# Secure Key Box 5.42

User Guide

# Table of Contents



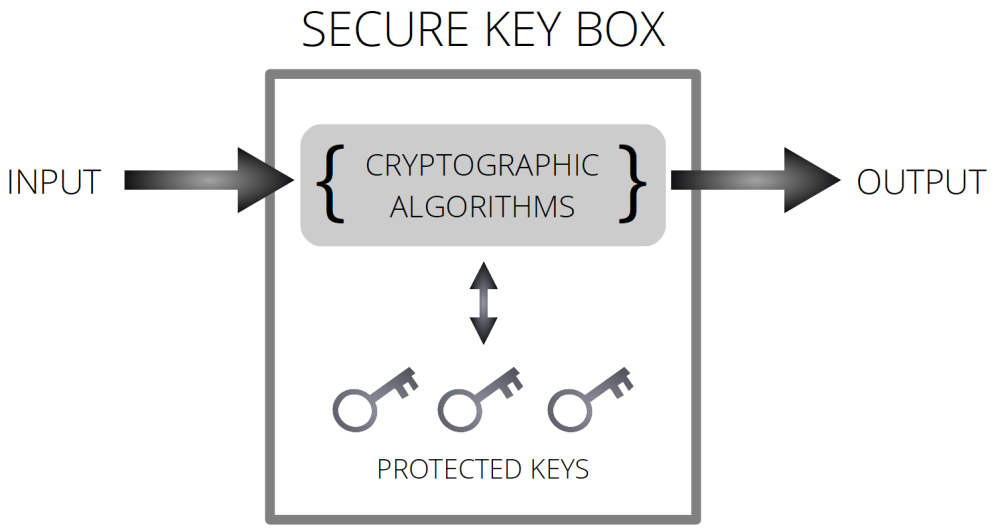
 

# 1. Introduction

This chapter provides a general overview of Secure Key Box (SKB).

## 1.1 What Is SKB?

SKB is a library that provides an API for executing the most common cryptographic algorithms in a secure manner. The main purpose of SKB is to hide and protect cryptographic keys so that they are never revealed in plain form, even during execution of cryptographic algorithms. With such security in place, it becomes extremely difficult for attackers to locate, modify, and extract the cryptographic keys.




The secure properties of SKB are based on white-box cryptography, which ensures that the implementation of cryptographic algorithms is secure in fully observable and modifiable execution environments. In other words, SKB is designed to protect cryptographic keys in open environments where anyone could potentially gain full control over the code, memory, and storage of the execution device. Typical examples of such open environments are smartphones, tablets, and desktop computers.

## 1.2 Terminology

This section describes several terms used throughout this User Guide as they should be understood in the context of SKB, because they may convey different and more specific meanings than what is commonly associated with such terms in everyday usage.

### 1.2.1 Security-Related Terms

This User Guide uses the terms "secure", "protect", "white-box protected", "safe", and variations of each to convey properties of SKB and the effect it has on the target application. As used herein, none of these terms describe an absolute condition. Use of SKB in compliance with this User Guide will not render any

application or data absolutely secure, absolutely protected or absolutely safe from unauthorized accessing, use or manipulation. In addition, use of these terms is not intended to convey a promise or warranty that SKB will never contain bugs, or that SKB will always operate without error.

## 1.2.2 Secure Data Objects

A secure data object is one of the basic concepts in the SKB protection scheme. It is a container within SKB of any sensitive data whose value is white-box protected. This means that the data is encrypted at all times, and even when SKB performs any operations with the data, it does so without ever exposing the contents in plain form. All keys stored and processed by SKB are secure data objects.

## 1.2.3 Raw Bytes

Raw bytes is a special case of a secure data object whose content is a buffer of arbitrary bytes. Most commonly, raw bytes are used to represent keys in symmetric-key algorithms, such as the AES, DES, and Speck ciphers. Raw bytes are not used for other ciphers, such as RSA and ECC, which demand a more sophisticated structure for keys.

## 1.2.4 Cryptographic Operations

When working with SKB, it is critical to understand what exactly is meant by certain terms related to cryptographic operations. Because different operations are intended for different purposes, and they work with different types of input and output, misusing the operations or combining them in a wrong way may render the use of SKB insecure or invalid.

The following table explains some of the most important operations as they are used in the context of SKB.

| **Term** | **Input** | **Output** | **Comments** |
|---|---|---|---|
| Encryption | Plain | Encrypted | Encrypts the input, and obtains a buffer of encrypted data. |
| Decryption | Encrypted | Plain | Decrypts the input, and obtains a buffer of plain data. |
| Wrapping | Secure data object | Encrypted | Encrypts the input key (stored within SKB), and outputs it as an encrypted buffer. The encrypted buffer can be decrypted by any other system (not just SKB) that has the corresponding unwrapping key. |
| Unwrapping | Encrypted | Secure data object | Decrypts the input without revealing it in plain form, and stores the decrypted data within SKB as a secure data object. |
| Wrapping plain data | Plain | Secure data object | Encrypts the plain input with a key, and stores the output as a new secure data object. |
| Derivation | Secure data object | Secure data object | Obtains a new key from an existing key by performing certain operations on it. Both keys are never revealed in plain form. |

# 1.3 Example Use Cases

SKB is used in all scenarios where there is a risk of cryptographic keys being discovered and used to cause harm. Here are some examples where this is the case.

## 1.3.1 Contactless Payments on Mobile Phones

Mobile phones are often used as contactless payment terminals. Developing payment software for general-purpose phones is significantly cheaper and more convenient than creating specialized hardware for traditional point-of-sale systems. The danger lies in the fact that software-based security systems are easier to reverse engineer than special-purpose hardware. By extracting the internal cryptographic keys, an attacker can collect sensitive data, steal money, or disrupt business operations. The industry-standard approach to mitigating key extraction risks on general-purpose devices is white-box cryptography. In fact, the Payment Card Industry Security Standards Council (PCI SSC) requires all vendors to employ white-box cryptography for mobile contactless payment apps to protect keys. SKB is a white-box cryptography library, which means that by using it in a software-based payment system you are able to satisfy the PCI SSC security and testing requirements and ensure strong security.

## 1.3.2 Device Personalization

An Internet of things (IoT) manufacturer produces devices that are distributed to customers. Each new device must be personalized upon startup to be able to authenticate with other devices. This personalization involves receiving a certified digital identity from the manufacturer’s server. The digital identity contains a unique encrypted authentication key. To decrypt the identity, the device uses a master key that is hard-coded into the device firmware. As can be quickly seen, this scenario has several critical parts that are susceptible to key attacks. For example, by extracting the master key the attacker can freely add unauthorized devices to the network and therefore cause loss of profit. Another attack vector is to intercept the authentication key once it is decrypted on the device; this would allow the attacker to impersonate devices, intercept secret communication, or mount denial of service attacks. These threats can be prevented by employing the secure cryptographic algorithms provided by SKB, which never reveal keys as plaintext and can withstand various key-extraction and side-channel attacks.

## 1.3.3 Remote Control App for Cars

A car manufacturer offers a mobile app that allows its customers to remotely park their cars. For security reasons, the app needs to maintain constant communication with the car to ensure that the owner is present while the car is being parked. This involves mutual authentication between the app and the car that must be protected to prevent spoofing of the owner's presence. An attacker who is able to extract the cryptographic keys used in the communication may impersonate the owner to steal the car or cause an accident. To prevent these risks, the remote control app can use SKB in its code to protect the keys. If the app is protected in this way, the attacker will find it almost impossible to reverse engineer the app, discover the internal secrets, and misuse the remote control functionality. Moreover, the internal

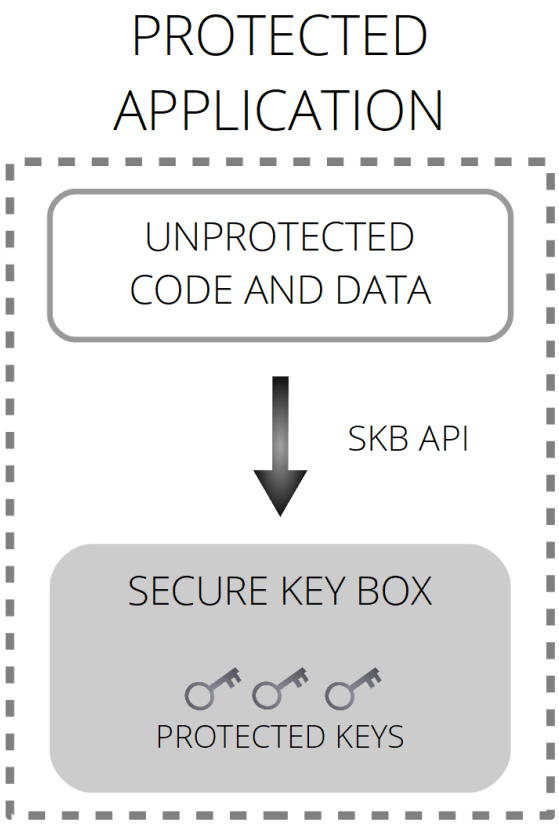
cryptographic modules within the car’s firmware can also benefit greatly from the secure properties of SKB, thus preventing reverse engineering of the cryptographic algorithms whose integrity is the foundation of most security features of modern cars.

## 1.4 Applying SKB Protection

Because SKB provides standards-compliant implementations of the most common cryptographic algorithms, SKB can, in most cases, simply replace the cryptographic module of the target application. Thus, the SKB-protected application will be functionally equivalent to the original application, but at the same time it will ensure robust protection of its keys.

So, the general procedure for applying SKB protection is as follows:

1. Include the SKB library into the target application you want to protect.
2. Rewrite the low-level cryptographic functions of the target application so that they employ the SKB API.




## 1.5 SKB APIs

SKB provides several APIs for accessing its core features as described in the following subsections. You must choose the right API depending on your software development environment.

### 1.5.1 Native API

The Native API is used when the cryptographic module of your application is written in C/C++. For this purpose, SKB is provided as a static library, which must be linked with your application. The native implementation supports all features of SKB on a wide range of target platforms. In most cases, this *User Guide* refers to the Native API.

### 1.5.2 Java API

The Java API is used when the secure cryptographic SKB functions must be called from Java. The Java API follows the design principles of Java Cryptographic Architecture (JCA), which means that the Java API is accessible through an implementation of the JCA `Provider` class. Internally, the Java API invokes functions of the Native API from a specially built shared library of SKB. The Java API supports only a subset of algorithms and target platforms when compared to the Native API.

### 1.5.3 JavaScript API

The JavaScript API is used when the secure cryptographic SKB functions must be called from JavaScript. The JavaScript API is the implementation of the Web Cryptography API. Internally, the JavaScript API invokes functions of the Native API from a specially built WebAssembly module. The JavaScript API supports only a subset of algorithms when compared to the Native API.

## 1.6 SKB Extensions

In addition to its core features, SKB also provides a number of extensions that expand its functionality for specific purposes. The available extensions are described in the following subsections.

### 1.6.1 Transport Layer Security Support

SKB provides the following two alternative Transport Layer Security (TLS) support extensions for implementing TLS communication in such a way that the secure keys involved in authentication and traffic encryption are always protected:

- An SKB-specific provider for OpenSSL 3 allows you to use the standard OpenSSL API with the secure properties of SKB.
- The proprietary TLS Library is a static library that exposes a small set of high-level methods that take care of the entire TLS communication process.

> **Important**
>
> The TLS Library is deprecated and will be removed in future releases of SKB.

### 1.6.2 Secure Database Library

The Secure Database Library provides the ability to create and use encrypted databases. This means that all data files produced and accessed via this library will be encrypted at all times, even when in use. The implementation is an extension of the SQLite library. Therefore, using the Secure Database Library is very similar to using SQLite.

### 1.6.3 DUKPT API

The DUKPT API is an extension API designed to implement working key derivation algorithms of the originating Secure Cryptographic Device according to the Derived Unique Key Per Transaction (DUKPT) key management scheme, which is specified by ANSI X9.24-3-2017.

### 1.6.4 Secure PIN

Secure PIN is an SKB extension that allows you to implement secure graphical user interface (GUI) based PIN entry in Android applications. Secure PIN is designed to satisfy dozens of requirements put forth by various standards associated with PIN entry on commercial off-the-shelf (COTS) devices.

### 1.6.5 "SkbTrustedStorage" Library

The `SkbTrustedStorage` library provides read-only access to the Trusted Storage of the Trustonic Application Protection (TAP) SDK. This library allows you to retrieve data and keys from the Trusted Storage to facilitate integration with and migration from the TAP SDK.

## 1.7 Supported Algorithms

This section lists the main cryptographic algorithms supported by SKB. Please contact Zimperium if you require algorithms that are not listed here.

**Important**

Your SKB package includes only those algorithms that you specifically requested from Zimperium.

### 1.7.1 Encryption

SKB supports the following encryption algorithms.

- DES in ECB mode (no padding) and CBC mode (no padding)
- Triple DES in ECB mode (no padding) and CBC mode (no padding) with two keying options:
  - All three keys are distinct.
  - Key 1 is the same as key 3.

- 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding), CTR mode, GCM mode, and CCM mode
- Speck using 128-bit keys and 64-bit blocks in ECB mode (no padding), CBC mode (no padding), and CTR mode

## 1.7.2 Decryption

SKB supports the following decryption algorithms.

- DES in ECB mode (no padding) and CBC mode (no padding)
- Triple DES in ECB mode (no padding) and CBC mode (no padding) with two keying options:
  - All three keys are distinct.
  - Key 1 is the same as key 3.
- 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding), CTR mode, GCM mode, and CCM mode
- Speck using 128-bit keys and 64-bit blocks in ECB mode (no padding), CBC mode (no padding), and CTR mode
- 1024-bit to 4096-bit RSA with no padding, PKCS#1 version 1.5 padding, and OAEP padding using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512

  The size of the private RSA keys must be a multiple of 128 bits.
- ElGamal ECC

  For supported curve types, see Supported ECC Curve Types.
- AES key decryption based on the unwrapping algorithm defined by NIST in the Special Publication 800-38F with 128-bit, 192-bit, and 256-bit AES keys

## 1.7.3 Signing

SKB supports the following signing algorithms.

- 128-bit, 192-bit, and 256-bit AES-CMAC
- Speck-CMAC using 128-bit keys and 64-bit blocks
- HMAC using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 as the hash function
- ISO/IEC 9797-1 MAC algorithm 3 for DES (also known as Retail MAC)

  Padding for this algorithm is not supported; the size of the input buffer should be a multiple of the DES block size, which is 64 bits.
- 1024-bit to 4096-bit RSA with PKCS#1 version 1.5 padding and PSS padding using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 as the hash function

The size of the private RSA keys must be a multiple of 128 bits.

A special variant of the RSA signing algorithm (with PKCS#1 version 1.5 padding) is provided that does not use a hash function. In this case, SKB will expect the input to be a short unencrypted message (such as a digest) prepared outside of SKB and formatted as described in SKB_Transform_AddBytes.

- DSA specified by the FIPS 186-4 standard

  The input is expected to be a short unencrypted message (such as a digest) prepared outside of SKB.

- ECDSA using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 as the hash function

  For supported curve types, see Supported ECC Curve Types.

  You may also use this algorithm without a hash function. In that case, SKB will expect that the input is a short unencrypted message (such as a digest) prepared outside of SKB.

- Elliptic curve signing algorithm EdDSA with the edwards25519 curve

## 1.7.4 Signature Verification

SKB supports the following signature verification algorithms.

- 128-bit, 192-bit, and 256-bit AES-CMAC
- Speck-CMAC using 128-bit keys and 64-bit blocks
- HMAC using MD5, SHA-1, SHA-224, SHA-256, SHA-384, or SHA-512 as the hash function
- ISO/IEC 9797-1 MAC algorithm 3 for DES (also known as Retail MAC)

  Padding for this algorithm is not supported; the size of the input buffer should be a multiple of the DES block size, which is 64 bits.

## 1.7.5 Hash Functions

SKB supports the following hash functions.

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

## 1.7.6 Unwrapping

SKB supports the following key unwrapping algorithms.

- unwrapping raw bytes using Triple DES in ECB mode (no padding and XML encryption padding) and CBC mode (no padding and XML encryption padding)
- unwrapping raw bytes using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding and XML encryption padding), CBC mode (no padding and XML encryption padding), CTR mode, and GCM mode

  The ECB mode and the CBC mode without padding can only be used to unwrap a buffer of raw bytes whose length is a multiple of 128 bits.
- unwrapping private 1024-bit to 4096-bit RSA keys using 128-bit, 192-bit, and 256-bit AES in CBC mode (XML encryption padding), CTR mode, and GCM mode
- unwrapping private 786-bit to 2048-bit RSA keys in the Chinese Reminder Theorem (CRT) format using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding) and CBC mode (no padding)
- unwrapping private ECC keys using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding and XML encryption padding), CTR mode, and GCM mode

  The ECB mode and the CBC mode without padding can be used only if the length of the private ECC key is a multiple of 128 bits.
- unwrapping private DSA keys using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding and XML encryption padding), CTR mode, and GCM mode
- unwrapping raw bytes using 1024-bit to 4096-bit RSA (no padding, PKCS#1 version 1.5 padding, and OAEP padding with SHA-1 or SHA-256)

  The size of the private RSA keys must be a multiple of 128 bits.
- unwrapping raw bytes using ElGamal ECC

  For supported curve types, see Supported ECC Curve Types.
- AES key unwrapping defined by NIST in the Special Publication 800-38F with 128-bit, 192-bit, and 256-bit AES keys
- unwrapping raw bytes as defined by ASC X9 TR 31-2018 with the following ciphers:
  - 128-bit, 192-bit, and 256-bit AES (Key Block Version ID "D")
  - two-key (128-bit) and three-key (192-bit) Triple DES (Key Block Version ID "B")
- unwrapping using XOR

## 1.7.7 Wrapping

SKB supports the following key wrapping algorithms.

- wrapping raw bytes using Triple DES in ECB mode (no padding) and CBC mode (XML encryption padding)
- wrapping raw bytes using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding), CBC mode (XML encryption padding), and CTR mode

The ECB mode can only be used to wrap a buffer of raw bytes whose length is a multiple of 128 bits.

- wrapping raw bytes using 1024-bit to 4096-bit RSA (no padding, PKCS#1 version 1.5 padding, and OAEP padding with SHA-1 or SHA-256)

  The size of the public RSA keys must be a multiple of 128 bits.

- wrapping private ECC keys using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding), CBC mode (XML encryption padding), and CTR mode

  The ECB mode can be used only if the length of the private ECC key is a multiple of 128 bits.

- wrapping private DSA keys using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding), CBC mode (XML encryption padding), and CTR mode
- AES key wrapping defined by NIST in the Special Publication 800-38F with 128-bit, 192-bit, and 256-bit AES keys
- wrapping raw bytes as defined by ASC X9 TR 31-2018 with the following ciphers:
  - 128-bit, 192-bit, and 256-bit AES (Key Block Version ID "D")
  - two-key (128-bit) and three-key (192-bit) Triple DES (Key Block Version ID "B")
- wrapping using XOR

## 1.7.8 Wrapping of Plain Data

SKB supports the following special algorithms that deal with wrapping of plain data.

- wrapping plain data using Triple DES in ECB mode (no padding) and CBC mode (no padding)
- wrapping plain data using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding) and CBC mode (no padding)

## 1.7.9 Key Generation

SKB supports the following key generation algorithms.

- random buffer of bytes (for example, DES and AES keys)
- 1024-bit to 4096-bit RSA key pairs

  **Important**

  In SKB, RSA key generation is a convenience feature that must not be used in security-critical environments. Before the generated RSA keys are obtained in the secure format, they are briefly exposed in the memory as plain keys.

- DSA key pairs

- ECC key pairs

  For supported curve types, see Supported ECC Curve Types.

## 1.7.10 Key Agreement

SKB supports the following key agreement algorithms.

- Classical Diffie-Hellman (DH) with up to 1024-bit prime P
- Elliptic curve Diffie-Hellman (ECDH)

  For supported curve types, see Supported ECC Curve Types.
- Montgomery-X-coordinate DH function using Curve25519

## 1.7.11 Key Derivation

SKB supports the following key derivation algorithms.

- slicing (selecting a substring of bytes from another key)
- concatenating two keys
- selecting odd or even bytes
- encrypting and decrypting raw bytes using the following algorithms:
  - 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding) and CBC mode (no padding)
  - DES in ECB mode (no padding) and CBC mode (no padding)
  - Triple DES in ECB mode (no padding) and CBC mode (no padding)
- iterated SHA-1
- SHA-256 and SHA-512 with plain prefix and suffix
- SHA-384
- HMAC using SHA-256, SHA-384, or SHA-512 as the hash function
- reversing the order of bytes
- NIST 800-108 key derivation with either 128-bit AES-CMAC or Speck-CMAC in counter mode
- KDF2 used in the RSAES-KEM-KWS scheme of the OMA DRM Specification
- deriving raw bytes from a private ECC key
- 128-bit AES encryption in ECB mode (no padding) with a concatenated key and optional SHA-1 function
- XOR-ing a key with either plain data or another key
- deriving a key by encrypting a plain 16-byte buffer two times using the AES cipher

- HMAC-based Extract-and-Expand Key Derivation Function (HKDF) with SHA-256 as defined by RFC 5869

## 1.8 Supported Platforms

The following table lists target platforms and architectures supported by SKB, and the build systems used to build each corresponding edition of the SKB library.

The "Java" column indicates whether the Java API is available for that platform. The "zShield" column indicates whether zShield tamper resistance is available for that platform.

| Platform | Architectures | Java | zShield | Built With |
|---|---|---|---|---|
| Android | x86, x86_64, 32-bit/64-bit ARM | Yes | Yes | Android NDK r23 |
| iOS/iPadOS (with and without bitcode) | 32-bit/64-bit ARM | No | Yes | Xcode 12.4 |
| macOS | x86_64, 64-bit ARM | Yes | Yes | Xcode 12.4 |
| tvOS | 64-bit ARM | No | Yes | Xcode 12.4 |
| watchOS | 32-bit/64-bit ARM | No | No | Xcode 12.4 |
| Mac Catalyst | x86_64, 64-bit ARM | No | Yes | Xcode 12.4 |
| Windows | x86, x86_64 | Yes | Yes | Visual Studio 2013, 2015, and 2017* |
| Universal Windows Platform | x86, x86_64 | No | No | Visual Studio 2017 |
| glibc/Linux (most popular Linux distributions) | x86, x86_64, 32-bit ARM (including hard float), 64-bit ARM | Yes | Yes | GCC 7.5 (x86, x86_64, 32-bit ARM, and 64-bit ARM) crosstool-NG 1.24.0 GCC 4.8 (CentOS) |
| uClibc/Linux | ARM (hard float), MIPS, MIPSel | Yes | Yes** | crosstool-NG 1.24.0 GCC 5.5 |
| musl/Linux | x86_64 | Yes | Yes | crosstool-NG 1.24.0 GCC 5.5 |
| WebAssembly | wasm32 | No | No | Emscripten 1.39.10 |
| MinGW | x86, x86_64 | No | No | MSYS2 Rev2 GCC 7.3 (x86) MSYS2 Rev3 GCC 8.2 (x86_64) |
| PlayStation 5 | Prospero | No | No | Visual Studio 2017 |
| Xbox (Series S and X) | x64 | No | No | LLVM (clang-cl) Visual Studio 2019 |

*SKB that is built with Visual Studio 2015 or 2017 is binary-compatible with later versions of Visual Studio as described in this article. Also, the Visual Studio 2013 and 2015 targets are deprecated and will be removed in future releases of SKB.

