
- Two SKB JCA Provider editions are provided:
  - The provider named "`Cryptanium`" provides the full scope of SKB algorithms (except those listed in §1.4), but it does not support the high-speed AES cipher implementation.
  - The provider named "`CryptaniumHighSpeedAes`" is specifically intended for the high-speed AES cipher implementation. No other ciphers and algorithms are provided.

2.3 Algorithms Provided by the SKB JCA Provider

This section lists algorithms supported by the SKB JCA Provider, organized into subsections corresponding to the main JCA classes.

2.3.1 Cipher

This section provides implementation details regarding the `Cipher` class.

2.3.1.1 Encryption

`Cipher.ENCRYPT_MODE` supports the following algorithms:

- `DES/ECB/NoPadding`
- `DES/CBC/NoPadding`
- `DESEDE/ECB/NoPadding`
- `DESEDE/CBC/NoPadding`
- `AES/ECB/NoPadding` (128-bit, 192-bit, and 256-bit keys)
- `AES/CBC/NoPadding` (128-bit, 192-bit, and 256-bit keys)
- `AES/CTR/NoPadding` (128-bit, 192-bit, and 256-bit keys)

2.3.1.2 Decryption

`Cipher.DECRYPT_MODE` supports the following algorithms:

- `DES/ECB/NoPadding`
- `DES/CBC/NoPadding`
- `DESEDE/ECB/NoPadding`
- `DESEDE/CBC/NoPadding`
- `AES/ECB/NoPadding` (128-bit, 192-bit, and 256-bit keys)
- `AES/CBC/NoPadding` (128-bit, 192-bit, and 256-bit keys)
- `AES/CTR/NoPadding` (128-bit, 192-bit, and 256-bit keys)
- `RSA/NONE/NoPadding` (1024-bit to 2048-bit keys)
- `RSA/NONE/PKCS1Padding` (1024-bit to 2048-bit keys)
- `RSA/NONE/OAEPWithMD5AndMGF1Padding` (1024-bit to 2048-bit keys)
- `RSA/NONE/OAEPWithSHA1AndMGF1Padding` (1024-bit to 2048-bit keys)
- `RSA/NONE/OAEPWithSHA224AndMGF1Padding` (1024-bit to 2048-bit keys)
- `RSA/NONE/OAEPWithSHA256AndMGF1Padding` (1024-bit to 2048-bit keys)
- `RSA/NONE/OAEPWithSHA384AndMGF1Padding` (1024-bit to 2048-bit keys)
- `RSA/NONE/OAEPWithSHA512AndMGF1Padding` (1024-bit to 2048-bit keys)
- `ECCElGamal/NONE/NoPadding` (only with the standard 160-bit, 192-bit, 224-bit, and 256-bit curves)

For RSA algorithms, you may also use “ECB” instead of “NONE”.

2.3.1.3 Wrapping

`Cipher.WRAP_MODE` supports the `AES/CBC/ISO10126Padding` wrapping algorithm with 128-bit, 192-bit, and 256-bit keys.

This algorithm can be used to wrap DES, AES, and ECC keys.

2.3.1.4 Unwrapping

`Cipher.UNWRAP_MODE` supports the following algorithms:

- `AES/ECB/NoPadding` (128-bit, 192-bit, and 256-bit keys) can unwrap DES and AES keys
- `AES/CBC/NoPadding` (128-bit, 192-bit, and 256-bit keys) can unwrap DES, AES, and ECC keys
- `AES/CBC/ISO10126Padding` (128-bit, 192-bit, and 256-bit keys) can unwrap DES, AES, RSA, and ECC keys
- `AES/CTR/NoPadding` (128-bit, 192-bit, and 256-bit keys) can unwrap DES, AES, RSA, and ECC keys
- `RSA/NONE/NoPadding` (1024-bit to 2048-bit keys) can unwrap DES and AES keys
- `RSA/NONE/PKCS1Padding` (1024-bit to 2048-bit keys) can unwrap DES and AES keys
- `RSA/NONE/OAEPWithMD5AndMGF1Padding` (1024-bit to 2048-bit keys) can unwrap DES and AES keys
- `RSA/NONE/OAEPWithSHA1AndMGF1Padding` (1024-bit to 2048-bit keys) can unwrap DES and AES keys
- `RSA/NONE/OAEPWithSHA224AndMGF1Padding` (1024-bit to 2048-bit keys) can unwrap DES and AES keys
- `RSA/NONE/OAEPWithSHA256AndMGF1Padding` (1024-bit to 2048-bit keys) can unwrap DES and AES keys
- `RSA/NONE/OAEPWithSHA384AndMGF1Padding` (1024-bit to 2048-bit keys) can unwrap DES and AES keys
- `RSA/NONE/OAEPWithSHA512AndMGF1Padding` (1024-bit to 2048-bit keys) can unwrap DES and AES keys
- `ECCElGamal/NONE/NoPadding` (only with the standard 160-bit, 192-bit, 224-bit, and 256-bit curves) can unwrap DES and AES keys
- `AesWrap/NONE/NoPadding` (corresponds to AES unwrapping defined by NIST) can unwrap AES keys
- `AES/CMLA/NoPadding` can unwrap DES and AES keys
- `RSA/CMLA/NoPadding` can unwrap DES and AES keys

⚠ When unwrapping AES keys, we recommend that you specify the wrapped key algorithm as "AES128", "AES192", or "AES256", because "AES" will automatically choose the largest possible AES key.

For RSA algorithms, you may also use "ECB" instead of "NONE".

2.3.1.5 Unwrapping In "NoPadding" Modes

If data is N bits long, and the key you want to get is K bits long, the following algorithm is executed in `NoPadding` modes:

1. SKB verifies that K is equal or less than N.
2. SKB takes the K most significant bits of the unwrapped data and uses them as the key.

2.3.1.6 Unwrapping Using the ElGamal ECC Algorithm

ECC uses the 4 lowest bytes to translate data into a valid ECC point. When decrypting, SKB ignores this, and thus using an N-bit key would give N bits of data. However, when unwrapping, SKB always crops the 4 bytes, and thus an N-bit key gives N-32 bits of data.

2.3.1.7 Default Values

The following are the default values for all cipher modes:

- `DES` defaults to `DES/ECB/NoPadding`
- `DESEDE` defaults to `DESEDE/ECB/NoPadding`
- `AES` defaults to `AES/ECB/PKCS5Padding` (currently, not supported)
- `RSA` defaults to `RSA/NONE/Pkcs1Padding`
- `ECCElGamal` defaults to `EccElGamal/NONE/NoPadding`
- `AesWrap` defaults to `AesWrap/NONE/NoPadding`

2.3.2 Algorithm Parameters

The SKB JCA Provider offers two implementations of the `AlgorithmParameterSpec` interface to handle initialization of specific key types. The following subsections explain these implementations.

2.3.2.1 SkbEcParameterSpec

This class is used to provide initialization parameters for ECC keys. Since SKB supports both standard and custom ECC curves, two constructors are available for both cases.

To use standardized ECC curve types, the following constructor is available:

```
SkbEcParameterSpec(int keySize)
```

where `keySize` is one of the following:

- 160: 160-bit prime curve recommended by SECG, SECP R1
- 192: 192-bit prime curve recommended by NIST (same as 192-bit SECG, SECP R1)
- 224: 224-bit prime curve recommended by NIST (same as 224-bit SECG, SECP R1)
- 256: 256-bit prime curve recommended by NIST (same as 256-bit SECG, SECP R1)
- 384: 384-bit prime curve recommended by NIST (same as 384-bit SECG, SECP R1)
- 521: 521-bit prime curve recommended by NIST (same as 521-bit SECG, SECP R1)

To use ECC curve types with custom domain parameters, the following constructor is available:

```
SkbEcParameterSpec(int[] prime,
                   int[] a,
                   int[] gx,
                   int[] gy,
                   int[] order,
                   int   primeBitLen,
                   int   orderBitLen,
                    int[] randomVal)
```

where `prime`, `a`, `gx`, `gy`, `order`, and `randomVal` must contain ECC domain parameters in protected form. You can obtain the protected form of each parameter using the Custom ECC Tool, which is included in the package of the native SKB library.