**Except MIPS.

Other target platforms, architectures, and build systems are not guaranteed to work with SKB. Please contact Zimperium if you have any questions in this regard, or if you require support for combinations of platforms, architectures, and build systems other than listed in the table above.

## 1.9 Supported ECC Curve Types

SKB supports ECC curves that correspond to the Weierstrass equation ($y^2 = x^3 + ax + b$) and whose curve parameters are at most 528 bits long.

While you can define and use ECC curves with arbitrary parameters (called custom curves), SKB provides the following predefined set of commonly used ECC curve types (called standard curves):

- secp160r1, which is the 160-bit prime curve recommended by SECG
- the following set of curves recommended by NIST:
  - P-192 (also known as secp192r1 and prime192v1)
  - P-224 (also known as secp224r1)
  - P-256 (also known as secp256r1 and prime256v1)
  - P-384 (also known as secp384r1)
  - P-521 (also known as secp521r1)

**Important**

ElGamal ECC decryption and ElGamal ECC key unwrapping support only the standard 160-bit, 192-bit, 224-bit, and 256-bit curve types. ECC key generation, ECDSA, and ECDH support the full set of the standard curve types, as well as the custom curve types.

## 1.10 SKB Editions

Two editions of SKB are available to customers as described in the following subsections.

### 1.10.1 Evaluation

The evaluation edition of SKB has an expiration date, after which SKB will no longer be usable. Also, evaluation editions use the same encoding to store secure data objects on a persistent storage as described in Understanding Key Exporting and Importing. This means that any user who has an evaluation edition of SKB can potentially load and use keys from all other evaluation editions.

Additionally, when an evaluation edition of SKB is run, it will produce a text file listing all the SKB features that were used during runtime. This information can help you remove SKB features that are not needed, thus reducing the overall binary size of your application. For more information on this text file, see Determining the Set of Used Modules.

If you are experiencing any problems when evaluating SKB, the evaluation edition of SKB allows you to enable debug logging.

### 1.10.2 Production

The production edition of SKB does not have an expiration date, and the way it stores secure data objects in memory and on a persistent storage is unique. This means that the production edition of SKB is protected against attacks where the attacker attempts to retrieve its keys by importing them into another SKB-protected application.

## 1.11 Package Structure

The SKB package delivered to you contains the following main directories and files that you should be aware of:

- `/bin/`

  Contains the following:

  - SKB command-line utilities:
    - Key Export Tool
    - Custom ECC Tool

      This utility will be present only if you selected it when requesting the SKB package.
    - Diffie-Hellman Tool

      This utility will be present only if you selected the Classical Diffie-Hellman algorithm when requesting the SKB package.
    - Key Embedding Tool

      This utility will be present only if you selected it when requesting the SKB package.
    - Binary Update Tool

      This utility will be present only if you selected zShield tamper resistance when requesting the SKB package.

> **Important**
>
> If you have ordered an SKB package that is protected with zShield, you must always run the Binary Update Tool on the final protected application once it is built. Otherwise, the protected application will crash at runtime with a segmentation fault.

- Secure Database Shell Tool

  This utility will be present only if you selected the Secure Database Library when requesting the SKB package.

- Dynamic library of the SKB provider for OpenSSL

  This library will be present only if you selected the SKB OpenSSL provider library when requesting the SKB package.

- `/Docs/`

  Contains SKB documentation.

- `/Examples/`

  Contains SKB examples.

  Each example has a `README.txt` file describing the purpose of the example and steps to build and run it.

- `/Include/`

  Contains the following header files:

  - `SkbSecureKeyBox.h`: Main interface of the Native API
  - `SkbInternalHelpers.h`: Interface of the Sensitive Operations Library
  - `SkbSetExportKey.h`: Interface for functions that deal with custom export keys produced by the Key Embedding Tool

    This file will be present only if you selected the Key Embedding Tool when requesting the SKB package.

  - `SkbSecureDatabase.h` and `sqlite3.h`: Interfaces required for using the Secure Database Library

    These files will be present only if you selected the Secure Database Library when requesting the SKB package.

  - `SkbOpenSslProvider.h` and `SkbTls.h`: Interfaces for implementing the TLS protocol using SKB

    These files will be present only if you selected the corresponding TLS features when requesting the SKB package.

  - `SkbDukpt.h`: Interface of the DUKPT API

    This file will be present only if you selected the DUKPT feature when requesting the SKB package.

  - `SkbTrustedStorage.h`: Interface of the `SkbTrustedStorage` library

This file will be present only if you selected the Trusted Storage feature when requesting the SKB package.

- `/lib/`

  Contains the following components:

  - main static SKB library (`libSecureKeyBox.a` or `SecureKeyBox.lib`) for every target platform
  - Sensitive Operations Library (`libSkbInternalHelpers.a` or `SkbInternalHelpers.lib`) for every target platform
  - Compiled edition of the Platform Library (`libSkbPlatform.a` or `SkbPlatform.lib`) for every target platform

    **Note**

    Source code of the Platform Library is provided in the `src` directory.

  - SKB Java API files:
    - Native shared SKB library that is used by the SKB Java API (`libSecureKeyBoxJava.so`, `SecureKeyBoxJava.dll`, or `libSecureKeyBoxJava.dylib`)

      A separate edition of this library is provided for every target platform supported by the Java API.
    - `skb.jar` file, which contains all Java classes of the SKB Java API
  - SKB JavaScript API files:
    - WebAssembly module (`SkbWebCrypto.wasm`) used by the JavaScript API
    - SKB JavaScript file (`SkbWebCrypto.js`), which serves as the primary interface for the JavaScript API

    These files will be present only if you selected the WebAssembly target platform when requesting the SKB package.
  - Secure Database Library files:
    - Shared library (`SkbSecureDatabase.lib` or `libSkbSecureDatabase.a`) that provides the Secure Database API
    - SQLite library (`sqlite3.lib` or `libsqlite3.a`)

    These libraries will be present only if you selected the Secure Database Library when requesting the SKB package.
  - TLS-specific static libraries:
    - SKB provider library for OpenSSL (`libSkbOpenSslProvider.a` or `SkbOpenSslProvider.lib`)
    - TLS Library (`libSkbTls.a` or `SkbTls.lib`)

    These libraries will be present only if you selected the corresponding TLS features when requesting the SKB package.

- Library for extracting data and keys from the Trusted Storage (`libSkbTrustedStorage.a`)

  This library will be present only if you selected the Trusted Storage feature when requesting the SKB package.

- `securepin.aar` and `libSecurePin.nwdb` files used to implement secure PIN entry

  **Note**

  The `libSecurePin.nwdb` will be present only if you have ordered a zShield-protected SKB package.

  These files will be present only if you selected the secure PIN entry feature when requesting the SKB package.

- `/src/SkbModules`

  Contains the necessary files that allow you to reduce the number of SKB modules linked into your application.

  This directory will be present only if you selected the **Link-time module configuration** option when requesting the SKB package.

- `/src/SecurePIN`

  Contains the Python script that allows you to generate layout configuration for Secure PIN.

  This directory will be present only if you selected the secure PIN entry feature when requesting the SKB package.

- `/src/SkbPlatform`

  Contains source code of the Platform Library.

  The Platform Library is built using CMake. For information on how to do it, see the `README.txt` file, which is located in the same directory.

- `/export.id`

  Text file containing the identifier of the export key included in this SKB package, as well as identifiers of all export keys whose exported data this SKB package is capable of upgrading.

  Contents of the file resemble the following:

```
Legacy key 0 ID: «export key identifier»
Legacy key 1 ID: «export key identifier»
Legacy key 2 ID: «export key identifier»
...
Current key version «N» ID: «export key identifier»
```

  "Legacy key" specifies identifiers of export keys whose exported data can be upgraded by this SKB. "Current key" specifies the identifier of the export key that is used by this SKB to encrypt exported data.

  This file will not be present in packages that contain the Key Embedding Tool.

- `/License.txt`

  Text file containing the SKB license and licenses of third-party products included in SKB.

- `/Release-Notes.txt`

  Release Notes listing the new features and changes introduced in this release of SKB, as well as a change log of previous releases.

# 2. Key Management

This chapter describes the general concepts and procedures related to correct and safe handling of keys in SKB.

## 2.1 Key Security in SKB

SKB provides a protected domain in which cryptographic keys are stored. The keys are mathematically transformed (encrypted) in such a way that all calculations with the keys can be performed without ever decrypting them. Keys never appear in plain at any moment. Even the application that is linked with SKB does not have access to keys in plain form.

## 2.2 Static and Dynamic Keys

Keys protected by SKB can be divided into the following categories, depending on the moment in the application's life cycle at which they enter the protected SKB domain.

| Category | Description |
|---|---|
| Static keys | Static keys are embedded into the target application's code at compile time. This arrangement is most appropriate for keys that are fixed and are not intended to change throughout the application's life cycle. The recommended way for creating SKB-compatible static keys is by using a special utility called the Key Export Tool within a safe environment, such as a closed-off facility or trusted development computer. The Key Export Tool is capable of converting keys to a format that is easy to include into source code, such as a definition of a C array or a hexadecimal string. |
| Dynamic keys | Dynamic keys are obtained by SKB at runtime. This option is convenient for keys that are not known at compile time or that are intended to change frequently. There are several ways for preparing SKB-compatible dynamic keys as described in the next section. |

## 2.3 Getting Keys Into and Out of SKB

SKB only operates on keys in memory. Therefore, getting keys into memory and moving them from memory to a persistent storage are essential operations. To avoid defeating the sole purpose of SKB, keys must never be revealed in plain form within open and unsafe environments, such as smartphones, tablets, desktop computers, or in network communication. Therefore, in such environments, keys must always be loaded, stored, and transported in a protected form.

### 2.3.1 Getting Keys Into SKB

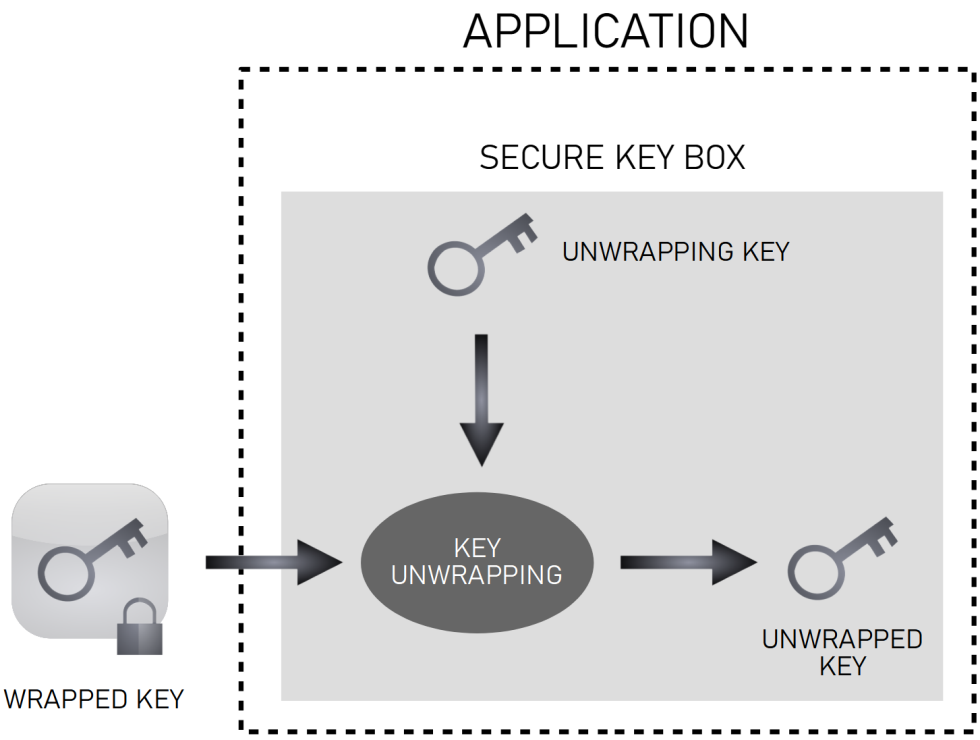
The following subsections describe the recommended safe options for getting keys into SKB.

## Importing Keys

Importing is a SKB-specific operation that loads a key that was previously exported by the SKB library or the Key Export Tool. SKB can export any of its secure data objects, and this is always done using a proprietary protected form. Depending on the export type, an exported key can be imported only by SKB from the very same package that exported it, or by SKB from any package that is built to be compatible with the exporting SKB. For more details on the available export types, see Understanding Key Exporting and Importing.

## Unwrapping Keys

A wrapped key is a key encrypted with another key. Wrapping is an established way of distributing keys in a secure manner, typically over network. If you have a pre-loaded unwrapping key in SKB, unwrapping is the recommended way of getting other keys into SKB. During unwrapping, SKB decrypts the wrapped key within its protected domain without exposing the plain form of the keys involved.



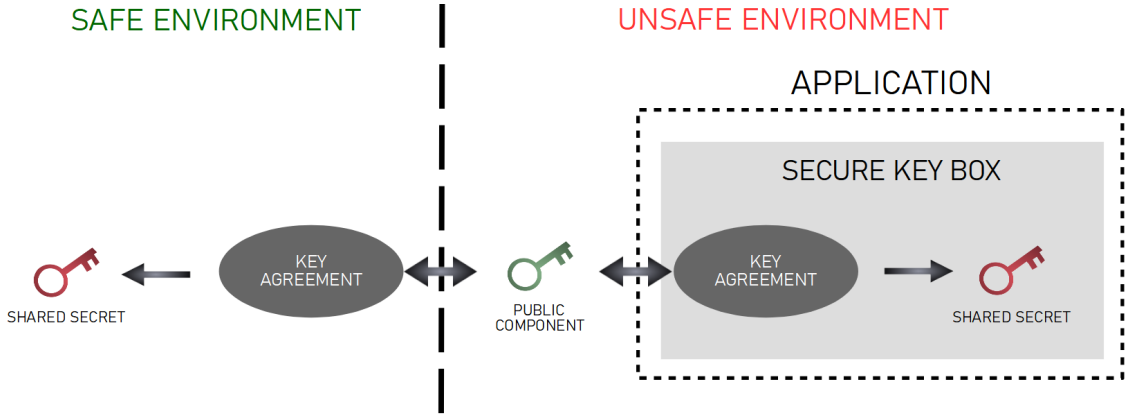

For information on executing the unwrapping operation, see Unwrapping a Key.

## Generating Keys

If your security model allows you to generate your own cryptographic keys within the SKB-protected application, you can generate new random keys using SKB. SKB provides all the necessary functions for doing this as described in Generating a Key.

### Using the Key Agreement Scheme

Key agreement is a method that allows two parties that have no prior knowledge of each other to jointly establish a shared secret (key) over an insecure channel. Diffie-Hellman key agreement is a commonly used algorithm for this purpose, and it is supported by SKB as well. Essentially, this approach allows you to obtain a new key within SKB without exposing they key itself or any intermediate sensitive parameters involved. If needed, key derivation algorithms can be applied to the shared secret to implement various standard key agreement schemes.




For information on implementing key agreement in SKB, see Executing the Key Agreement Algorithm.

## 2.3.2 Getting Keys Out of SKB

The following subsections describe the recommended safe options for moving keys from memory to a external data buffer, which can then be stored.

### Exporting Keys

Exporting is a SKB-specific operation that stores a key from memory into a data buffer using a protected format. Depending on the export type, an exported key can be imported only by SKB from the very same package that exported it, or by SKB from any package that is built to be compatible with the exporting SKB. For more details on the available export types, see Understanding Key Exporting and Importing.

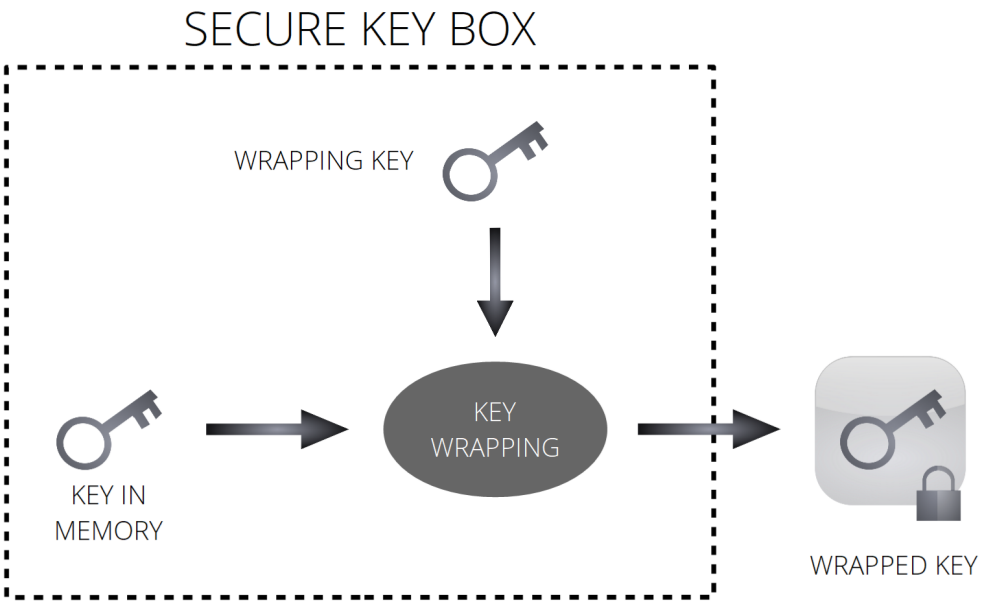
**Important**

Exported keys are stored in a secure but proprietary format. Therefore, it is recommended to use standard key wrapping algorithms for transferring keys between systems. For information on key wrapping, see Wrapping Keys.

To export a key, call the `SKB_SecureData_Export` method, specify the secure data object to be exported and the export type to be used, and provide a memory buffer where the exported data must be written.

### Wrapping Keys

A wrapped key is a key encrypted with another key. SKB provides a set of functions for wrapping secure data objects within its protected domain without exposing the plain form of any of the keys involved. Wrapped keys can then be unwrapped by any system that has the corresponding unwrapping key.




One of the advantages of wrapping is that in this way you can safely supply keys to external systems (not just SKB) over insecure channels.

For information on executing the wrapping operation, see Wrapping a Key.

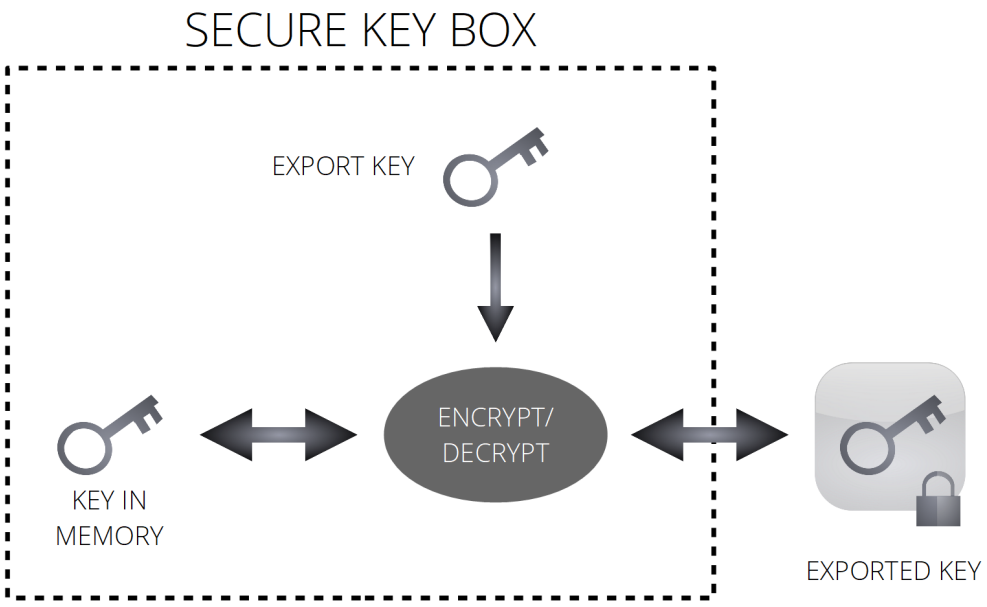
## 2.4 Understanding Key Exporting and Importing

Exporting and importing is an SKB feature that allows embedding static keys into the SKB-protected application, and ensuring persistence of dynamic keys. For example, you can prepare a data buffer containing a static key in the protected form (exported key) and embed it into the source code of your application; then, when the key is needed, SKB can import it from the embedded data buffer. Or, you can obtain a data buffer containing a protected dynamic key (exported key) at runtime from an external source, and then import the key from the data buffer.

The following subsections describe the two types of key exporting provided by SKB.

## 2.4.1 Persistent Exporting

When SKB is requested to use the persistent export type, it encrypts the contents of the key to be exported with a special key called the export key. The export key is a secret key embedded into SKB, and its sole purpose is to encrypt keys to be exported, and decrypt keys to be imported. This means that in order for importing to be successful, the importing SKB must have the same export key as SKB that exported the key.




All evaluation editions of SKB have the same export key, whereas each production edition of SKB has a unique export key (unless you have specifically requested to use an export key from an earlier package).

The advantage of using the persistent export type is that you can ensure compatibility between different SKB packages and even SKB versions. As long as two SKB packages have the same export key, they will be able to exchange keys via persistent exporting. Such SKB packages are called compatible. When you submit a request for the production edition of SKB through the Developer Portal, you can provide compatibility information to ensure that the new SKB package that you will receive is capable of importing keys exported by the previous SKB package you already have. Also, persistent export type allows you to use the one-way key upgrading scheme.

The disadvantage of persistent exporting is that key encryption and decryption operations are involved, which render this export type slower than the cross-engine export type described in the next section. However, SKB has a feature called key caching, which improves performance of operations associated with persistent exporting and importing.

You may request an SKB package that does not have the export key set. In that case, the package will contain a special utility named the Key Embedding Tool, and you will have to use this utility to prepare the export key. The prepared export key must then be loaded into SKB at runtime.

### 2.4.2 Cross-Engine Exporting

When the cross-engine export type is used, the key to be exported is transformed to a format that is unique to each SKB.

The advantage of cross-engine exporting is that this approach does not involve key encryption or decryption, which makes it faster than persistent exporting described in the previous section.

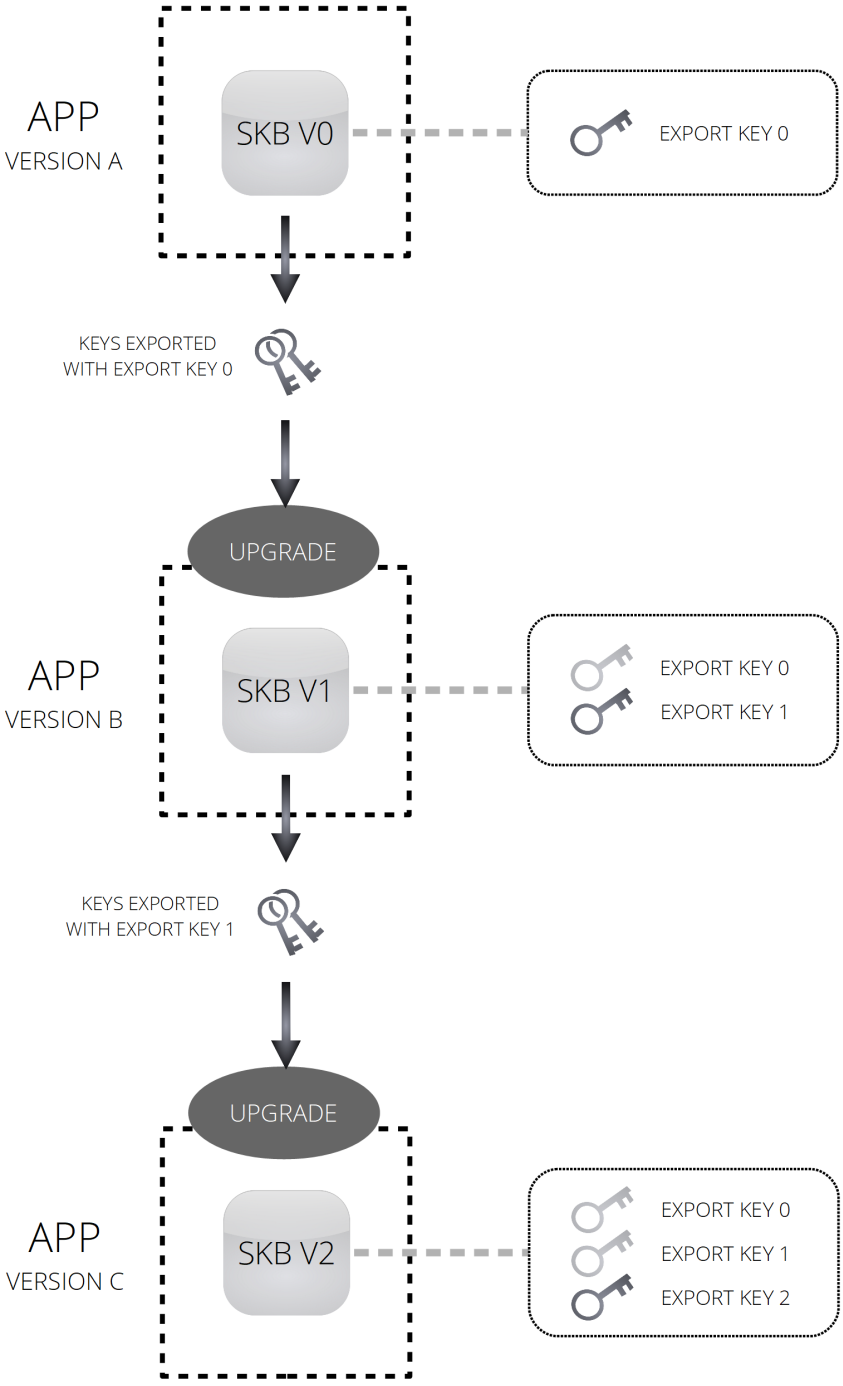
However, the disadvantage is that keys exported in this way can only be imported by SKB from the very same SKB package that was used to export them. This means that you cannot use cross-engine exporting to ensure key compatibility or one-way upgrading between different SKB packages or versions.

## 2.5 One-Way Key Upgrading

SKB provides an optional security feature that can be used together with the persistent export type to enable forward-only key compatibility called one-way key upgrading. This means that you have an option to request multiple SKB packages from the Developer Portal, where each package is assigned an internal version number, starting with version 0. These packages will be configured so that a package with a greater version number will be able to upgrade and import keys exported by all SKB packages with a smaller version number, but not the other way round.

The practical application of this is that older versions of your protected application will not be able to import keys exported by newer versions of that application. For example, if someone successfully cracks your application, the attack will not be directly applicable to newer releases of the application.

One-way key upgrading is implemented by embedding into each delivered SKB package all export keys of its previous versions as shown in the following diagram. For this reason, one-way key upgrading can be applied only to keys exported using the persistent export type.




To implement the one-way key upgrading scheme, proceed as follows:

1. Order multiple versioned SKB packages through the Developer Portal.
2. Link SKB from the package that has version 0 with your application.
3. When creating an updated version of your application, execute the following steps:
   - In the application code, replace SKB with the subsequent version.

- In the process of upgrading your application, execute the SKB_Engine_UpgradeExportedData method on each key exported by the previous SKB version.

  This function will upgrade the exported keys so that they are no longer readable by the previous SKB version. With this approach, you can even upgrade keys exported by older SKB releases.

  **Note**

  Alternatively, you can upgrade exported keys with the Key Export Tool.

**Important**

To avoid security risks, you must always delete all previous versions of keys you have upgraded. Key upgrading should be performed only once if possible.

## 2.6 Binding Keys to a Specific Device

Normally, exported keys can be imported by SKB from any compatible package, regardless of the device SKB is run on. However, you may want to introduce an additional security layer by binding exported keys to specific devices, so that they cannot be imported on any other device.

SKB provides device binding via the SKB_Engine_SetDeviceId method for the Native API, and the setDeviceID method for the JavaScript API. By calling this method, you set the device ID, which is a byte array of arbitrary length, typically derived from the hardware details or other environment-specific parameters. This ID is then combined with contents of every exported key. Thus, prior to importing, SKB that needs to import these keys must have the same device ID set as SKB that exported them.

Please note that this feature has a fallback mechanism. If key importing fails, SKB will try to import the same key again, but this time it will try to do so as if the device ID was not set in SKB. In other words, the logic of importing a key is as follows:

1. Try importing the key using the device ID currently set in SKB.
2. If it fails, try importing the key with no device ID set in SKB.

If neither of these steps succeeds, it means the key was exported with a different device ID and cannot be imported.

You may call the SKB_Engine_SetDeviceId or setDeviceId method multiple times, but SKB will always use the device ID that was set in the last method call.

When the device ID is no longer needed, you can reset SKB to use the default device-independent export format by calling the same method with an input buffer of size 0.

# 2.7 Loading Plain Keys

The purpose of SKB is to always work with keys in protected form. However, for very rare cases, SKB does support direct loading of plain keys, which is a special case of the unwrapping operation.

**Important**

Loading a plain key is a very insecure operation and must be avoided at all costs if possible. There are better alternatives of obtaining keys. Therefore, under normal circumstances loading of plain keys is disabled in SKB.

If, in your case, SKB is configured to allow loading of plain keys, to convert a plain key into a secure data object, call the `SKB_Engine_CreateDataFromWrapped` method and choose the unwrapping algorithm `SKB_CIPHER_ALGORITHM_NULL`. In this case, the unwrapping key and other parameters do not have to be provided.

SKB supports loading the following types of plain keys:

- raw bytes
- private and public RSA keys
- private ECC keys

# 3. Building Applications Protected by SKB

This chapter describes the recommended procedure for building and deploying an application that is linked with the native SKB library. Following these steps is important to achieve the highest level of security.

For information on building applications that use the Java implementation of SKB, see Java API. Similarly, if you are building a Web application that uses SKB, see JavaScript API.

## 3.1 Building the Protected Application

SKB is delivered as a precompiled static library. The public interface to this library is described in the `SkbSecureKeyBox.h` file, which is located in the `Include` directory.

To build an application protected by SKB, you must perform the following main steps:

1. Optionally, to reduce the binary size of the application, disable the modules you do not use.
2. Link your application with the following libraries from the `lib` directory:
   - SKB library (`libSecureKeyBox.a` or `SecureKeyBox.lib`, depending on the target platform)
   - Platform Library (`libSkbPlatform.a` or `SkbPlatform.lib`, depending on the target platform)

   **Important**

   Make sure you are not linking or distributing any of the unsafe SKB components.

   Also, a special note applies to the Android target. For Android applications, you have to specify the Platform Library using the `LOCAL_WHOLE_STATIC_LIBRARIES` option in the `android.mk` file, not using `LOCAL_STATIC_LIBRARIES`.
3. If you selected the **Link-time module configuration** option when ordering the SKB package from Developer Portal, link the `SkbLinkTimeConfiguration.cpp` file (located in the `src/SkbModules` directory).

   For more information on this file, see Reducing the Number of Modules at Link Time.
4. Build your application and, to remove unnecessary code and sensitive information, make sure you use the following compiler and linker settings depending on your build system:
   - Visual Studio
     - Enable references (`/OPT:REF`).
     - Enable `COMDAT` folding (`/OPT:ICF`).
   - GCC
     - If compiling for Linux, use the `-Wl,-gc-sections` option.
   - Xcode

- Enable the **Deployment Postprocessing** option.
- Enable the **Strip Linked Product** option.
- Set the **Strip Style** option to **All Symbols**.

◦ Xcode with a custom build system

- Use the `-Wl,-dead_strip` option.

5. To ensure that symbol information is correctly stripped from the executable, open the executable in a binary editor and search for a string "WHITEBOX".

   This string must not be present in the code. If it is, ensure you have configured settings as described in the preceding step.

6. If the SKB package delivered to you has tamper resistance applied, run the Binary Update Tool on the final built application as described in Applying Tamper Resistance with zShield.

## 3.2 Understanding Modules

The codebase of SKB is organized into logical parts named modules. Each module contains code for a specific algorithm or feature provided by SKB.

When you request an SKB package from the Developer Portal, you must specify the modules to be included in the package. Normally, you would request only the modules you need. However, it is not always easy to determine the exact set of required modules. Or your build infrastructure may involve building several editions of the final application where each edition requires a different set of modules. The following subsections describe the steps you may take to address these issues.

### 3.2.1 Determining the Set of Used Modules

If you are using an evaluation edition of SKB, there is an automated way of determining the set of SKB modules your application is actually using. When an evaluation edition of SKB is deinitialized (the last `SKB_Engine_Release` method is called), SKB produces a text file where it lists all the modules included in SKB, and marks each one based on whether it was used or not during execution (this does not happen with production editions of SKB). The obtained information may be very useful for requesting a new SKB package that contains only the modules you use.

The generated text file is named `skb_config_«unique ID».cfg`, and you can specify the directory where this file must be placed by using the `SKB_SetTempDirc` function.

**Important**

This function must be called before the first invocation of the `SKB_Engine_GetInstance` method; otherwise, SKB will ignore the specified directory.

**Note**

The directory set with this function will also be used to save the persistent key cache storage as described in Runtime Configuration.

The contents of the generated text file resemble the following:

```
AES128_ENCRYPT 1
AES128_DECRYPT 0
AES128_HIGH_SPEED_ENCRYPT 1
AES192_HIGH_SPEED_ENCRYPT 1
AES256_HIGH_SPEED_ENCRYPT 1
AES128_HIGH_SPEED_DECRYPT 0
AES192_HIGH_SPEED_DECRYPT 0
AES256_HIGH_SPEED_ENCRYPT 0
...
```

where "0" means the corresponding feature was not used, and "1" means it was used.

If the text file already exists when SKB is run, SKB will not rewrite the text file, but will update its contents instead. An exception is when the SKB feature set was changed since the previous run. For example, a newer version of SKB was installed, SKB from a different package was used, or SKB modules were enabled or disabled. In these cases, the existing text file will be overwritten.

Updating of the text file takes place as follows:

- If a feature that was previously marked with "0" was used this time, it will be changed to "1".
- If a feature that was previously marked with "1" was not used this time, it will remain marked as "1".

Therefore, if you want to obtain module usage statistics for one particular application, it is recommended you delete the text file first. Otherwise, the data may be misleading.

To understand what each of the features in the text file mean, you may consult with Zimperium. Or you can send the text file to Zimperium to help you request a new SKB package containing only the necessary features.

## 3.2.2 Reducing the Number of Modules at Link Time

SKB provides a mechanism that allows you to drop unnecessary algorithms and features from the final binary (and reduce its size) by letting the linker remove unused modules. Here are the main steps you have to do:

- Select the **Link-time module configuration** option when ordering the SKB package. If you have done that, there will be an additional directory `SkbModules` present in the `src` directory of the SKB package. This directory contains the necessary files that allow you to reduce the number of modules linked into your application.

- Inside the `src/SkbModules` directory, there is a file named `SkbModules.h`. This file contains a list of enabled SKB modules. You can reduce the number of modules included in the final executable by commenting out or deleting individual lines in the `SkbModules.h` file. The file contains comments that will help you identify the algorithms and features.
- Link the `SkbLinkTimeConfiguration.cpp` file (located in the `src/SkbModules` directory) into the final executable. This file must always be included if you selected the **Link-time module configuration** option. If you did not select the option, this file will not be present and you do not have to worry about linking it.

**Important**

This feature is deprecated and will be removed in future releases of SKB.

# 3.3 Improving Security

This section provides some general recommendations that can keep the final protected application safe.

## 3.3.1 Avoiding Distribution of Unsafe Components

SKB consists of a number of binary libraries and supporting files. Some of these components are secure and can be safely included in the final protected application. However, some components expose sensitive operations that can lead to key exposure and therefore must be considered insecure. These components serve a specific purpose and are usually not required in the final deliverable that is delivered to end users. The following components must be considered unsafe and must never be included in an application that is deployed in an open environment:

- Key Export Tool
- Custom ECC Tool
- Diffie-Hellman Tool
- Key Embedding Tool
- Sensitive Operations Library

These components can only be used on a protected computer that is not accessible to end users.

## 3.3.2 Restricting Key Usage

Some cryptographic algorithms are designed so that the same key could potentially be used for different operations. For example, an AES key can be used both for encryption and unwrapping, or an RSA key can be used for decryption and signing. This may open up security risks and vulnerabilities in the protected

application. Therefore, for some algorithms, SKB provides the ability to specify the intended purpose of a key, which is determined by the key data type. This feature restricts the usage of a particular key to its only intended operation and nothing else. Here are the options that are available to achieve that:

- If a symmetric key is only going to be used for unwrapping, choose the `SKB_DATA_TYPE_UNWRAP_BYTES` data type instead of the general purpose `SKB_DATA_TYPE_BYTES`.
- If a private RSA key is only going to be used for signing and the key is known during compile time, choose the `SKB_DATA_TYPE_RSA_STATIC_SIGN_CONTEXT` data type instead of the general purpose `SKB_DATA_TYPE_RSA_PRIVATE_KEY`.
- If a private RSA key is only going to be used for unwrapping and the key is known during compile time, choose the `SKB_DATA_TYPE_RSA_STATIC_UNWRAP_CONTEXT` data type instead of the general purpose `SKB_DATA_TYPE_RSA_PRIVATE_KEY`.

**Note**

Keys of the types `SKB_DATA_TYPE_RSA_STATIC_SIGN_CONTEXT` and `SKB_DATA_TYPE_RSA_STATIC_UNWRAP_CONTEXT` can only be created by importing data prepared using the Key Export Tool.

### 3.3.3 Removing Unnecessary Algorithms

You must make sure that your application only contains the cryptographic algorithms and SKB features it actually needs. Otherwise, if an unneeded algorithm or feature is included, an attacker may misuse it to expose secrets and use the application in an unintended way. For example, if an AES key is only used for decryption, you should not include the AES encryption algorithm in your SKB package. Besides, removing unneeded algorithms and features will reduce the binary size of your application.

To remove unnecessary algorithms, you should determine the used SKB modules and then request a new SKB package that has only these modules selected.

## 3.4 Improving Performance

This section provides recommendations for optimizing several SKB operations in order to achieve better execution speed for time-critical applications.

### 3.4.1 Optimizing the Use of Cipher and Transform Objects

A cipher is an object that performs encryption or decryption of data. A transform is an object that performs signing and verification. These objects must be created prior to executing the operations they provide. Because creation of cipher and transform objects is a time-consuming operation, consider the following recommendations:

- Create the cipher and transform objects at an appropriate moment ahead of time (for example, immediately after obtaining the keys they will require), not within time-critical functions.
- Release the cipher and transform objects only when you are absolutely sure they will not be used again.

## 3.4.2 Using Key Caching

In SKB, operations associated with preparation of keys may take significant computing time. This can create unwanted performance impact on speed-sensitive applications. To address this problem, SKB provides an optional feature called key caching, which significantly improves certain operations associated with keys.

To improve SKB execution speed with key caching, the following are the general recommendations:

- Always keep key caching enabled, which is the default configuration.
- If you have the permanent storage of the key cache enabled (which is the default setting for most targets), avoid creating and releasing the `SKB_Engine` object frequently, because these operations trigger loading and saving the key cache to the local storage, which may create some performance loss.

  If possible, you should create the `SKB_Engine` object only once, and release it when you are sure it will no longer be used.

## 3.4.3 Using the Cross-Engine Export Type

Whenever possible, you should use the cross-engine export type for exporting keys. Cross-engine exporting is a faster operation than persistent exporting.

However, you should be aware that keys exported using the cross-engine export type can only be imported by the very same SKB package that exported it; other SKB packages and versions will not be able to import such keys.

# 4. Executing Cryptographic Operations

This chapter provides high-level information on how to use SKB to execute the standard cryptographic operations.

All the operations described in this chapter assume that you already have the SKB_Engine object initialized, and all the necessary SKB_SecureData objects (keys) prepared.

## 4.1 Encrypting and Decrypting Data

This section describes operations related to encrypting and decrypting data.

### 4.1.1 Encrypting Data

Encryption is a process where a cryptographic cipher and a cryptographic key are applied to plain data to produce encrypted data.

DES, Triple DES, Speck, and AES are the only supported encryption ciphers. The main reason for this is that encryption for asymmetric key ciphers (such as RSA and ElGamal ECC) require a public key, which is usually known and therefore does not require protection.

To encrypt data, proceed as follows:

1. Create a cipher object using the SKB_Engine_CreateCipher method, specify the direction SKB_CIPHER_DIRECTION_ENCRYPT, and provide all the necessary parameters.
2. Encrypt the input buffer by calling the SKB_Cipher_ProcessBuffer method one or several times.

   This method returns a byte buffer containing the encrypted data.
3. When no longer needed, release the cipher object by calling the SKB_Cipher_Release method.

**Important**

If you are encrypting data using AES in the GCM or CCM mode, you need to execute some additional steps.

### 4.1.2 Decrypting Data

Decryption is a process where a cryptographic cipher and a cryptographic key are applied to encrypted data to produce plain data.

To decrypt data, proceed as follows:

1. Create a cipher object by calling the SKB_Engine_CreateCipher method, specify the direction SKB_CIPHER_DIRECTION_DECRYPT, and provide all the necessary parameters.
2. Decrypt the input buffer by calling the SKB_Cipher_ProcessBuffer method one or several times.

This method returns a byte buffer containing the decrypted data.

3. When no longer needed, release the cipher object by calling the SKB_Cipher_Release method.

**Important**

If you are decrypting data using AES in the GCM or CCM mode, you need to execute some additional steps.

## 4.1.3 Using High-Speed AES

SKB provides three implementations of AES. The default implementation is the most secure, but also the slowest. If you intend to encrypt or decrypt high-volume data, such as a video stream, one of the other two implementations might be a better choice; they ensure much faster execution speed while maintaining a decent level of security.

To enable one of the high-speed AES implementations, proceed as follows depending on the type of API you are using:

- If you are using the Native API, specify one of the following flags when creating the cipher object:
  - SKB_CIPHER_FLAG_BALANCED: This implementation ensures a good balance between performance and security, but it takes up more binary size. We recommend trying this implementation first, and switch to the SKB_CIPHER_FLAG_HIGH_SPEED implementation only if the performance or binary size is not satisfactory.
  - SKB_CIPHER_FLAG_HIGH_SPEED: This implementation ensures execution speed that is very close to unprotected AES, but is also the least secure. We suggest choosing this implementation when the other two implementations do not fit your requirements in terms of speed or size.
- If you are using the Java API, select the SkbHighSpeedAesProvider class when specifying the Provider.

  This configuration corresponds to the SKB_CIPHER_FLAG_HIGH_SPEED implementation; the SKB_CIPHER_FLAG_BALANCED implementation is not supported in the Java API.
- If you are using the JavaScript API, in the Algorithm dictionary of the corresponding AES mode, set the high_speed member to true to use the SKB_CIPHER_FLAG_HIGH_SPEED implementation, or set the balanced member to true to use the SKB_CIPHER_FLAG_BALANCED implementation, as described in Technical Details.

## 4.1.4 Using Authenticated Encryption With Additional Data

SKB supports authenticated encryption with additional data (AEAD) using the AES cipher in the GCM and CCM modes. The following subsections provide details about each mode.

## GCM Mode

When the AES cipher is used in the GCM mode, the following main steps need to be executed:

1. Create a cipher object by calling the SKB_Engine_CreateCipher method, provide the SKB_GcmCipherParameters structure as the cipher_parameters argument, and provide other input settings.
2. Supply the authenticated data input buffer by calling the SKB_Cipher_ProcessAad method one or several times.

   You may also choose not to call this method at all, in which case the additional authenticated data buffer will be assumed to be an empty string.
3. Encrypt or decrypt the input buffer by calling the SKB_Cipher_ProcessBuffer method one or several times.

   You may also choose not to call this method at all, in which case the algorithm will be executed in the authentication-only mode, which is called Galois Message Authentication Code (GMAC).
4. When the entire input buffer is processed, call the SKB_Cipher_ProcessFinal method and supply the SKB_AuthenticationParameters structure as input to calculate the authentication tag (in case of encryption) or verify the provided authentication tag (in case of decryption).
5. When no longer needed, release the cipher object by calling the SKB_Cipher_Release method.

## CCM Mode

When the AES cipher is used in the CCM mode, the following main steps need to be executed:

1. Create a cipher object by calling the SKB_Engine_CreateCipher method, provide the SKB_AuthenticationParameters structure as the cipher_parameters parameter, and provide other input settings.
2. Supply the authenticated data input buffer by calling the SKB_Cipher_ProcessAad method.

   You may choose not to call this method, in which case the additional authenticated data buffer will be assumed to be an empty string.
3. Encrypt or decrypt the input buffer by calling the SKB_Cipher_ProcessBuffer method once.

   When encrypting, the output buffer will contain the authentication tag at the end. When decrypting, the authentication tag is expected to be located at the end of the input buffer. The output buffer will contain decrypted data only if authentication is successful.

   You may call SKB_Cipher_ProcessBuffer method repeatedly, without calling any other methods in between. Doing so will reuse the same key and the same additional authenticated data. Please note that a different nonce value should be supplied for each different input when reusing the same key in the CCM mode.

**Important**

The SKB_Cipher_ProcessFinal method must not be called in the CCM mode.

4. When no longer needed, release the cipher object by calling the SKB_Cipher_Release method.

## 4.2 Calculating a Signature

Calculating a signature involves executing the signing algorithm on a buffer of plain or secure data using a particular signing key. The output is a plain buffer of bytes containing the signature.

To calculate a signature, proceed as follows:

1. Create a transform object by calling the SKB_Engine_CreateTransform method, select the SKB_TRANSFORM_TYPE_SIGN type, and specify all the necessary parameters.
2. Supply a buffer of data as input to the transform object by calling the following methods:
   - SKB_Transform_AddBytes

     This method appends a buffer of plain data to the input.
   - SKB_Transform_AddSecureBytes

     This method appends contents of a secure data object to the input.

     **Important**

     For the ECDSA and RSA signing algorithms, the SKB_Transform_AddSecureData method is available only if the corresponding digest algorithm is also included in SKB. For instance, if you want to calculate a signature of a cryptographic key using the ECDSA signing algorithm with SHA-384 as the hash function, the SHA-384 digest algorithm must also be included in SKB. Otherwise, SKB will accept only plain data as the input buffer.

     The ISO/IEC 9797-1 MAC algorithm 3 for DES (also known as Retail MAC) and the Speck-CMAC algorithm do not support the SKB_Transform_AddSecureData method.

   You can call these methods more than once to pass a large buffer of input data consisting of several smaller data chunks. An exception is those signing algorithms that do not have their own hash functions (SKB_SIGNATURE_ALGORITHM_RSA, SKB_SIGNATURE_ALGORITHM_DSA, and SKB_SIGNATURE_ALGORITHM_ECDSA). These algorithms assume that the input is a short unencrypted message (such as a digest) prepared outside of SKB. Therefore, they will accept only one data buffer of plain data as an input. This means that only the SKB_Transform_AddBytes method can be used (not SKB_Transform_AddSecureData), and only once.
3. To calculate the signature, call the SKB_Transform_GetOutput method.
4. When no longer needed, release the transform object by calling the SKB_Transform_Release method.