2.3.2.2 SkbDhParameterSpec

This class is used to provide initialization parameters for the Classical Diffie-Hellman algorithm implementation. The constructor to be used has the following signature:

```
SkbDhParameterSpec(byte[] data, int[] randomValue, DHParameterSpec dhParamSpec)
```

`data` and `randomValue` contain input parameters in protected form. You can obtain the protected form of both parameters using the Diffie-Hellman Tool, which is included in the package of the native SKB library.

`dhParamSpec` must contain the same parameters but in plain form. The algorithm requires this to be able to provide the public key.

2.3.3 KeyFactory

`KeyFactory` supports the `RSA` and `EC` algorithms.

The following key specifications are supported:

- `SkbExportedKeySpec` represents the safe SKB-protected format of keys that can be saved on a persistent storage. You can use this specification to safely import and export keys.

  To create an instance of `SkbExportedKeySpec` from a buffer of exported data, you must use its constructor that has the following signature:

  ```
  SkbExportedKeySpec(String algorithm, byte[] encoded)
  ```

  `encoded` is the buffer of exported data.

- `SkbEcPublicKeySpec` is used to obtain the `SkbEcPublicKey` object (public ECC key) from the `SkbEcPrivateKey` object (private ECC key). To do this, you have to manually call the specification's constructor, which has the following signature:

  ```
  SkbEcPublicKeySpec(String             algorithm,
                     SkbEcParameterSpec skbEcParamSpec,
                     PrivateKey         ecPrivateKey)
  ```

  `skbEcParamSpec` is an object that specifies the ECC curve type used. For information on this object, see §2.3.2.1.

  `ecPrivateKey` must be the `SkbEcPrivateKey` object generated using either `KeyFactory` or `KeyPairGenerator` (see §2.3.6).

- `PKCS8EncodedKeySpec` is used to load plain RSA keys (encoded in the PKCS #8 format) and convert them to the SKB-protected format. You cannot use this specification to save RSA keys in plain form.

- `ECPrivateKeySpec` is used to load plain ECC keys and convert them to the SKB-protected format. You cannot use this specification to save ECC keys in plain form.

Using `PKCS8EncodedKeySpec` and `ECPrivateKeySpec` specifications is insecure. These operations will only be available if loading of plain keys is enabled in SKB.

2.3.4 SecretKeyFactory

`SecretKeyFactory` supports the following algorithms:

- `DES`
- `DESEDE`
- `AES` (128-bit, 192-bit, and 256-bit keys)
- `HmacSHA1`
- `HmacSHA224`
- `HmacSHA256`
- `HmacSHA384`
- `HmacSHA512`
- `HmacMD5`

The following key specifications are supported:

- `SkbExportedKeySpec` represents the safe SKB-protected format of keys that can be saved on a persistent storage. You can use this specification to safely import and export keys.

  To create an instance of `SkbExportedKeySpec` from a buffer of exported data, you must use its constructor that has the following signature:

  ```
  SkbExportedKeySpec(String algorithm, byte[] encoded)
  ```

  `encoded` is the buffer of exported data.

- `SecretKeySpec` is used to load plain keys and convert them to the SKB-protected format. You cannot use this specification to save keys in plain form.
- `DESKeySpec` is used to load plain DES keys and convert them to the SKB-protected format. You cannot use this specification to save keys in plain form.
- `DESedeKeySpec` is used to load plain Triple DES keys and convert them to the SKB-protected format. You cannot use this specification to save keys in plain form.

  Using `SecretKeySpec`, `DESKeySpec`, and `DESedeKeySpec` specifications is insecure. These operations will only be available if loading of plain keys is enabled in SKB.