## 4.3 Verifying a Signature

Verifying a signature involves executing the verification algorithm that requires the following main inputs:

- data buffer for which the signature was calculated
- signature to be verified
- verification key

To verify a signature, proceed as follows:

1. Create a transform object by calling the SKB_Engine_CreateTransform method, select the SKB_TRANSFORM_TYPE_VERIFY transform type, and provide a parameters structure that specifies the verification algorithm, and supplies the signature and the verification key.
2. To provide data buffer for which the signature was calculated, supply the input buffer of plain or secure data as an input to the transform object by calling the SKB_Transform_AddBytes method and the SKB_Transform_AddSecureBytes method.

   You can call these methods more than once to pass a large buffer of input data consisting of several smaller data chunks.

   **Important**

   The ISO/IEC 9797-1 MAC algorithm 3 for DES (also known as Retail MAC) and the Speck-CMAC algorithm do not support the SKB_Transform_AddSecureData method.

3. To execute the verification algorithm, call the SKB_Transform_GetOutput method.

   The output will be 1 if the signature matches the provided data buffer, and 0 if it does not.
4. When no longer needed, release the transform object by calling the SKB_Transform_Release method.

## 4.4 Calculating a Digest

Calculating a digest involves taking a buffer of plain or secure data and calculating the hash value. The output is a plain buffer of bytes.

To calculate a digest, proceed as follows:

1. Create a transform object by calling the SKB_Engine_CreateTransform method, select the SKB_TRANSFORM_TYPE_DIGEST type, and specify all the necessary parameters.
2. Supply a buffer of plain or secure data as an input to the transform object by calling the SKB_Transform_AddBytes method and the SKB_Transform_AddSecureBytes method.

   You can call these methods more than once to pass a large buffer of input data consisting of several smaller data chunks.

**Important**

The ISO/IEC 9797-1 MAC algorithm 3 for DES (also known as Retail MAC) does not support the `SKB_Transform_AddSecureData` method.

3. To calculate the digest, call the `SKB_Transform_GetOutput` method.
4. When no longer needed, release the transform object by calling the `SKB_Transform_Release` method.

## 4.5 Unwrapping a Key

For general information about unwrapping, see Unwrapping Keys.

To unwrap a key, call the `SKB_Engine_CreateDataFromWrapped` method and provide the necessary parameters, such as the following:

- wrapped key
- type and format of the wrapped key
- algorithm for unwrapping the data

  For the special case of using the ElGamal ECC unwrapping algorithm, see the next section.
- unwrapping key

**Important**

For AES and Triple DES unwrapping algorithms, it is recommended to always use the unwrapping key of type `SKB_DATA_TYPE_UNWRAP_BYTES`, because this type of keys cannot be misused in other cryptographic operations, such as decryption.

- additional parameters for the unwrapping algorithm

### 4.5.1 Unwrapping a Key Wrapped with the ElGamal ECC Algorithm

Since there are no widely accepted standards for storing the output of ElGamal ECC decryption, this section describes the format used by SKB. In connection with this, you may have to perform additional steps to extract the actual unwrapped key from the output as described below.

In the case of the ElGamal ECC unwrapping algorithm, the `wrapped` buffer of the `SKB_Engine_CreateDataFromWrapped` method must contain two points on an ECC curve as described in Input Buffer for the ElGamal ECC Cipher.

After the unwrapping method is successfully executed, the `data` variable will point to a buffer that contains the X coordinate of the decrypted point on the ECC curve. The actual unwrapped key is stored within the X coordinate using the big-endian format. You must then extract the unwrapped key bytes from the X

coordinate using the SKB_SecureData_Derive method and the SKB_DERIVATION_ALGORITHM_SLICE algorithm according to your ElGamal ECC encryption padding scheme used. With this approach, you can use any padding scheme for encryption.

For example, assume you use ElGamal ECC with the P-256 curve to wrap a secret 16-byte AES key by adding 4 bytes to its beginning to map it to a point on an ECC curve. Then the unwrapping code would resemble the following:

```
SKB_SecureData* secret_key; // This will contain the unwrapped AES key
SKB_SecureData* temp_data;
SKB_SecureData* ecc_key = ...; // Previously obtained private ECC key
SKB_Byte wrapped_buffer[256/8 * 4] = { ... };
SKB_Size wrapped_buffer_size = sizeof(wrapped_buffer);

// ECC parameters
SKB_EccParameters params = {};
params.curve = SKB_ECC_CURVE_NIST_256;
params.domain_parameters = NULL;
params.random_value = NULL;

SKB_Engine_CreateDataFromWrapped(engine,
                                 wrapped_buffer,
                                 wrapped_buffer_size,
                                 SKB_DATA_TYPE_BYTES,
                                 SKB_DATA_FORMAT_RAW,
                                 SKB_CIPHER_ALGORITHM_ECC_ELGAMAL,
                                 &params,
                                 ecc_key,
                                 &temp_data);
// Now temp_data contains 256/8 = 32 bytes. The secret AES key is stored in bytes
// with indices 12 to 27. Remember that data is in big-endian, so when you add 4
// bytes before the 16-byte AES key in the encryption process, all 20 bytes go to
// bytes with indices 12 to 31 (the 4 added bytes are stored in bytes with
// indices 28 to 31).

// Extract the AES key from bytes 12 to 27
SKB_SliceDerivationParameters params = { 12, 16 }; // from, size
SKB_SecureData_Derive(temp_data,
                      SKB_DERIVATION_ALGORITHM_SLICE,
                      &params,
                      &secret_key);
```

```
// Release temporary data
SKB_SecureData_Release(temp_data);

// Now use secret_key that contains the 16-byte AES key
// ...

// Release the secret key when it is no longer needed
SKB_SecureData_Release(secret_key);
```

## 4.6 Wrapping a Key

For general information about wrapping, see Wrapping Keys.

To wrap a key contained within a secure data object, call the SKB_SecureData_Wrap method and specify the necessary parameters.

## 4.7 Wrapping Plain Data

In some cases, you may want to take a plain input buffer, encrypt it with a key, and store the output as a new secure data object. For example, this operation is suitable for deriving new keys from some input seed and a specific key.

To wrap plain data, call the SKB_Engine_WrapDataFromPlain method. The input is plain data, but the encryption key and the output of the method are secure data objects.

## 4.8 Generating a Key

SKB provides a way for generating new symmetric and private keys. The generated keys will contain random content based on the native system's random generator as follows:

- On Windows, CryptoAPI is used.
- On other systems, the `/dev/urandom` device is used, if available; otherwise, the `/dev/random` device is used.

If necessary, you can create a custom implementation for the random generator function as described in Platform Library.

To generate a new random key, call the SKB_Engine_GenerateSecureData method, specify what type of key you want to generate, and provide the necessary parameters.

With this method, you can generate:

- raw bytes

**Important**

Please note that if the generated raw bytes are used as a DES key, the 8 parity bits of the key, which are sometimes used for error detection, will contain random values and will not be considered valid by external systems that perform validation of those 8 bits.

- private RSA keys

**Important**

In SKB, RSA key generation is a convenience feature that must not be used in security-critical environments. Before the generated RSA keys are obtained in the secure format, they are briefly exposed in the memory as plain keys.

- private ECC keys

## 4.9 Deriving a Public Key from a Private Key

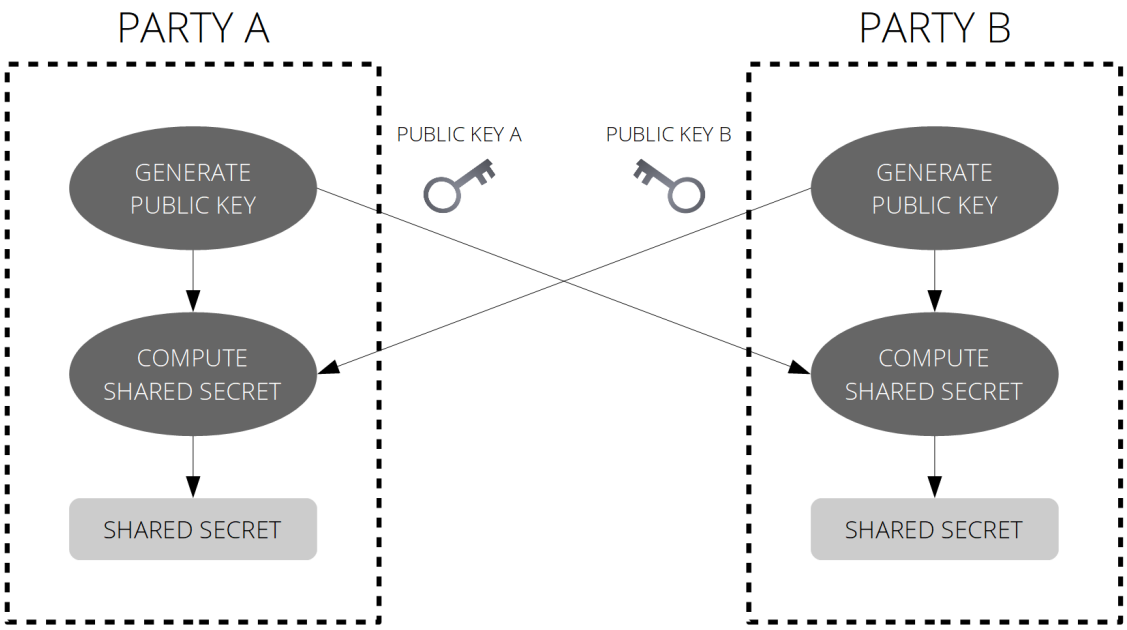
As described in the previous section, SKB can generate private keys of various types. In connection with this, it may be necessary to get the corresponding public keys as well. A public key can be derived from a private key.

To derive a public key from a private key, call the SKB_SecureData_GetPublicKey method, provide the secure data object containing the private key, and supply the necessary parameters. This method will return a buffer of bytes containing the corresponding public key.

## 4.10 Executing the Key Agreement Algorithm

The key agreement algorithm involves two parties that want to obtain a shared secret (usually a cryptographic key) that must be known only to them.

First, both parties each generate a public value or key that is given to the other party. Then each party takes the other party’s public key and generates a shared secret. The shared secret is identical to both parties. This algorithm is shown in the following diagram.




To perform the key agreement algorithm using SKB, proceed as follows:

1. Create a key agreement object by calling the SKB_Engine_CreateKeyAgreement method and specify the necessary parameters.
2. Generate a public key by calling the SKB_KeyAgreement_GetPublicKey method.
3. Exchange the public keys with the other party.
4. With the other party's public key on hand, compute the shared secret by calling the SKB_KeyAgreement_ComputeSecret method.
5. When no longer needed, release the key agreement object by calling the SKB_KeyAgreement_Release method.

## 4.11 Deriving a Key

This section describes several operations that can be used to derive one cryptographic key from another.

### 4.11.1 Deriving a Key as a Substring of Bytes of Another Key

In some cases, it is necessary to securely derive a new key as a substring of bytes of another key. To do this, call the SKB_SecureData_Derive method, select either the SKB_DERIVATION_ALGORITHM_SLICE or SKB_DERIVATION_ALGORITHM_BLOCK_SLICE algorithm, and specify the range of bytes to be derived as a new key.

The only difference between the SKB_DERIVATION_ALGORITHM_SLICE and SKB_DERIVATION_ALGORITHM_BLOCK_SLICE algorithms is that the latter requires the index of the first byte and the number of bytes in the substring to be multiples of 8 or 16 depending on which block slicing algorithm variation is enabled in the SKB package.

The SKB_DERIVATION_ALGORITHM_SLICE and SKB_DERIVATION_ALGORITHM_BLOCK_SLICE algorithms can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

**Note**

You can use the SKB_DERIVATION_ALGORITHM_SLICE algorithm to extract the unwrapped key from the output of the ElGamal ECC unwrapping algorithm, as described in Unwrapping a Key Wrapped with the ElGamal ECC Algorithm.

## 4.11.2 Concatenating Two Keys

SKB provides a special derivation algorithm that produces a new key by concatenating two existing keys. The output key consists of the first key followed by the second key.

To use this algorithm, call the SKB_SecureData_Derive method, provide the first input key, select the SKB_DERIVATION_ALGORITHM_BLOCK_CONCATENATE algorithm, and specify the necessary parameters, among which the second input key must also be supplied.

Both input keys must be secure data objects that contain raw bytes, and their lengths must be multiples of 16 bytes.

## 4.11.3 Deriving a Key as Odd or Even Bytes of Another Key

SKB allows you to derive new keys from an existing key by selecting a number of its odd or even bytes. For example, if you have a 256-byte key, you can derive two 128-byte keys from it (the size of the derived keys can be smaller). One key would have the bytes of the input key with indices 0, 2, 4, 6, and so on (odd bytes); the other key would have the bytes of the input key with indices 1, 3, 5, 7, and so on (even bytes).

To use this algorithm, call the SKB_SecureData_Derive method, select the SKB_DERIVATION_ALGORITHM_SELECT_BYTES algorithm, and specify the necessary parameters.

This operation can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

## 4.11.4 Deriving a Key by Encrypting or Decrypting an Existing Key

One way of obtaining a new key is by taking an existing key and encrypting or decrypting it with another key. Since keys cannot appear in plain form, the input key, the encrypting or decrypting key, and the output key have to be secure data objects. SKB supplies a special derivation algorithm for this purpose.

To use this algorithm, call the SKB_SecureData_Derive method, select the SKB_DERIVATION_ALGORITHM_CIPHER algorithm, and specify the necessary parameters.

This operation can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

## 4.11.5 Deriving a Key as a Protected Hash Value of Another Key

SKB provides the following special key derivation algorithms that allow obtaining a new key from a hash value calculated from another key:

- iterated SHA-1 derivation
- SHA-256 or SHA-512 derivation with plain prefix and suffix
- SHA-384 derivation

These algorithms are described in the following subsections.

The main difference when compared to the standard SHA operations (provided by the `SKB_Transform` class) is that the output of these special algorithms is a secure data object, whereas the `SKB_Transform` class provides the hash value in plain form. This feature makes these algorithms suitable for deriving new keys.

### Iterated SHA-1 Derivation

The iterated SHA-1 derivation algorithm creates a new key as a substring of bytes from a SHA-1 hash value obtained from another key.

This algorithm functions as follows:

1. The SHA-1 hash value is calculated from the contents of the provided secure data object (key).

   The result is 20 bytes containing the hash value.

2. Optionally, if requested by the caller (number of rounds is greater than 1), the specified number of bytes is taken from the beginning of the 20-byte hash value and passed to the SHA-1 algorithm again one or several times.

   Each time, the result again is 20 bytes containing the hash value.

3. Finally, the specified number of bytes is taken from the beginning of the 20-byte hash value and returned as a new secure data object.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_SHA_1` algorithm, and specify the necessary parameters.

This operation can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

### SHA Derivation with Plain Prefix and Suffix

This derivation algorithm creates a SHA-256 or SHA-512 hash value of a buffer that contains three parts in the following sequence:

1. plain data of arbitrary size
2. secure data object (key)
3. plain data of arbitrary size

The output is stored as a new secure data object, which can serve as a new key.

To use this algorithm, call the SKB_SecureData_Derive method, select the algorithm (SKB_DERIVATION_ALGORITHM_SHA_256 or SKB_DERIVATION_ALGORITHM_SHA_512), and specify the plain prefix and suffix buffers.

This operation can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

### SHA-384 Derivation

The SHA-384 derivation algorithm applies SHA-384 to the input secure data object (key) and stores the output as a new secure data object (key). Unlike the SHA-1 derivation algorithm, this operation is executed only once, and the entire 48-byte hash value is returned as an output.

To use this algorithm, call the SKB_SecureData_Derive method and select the SKB_DERIVATION_ALGORITHM_SHA_384 algorithm.

This operation can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

### HMAC Derivation

The HMAC key derivation algorithm applies the HMAC algorithm to the input, which may be either a secure data object (key containing raw bytes) or plain data. Another secure data object is used as the HMAC key. The output is returned as a new secure data object (key). Three hash functions are supported—SHA-256, SHA-384, and SHA-512.

To use this algorithm, call the SKB_SecureData_Derive method and select one of the following algorithms:

- SKB_DERIVATION_ALGORITHM_HMAC_SHA256
- SKB_DERIVATION_ALGORITHM_HMAC_SHA384
- SKB_DERIVATION_ALGORITHM_HMAC_SHA512

The secure data object provided to the SKB_SecureData_Derive method will be used as the HMAC key. The secure data object or plain data provided in the parameters structure will be used as the input to the HMAC-SHA algorithm. The output will be a secure data object, which is 32, 48, or 64 bytes long depending on the chosen SHA version.

## 4.11.6 Reversing the Order of Bytes of a Key

SKB provides a simple derivation algorithm that allows you to reverse the order of bytes within a secure data object. With this method, you can not only derive new keys but also convert a protected little-endian data buffer to big-endian and vice versa.

To use this algorithm, call the SKB_SecureData_Derive method and select the SKB_DERIVATION_ALGORITHM_REVERSE_BYTES algorithm.

This operation can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

## 4.11.7 Using the NIST 800-108 Key Derivation Function

SKB supports the NIST 800-108 key derivation algorithm.

The following special notes apply to the SKB implementation:

- Either 128-bit AES-CMAC or Speck-CMAC (128-bit keys and 64-bit blocks) can be used as the pseudo-random function.
- The key derivation function works in counter mode.
- The size of the iteration counter and its binary representation (parameters i and r) is 8 bits.
- If AES-CMAC is used, the size of the integer specifying the length of the derived key (parameter L) is either 16 bits or 32 bits (depending on the algorithm you choose) and is encoded using the big-endian format.
- If Speck-CMAC is used, the size of the integer specifying the length of the derived key (parameter L) is 16 bits and is encoded using the little-endian format.

To use this algorithm, call the SKB_SecureData_Derive method and select one of the following algorithms:

- SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128 (AES-CMAC and the 32-bit L parameter)
- SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT (AES-CMAC and the 16-bit L parameter)
- SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_SPECK_L16BITLE (Speck-CMAC and the 16-bit L parameter)

This operation can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

## 4.11.8 Using KDF2 of the RSAES-KEM-KWS Scheme Defined in the OMA DRM Specification

SKB provides a derivation algorithm that is based on KDF2 used in the RSAES-KEM-KWS scheme of the OMA DRM Specification.

To use this algorithm, call the SKB_SecureData_Derive method, select the SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2 algorithm, and specify the necessary parameters.

This operation can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

## 4.11.9 Deriving a Key as Raw Bytes from a Private ECC Key

In some scenarios, you may want to derive a new key as raw bytes from a private ECC key.

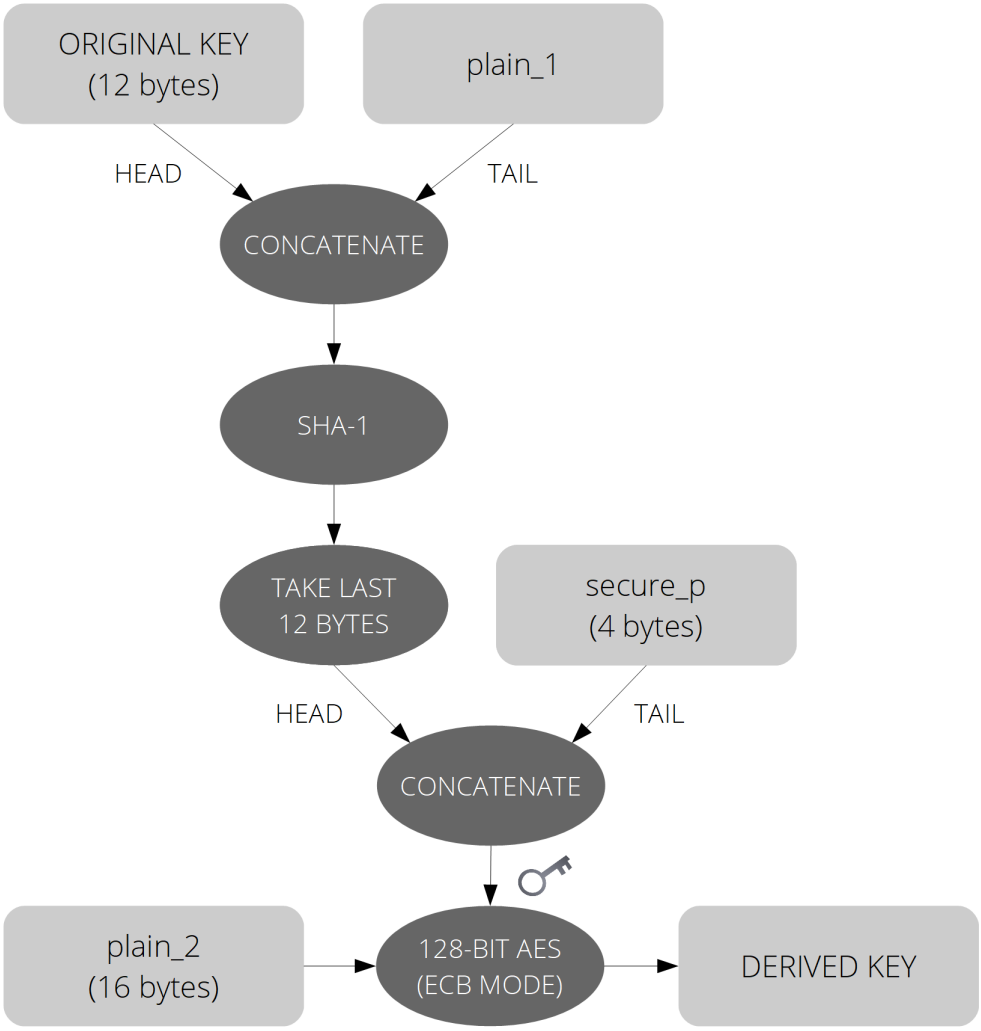
To use this algorithm, call the SKB_SecureData_Derive method, select the SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE algorithm, and specify the necessary parameters.

The derived data buffer will contain the private ECC key in the little-endian or big-endian format (depending on the selected parameters), and its size will be the same as the size of the private ECC key rounded up to whole bytes. You can then use other derivation algorithms to obtain new keys.

This operation can only be performed on a secure data object that contains a private ECC key.

## 4.11.10 Deriving a Key By Encrypting Data Using 128-bit AES With a Concatenated Key

This derivation algorithm consists of several steps executed one after another as shown in the following diagram.



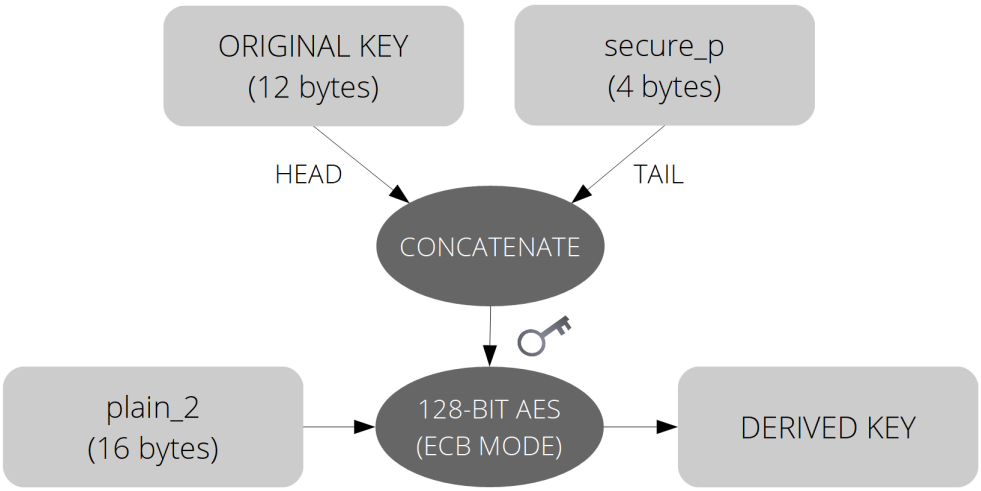

The diagram contains the following elements:

- "ORIGINAL KEY" is the secure data object used as an input of the derivation algorithm (must be 12 bytes long).
- "plain_1" and "plain_2" are plain data buffers provided as input parameters to the algorithm.

  "plain_2" must be 16 bytes long.
- "secure_p" is a secure data object provided as an input parameter to the algorithm (must be 4 bytes long).
- "DERIVED KEY" is a new secure data object obtained in the output.

This derivation algorithm supports a simplified mode of operation when "plain_1" is not provided (is NULL). Then the algorithm is executed as shown in the following diagram.




As can be seen, this algorithm is similar to the first one, except the SHA-1 step involving "plain_1" parameter is omitted.

To execute this derivation algorithm, call the SKB_SecureData_Derive method, select the SKB_DERIVATION_ALGORITHM_SHA_AES algorithm, and supply SKB_ShaAesDerivationParameters as the parameters structure.

This operation can only be performed on a secure data object that contains raw bytes and is 12 bytes long.

## 4.11.11 Deriving a Key By XOR-ing It with Plain Data or Another Key

SKB provides a derivation algorithm that obtains a new key by taking an existing key as input and executing the XOR operation on it using plain data or another key. The input key and the output key are secure data objects.

To use this algorithm, call the SKB_SecureData_Derive method, select the SKB_DERIVATION_ALGORITHM_XOR algorithm, and specify the necessary parameters.

This operation can only be performed on a secure data object that contains raw bytes, but not an RSA or ECC key.

## 4.11.12 Deriving a Key By Encrypting a Plain Buffer Two Times

This derivation algorithm takes a plain buffer of bytes, encrypts it two times using the AES cipher, and obtains the output as a new secure data object. The following are the specific details of this algorithm:

- 128-bit AES is used for encryption.
- The input secure data object is used as the AES encryption key both times.
- The input secure data object must be of type SKB_DATA_TYPE_UNWRAP_BYTES.
- The buffer of plain bytes used as input must be 16 bytes long.
- The output of the first AES encryption is passed as input to the second AES encryption.

- The output of this algorithm is a secure data object of type `SKB_DATA_TYPE_UNWRAP_BYTES`.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_DOUBLE_AES_ENCRYPT` algorithm, and specify the necessary parameters.

## 4.11.13 Deriving a Key Using HKDF with SHA-256

SKB provides an implementation of the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) specified by RFC 5869.

The following points apply to the SKB implementation of this algorithm:

- Only SHA-256 is available as the hash function.
- Salt is supported but optional.
- IKM is a `SKB_SecureData` object that serves as the input key for the derivation algorithm.
- OKM is a `SKB_SecureData` object that is the result of the derivation algorithm.
- `info` of any length (including zero) is supported.
- L must be a multiple of 16.

To use this algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_HKDF_SHA256` algorithm, and specify the necessary parameters.

# 5. Key Caching

This chapter describes how SKB uses key caching to improve performance of certain operations related to importing and processing of keys.

## 5.1 Key Caching Overview

In SKB, operations associated with preparation of keys may take significant computing time. This can create unwanted performance impact on speed-sensitive applications. To address this problem, SKB provides an optional feature called key caching. By caching processed keys, this feature reduces time required to operate with secure data objects that have already been used by SKB at some time in the past. The cached keys may be saved in the permanent device storage (which is the default setting), or kept only in the device memory for the duration of SKB execution. The key cache is used only for the 100 most recently used keys that can be cached.

The following subsections describe the specific key caching mechanisms used by SKB.

### 5.1.1 Caching of Exported and Imported Keys

When SKB imports a secure data object that was previously exported using the persistent export type, it decrypts the secure data object with the export key. Decryption of keys is a time-consuming operation. Therefore, SKB provides the ability to cache keys exported and imported using the persistent export type.

Caching of exported and imported keys works as follows:

- Upon exporting a key, SKB stores the key's pre-encrypted form in the key cache in a secure manner (if the key is not already present in the key cache).
- When SKB is requested to import a key, it tries to identify the key in the key cache first. If the key is present in the key cache, the cached version is imported, bypassing the decryption operation. If the key is not present in the key cache, the key is decrypted using the export key, and the decrypted version is stored in the key cache in a secure manner.

### 5.1.2 Caching of Private RSA Keys

Initialization of a secure data object that contains a private RSA key (for example, when creating a cipher object) may be a very time-consuming operation. To address this problem, SKB provides the ability to cache the initialized form of private RSA keys.

Caching of private RSA keys works as follows:

- When a secure data object containing a new private key is initialized for the first time, the initialized form of the key is prepared and then stored in the key cache.
- Whenever the same private key is required again, instead of going through the time-consuming key initialization phase again, the initialized form of the key is immediately retrieved from the key cache.

**Note**

Caching is not performed for private RSA keys of the `SKB_DATA_TYPE_RSA_STATIC_UNWRAP_CONTEXT` and `SKB_DATA_TYPE_RSA_STATIC_SIGN_CONTEXT` data types, because they are pre-initialized in the Key Export Tool.

# 5.2 Configuring the Key Cache

This section describes various configuration settings that you can set at runtime and compile time to configure the key cache.

## 5.2.1 Runtime Configuration

The Platform Library exposes the following functions that you can use to configure the key cache at runtime:

- `SKB_SetTempDir`

  This function allows you to specify the directory on the device where the key cache persistent storage should be placed.

- `SKB_SetFlags`

  This function enables specific features of the key cache.

- `SKB_GetFlags`

  This function returns the currently set key cache flags.

**Important**

You must make sure these functions are called before the first invocation of the `SKB_Engine_GetInstance` method; otherwise, SKB will ignore them.

## 5.2.2 Compile-Time Configuration

SKB supports the following compiler macros that you may set when compiling your application to configure the key cache:

- `SKB_USE_KEY_CACHE_IN_MEMORY`

  This macro tells SKB to keep key cache items only in the memory and never save them in a persistent storage. It is equivalent to the following set of values of the `SKB_PlatformFlags` enumeration, which is described in SKB_SetFlags:

  - `SKB_USE_COMPACT_KEYCACHE`
  - `SKB_USE_EXPORT_KEYCACHE`
  - `SKB_USE_RSA_KEYCACHE`

- `SKB_USE_KEY_CACHE_NONE`

  This macro tells SKB to disable all key caching features. This macro has the same effect as if none of the values of the `SKB_PlatformFlags` enumeration were set.

# 6. Utilities

This chapter describes the command-line utilities provided in the SKB package. The available utilities are listed in the following table.

| Utility | Description |
|---|---|
| Key Export Tool | Creates secure data objects from plain input data, and upgrades previously exported secure data objects. |
| Custom ECC Tool | Generates definitions of custom ECC curves containing parameters in protected form. |
| Diffie-Hellman Tool | Generates the protected form of parameters for the Diffie-Hellman key agreement algorithm. |
| Key Embedding Tool | Prepares an export key to be loaded into SKB at runtime. |

## 6.1 System Requirements

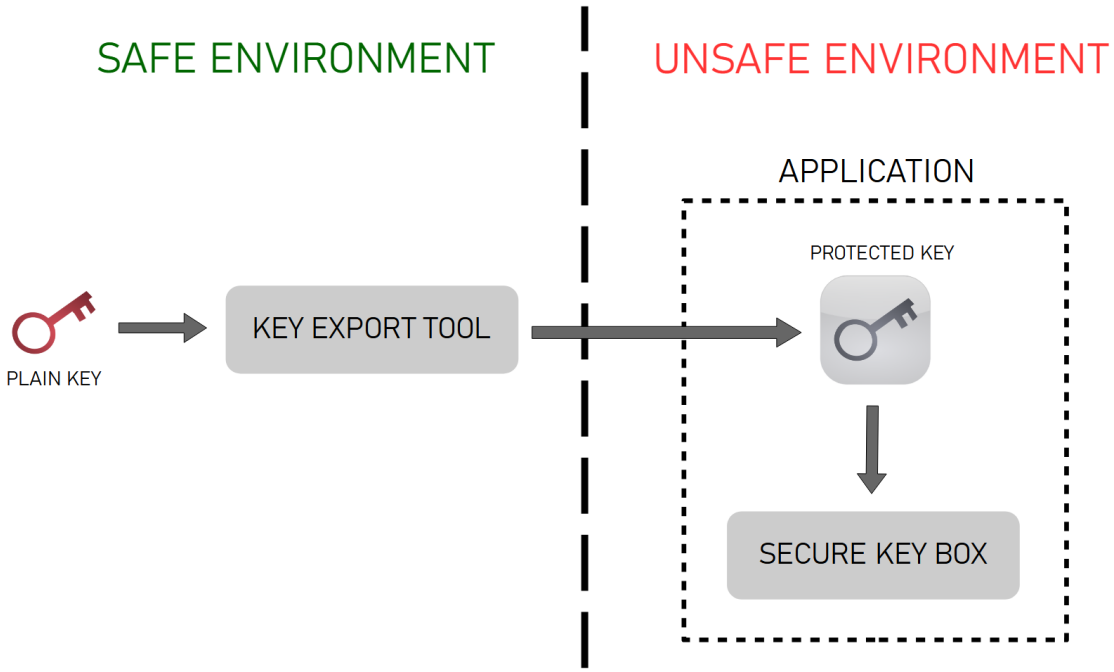
Each of the SKB utilities is provided in three editions—for Windows, macOS, and Linux. The following are the specific system requirements the utilities have for these operating systems:

- Windows 7 or later (x86-64)
- macOS 10.9 or later
- Linux (x86-64)

## 6.2 Key Export Tool

The Key Export Tool is used for the following purposes:

- creating a protected exported form of an `SKB_SecureData` object from plain input data, which is then ready for importing into SKB




The input can be raw bytes, a private RSA key, a public RSA key, or a private ECC key.

- upgrading previously exported data to the current version in the one-way data upgrade scheme

**Important**

The Key Export Tool must always be used within a safe environment, such as a closed-off facility or trusted development computer, and must never be delivered together with the target application.

## 6.2.1 Key Export Tool Overview

The Key Export Tool performs the following actions:

1. Depending on the input format, do one of the following:
   - If the input is in plain form, load it as an `SKB_SecureData` object.
   - If the input is previously exported data containing an old key version, upgrade the data.
2. Save the output in a protected format to a file or to the standard output.

   The output format can be binary data, a hexadecimal string, or source code (C, Java, or JavaScript), in which the exported data is defined as an array of bytes.

Once the output is created, you can import it into SKB using the `SKB_Engine_CreateDataFromExported` method.

## 6.2.2 Running the Key Export Tool

The Key Export Tool is located in the `bin` directory of the SKB package. To run the utility, execute it at the command line and pass several parameters to it as follows:

```
KeyExportTool
    --input-format «input format»
    --output-format «output format»
    (--input «input file» | --input-hex «hexadecimal string»)
    ([--output «output file»] [--output-stdout])
    [--array-name]
    [--input-custom]
    [--static-key-sign | --static-key-unwrap ]
    [--use-export-key «file»]
    [--cross-engine]
    [--device-id «file» | --device-id-string «string literal» |
    --device-id-hex «hexadecimal string»]
```

Please note the following special rules:

- Exactly one of the parameters `--input` and `--input-hex` must be provided.
- At least one or both of the parameters `--output` and `--output-stdout` must be provided.
- Either exactly one or none of the parameters `--static-key-sign` and `--static-key-unwrap` must be provided.
- Either exactly one or none of the parameters `--device-id`, `--device-id-string`, and `--device-id-hex` must be provided.

The following are the input parameters:

- `--input-format`

  Specifies the format of the input file. Possible values are the following:

  - "bytes": raw bytes in plain (for example, a DES or AES key)

    For this input format, SKB will accept only buffers of size 8, 16, 24, and 32 bytes, which correspond to key sizes of the supported DES and AES algorithms. If you need to import a data buffer of a different size (for example, when you use a specific key derivation algorithm), you must also provide the `--input-custom` parameter to the Key Export Tool. Otherwise, the Key Export Tool will return an error.

    If you are loading a key for the Triple DES algorithm, make sure the input corresponds to the format described in Key Format for the Triple DES Cipher.

  - "unwrap-bytes": unwrap bytes in plain

For more information about this input format, see SKB_DataType. Please note that, in this case, SKB will accept only buffers of size 16, 24, and 32 bytes, which correspond to key sizes of the supported AES unwrapping algorithms.

- "rsa": plain private RSA key in any of the following formats:
  - DER-encoded PKCS#1
  - PEM-encoded PKCS#1
  - DER-encoded PKCS#8
  - PEM-encoded PKCS#8
- "rsa-public": plain public RSA key in either of the following formats:
  - DER-encoded public RSA key stored according to the format of the `SubjectPublicKeyInfo` structure as defined in RFC 5280
  - PEM-encoded public RSA key stored according to the format of the `SubjectPublicKeyInfo` structure as defined in RFC 5280

  **Important**

  Public RSA keys can only be exported using the cross-engine export format. This means that you must pass the `--cross-engine` parameter to the Key Export Tool.

- "dsa": plain private DSA key, which is a number encoded as a big-endian buffer of bytes
- "ecc": plain private ECC key in any of the following formats:
  - DER-encoded SEC 1
  - PEM-encoded SEC 1
  - DER-encoded PKCS#8
  - PEM-encoded PKCS#8
  - SKB-specific format described in Private ECC Key
- "upgrade": previously exported data that needs to be upgraded to the current version as described in One-Way Key Upgrading

- `--output-format`

  Specifies the format of the output. Possible values are the following:

  - "binary": binary buffer of bytes (cannot be used if the `--output-stdout` parameter is specified)
  - "hex": hexadecimal string with each byte represented as two symbols
  - "java": file containing a Java byte array representing the protected data buffer, which can be used by the Java API

  - "javascript": file containing a JavaScript byte array representing the protected data buffer, which can be used by the JavaScript API
  - "source": header file containing a C array representing the protected data buffer, which you must then include into your source code

- `--input`

  Name of the file to be used as the input.

- `--input-hex`

  Specifies the input as a command-line argument, rather than a file.

  This parameter must be followed by an even number of hexadecimal characters. For example, to specify a key "ABCDEFGHIJKLMNOP", you would write this parameter as follows:

  ```
  --input-hex 41424344454647484 94A4B4C4D4E4F50
  ```

- `--output`

  File name of the output file generated by the Key Export Tool.

- `--output-stdout`

  Tells the Key Export Tool to print the output to the standard output.

  Using this parameter will cause an error if the `--output-format` parameter value is set to "binary".

- `--array-name`

  Specifies the name of the generated byte array when you choose the "source" or "java" output format.

  If this parameter is omitted, the value of the `--output` parameter will be used as the name of the array.

- `--input-custom`

  This parameter is only used together with the `bytes` input format when you want to import a buffer whose size does not correspond to the standard DES or AES key sizes (8, 16, 24, or 32 bytes). If you try to import such a buffer without this parameter, the Key Export Tool will return an error.

- `--static-key-sign`

  This parameter can only be used for a private RSA key. If set, the output will correspond to the `SKB_DATA_TYPE_RSA_STATIC_SIGN_CONTEXT` data type, which means that this key can only be used for signing. You should choose this option for private RSA keys that are known at compile time.

- `--static-key-unwrap`

  This parameter can only be used for a private RSA key. If set, the output will correspond to the `SKB_DATA_TYPE_RSA_STATIC_UNWRAP_CONTEXT` data type, which means that this key can only be used for unwrapping. You should choose this option for private RSA keys that are known at compile time.

- `--use-export-key`

This parameter is required if you have an SKB package that does not have the export key set as described in Key Embedding Tool. This parameter must be followed by a path to a binary file containing the export key prepared by the Key Embedding Tool.

- `--cross-engine`

  Tells the Key Export Tool to generate the output using the cross-engine export format.

  If the `--cross-engine` parameter is omitted, the persistent export format is used.

- `--device-id`, `--device-id-string`, `--device-id-hex`

  Optional parameters that allow you to set the device ID. The device ID is applied to every exported key and every key stored in the key cache as described in Binding Keys to a Specific Device.

  Only one of the parameters may be provided. The value to be provided depends on the parameter as follows:

  - `device-id`: path to a file that contains the device ID in binary form (for example, "device_id.bin")
  - `device-id-string`: string literal (for example, "Device ABCD")
  - `device-id-hex`: case-insensitive hexadecimal string (for example, "3Eadb54fC0A1Bab9")

  Note that these parameters are available only if the SKB package you requested has the device binding feature enabled.

By passing only the `--version` parameter to the Key Export Tool you can find out its version number.

You can see a brief description of all available parameters by running the Key Export Tool with the `--help` parameter.

## 6.3 Custom ECC Tool

For security reasons, SKB requires that ECC curve type parameters are provided in protected form. If you are not using any of the predefined standard ECC curve types, you have to use the Custom ECC Tool to generate custom curve types. More specifically, the Custom ECC Tool generates definitions of the `SKB_EccCurveParameters` structure in protected form. These generated definitions must then be included into your application's source code.

### 6.3.1 Custom ECC Tool Overview

The Custom ECC Tool generates definitions of the `SKB_EccCurveParameters` structure, which contain the following parameters in protected form:

- prime of the elliptic curve
- order of the base point
- constant "a" in the curve equation

- X coordinate of the base point
- Y coordinate of the base point

The Custom ECC Tool is executed as a command line utility. It outputs an `.h` file, which contains C code defining the custom curve in protected form. You must then include this file into your source code.

## 6.3.2 Parameter Size and Value Restrictions

The size of the "prime", "order", "X", and "Y" parameters must be between 150 and 528 bits, and none of the parameters should have an equal or greater value than the value of "prime".

Note that SKB contains two runtime instances of ECC (you may specify these instances when requesting SKB from Zimperium). One instance supports 150 to 264 bit curves, and the other supports 150 to 528 bit curves. While the second instance supports the entire bit range of ECC curves, it is slower than the first. Therefore, when generating definitions of custom curve types, you must also specify the ECC runtime instance you want to use.

## 6.3.3 Running the Custom ECC Tool

The Custom ECC Tool is located in the `bin` directory of the SKB package. You can run the utility by executing it at the command line and passing the following parameters to it:

- `-n`

  Name of the `SKB_EccCurveParameters` definition as it will be used in the code. The generated `.h` file will also have this name.

- `-p`

  Value of the prime of the elliptic curve in plain hexadecimal format.

- `-o`

  Value of the order of the base point in plain hexadecimal format.

- `-a`

  Value of the constant "a" in the curve equation in plain hexadecimal format.

  If you are passing a negative value for this parameter, it must be provided as "p-a" where "p" is the prime of the elliptic curve.

- `-x`

  Value of the X coordinate of the base point in plain hexadecimal format.

- `-y`

  Value of the Y coordinate of the base point in plain hexadecimal format.

- `-s`

ECC runtime instance to be used. The following are the available values:

- "264" corresponds to the 150 to 264 bit instance (faster)
- "528" corresponds to the 150 to 528 bit instance (slower)

You can see a brief description of all available parameters by running the Custom ECC Tool with the `--help` parameter.

The following is an example invocation of the Custom ECC Tool, which will generate a custom curve named "my_curve":

```
CustomEccTool
    -n my_curve
    -p 0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff
    -o 0xFFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551
    -a 0xffffffff00000001000000000000000000000000fffffffffffffffffffffffc
    -x 0x4A96B5688EF573284664698968C38BB913CBFC82
    -y 0x23A628553168947D59DCC912042351377AC5FB32
    -s 528
```

# 6.4 Diffie-Hellman Tool

The SKB implementation of Diffie-Hellman key agreement algorithm operates on encrypted parameters. The Diffie-Hellman Tool is used to generate the protected forms of these parameters, which must then be provided to the `SKB_PrimeDhParameters` structure when you use the Diffie-Hellman key agreement algorithm.

## 6.4.1 Diffie-Hellman Tool Overview

The Diffie-Hellman algorithm requires the following main input parameters:

- prime "p"
- generator "g"

By design, these parameters are used in plain form, but for increased security, the SKB implementation requires that they are operated on in protected form.

The Diffie-Hellman Tool is executed as a command line utility. It will then produce a single data buffer containing the input parameters in protected form. This data buffer must then be provided as the `context` parameter to the `SKB_PrimeDhParameters` structure in your source code.

## 6.4.2 Running the Diffie-Hellman Tool

The Diffie-Hellman Tool is located in the `bin` directory of the SKB package. You can run the utility by executing it at the command prompt and passing several parameters to it.

The following is the pattern to be used to run the Diffie-Hellman Tool:

```
PrimeDhTool «arguments»
```

The following are the input arguments:

- `-p`

  Prime "p" as an unsigned integer.

  The greatest common divisor of "p" and "g" must be 1.

- `-g`

  Generator "g" as an unsigned integer.

  The value of this parameter must be less than "p", and the greatest common divisor of "p" and "g" must be 1.

- `--output_format`

  Specifies the format of the output. Possible values are the following:

  ◦ "binary": binary file containing a buffer of bytes

  ◦ "source": `.h` file containing a C array representing the protected data buffer, which you must then include into your source code

- `--output`

  File name of the output file generated by the Diffie-Hellman Tool.

You can see a brief description of all available parameters by running the Diffie-Hellman Tool with the `--help` parameter.

The following is an example invocation of the Diffie-Hellman Tool, which will generate a binary file `context.bin` containing the combined protected form of the prime "p" and generator "g":

```
PrimeDhTool
    --output-format binary
    --output context.bin
    -p 0xdcb4ccd800da26b874b3350b49340fe41789f945ebd090d071c023cf777e570de3c...
    -g 0x491c9f8a3e74e82dcbf1e1511b58f24fc7013c1661da4b8268e953fde8e570c9305...
```

**Note**

Due to the physical limitation of the page width, the values of the `-p` and `-g` parameters are not displayed in the full form above.

# 6.5 Key Embedding Tool

Usually, each SKB instance has an export key embedded into its code as described in Persistent Exporting. This key is used to encrypt and decrypt other keys exported and imported into SKB. In some scenarios, you may want to control the export key used by SKB. For instance, you may want to set the key yourself; or you may need to distribute several SKB-protected applications, each with a different export key. For these cases, you have the option to request an SKB package that contains a special utility named the Key Embedding Tool. Such SKB packages will not have an export key set; you will have to prepare the export key using the Key Embedding Tool and set it yourself.

**Important**

Every SKB package has a unique binary footprint, which affects how the Key Embedding Tool encodes the key. Therefore, you cannot use an export key prepared by the Key Embedding Tool in another SKB package. Instead, you have to re-generate the export key by the Key Embedding Tool of the corresponding package.

## 6.5.1 Running the Key Embedding Tool

The Key Embedding Tool is located in the `bin` directory of the SKB package. You can run the utility by executing it at the command prompt as follows:

```
KeyEmbeddingTool «input filename» «output filename»
```

The following are the command line arguments:

- `«input filename»`

  File where the export key is stored in plain. The file can have an arbitrary name, and it can be in either of the following formats:

  - binary format if the size of the file is 16 bytes
  - text format, in which case the utility will attempt to read 32 hexadecimal characters as a representation of 16 bytes

    The hexadecimal format allows you to input multiple export keys, each on a separate line. If that is the case, the last key in the file will be used as the export key, but SKB will be able to upgrade keys exported with any of the other keys specified in the file.