2.3.5 KeyGenerator

`KeyGenerator` supports the following algorithms:

- `DES`
- `DESEDE` with 128-bit an 192-bit keys (192-bit keys are generated by default)
- `AES` with 128-bit, 192-bit, and 256-bit keys (128-bit keys are generated by default)
- `HmacMD5`
- `HmacSHA1`
- `HmacSHA224`
- `HmacSHA256`
- `HmacSHA384`
- `HmacSHA512`

2.3.6 KeyPairGenerator

`KeyPairGenerator` supports the following algorithms:

- `EC`
- `ECDH`
- `DH`

If you use the `EC` or `ECDH` algorithm, you have to use the `SkbEcParameterSpec` object to specify the ECC curve type as described in §2.3.2.1.

For the `DH` algorithm, you have to use the `SkbDhParameterSpec` object to provide the input parameters as described in §2.3.2.2.

EC and ECDH are not compatible formats.

2.3.7 Signature

`Signature` supports the following algorithms:

- `NONEWITHRSA`
- `SHA1WITHRSA`
- `SHA224WITHRSA`
- `SHA256WITHRSA`
- `SHA384WITHRSA`
- `SHA512WITHRSA`

- `MD5WITHRSA`
- `SHA1WITHRSAANDMGF1`
- `SHA224WITHRSAANDMGF1`
- `SHA256WITHRSAANDMGF1`
- `SHA384WITHRSAANDMGF1`
- `SHA512WITHRSAANDMGF1`
- `MD5WITHRSAANDMGF1`
- `ECDSA`
- `NONEWITHECDSA`
- `SHA1WITHECDSA`
- `SHA224WITHECDSA`
- `SHA256WITHECDSA`
- `SHA384WITHECDSA`
- `SHA512WITHECDSA`
- `MD5WITHECDSA`

These algorithms can only be used for signing. This means that only the `SIGN` state of the `Signature` class is supported.

If you use the `ECDSA` algorithm, you have to use the `SkbEcParameterSpec` object to specify the curve type as described in §2.3.2.1.

ECDSA signatures are DER-encoded for compatibility with other JCA providers. Note that SKB does not encode the signature.

2.3.8 Mac

`Mac` supports the following algorithms:

- `HmacSHA1`
- `HmacSHA224`
- `HmacSHA256`
- `HmacSHA384`
- `HmacSHA512`
- `HmacMD5`
- `AESCMAC` (128-bit AES only)

2.3.9 KeyAgreement

`KeyAgreement` supports the `ECDH` and `DH` algorithms. For these algorithms, you have to use the `ECDH` or `DH` keys generated by `KeyPairGenerator` as described in §2.3.6.

⚠ `ECDH` format is not compatible with the `EC` format.

Currently, only secret keys (AES and DES) can be generated.

⚠ When generating AES keys, we recommend that you specify the key algorithm as “AES128”, “AES192”, or “AES256”, because “AES” will automatically choose the largest possible AES key.

2.3.10 MessageDigest

`MessageDigest` supports the following algorithms:

- `SHA-1`
- `SHA-224`
- `SHA-256`
- `SHA-384`
- `SHA-512`
- `MD5`

3 SKB Java Binding

This chapter provides a general overview of the SKB Java Binding. Information about individual classes and methods is available in the Javadoc source annotations.

⚠ We strongly recommend using the SKB JCA Provider instead of the SKB Java Binding, because backward compatibility of the SKB Java Binding API in future releases is not guaranteed.

3.1 SKB Java Binding Overview

The SKB Java Binding is a low-level API that provides an almost one-to-one mapping between native SKB elements and their Java counterparts. For example, the native `SKB_SecureData` class is exposed as the `SecureData` Java class; and the native method `SKB_SecureData_Export` is exposed as the method `export` of the `SecureData` Java class. Every method in the SKB Java API actually calls the corresponding method in the native SKB API.

The SKB Java Binding code is located in the `com.cryptanium.skb package`. For a complete reference of mappings between Java elements and native SKB elements, see the Javadoc.

The SKB JCA Provider is implemented using classes and methods of the SKB Java Binding.