- `«output filename»`

  Name of the following four files that will be produced by the Key Embedding Tool as output:

  - C++ header file, which contains the converted key as a byte array that you can then include in C++ source code.
  - Binary file (extension `.bin`), which contains the converted key in binary format.

This is the only format supported by the Key Export Tool.

- Java file (extension `.java`), which contains the converted key as a byte array that you can then include in Java source code in case you use the Java API.
- JavaScript file (extension `.js`), which contains the converted key as an array that you can pass to the `setExportKey` method of the JavaScript API.

For example, if «`output filename`» is "export_key.h", then the Key Embedding Tool will produce four files, namely, `export_key.h`, `export_key.h.bin`, `export_key.h.java`, and `export_key.h.js`.

## 6.5.2 Using the Converted Export Key

Once you have obtained the converted format of the export key you want to use, you can set that export key in SKB as follows:

1. If you are using the native SKB API, include the `SkbSetExportKey.h` file in source code, which is provided in the `Include` directory.
2. After initializing the SKB engine, set the export key as follows:
   - If you are using the native SKB API, call the `SKB_Engine_SetExportKey` method, which is declared as follows:

     ```
     SKB_Result
     SKB_Engine_SetExportKey(SKB_Engine*     self,
                             const SKB_Byte* buffer,
                             SKB_Size        buffer_size);
     ```

     `buffer` is a pointer to the data produced by the Key Embedding Tool, and `buffer_size` is its size in bytes.

     Once this function is successfully executed, the export key will be functional.

   - If you are using the SKB Java API, call the following method of the `com.whitecryption.skb.Engine` class:

     ```
     public static native void setExportKey(byte[] exportKeyData) throws SkbException;
     ```

     `exportKeyData` must contain data produced by the Key Embedding Tool. You can directly use the byte array from the generated file with the `.java` extension.

   - If you are using the SKB JavaScript API, call the following method:

     ```
     window.skb_crypto.subtle.setExportKey(data)
     ```

     where `data` is the byte array from the generated file with the `.js` extension.

     For more information on this method, see the technical details of the JavaScript API.

3. Optionally, at any point during application execution, check if a custom export key is currently set as follows:
   - If you are using the Native API, call the `SKB_Engine_IsExportKeySet` method, which is declared as follows:

     ```
     SKB_Result
     SKB_Engine_IsExportKeySet(SKB_Engine* self,
                               SKB_Byte*   is_key_set);
     ```

     `is_key_set` will be 1 if the export key is set, and 0 otherwise.
   - If you are using the Java API, call the following method of the `com.whitecryption.skb.Engine` class:

     ```
     public static native boolean isExportKeySet() throws SkbException;
     ```

     This method will return `true` if the export key is set, and `false` otherwise.

**Important**

The `SKB_Engine_SetExportKey` and `SKB_Engine_IsExportKeySet` methods are not thread-safe; they must be called only when it can be ensured that no other threads call them or perform key exporting or importing.

If your SKB instance does not have a default export key set (indicated by the presence of the Key Embedding Tool in the SKB package), and you want to use the Key Export Tool, you must pass the binary format of the converted export key to the Key Export Tool by setting the `--use-export-key` parameter.

# 7. Supporting Libraries

The core of SKB is delivered as a single static library. However, for several reasons certain functions are externalized as separate libraries that are delivered together with SKB.

This chapter describes the following supporting libraries of SKB.

| Library | Overview |
|---|---|
| Platform Library | Contains functions whose implementation and configuration details vary between different platforms and may be modified. It also provides means for you to configure some SKB settings at runtime. |
| Sensitive Operations Library | Contains functions for loading and saving plain keys. |

## 7.1 Platform Library

This section describes the purpose and details of the Platform Library delivered together with SKB.

### 7.1.1 Externalization of Platform-Specific Functions

Although the largest part of SKB is delivered as a single library, a small subset of functions used by SKB depends on the target operating system and may be implemented differently on the same architecture. Therefore, these functions are externalized as a separate module called the Platform Library.

This library is available both as source code in the `src/SkbPlatform` directory, and as a precompiled binary in the `lib` directory. In most cases, you would use the precompiled library as is. However, if you want to modify or rewrite some of the platform-specific features or adjust the SKB configuration, you have to modify the Platform Library source code and rebuild it. The Platform Library has its own interface defined in the `SkbPlatform.h` file, which is located in the `src/SkbPlatform` directory, and has inline comments providing all the information you need. Additionally, you may even modify the way SKB allocates and releases memory; to do it, you must modify the implementation of functions `new` and `delete` in the `SkbPlatformUtils.cpp` file, which is located in the `src\SkbPlatform` directory.

The Platform Library is built using CMake. For information on how to do it, see the `README.txt` file, which is located in the `src/SkbPlatform` directory.

### 7.1.2 Configuring SKB Features at Runtime

The Platform Library exposes a few functions that can be called to adjust SKB behavior at runtime (after the target application is built). In most cases, these functions are used for configuring key caching.

**Important**

You must make sure that these functions are called before the first invocation of the SKB_Engine_GetInstance method; otherwise, SKB will ignore them.

### SKB_SetTempDir

This function allows you to specify the directory on the device that will serve as the location for the following items:

- persistent storage of the key cache
- `skb_config_«unique ID».cfg` file, which is described in Determining the Set of Used Modules

The function is declared as follows:

```
SKB_Result SKB_SetTempDir(const char* dir_path);
```

`dir_path` is a pointer to the string containing a path to a directory on the device. You must make sure that this directory is writable by your application.

On Android, you may take advantage of the `getFilesDir()`, `getExternalFilesDir()`, or `getExternalCacheDir()` methods to determine the most appropriate location. Please note that in some cases this may require that you enable the `WRITE_EXTERNAL_STORAGE` permission in your Android application.

### SKB_SetFlags

This function enables specific SKB features.

The function is declared as follows:

```
void SKB_SetFlags(unsigned int flags);
```

`flags` is a set of flags that correspond to the values of the `SKB_PlatformFlags` enumeration. By modifying the bits in this value you may enable or disable certain SKB features.

If you want to set a particular flag without losing the existing flags, you may take advantage of the SKB_GetFlags function.

The `SKB_PlatformFlags` enumeration is defined as follows:

```
typedef enum {
    SKB_USE_COMPACT_KEYCACHE     = 1 << 0,
    SKB_USE_EXPORT_KEYCACHE      = 1 << 1,
    SKB_USE_RSA_KEYCACHE         = 1 << 2,
    SKB_USE_PERSISTENT_KEYCACHE  = 1 << 3,
    SKB_SAVE_KEYCACHE_FREQUENTLY = 1 << 4,
```

```
    SKB_EVAL_ENABLE_LOGGING      = 1 << 5
} SKB_PlatformFlags;
```

The following is the meaning of the flags:

- SKB_USE_COMPACT_KEYCACHE

  With this feature enabled, all keys in the key cache are identified by a hash value calculated from the input data. This reduces the key cache size, but in extremely rare cases it may lead to a situation when a wrong key is returned from the key cache.

  By default, this feature is enabled.

- SKB_USE_EXPORT_KEYCACHE

  Enables caching of keys exported and imported using the persistent export type.

  By default, this feature is enabled.

- SKB_USE_RSA_KEYCACHE

  Enables caching of the initialized form of private RSA keys as described in Caching of Private RSA Keys.

  By default, this feature is enabled.

- SKB_USE_PERSISTENT_KEYCACHE

  If enabled, SKB will load contents of the key cache from the local storage when the SKB_Engine_GetInstance method is called for the first time, and will save contents of the key cache in the same storage when the last SKB_Engine_Release method is called.

  Without this flag, the key cache will be reset every time SKB is initialized, and its contents will always exist only in the device memory.

  You can specify the directory where the persistent storage should be placed by using the SKB_SetTempDir function.

  If, for some reason, SKB cannot read or write the key cache file, your application will continue functioning as if this flag is not set.

  By default, this feature is enabled on all platforms, except WebAssembly, Xbox, and PlayStation.

- SKB_SAVE_KEYCACHE_FREQUENTLY

  If enabled, SKB will save key cache contents to the local storage whenever any data is modified in the cache. This creates some performance penalty, but it also ensures that the latest updates in the key cache are not lost if your application is terminated unexpectedly. This flag is only usable if you have also set the SKB_USE_PERSISTENT_KEYCACHE flag.

  By default, this feature is disabled.

- SKB_EVAL_ENABLE_LOGGING

Enables logging to the debug output of the device. This feature can be enabled only in the evaluation edition of SKB. If you run into a problem when using SKB, you can enable this feature, repeat the problem, and send the log output to Zimperium specialists.

By default, this feature is disabled.

### SKB_GetFlags

This function returns the currently set SKB flags.

The function is declared as follows:

```
unsigned int SKB_GetFlags();
```

This function can be very useful when used together with the SKB_SetFlags function. For example, if you want to add the SKB_SAVE_KEYCACHE_FREQUENTLY flag to the existing flags, you can use the following combination of both functions:

```
SKB_SetFlags(SKB_GetFlags() | SKB_SAVE_KEYCACHE_FREQUENTLY);
```

## 7.2 Sensitive Operations Library

This section describes the Sensitive Operations Library, which is the `SkbInternalHelpers.lib` or `libSkbInternalHelpers.a` file (depending on the target platform) in the `lib` directory.

### 7.2.1 Overview

The Sensitive Operations Library is a static library that can convert plain keys to secure data objects, and vice versa.

**Important**

The Sensitive Operations Library is only intended for testing and debugging your application. Because functions provided by the Sensitive Operations Library are extremely insecure, this library is separated from the main API and you must never deliver it or link it with the target application.

### 7.2.2 Library Functions

The Sensitive Operations Library has its own interface, defined in the `SkbInternalHelpers.h` file, which is located in the `Include` directory.

This section describes the functions declared in the `SkbInternalHelpers.h` interface.

### SKB_CreateRawBytesFromPlain

This function creates an SKB_SecureData object from a plain data buffer. The type of the created SKB_SecureData object will be SKB_DATA_TYPE_BYTES as described in SKB_DataType.

The function is declared as follows:

```
SKB_Result
SKB_CreateRawBytesFromPlain(const SKB_Engine* engine,
                            const SKB_Byte*   plain,
                            SKB_Size          plain_size,
                            SKB_SecureData**  data);
```

The following are the parameters used:

- `engine`

  Pointer to the pre-initialized engine.

- `plain`

  Pointer to the data buffer containing the plain key.

- `plain_size`

  Size of the `plain` buffer in bytes.

- `data`

  Address of a pointer to the SKB_SecureData object that will contain the created key after this function is executed.

### SKB_CreatePlainFromRawBytes

This function returns a plain data buffer from an SKB_SecureData object. The type of the provided SKB_SecureData object must be SKB_DATA_TYPE_BYTES as described in SKB_DataType.

The function is declared as follows:

```
SKB_Result
SKB_CreatePlainFromRawBytes(const SKB_SecureData* data,
                            SKB_Byte*             plain,
                            SKB_Size*             plain_size);
```

The following are the parameters used:

- `data`

  Pointer to the SKB_SecureData object from which the plain data buffer must be created.

- `plain`

This parameter is either NULL or a pointer to the memory buffer where the plain key is to be written.

If this parameter is NULL, the method simply returns, in `plain_size`, the number of bytes that would be sufficient to hold the plain key, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the plain key, the method stores the plain key there, sets `plain_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `plain_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

- `plain_size`

  Pointer to the size of the `plain` buffer in bytes.

### SKB_CreateEccPrivateFromPlain

This function creates an `SKB_SecureData` object from a plain private ECC key. The type of the created `SKB_SecureData` object will be `SKB_DATA_TYPE_ECC_PRIVATE_KEY` as described in SKB_DataType.

The function is declared as follows:

```
SKB_Result
SKB_CreateEccPrivateFromPlain(const SKB_Engine* engine,
                              const SKB_Byte*   plain,
                              SKB_Size          plain_size,
                              SKB_SecureData**  data);
```

The following are the parameters used:

- `engine`

  Pointer to the pre-initialized engine.

- `plain`

  Pointer to the data buffer containing the private ECC key. For information on the ECC key format, see Private ECC Key.

- `plain_size`

  Size of the `plain` buffer in bytes.

- `data`

  Address of a pointer to the `SKB_SecureData` that will contain the created key after this function is executed.

### SKB_CreatePlainFromEccPrivate

This function derives a plain private ECC key from an `SKB_SecureData` object. The type of the provided `SKB_SecureData` object must be `SKB_DATA_TYPE_ECC_PRIVATE_KEY` as described in SKB_DataType.

The function is declared as follows:

```
SKB_Result
SKB_CreatePlainFromEccPrivate(const SKB_SecureData* data,
                              SKB_Byte*             plain,
                              SKB_Size*             plain_size);
```

The following are the parameters used:

- `data`

  Pointer to the SKB_SecureData from which the plain private ECC key must be derived.

- `plain`

  This parameter is either NULL or a pointer to the memory buffer where the plain key is to be written.

  If this parameter is NULL, the method simply returns, in `plain_size`, the number of bytes that would be sufficient to hold the plain key, and returns SKB_SUCCESS.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the plain key, the method stores the plain key there, sets `plain_size` to the exact number of bytes stored, and returns SKB_SUCCESS. If the buffer is not large enough, then the method sets `plain_size` to the number of bytes that would be sufficient, and returns SKB_ERROR_BUFFER_TOO_SMALL.

  The data will be provided using the big-endian format.

- `plain_size`

  Pointer to the size of the `plain` buffer in bytes.

## SKB_CreateRsaPrivateFromPlainPKCS8

This function creates an SKB_SecureData object from a plain private RSA key stored according to the PKCS#8 standard. The type of the created SKB_SecureData object will be SKB_DATA_TYPE_RSA_PRIVATE_KEY as described in SKB_DataType.

The function is declared as follows:

```
SKB_Result
SKB_CreateRsaPrivateFromPlainPKCS8(const SKB_Engine* engine,
                                   const SKB_Byte*   plain,
                                   SKB_Size          plain_size,
                                   SKB_SecureData**  data);
```

The following are the parameters used:

- `engine`

  Pointer to the pre-initialized engine.

- `plain`

  Pointer to the data buffer containing the private RSA key stored according to the PKCS#8 standard.

- `plain_size`

  Size of the plain buffer in bytes.

- `data`

  Address of a pointer to the `SKB_SecureData` that will contain the created key after this function is executed.

### SKB_CreateRsaPrivateFromPlain

This function creates an `SKB_SecureData` object from a plain private RSA key defined as a set of key components. The type of the created `SKB_SecureData` object will be `SKB_DATA_TYPE_RSA_PRIVATE_KEY` as described in `SKB_DataType`.

**Important**

The input parameters must be provided in the big-endian format.

The function is declared as follows:

```
SKB_Result
SKB_CreateRsaPrivateFromPlain(const SKB_Engine* engine,
                              void*             plain_p,
                              void*             plain_q,
                              void*             plain_d,
                              void*             plain_n,
                              SKB_Size          key_size,
                              SKB_SecureData**  data);
```

The following are the parameters used:

- `engine`

  Pointer to the pre-initialized engine.

- `plain_p`

  Pointer to the prime number "p".

- `plain_q`

  Pointer to the prime number "q".

- `plain_d`

  Pointer to the decryption exponent "d".

- `plain_n`

  Pointer to the modulus "n".

- `key_size`

  Size of the key in bytes.

- `data`

  Address of a pointer to the `SKB_SecureData` that will contain the created key after this function is executed.

### SKB_CreatePlainFromRsaPrivate

This function derives plain private RSA key components from an `SKB_SecureData` object. The type of the provided `SKB_SecureData` object must be `SKB_DATA_TYPE_RSA_PRIVATE_KEY` as described in `SKB_DataType`.

**Important**

The output data buffers will be provided in the big-endian format.

The function is declared as follows:

```
SKB_Result
SKB_CreatePlainFromRsaPrivate(const SKB_SecureData* data,
                              SKB_Byte*             p,
                              SKB_Byte*             q,
                              SKB_Byte*             d,
                              SKB_Byte*             n,
                              SKB_Size*             key_size);
```

The following are the parameters used:

- `data`

  Pointer to the `SKB_SecureData` from which the plain private RSA key components must be derived.

- `plain_p`

  This parameter is either NULL or a pointer to the memory buffer where the prime number "p" is to be written.

  If this parameter is NULL, the method simply returns, in `key_size`, the number of bytes that would be sufficient to hold the prime number "p", and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the prime number "p", the method stores the value there, sets `key_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `key_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

- `plain_q`

  Pointer to the prime number "q". This parameter works similar to `plain_p` and will have the same size.

- `plain_d`

  Pointer to the decryption exponent "d". This parameter works similar to `plain_p` and will have the same size.

- `plain_n`

  Pointer to the modulus "n". This parameter works similar to `plain_p` and will have the same size.

- `key_size`

  Pointer to the size of the prime number "p", prime number "q", decryption exponent "d", and modulus "n".

## SKB_CreateRsaPublicFromPlainPKCS1

This function creates an `SKB_SecureData` object from a plain public RSA key stored according to the PKCS#1 standard. The type of the created `SKB_SecureData` object will be `SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT` as described in SKB_DataType.

The function is declared as follows:

```
SKB_Result
SKB_CreateRsaPublicFromPlainPKCS1(const SKB_Engine* engine,
                                  const SKB_Byte*   plain,
                                  SKB_Size          plain_size,
                                  SKB_SecureData**  data);
```

The following are the parameters used:

- `engine`

  Pointer to the pre-initialized engine.

- `plain`

  Pointer to the data buffer containing the public RSA key stored according to the PKCS#1 standard.

- `plain_size`

  Size of the `plain` buffer in bytes.

- `data`

Address of a pointer to the SKB_SecureData that will contain the created key after this function is executed.

## SKB_CreateRsaPublicFromPlain

This function creates an SKB_SecureData object from a plain public RSA key defined as a set of key components. The type of the created SKB_SecureData object will be SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT as described in SKB_DataType.

**Important**

The input parameters must be provided in the big-endian format.

The function is declared as follows:

```
SKB_Result
SKB_CreateRsaPublicFromPlain(const SKB_Engine* engine,
                             const SKB_Byte*   plain_e,
                             const SKB_Byte*   plain_n,
                             const SKB_Size    key_size,
                             SKB_SecureData**  data);
```

The following are the parameters used:

- engine

  Pointer to the pre-initialized engine.

- plain_e

  Pointer to the public exponent "e".

- plain_n

  Pointer to the modulus "n".

- key_size

  Size of the key in bytes.

- data

  Address of a pointer to the SKB_SecureData that will contain the created key after this function is executed.

## SKB_CreateUnwrapBytesFromPlain

This function creates an SKB_SecureData object from a plain data buffer. The type of the created SKB_SecureData object will be SKB_DATA_TYPE_UNWRAP_BYTES as described in SKB_DataType.

The function is declared as follows:

```
SKB_Result
SKB_CreateUnwrapBytesFromPlain(const SKB_Engine* engine,
                               const SKB_Byte*   plain,
                               SKB_Size          plain_size,
                               SKB_SecureData**  data);
```

The following are the parameters used:

- `engine`

  Pointer to the pre-initialized engine.

- `plain`

  Pointer to the data buffer containing the plain key.

- `plain_size`

  Size of the `plain` buffer in bytes.

- `data`

  Address of a pointer to the `SKB_SecureData` object that will contain the created key after this function is executed.

## SKB_CreatePlainFromUnwrapBytes

This function returns a plain data buffer from an `SKB_SecureData` object. The type of the provided `SKB_SecureData` object must be `SKB_DATA_TYPE_UNWRAP_BYTES` as described in `SKB_DataType`.

The function is declared as follows:

```
SKB_Result
SKB_CreatePlainFromUnwrapBytes(const SKB_SecureData* data,
                               SKB_Byte*             plain,
                               SKB_Size*             plain_size);
```

The following are the parameters used:

- `data`

  Pointer to the `SKB_SecureData` object from which the plain data buffer must be created.

- `plain`

  This parameter is either NULL or a pointer to the memory buffer where the plain key is to be written.

  If this parameter is NULL, the method simply returns, in `plain_size`, the number of bytes that would be sufficient to hold the plain key, and returns `SKB_SUCCESS`.

If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the plain key, the method stores the plain key there, sets `plain_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `plain_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

- `plain_size`

  Pointer to the size of the `plain` buffer in bytes.

# 8. Applying Tamper Resistance with zShield

This chapter describes the benefits of applying zShield tamper resistance to SKB, and the steps necessary to do it.

## 8.1 Tamper Resistance Overview

When ordering an SKB package, you have the option to receive an edition of SKB that is protected by zShield. This means, that the SKB code base will be altered in a special way that enables a number of additional security features as described in the following table.

| Feature | Description |
|---|---|
| Integrity protection | Hundreds of embedded overlapping checksums prevent modifications of the binary code of the entire executable (not just SKB). |
| Code obfuscation | SKB code is transformed to make it difficult to analyze and reverse engineer. |
| Anti-debug protection | Platform-specific anti-debug code prevents the use of main-stream debuggers on the SKB-protected application. |
| Jailbreak detection | Normally, a cracked or modified iOS/iPadOS application can be run only on jailbroken devices. Jailbreak detection protects the SKB-protected application from being executed on such devices. |
| Rooting detection | Rooting creates a security risk for Android applications that deal with sensitive data or enforce certain usage restrictions. Rooting detection will crash the SKB-protected application if a rooted device is detected. |
| Inlining of static functions | SKB's static functions with simple declarations are inlined into the calling functions. Such an operation increases the obfuscation level of the SKB code and makes it more difficult to trace. |
| String literal obfuscation | Large portion of SKB's string literals, or string constants, get encrypted in the code and are decrypted only before they are actually used. The purpose of this feature is to hide useful and sensitive information from potential attackers. |
| Customizable defense action | Optionally, you can request an SKB package that is configured to execute specific callback functions depending on the type of attack it detects. Additionally, when requesting the package, you can choose whether the program state should be corrupted, or the application should be left running after a callback function is invoked. |

Applying tamper resistance does not directly improve the way SKB protects keys. Key protection is a central feature of SKB that is equally strong regardless of whether tamper resistance is applied or not.

## 8.2 Supported Platforms

Tamper resistant SKB is currently available only for a subset of the supported target platforms as indicated in Supported Platforms.

## 8.3 Callback Functions

When you request SKB that is protected with zShield, you may optionally specify whether you want SKB to invoke specific callback functions when threats are detected. If you do not choose to use callback functions, SKB will corrupt the application state (resulting in a crash) whenever it detects a threat. Callback functions allow you to implement other responses to attacks. Additionally, when requesting an SKB package that uses callback functions, you may also specify if you want the attacked application to continue execution after a callback function is invoked.

If the SKB package that you received is configured to use callbacks, you have to provide implementation for the callback functions in the source code. The following are the threats for which callback functions are supported in SKB:

- Debugger

  The following function is called by SKB when it detects that the application is run under a debugger:

  ```
  void SKB_Callback_AntiDebug();
  ```

- Rooted Android device

  The following function is called by SKB when it detects that the application is run on a rooted Android device:

  ```
  void SKB_Callback_Root();
  ```

- Jailbroken iOS/iPadOS device

  The following function is called by SKB when it detects that the application is run on a jailbroken iOS/iPadOS device:

  ```
  void SKB_Callback_Jailbreak();
  ```

## 8.4 Binary Update Tool

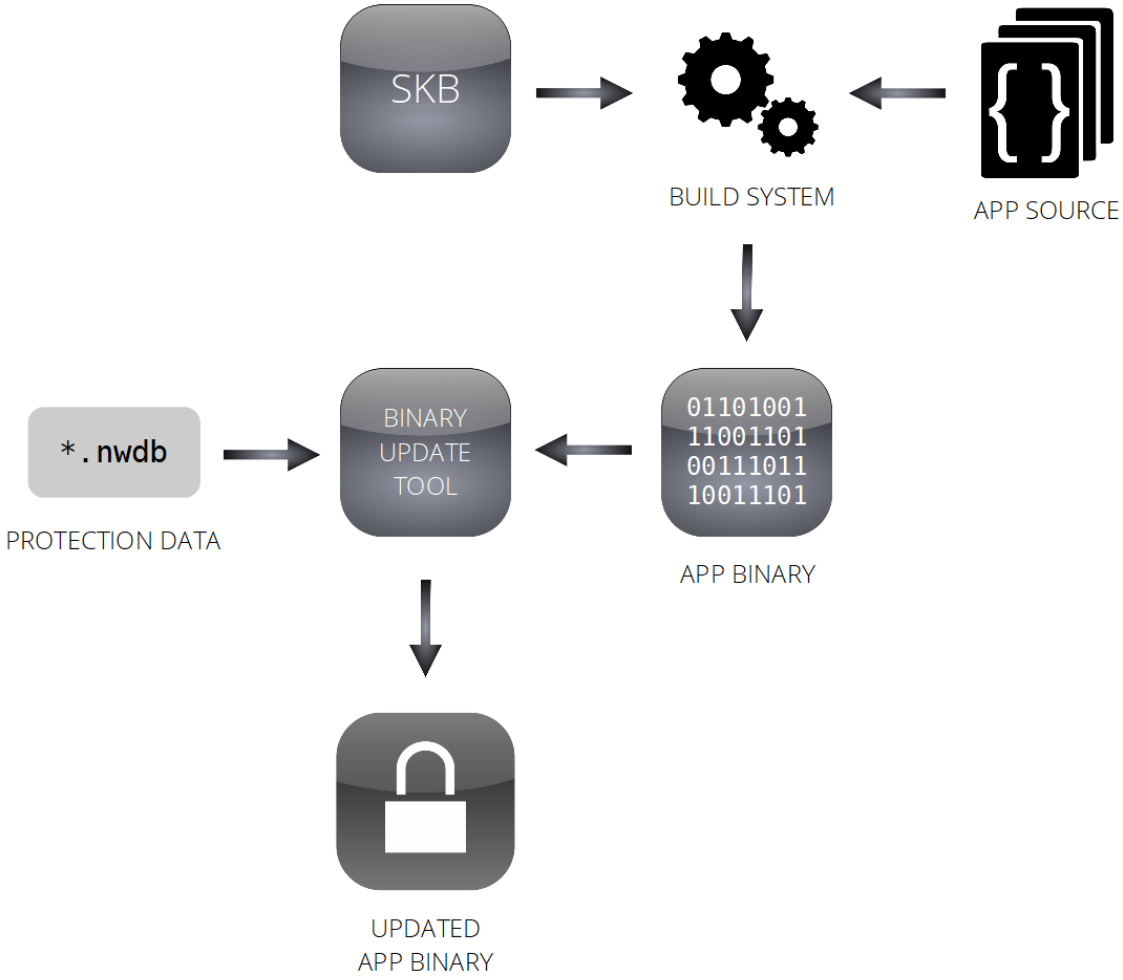
If you are using SKB that is protected with zShield, you have to run the final built application executable through a binary update process to correctly adjust the embedded integrity protection checksums. Adjustment of the binary code is done using the command-line utility `scp-update-binary`, called the Binary Update Tool, which is available in the SKB package.

**Important**

If the binary update process is not executed, the protected application will crash at runtime with a segmentation fault.

## 8.4.1 Binary Update Tool Overview

Every time the SKB-protected application is built, its binary code must be adjusted using the Binary Update Tool. As input, the Binary Update Tool requires the binary executable of the protected application and the `.nwdb` file that is delivered together with SKB. If you are using Secure PIN, its `libSecurePin.nwdb` file must also be supplied.




**Important**

macOS and iOS/iPadOS applications must be re-signed after running the Binary Update Tool, because the binary footprint will be modified. You can perform signing using the `codesign` tool.

You do not have to run the Binary Update Tool if your application (that is linked with the SKB library) itself is being protected by zShield. In that case, you only need to make sure the `.nwdb` files delivered together with SKB are placed next to the SKB library. zShield will automatically detect them and adjust the security features in the final executable.

## 8.4.2 Binary Update Tool Requirements

The SKB package contains three editions of the Binary Update Tool—for Windows, macOS, and Linux. The following are the specific system requirements the Binary Update Tool has for these operating systems:

- Windows 10 or 11 (x86-64)
- macOS 10.14 or later
- Linux (x86-64) with glibc 2.27 or later and zlib 1.2.11 or later

## 8.4.3 Running the Binary Update Tool

The Binary Update Tool is provided in a separate subdirectory named `scp-update-binary`, which is located in the `bin` directory. The `scp-update-binary` directory contains the utility executable, an instructions text file, and libraries the utility requires.

The `.nwdb` file, which needs to be provided to the Binary Update Tool, is located in the `lib` directory. Each target platform, for which tamper resistance is supported, has a separate `.nwdb` file. For Windows targets, this file is named `SecureKeyBox.nwdb`; for all other targets, it is named `libSecureKeyBox.nwdb`. If you are using Secure PIN, its `libSecurePin.nwdb` file must also be processed; you may do this either together with the SKB `.nwdb` file at the same time, or by running the Binary Update Tool separately at another time.

To process the SKB-protected application with the Binary Update Tool, execute the following command:

```
scp-update-binary --binary=«compiled executable» «.nwdb file(s)»
```

The `--binary` parameter specifies the path to the application executable that is linked with the SKB library.

As the final parameter, you must provide the `.nwdb` file of the particular target platform. If you are using Secure PIN and want to update the final application at the same time, the `libSecurePin.nwdb` file must also be provided separated by a space.

**Note**

You can see a brief description of the available parameters by running the Binary Update Tool with the `--help` parameter.

After the application executable is successfully processed by the Binary Update Tool, you can safely distribute the application to your customers.

# 9. Native API

This chapter provides full reference information about the SKB Native API.

## 9.1 Overview of the Native API

The Native API is a C interface, composed of a number of object classes. Even though the interface is an ANSI C interface, it adopts an object-oriented style. The header file declares a set of classes and class methods. Each method of a class interface is a function whose first argument is a reference to an instance of the same class. The data type that represents references to object instances is a pointer to an opaque C structure. It may be considered as analogous to a pointer to a C++ object.

For example, for the class named `SKB_Cipher`, the data type `SKB_Cipher` is the name of a C structure. The function name for one of the methods of `SKB_Cipher` is `SKB_Cipher_ProcessBuffer`, and the function takes `SKB_Cipher*` as its first parameter.

## 9.2 Initializing SKB Objects

An SKB object is obtained by declaring a pointer that will point to the object that needs to be created, and passing the address of that pointer to a particular method. The method creates the object and sets the pointer to refer to it.

For example, the first object you need to create is `SKB_Engine`, which represents an instance of an engine that can initialize other API objects. `SKB_Engine` is obtained by calling the method `SKB_Engine_GetInstance`, which is declared as follows:

```
SKB_Result SKB_Engine_GetInstance(SKB_Engine** engine);
```

The parameter `engine` is the address of a pointer to an `SKB_Engine` object. This method creates an `SKB_Engine` instance and sets the pointer to refer to it. Here is a sample call:

```
SKB_Engine* engine = NULL;
SKB_Result result;
result = SKB_Engine_GetInstance(&engine);
```

## 9.3 Releasing SKB Objects

To avoid exceptions and correctly release memory, all SKB objects must be released when they are no longer needed by calling the corresponding release methods. `SKB_Engine`, which is a singleton object, has internal counters for SKB objects created. When an object is created, the corresponding counter is incremented, and when an object is released, the counter is decremented. The engine itself is released only after the last release method is called when the counters reach zero. For this reason, every object creation method invocation must have a corresponding release method invocation.

## 9.4 Making Method Calls

A call to a method of a particular instance is done by calling a function and passing a pointer to the instance as the first parameter.

For example, once an SKB_Engine object is created, as shown in the previous section, all the SKB_Engine methods can be called to operate on that instance. One such method is SKB_Engine_GetInfo, which is used to obtain information about the engine (version number, properties, and so on). This method is declared as follows:

```
SKB_Result SKB_Engine_GetInfo(const SKB_Engine* self, SKB_EngineInfo* info);
```

It stores the engine information in the SKB_EngineInfo structure pointed to by the info parameter. Assuming engine is a pointer previously set by SKB_Engine_GetInstance to refer to the SKB_Engine instance it created, SKB_Engine_GetInfo can be invoked as follows:

```
SKB_Result result;
SKB_EngineInfo engineInfo;
result = SKB_Engine_GetInfo(engine, &engineInfo);
```

## 9.5 Method Return Values

All SKB API methods return a value of type SKB_Result. All possible return values are declared using #define macros in the SKB header file.

**Important**

The actual macro values are randomized, which means that they differ from one SKB package to another. Therefore, you must always refer to the return values by their macro names.

When a method call succeeds, the return value is SKB_SUCCESS; in all other cases the returned value signifies an error. The following return values are used by SKB:

- SKB_SUCCESS

  The called method was executed successfully.

- SKB_ERROR_INTERNAL

  An internal SKB error occurred. Please consult with Zimperium for assistance.

- SKB_ERROR_INVALID_PARAMETERS

  Invalid parameters were supplied to the method.

- SKB_ERROR_NOT_SUPPORTED

  The configuration provided to the method is not supported by SKB. It may also mean that you tried to execute an algorithm that is not included in the SKB package you requested.

- SKB_ERROR_OUT_OF_RESOURCES

  The method failed to allocate the required amount of memory.

- SKB_ERROR_BUFFER_TOO_SMALL

  The provided memory buffer was not large enough to contain the output.

- SKB_ERROR_INVALID_FORMAT

  This value is returned in the following cases:

  - The format of the input buffer was invalid.
  - You are importing a secure data object when the device ID set in SKB does not match the device ID that was used when exporting the data. For more information on this feature, see Binding Keys to a Specific Device.
  - Integrity check failed during decryption or unwrapping of the SKB_CIPHER_ALGORITHM_NIST_AES algorithm.

- SKB_ERROR_ILLEGAL_OPERATION

  Currently, this return value is never used by SKB.

- SKB_ERROR_INVALID_STATE

  You attempted to perform an invalid operation on the SKB_Transform object, such as the following:

  - You tried to add an input buffer to an SKB_Transform object after its SKB_Transform_GetOutput method was called.
  - You tried to call the SKB_Transform_GetOutput method again after it was already executed.
  - You executed the SKB_Transform_AddBytes method more than once on an SKB_Transform object that is associated with a signing algorithm that does not have its own hash function (SKB_SIGNATURE_ALGORITHM_RSA, SKB_SIGNATURE_ALGORITHM_DSA or SKB_SIGNATURE_ALGORITHM_ECDSA). For these algorithms, the SKB_Transform_AddBytes method may be called only once. For more information, see Calculating a Signature.
  - You executed the SKB_Cipher_ProcessAad or SKB_Cipher_ProcessBuffer method after the SKB_Cipher_ProcessFinal method was executed on the same SKB_Cipher object. For information on the general procedure to be followed in this case, see Using Authenticated Encryption with Additional Data.

- SKB_ERROR_OUT_OF_RANGE

  The specified offset or index of the input buffer was out of range.

- SKB_ERROR_EVALUATION_EXPIRED

  The evaluation period of the current SKB package has expired.

- SKB_ERROR_KEY_CACHE_FAILED

A key cache operation failed.

- `SKB_ERROR_INVALID_EXPORT_KEY_VERSION`

  Either you were trying to upgrade a key whose version number is equal to or greater than that of the current SKB package, or you were trying to import a key whose version is not equal to that of the current SKB package. For information on this fature, see One Way Key Upgrading.

- `SKB_ERROR_INVALID_EXPORT_KEY`

  This error may result from one of the following causes:

  - The export key of the current SKB package does not match the export key that was used for exporting the data you were trying to import or upgrade. For details on this, see Persistent Exporting.
  - The current SKB package does not have the export key set as described in Key Embedding Tool.

- `SKB_ERROR_AUTHENTICATION_FAILURE`

  This value is returned in the following cases:

  - When performing AES decryption in the GCM or CCM mode, this error is returned by the `SKB_Cipher_ProcessFinal` method (in the GCM mode) or the `SKB_Cipher_ProcessBuffer` method (in the CCM mode) if the provided authentication tag does not match the one calculated during decryption.
  - This error is returned by the `SKB_Engine_CreateDataFromWrapped` method if the `SKB_CIPHER_ALGORITHM_ASC_X9_TR_31` algorithm is used and MAC validation fails.

## 9.6 Restrictions of Multi-Threading

As a general rule, SKB objects are not synchronized and therefore must not be shared between multiple threads. However, there are two exceptions to this rule:

- The `SKB_Engine` object is thread-safe and can be shared between multiple threads using the `SKB_Engine_GetInstance` method. This method will always return the same `SKB_Engine` instance. However, the `SKB_Engine_SetExportKey` and `SKB_Engine_IsExportKeySet` methods, which are described in Using the Converted Export Key, are not thread-safe; they must be called only when it can be ensured that no other threads call them or perform key exporting or importing.
- Since the `SKB_SecureData` object is immutable, it can also be shared between multiple threads.

## 9.7 Classes

This section describes the classes of the API. Most operations are performed via these classes and their related methods.

## 9.7.1 SKB_Engine

SKB_Engine is the first object that you create before using the SKB API. It is used to initialize other API objects, and it also stores the key cache and the device ID.

SKB_Engine is a singleton object, which means that the SKB_Engine_GetInstance method, which is used to obtain the engine object, will always return the same SKB_Engine instance. Therefore, you may choose to create the SKB_Engine object in the global scope and retain it for the entire duration of your application's execution. This will improve the performance and eliminate the need for you to set the device ID and configure the key cache multiple times in your application.

## 9.7.2 SKB_SecureData

SKB_SecureData contains any data whose value is white-box protected and hidden from the outside world but can be internally operated on by SKB. Usually, the SKB_SecureData object is the container for cryptographic keys protected by SKB. Secure data objects can be operated by SKB cryptographic functions but their contents cannot be accessed.

There are several ways how SKB_SecureData objects are obtained:

- unwrapping encrypted keys
- importing previously exported keys
- obtaining as a shared secret via a key agreement algorithm
- generating a new random SKB_SecureData object to be used as a cryptographic key
- deriving an SKB_SecureData object from another SKB_SecureData object
- wrapping plain keys

## 9.7.3 SKB_Cipher

SKB_Cipher is an object that can encrypt or decrypt data. It encapsulates the attributes and parameters necessary to perform cryptographic operations on data buffers.

## 9.7.4 SKB_Transform

SKB_Transform is an object that can calculate a digest, sign data, or verify a signature. This object can operate both on plain data and secure data. The output is always plain data.

## 9.7.5 SKB_KeyAgreement

SKB_KeyAgreement is an object used to execute the key agreement algorithm.

# 9.8 Methods

This section describes all the methods provided by the SKB API.

## 9.8.1 SKB_Engine_GetInstance

This method creates an SKB_Engine object, which is the first object you must obtain before using the Native API.

**Important**

Make sure that every SKB_Engine_GetInstance call has a corresponding SKB_Engine_Release call to correctly release the memory.

The method is declared as follows:

```
SKB_Result
SKB_Engine_GetInstance(SKB_Engine** engine);
```

The parameter engine is an address of a pointer to the SKB_Engine object. After execution, this method creates an SKB_Engine instance and sets the pointer to refer to the new instance. Every subsequent call of the SKB_Engine_GetInstance method will return the same SKB_Engine object until this object is released.

## 9.8.2 SKB_Engine_Release

This method releases an SKB_Engine instance from the memory when it is no longer needed.

**Important**

Make sure that every SKB_Engine_GetInstance call has a corresponding SKB_Engine_Release call to correctly release the memory. Also, all other SKB objects created via the SKB_Engine object must be released before you call the SKB_Engine_Release method.

The method is declared as follows:

```
SKB_Result
SKB_Engine_Release(SKB_Engine* self);
```

The parameter `self` is a pointer to the engine instance to be released.

## 9.8.3 SKB_Engine_SetDeviceId

This method sets the device ID, which is a byte array of arbitrary length, that will be combined with every exported key and every key stored in the key cache. This method enables you to bind all stored keys to a specific device. By default, when an SKB engine is initialized, there is no device ID set, which means that exported keys will be device-independent.

The method is declared as follows:

```
SKB_Result
SKB_Engine_SetDeviceId(SKB_Engine*     self,
                       const SKB_Byte* id,
                       SKB_Size        size);
```

The following are the parameters used:

- `self`

  Pointer to the pre-initialized engine.

- `id`

  Pointer to the byte array containing the device ID.

  You have to generate this byte array yourself based on some hardware details or other environment-specific parameters.

- `size`

  Number of bytes in the `id` parameter. The device ID can be of arbitrary length.

  If the size is 0, SKB will remove the previously set device ID. This can be useful when the device ID is no longer needed and the default export format (based only on the export key) needs to be restored.

## 9.8.4 SKB_Engine_GetInfo

This method populates the SKB_EngineInfo structure with generic information about the initialized engine.

**Important**

The contents of the populated SKB_EngineInfo structure will not be valid after the corresponding SKB_Engine object is released from memory. During examination of the SKB_EngineInfo object, the SKB_Engine object must exist.

The method is declared as follows:

```
SKB_Result
SKB_Engine_GetInfo(const SKB_Engine* self,
                   SKB_EngineInfo*   info);
```

The following are the parameters used:

- `self`

  Pointer to the pre-initialized engine that you want to get the information about.

- `info`

  Pointer to the `SKB_EngineInfo` structure to be populated with the engine information.

## 9.8.5 SKB_Engine_CreateDataFromWrapped

This method creates a new `SKB_SecureData` object from a wrapped buffer of data by unwrapping it with a previously loaded key. The unwrapped data is never exposed in plain form.

As a special case of calling this method, you can also load a plain buffer of data as an `SKB_SecureData` object. In this case, the unwrapping algorithm and the unwrapping key are not specified. This operation should be used with extreme care because you are providing the key in plain form. Use this approach only in a highly protected environment. Loading of plain keys can be executed only if this feature is enabled in SKB.

The `SKB_Engine_CreateDataFromWrapped` method is declared as follows:

```
SKB_Result
SKB_Engine_CreateDataFromWrapped(SKB_Engine*           self,
                                 const SKB_Byte*       wrapped,
                                 SKB_Size              wrapped_size,
                                 SKB_DataType          wrapped_type,
                                 SKB_DataFormat        wrapped_format,
                                 SKB_CipherAlgorithm   wrapping_algorithm,
                                 const void*           wrapping_parameters,
                                 const SKB_SecureData* unwrapping_key,
                                 SKB_SecureData**      data);
```

The following are the parameters used:

- `self`

  Pointer to the pre-initialized engine.

- `wrapped`

  Pointer to the buffer of encrypted data (cryptographic key) to be unwrapped.

  If you are unwrapping raw bytes using the Triple DES or AES algorithm, the buffer must follow the formatting described in Wrapped Data Buffer.

If you are unwrapping a private RSA key using AES, the input buffer must be DER-encoded PKCS#8 format without encryption. If the private RSA key consists of components optimized according to the Chinese Remainder Theorem (CRT), this buffer must be formatted as described in AES-Wrapped Private RSA Key in the CRT Format.

If you are unwrapping an AES-wrapped private ECC key that has the SKB_DATA_FORMAT_ECC_PRIVATE data format, see AES-Wrapped Private ECC Key for information on how the input buffer must be formatted.

If you are unwrapping a Triple DES key, make sure it is formatted as described in Key Format for the Triple DES Cipher.

If you are unwrapping raw bytes using the SKB_CIPHER_ALGORITHM_ASC_X9_TR_31 algorithm, please make sure the following requirements are met:

- The encoding of the buffer corresponds to the way it is defined in ASC X9 TR 31-2018. Namely, the buffer must include the key block header, and the header must be encoded as described in section "A.2. Key Block Header (KBH)"; the encrypted data and the MAC must be encoded as hex-ASCII characters.
- The size of the key being unwrapped does not exceed 78 bytes.

If the SKB_CIPHER_ALGORITHM_NULL algorithm is used, which means that you are importing a plain key, only keys of the following types and formats are supported:

- SKB_DATA_TYPE_BYTES key type using the SKB_DATA_FORMAT_RAW format
- SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT key type using the SKB_DATA_FORMAT_PKCS1 format
- SKB_DATA_TYPE_RSA_PRIVATE_KEY key type using the SKB_DATA_FORMAT_PKCS8 format
- SKB_DATA_TYPE_ECC_PRIVATE_KEY key type using the SKB_DATA_FORMAT_ECC_PRIVATE format

- `wrapped_size`

  Size of the `wrapped` buffer in bytes.

- `wrapped_type`

  Type of the wrapped key. The available types are defined in the SKB_DataType enumeration.

  If you are unwrapping a private RSA key that follows the optimization principles of CRT, this type must be SKB_DATA_TYPE_RSA_PRIVATE_KEY.

- `wrapped_format`

  Format how the wrapped key is stored in the input data buffer. The available formats are defined in the SKB_DataFormat enumeration.

  If you are unwrapping a private RSA key that follows the optimization principles of CRT, you must choose one of these formats:

  - SKB_DATA_FORMAT_CRT: Components of the wrapped key are of variable length as described here.

- SKB_DATA_FORMAT_CRT_EQUAL_LEN: Components of the wrapped key are of equal length as described here.

- wrapping_algorithm

  Cryptographic algorithm to be used for unwrapping the data. The available algorithms are defined in the SKB_CipherAlgorithm enumeration.

  The following is a list of algorithms that support unwrapping (the supported input data types are listed in the parentheses):

  - SKB_CIPHER_ALGORITHM_NULL (all data types)
  - SKB_CIPHER_ALGORITHM_AES_128_ECB (raw bytes, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_128_CBC (raw bytes, private RSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_128_CTR (raw bytes, private RSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_128_GCM (raw bytes, private RSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_192_ECB (raw bytes, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_192_CBC (raw bytes, private RSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_192_CTR (raw bytes, private RSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_192_GCM (raw bytes, private RSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_256_ECB (raw bytes, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_256_CBC (raw bytes, private RSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_256_CTR (raw bytes, private RSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_256_GCM (raw bytes, private RSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_RSA (raw bytes)
  - SKB_CIPHER_ALGORITHM_RSA_1_5 (raw bytes)
  - SKB_CIPHER_ALGORITHM_RSA_OAEP (raw bytes)
  - SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA256 (raw bytes)
  - SKB_CIPHER_ALGORITHM_ECC_ELGAMAL (raw bytes)
  - SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB (raw bytes)
  - SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC (raw bytes)
  - SKB_CIPHER_ALGORITHM_NIST_AES (raw bytes)
  - SKB_CIPHER_ALGORITHM_XOR (raw bytes)
  - SKB_CIPHER_ALGORITHM_ASC_X9_TR_31 (raw bytes)

If the SKB_CIPHER_ALGORITHM_NULL algorithm is used, the method assumes that the key in the input buffer is in plain form. Then you do not have to provide the unwrapping key or unwrapping parameters.

If the SKB_CIPHER_ALGORITHM_AES_128_CTR, SKB_CIPHER_ALGORITHM_AES_192_CTR, or SKB_CIPHER_ALGORITHM_AES_256_CTR algorithm is used, the counter size will be 16 bytes.