3.2 Differences Between the SKB Java Binding and the Native SKB API

The SKB Java Binding is designed to be as similar to the native SKB API as possible. However, due to differences between the C and Java programming languages, some aspects of the APIs vary. This section describes the main differences.

3.2.1 Memory Management

There are no specific object releasing methods in the SKB Java Binding. SKB objects will be released automatically by the Java garbage collector when they are no longer used and needed. In Java, you do not have to worry about object lifecycle.

3.2.2 Pointers to the Owner Object in Method Calls

When you use methods of the native SKB API, the first parameter is always a pointer to the corresponding SKB object. For example, if you call the method `SKB_Engine_GetInfo(const SKB_Engine* self, SKB_EngineInfo* info)`, the parameter `self` is a pointer to the `SKB_Engine` object.

In the SKB Java Binding, every method belongs to a corresponding class. Therefore, the object parameter is not needed. For example, in the SKB Java Binding, the `SKB_Engine_GetInfo method` shown above is represented by the method `Engine.getInfo()`. As you can see, the parameter `self` is omitted.

For information on why the `info` parameter is also omitted, see §3.2.3.

3.2.3 Method Return Values

In the native SKB API, methods return a numeric value that represents the outcome of the execution. In cases when a native method is expected to return a particular data object, this is done by assigning a pointer to one of the method parameters. This parameter then points to the data object.

For example, if you call the method
`SKB_Engine_GetInfo(const SKB_Engine* self, SKB_EngineInfo* info)`, it does two things:

- returns an integer that represents the outcome of the execution (0 if execution was successful, and another value if an error occurred)
- if execution is successful, sets the `info` parameter to point to an `SKB_EngineInfo` structure, which describes the engine

The SKB Java Binding uses a completely different approach of returning method values. Firstly, it uses a special type of exception instead of the numeric method return values (see §3.2.4). Secondly, the data object that is logically to be returned by a method is always the actual return value of the method.

For example, if you call the method `Engine.getInfo()`, it does two things:

- throws an exception if an error occurs (see §3.2.4)
- if execution is successful, returns the `Info` object, which describes the engine

3.2.4 Exceptions

Unlike the native SKB API, the SKB Java Binding does not use numeric method return values to indicate the outcome and problems of SKB method execution. Instead, the SKB Java Binding uses a special exception class `com.cryptanium.skb.SkbException`. This type of exception is thrown from within the SKB Java Binding methods when some SKB-related problems are encountered. If, upon calling a particular SKB method, the `SkbException` is not thrown, you can assume the method was executed successfully.

You can use the following methods of the `SkbException` class to determine the cause of the exception:

`int getSkbResult()`

Returns an integer whose value is the actual error value returned by the native SKB method.

For information on SKB error values, see the *Secure Key Box User Guide*.

`String getMessage()`

Returns a `String` object that contains the error code and a brief explanation of what the error code means.

3.2.5 Buffer Size Parameters

Several methods of the native SKB API that work with data buffers require a separate parameter to specify the buffer size. In Java, data buffers are byte arrays and their size can be deducted from the array itself. Therefore, in the corresponding SKB Java Binding methods, buffer size parameters are omitted.

3.2.6 Parameter Classes

The native SKB API employs a number of parameter structures, which are used to hold a set of parameters to be passed to or returned by various SKB objects. For example, the `SKB_SignTransformParameters` structure specifies the signature algorithm and signing key for the signing transform object.

The SKB Java Binding defines a special class for each parameter structure of the native SKB API. These classes are located in the `com.cryptanium.skb.parameters` package. These classes have similar names and serve similar purposes as their native counterparts. For example, the native `SKB_SignTransformParameters` structure mentioned before is represented by the `SignTransformParameters` class in the SKB Java Binding.

What must be noted is that the SKB Java Binding organizes these classes into related groups. All classes of the same group implement the same interface. For example, the `DigestTransformParameters`, `SignTransformParameters`, and `VerifyTransformParameters` classes are implementations of the `TransformParameters` interface. Interfaces are frequently used as types of method parameters. This mechanism limits the range of parameter objects that can be passed as parameters to specific SKB object methods.