If the SKB_CIPHER_ALGORITHM_NIST_AES algorithm is used, in the case of integrity check failure this method will return the SKB_ERROR_INVALID_FORMAT value.

If the SKB_CIPHER_ALGORITHM_ECC_ELGAMAL algorithm is used, see the special instructions.

If the SKB_CIPHER_ALGORITHM_ASC_X9_TR_31 algorithm is used and MAC validation fails, the method will return the SKB_ERROR_AUTHENTICATION_FAILURE value.

If you are unwrapping a private RSA key using the optimization principles of CRT (wrapped_format is SKB_DATA_FORMAT_CRT or SKB_DATA_FORMAT_CRT_EQUAL_LEN), only the following algorithms are supported with no padding:

- SKB_CIPHER_ALGORITHM_AES_128_ECB
- SKB_CIPHER_ALGORITHM_AES_192_ECB
- SKB_CIPHER_ALGORITHM_AES_256_ECB
- SKB_CIPHER_ALGORITHM_AES_128_CBC
- SKB_CIPHER_ALGORITHM_AES_192_CBC
- SKB_CIPHER_ALGORITHM_AES_256_CBC

For detailed information on the supported unwrapping algorithms and their restrictions, see Supported Algorithms.

- wrapping_parameters

  Additional parameters for the unwrapping algorithm.

  If you are using one of the following algorithms, you can optionally point this parameter to the SKB_BlockCipherUnwrapParameters structure to specify the padding type:

  - SKB_CIPHER_ALGORITHM_AES_128_ECB
  - SKB_CIPHER_ALGORITHM_AES_192_ECB
  - SKB_CIPHER_ALGORITHM_AES_256_ECB
  - SKB_CIPHER_ALGORITHM_AES_128_CBC
  - SKB_CIPHER_ALGORITHM_AES_192_CBC
  - SKB_CIPHER_ALGORITHM_AES_256_CBC
  - SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB
  - SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC

If you use any of the algorithms above and set the `wrapping_parameters` value to NULL, then the padding type will be chosen as follows:

- In ECB mode, no padding will be used.
- In CBC mode, the XML encryption padding will be used, which is the equivalent of the `SKB_PADDING_TYPE_XMLENC` value of the `SKB_PaddingType` enumeration.

If you are unwrapping a private RSA key that follows the optimization principles of CRT, only the no-padding variants of the ECB and CBC modes are supported.

If you are using one of the following algorithms, you must point this parameter to the `SKB_GcmUnwrapParameters` structure:

- `SKB_CIPHER_ALGORITHM_AES_128_GCM`
- `SKB_CIPHER_ALGORITHM_AES_192_GCM`
- `SKB_CIPHER_ALGORITHM_AES_256_GCM`

If you are using the `SKB_CIPHER_ALGORITHM_ECC_ELGAMAL` algorithm, this parameter must be a pointer to the `SKB_EccParameters` structure. For special instructions on using the ElGamal ECC unwrapping algorithm, see Unwrapping a Key Wrapped with the ElGamal ECC Algorithm.

For all other cases, set this parameter to NULL.

- `unwrapping_key`

  `SKB_SecureData` object containing the key needed to unwrap the data.

  If the `SKB_CIPHER_ALGORITHM_NULL` algorithm is used, this parameter should be set to NULL.

  **Important**

  For AES and Triple DES unwrapping algorithms, it is recommended to always use the unwrapping key of type `SKB_DATA_TYPE_UNWRAP_BYTES`, because this type of keys cannot be insecurely misused in other cryptographic operations, such as decryption.

- `data`

  Address of a pointer to the `SKB_SecureData` object that will contain the unwrapped key after this method is executed.

## 9.8.6 SKB_Engine_CreateDataFromExported

This method imports data that was previously exported using the `SKB_SecureData_Export` method or prepared using the Key Export Tool.

The method is declared as follows:

```
SKB_Result
SKB_Engine_CreateDataFromExported(SKB_Engine*      self,
                                  const SKB_Byte*  exported,
                                  SKB_Size         exported_size,
                                  SKB_SecureData** data);
```

The following are the parameters used:

- `self`

  Pointer to the pre-initialized engine.

- `exported`

  Pointer to the memory buffer containing the exported data.

- `exported_size`

  Size of the `exported` buffer in bytes.

- `data`

  Address of a pointer to the `SKB_SecureData` object that will be created by this method. This object will contain the imported data.

## 9.8.7 SKB_Engine_WrapDataFromPlain

This method takes a plain data buffer, encrypts it with a key stored in an `SKB_SecureData` object, and stores the output as a new `SKB_SecureData` object. For more information on this method, see Wrapping Plain Data.

The method is declared as follows:

```
SKB_Result
SKB_Engine_WrapDataFromPlain(SKB_Engine*           self,
                             const SKB_Byte*       plain,
                             SKB_Size*             plain_size,
                             SKB_DataType          data_type,
                             SKB_DataFormat        plain_format,
                             SKB_CipherAlgorithm   algorithm,
                             const void*           encryption_parameters,
                             const SKB_SecureData* encryption_key,
                             const SKB_Byte*       iv,
                             SKB_Size              iv_size,
                             SKB_SecureData**      data);
```

The following are the parameters used:

- `self`

Pointer to the pre-initialized engine.

- `plain`

  Pointer to the memory buffer where the plain input data is stored.

- `plain_size`

  Pointer to a variable that holds the size of the input data in bytes.

- `data_type`

  Type of data stored in the input buffer. The available types are defined in the SKB_DataType enumeration.

  Currently, this method supports only the SKB_DATA_TYPE_BYTES data type.

- `plain_format`

  Format how the plain data is stored in the input buffer. The available formats are defined in the SKB_DataFormat enumeration.

  Currently, this method supports only the SKB_DATA_FORMAT_RAW data type.

- `algorithm`

  Algorithm to be used for encrypting the input data. Available algorithms are defined in the SKB_CipherAlgorithm enumeration.

  Currently, this method supports only the following algorithms:

  - SKB_CIPHER_ALGORITHM_AES_128_ECB
  - SKB_CIPHER_ALGORITHM_AES_128_CBC
  - SKB_CIPHER_ALGORITHM_AES_192_ECB
  - SKB_CIPHER_ALGORITHM_AES_192_CBC
  - SKB_CIPHER_ALGORITHM_AES_256_ECB
  - SKB_CIPHER_ALGORITHM_AES_256_CBC
  - SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB
  - SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC

- `encryption_parameters`

  Pointer to a structure that provides additional parameters for the cipher.

  Currently, this parameter must always be NULL.

- `encryption_key`

  Pointer to the SKB_SecureData object containing the encryption key.

  The encryption key must be of type SKB_DATA_TYPE_BYTES or SKB_DATA_TYPE_UNWRAP_BYTES.

- `iv`

  Pointer to the initialization vector if the CBC mode is used for the selected encryption algorithm.

  If the ECB mode is used, the value of this parameter must be NULL.

- `iv_size`

  Size of the initialization vector in bytes.

  If the value of the `iv` parameter is NULL, this parameter must be 0.

- `data`

  Address of a pointer to the `SKB_SecureData` object that will contain the output when this method is executed.

## 9.8.8 SKB_Engine_GenerateSecureData

This method creates a new random `SKB_SecureData` object based on the provided parameters. This operation is typically used for generating new random keys.

The method is declared as follows:

```
SKB_Result
SKB_Engine_GenerateSecureData(SKB_Engine*      self,
                              SKB_DataType     data_type,
                              const void*      generate_parameters,
                              SKB_SecureData** data);
```

The following are the parameters used:

- `self`

  Pointer to the pre-initialized engine.

- `data_type`

  Type of data to be generated. The available types are defined in the SKB_DataType enumeration.

  Currently, only the following data types are supported for generating secure data objects:

  - `SKB_DATA_TYPE_BYTES`

    **Important**

    Please note that if the generated raw bytes are used as a DES key, the 8 parity bits of the key, which are sometimes used for error detection, will contain random values and will not be considered valid by external systems that perform validation of those 8 bits.

  - `SKB_DATA_TYPE_RSA_PRIVATE_KEY`

> **Important**
>
> In SKB, RSA key generation is a convenience feature that must not be used in security-critical environments. Before the generated RSA keys are obtained in the secure format, they are briefly exposed in the memory as plain keys.

  - SKB_DATA_TYPE_DSA_PRIVATE_KEY
  - SKB_DATA_TYPE_ECC_PRIVATE_KEY

- generate_parameters

  Pointer to a structure that specifies the necessary parameters for generating the secure data object.

  For different secure data types, different structures must be provided as follows:

  - For SKB_DATA_TYPE_BYTES, this parameter must point to the SKB_RawBytesParameters structure, which specifies the number of bytes to be generated.
  - For SKB_DATA_TYPE_RSA_PRIVATE_KEY, this parameter must point to the SKB_RsaParameters structure, which specifies the size and public exponent of the key to be generated.
  - For SKB_DATA_TYPE_DSA_PRIVATE_KEY, this parameter must point to the SKB_DsaParameters structure, which specifies the parameters to be used.
  - For SKB_DATA_TYPE_ECC_PRIVATE_KEY, this parameter must point to the SKB_EccParameters structure, which specifies the ECC curve type to be used.

- data

  Address of a pointer to the SKB_SecureData object that will be created by this method. This object will contain the generated data.

## 9.8.9 SKB_Engine_CreateCipher

This method creates a new SKB_Cipher object based on the provided parameters. The SKB_Cipher object is used to encrypt or decrypt data.

The method is declared as follows:

```
SKB_Result
SKB_Engine_CreateCipher(SKB_Engine*           self,
                        SKB_CipherAlgorithm   cipher_algorithm,
                        SKB_CipherDirection   cipher_direction,
                        unsigned int          cipher_flags,
                        const void*           cipher_parameters,
                        const SKB_SecureData* cipher_key,
                        SKB_Cipher**          cipher);
```

The following are the parameters used:

- `self`

  Pointer to the pre-initialized engine.

- `cipher_algorithm`

  Algorithm to be used for encrypting or decrypting data. Available algorithms are defined in the `SKB_CipherAlgorithm` enumeration.

- `cipher_direction`

  Parameter that specifies whether the provided data should be encrypted or decrypted. Available directions are defined in the `SKB_CipherDirection` enumeration.

  Encryption is supported only for the DES, Triple DES, Speck, and AES ciphers.

- `cipher_flags`

  Optional flags for the cipher. Currently, these flags are used only for the AES cipher to specify the AES implementation to be used.

  The following flags are supported:

  - `SKB_CIPHER_FLAG_BALANCED`: This implementation ensures a good balance between performance and security. We recommend trying this implementation first, and switch to the `SKB_CIPHER_FLAG_HIGH_SPEED` implementation only if the performance is not satisfactory.
  - `SKB_CIPHER_FLAG_HIGH_SPEED`: This implementation ensures execution speed that is very close to unprotected AES, but is also the least secure.

- `cipher_parameters`

  Pointer to a structure that provides additional parameters for the cipher, as follows:

  - For the `SKB_CIPHER_ALGORITHM_SPECK_64_128_CTR`, `SKB_CIPHER_ALGORITHM_AES_128_CTR`, `SKB_CIPHER_ALGORITHM_AES_192_CTR`, and `SKB_CIPHER_ALGORITHM_AES_256_CTR` ciphers, it must point to the `SKB_CtrModeCipherParameters` structure, or NULL if the default counter size is to be used (16 for AES, and 8 for Speck).
  - For the `SKB_CIPHER_ALGORITHM_AES_128_GCM`, `SKB_CIPHER_ALGORITHM_AES_192_GCM`, and `SKB_CIPHER_ALGORITHM_AES_256_GCM` ciphers, it must point to the `SKB_GcmCipherParameters` structure.
  - For the `SKB_CIPHER_ALGORITHM_AES_128_CCM`, `SKB_CIPHER_ALGORITHM_AES_192_CCM`, and `SKB_CIPHER_ALGORITHM_AES_256_CCM` ciphers, it must point to the `SKB_AuthenticationParameters` structure.
  - For the `SKB_CIPHER_ALGORITHM_ECC_ELGAMAL` cipher, it must point to the `SKB_EccParameters` structure, which specifies the curve type.
  - For all other ciphers, this parameter must be NULL.

- `cipher_key`

  Pointer to the `SKB_SecureData` object containing the encryption or decryption key.

- `cipher`

  Address of a pointer to the `SKB_Cipher` object that will be created by this method.

## 9.8.10 SKB_Engine_CreateTransform

This method creates a new `SKB_Transform` object based on the provided parameters. The `SKB_Transform` object is used to calculate a digest, sign data, or verify a signature.

The method is declared as follows:

```
SKB_Result
SKB_Engine_CreateTransform(SKB_Engine*       self,
                           SKB_TransformType transform_type,
                           const void*       transform_parameters,
                           SKB_Transform**   transform);
```

The following are the parameters used:

- `self`

  Pointer to the pre-initialized engine.

- `transform_type`

  Transform type to be created. Available transform types are defined in the `SKB_TransformType` enumeration.

- `transform_parameters`

  Pointer to a structure that provides the necessary parameters for the transform. For different transform types, a different structure must be provided, as follows:

  - For the `SKB_TRANSFORM_TYPE_DIGEST` transform, this parameter must point to the `SKB_DigestTransformParameters` structure.
  - For the `SKB_TRANSFORM_TYPE_SIGN` transform, this parameter must point to one of the following structures:
    - If one of the following algorithms is to be used, this parameter must point to the `SKB_SignTransformParametersEx` structure:
      - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX`
      - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX`
      - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX`
      - `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX`

        - SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384_EX
        - SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512_EX
        - SKB_SIGNATURE_ALGORITHM_ECDSA
        - SKB_SIGNATURE_ALGORITHM_ECDSA_MD5
        - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1
        - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA224
        - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256
        - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA384
        - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA512
      - If the SKB_SIGNATURE_ALGORITHM_DSA algorithm is to be used, this parameter must point to the SKB_DsaParameters structure.
      - For all other algorithms, this parameter must point to the SKB_SignTransformParameters structure.
    - For the SKB_TRANSFORM_TYPE_VERIFY transform, this parameter must point to the SKB_VerifyTransformParameters structure.

- transform

  Address of a pointer to the SKB_Transform object that will be created by this method.

## 9.8.11 SKB_Engine_CreateKeyAgreement

This method creates a new SKB_KeyAgreement object based on the provided parameters. The SKB_KeyAgreement object is used to calculate a shared secret based on the key agreement algorithm.

The method is declared as follows:

```
SKB_Result
SKB_Engine_CreateKeyAgreement(SKB_Engine*               self,
                              SKB_KeyAgreementAlgorithm key_agreement_algorithm,
                              const void*               key_agreement_parameters,
                              SKB_KeyAgreement**        key_agreement);
```

The following are the parameters used:

- self

  Pointer to the pre-initialized engine.

- key_agreement_algorithm

  Key agreement algorithm to be used. Available algorithms are defined in the SKB_KeyAgreementAlgorithm enumeration.

- `key_agreement_parameters`

  Pointer to a structure providing the necessary parameters for the particular key agreement algorithm, as follows:

  - For the `SKB_KEY_AGREEMENT_ALGORITHM_ECDH` algorithm, this parameter must point to the `SKB_EccParameters` structure.
  - For the `SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH` algorithm, this parameter must point to the `SKB_PrimeDhParameters` structure.
  - For the `SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC` algorithm, this parameter must point to the `SKB_EcdhParameters` structure.
  - For the `SKB_KEY_AGREEMENT_ALGORITHM_X25519` algorithm, this parameter must be one of the following:
    - NULL: In this case, a random private key will be generated.
    - Pointer to the `SKB_EddhParameters` structure: In this case, a fixed private key specified by the `SKB_EddhParameters` structure will be used.

- `key_agreement`

  Address of a pointer to the `SKB_KeyAgreement` object that will be created by this method.

## 9.8.12 SKB_Engine_UpgradeExportedData

This method upgrades an exported `SKB_SecureData` object to the latest version as described in One-Way Key Upgrading.

The method is declared as follows:

```
SKB_Result
SKB_Engine_UpgradeExportedData(SKB_Engine*     engine,
                               const SKB_Byte* input,
                               SKB_Size        input_size,
                               SKB_Byte*       buffer,
                               SKB_Size*       buffer_size);
```

The following are the parameters used:

- `engine`

  Pointer to the pre-initialized engine.

- `input`

  Input data buffer containing the previously exported `SKB_SecureData` object that needs to be upgraded to the latest export format.

- `input_size`

Size of the `input` buffer in bytes.

- `buffer`

  This parameter is either NULL or a pointer to the memory buffer where the upgraded data is to be written.

  If this parameter is NULL, the method simply returns, in `buffer_size`, the number of bytes that would be sufficient to hold the output, and returns SKB_SUCCESS.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there, sets `buffer_size` to the exact number of bytes stored, and returns SKB_SUCCESS. If the buffer is not large enough, then the method sets `buffer_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

- `buffer_size`

  Pointer to a variable that holds the size of the memory buffer in bytes where the output is to be stored. For more details, see the description of the `buffer` parameter.

## 9.8.13 SKB_SecureData_GetInfo

This method provides information about the size and type of contents stored within a particular `SKB_SecureData` object.

The method is declared as follows:

```
SKB_Result
SKB_SecureData_GetInfo(const SKB_SecureData* self,
                       SKB_DataInfo*         info);
```

The following are the parameters used:

- `self`

  Pointer to the `SKB_SecureData` object whose size and type you want to know.

- `info`

  Pointer to the `SKB_DataInfo` structure, which will be populated by this method to return the characteristics of the `SKB_SecureData` object.

## 9.8.14 SKB_SecureData_Export

This method returns a protected form of contents of a particular `SKB_SecureData` object. This protected data is intended for exporting keys to a persistent storage. Later the exported data can be imported back into SKB using the `SKB_Engine_CreatedDataFromWrapped` method.

The method is declared as follows:

```
```

```
SKB_Result
SKB_SecureData_Export(const SKB_SecureData* self,
                      SKB_ExportTarget      target,
                      const void*           target_parameters,
                      SKB_Byte*             buffer,
                      SKB_Size*             buffer_size);
```

The following are the parameters used:

- `self`

  Pointer to the SKB_SecureData object to be exported.

- `target`

  Export type to be used. Available export types are defined in the SKB_ExportTarget enumeration.

- `target_parameters`

  Currently, this parameter is not used.

- `buffer`

  This parameter is either NULL or a pointer to the memory buffer where the exported data is to be written.

  If this parameter is NULL, the method simply returns, in `buffer_size`, the number of bytes that would be sufficient to hold the exported data, and returns SKB_SUCCESS.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the exported data, the method stores the exported data there, sets `buffer_size` to the exact number of bytes stored, and returns SKB_SUCCESS. If the buffer is not large enough, then the method sets `buffer_size` to the number of bytes that would be sufficient, and returns SKB_ERROR_BUFFER_TOO_SMALL.

- `buffer_size`

  Pointer to a variable that holds the size of the memory buffer in bytes where the exported data is to be stored.

## 9.8.15 SKB_SecureData_Wrap

This method wraps the contents of a particular SKB_SecureData object using a specified cipher and wrapping key as described in Wrapping Keys.

The method is declared as follows:

```
SKB_Result
SKB_SecureData_Wrap(const SKB_SecureData* self,
                    SKB_CipherAlgorithm   wrapping_algorithm,
                    const void*           wrapping_parameters,
```

```
                    const SKB_SecureData* wrapping_key,
                    SKB_Byte*             buffer,
                    SKB_Size*             buffer_size);
```

The following are the parameters used:

- `self`

  Pointer to the SKB_SecureData object whose contents need to be wrapped.

  If you are using the SKB_CIPHER_ALGORITHM_ASC_X9_TR_31 wrapping algorithm, the size of the key encapsulated by the SKB_SecureData object must not exceed 78 bytes.

- `wrapping_algorithm`

  Wrapping algorithm to be used. The available algorithms are defined in the SKB_CipherAlgorithm enumeration.

  The following is a list of algorithms that support wrapping (the supported input data types are listed in the parentheses):

  - SKB_CIPHER_ALGORITHM_AES_128_ECB (raw bytes, private DSA keys)
  - SKB_CIPHER_ALGORITHM_AES_128_CBC (raw bytes, private DSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_128_CTR (raw bytes, private DSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_192_ECB (raw bytes, private DSA keys)
  - SKB_CIPHER_ALGORITHM_AES_192_CBC (raw bytes, private DSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_192_CTR (raw bytes, private DSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_256_ECB (raw bytes, private DSA keys)
  - SKB_CIPHER_ALGORITHM_AES_256_CBC (raw bytes, private DSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_AES_256_CTR (raw bytes, private DSA keys, private ECC keys)
  - SKB_CIPHER_ALGORITHM_RSA (raw bytes)
  - SKB_CIPHER_ALGORITHM_RSA_1_5 (raw bytes)
  - SKB_CIPHER_ALGORITHM_RSA_OAEP (raw bytes)
  - SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA256 (raw bytes)
  - SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB (raw bytes)
  - SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC (raw bytes)
  - SKB_CIPHER_ALGORITHM_NIST_AES (raw bytes)
  - SKB_CIPHER_ALGORITHM_XOR (raw bytes)
  - SKB_CIPHER_ALGORITHM_ASC_X9_TR_31 (raw bytes)

If the SKB_CIPHER_ALGORITHM_AES_128_CTR, SKB_CIPHER_ALGORITHM_AES_192_CTR, or SKB_CIPHER_ALGORITHM_AES_256_CTR algorithm is used, the counter size will be 16 bytes.

If the SKB_CIPHER_ALGORITHM_NIST_AES algorithm is used, in the case of integrity check failure this method will return the SKB_ERROR_INVALID_FORMAT error.

For detailed information on the supported wrapping algorithms and their restrictions, see Supported Algorithms.

- wrapping_parameters

  Pointer to a structure that provides additional parameters for the wrapping algorithm.

  If you are using one of the following algorithms, you may optionally point this parameter to the SKB_BlockCipherWrapParameters structure to provide a specific initialization vector to the wrapping algorithm:

  - SKB_CIPHER_ALGORITHM_AES_128_CBC
  - SKB_CIPHER_ALGORITHM_AES_192_CBC
  - SKB_CIPHER_ALGORITHM_AES_256_CBC
  - SKB_CIPHER_ALGORITHM_AES_128_CTR
  - SKB_CIPHER_ALGORITHM_AES_192_CTR
  - SKB_CIPHER_ALGORITHM_AES_256_CTR
  - SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC

  If the SKB_BlockCipherWrapParameters structure is not provided (wrapping_parameters is NULL), a random initialization vector will be generated.

  If you are using the SKB_CIPHER_ALGORITHM_ASC_X9_TR_31 algorithm, this parameter must be a pointer to the SKB_AscX9Tr31WrapParameters structure, which will provide the key block header.

  For all other cases, set this parameter to NULL.

- wrapping_key

  Pointer to the SKB_SecureData object containing the wrapping key.

  If one of the RSA-based algorithms is used, the data type of this SKB_SecureData object must be SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT.

- buffer

  This parameter is either NULL or a pointer to the memory buffer where the output is to be stored.

  If this parameter is NULL, the method simply returns, in buffer_size, the number of bytes that would be sufficient to hold the output, and returns SKB_SUCCESS.

If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there, sets `buffer_size` to the exact number of bytes stored, and returns SKB_SUCCESS. If the buffer is not large enough, then the method sets `buffer_size` to the number of bytes that would be sufficient, and returns SKB_ERROR_BUFFER_TOO_SMALL.

If you are using the Triple DES or AES algorithm, the buffer will be formatted as described in Wrapped Data Buffer.

If you use the AES algorithm in the CBC mode with XML encryption padding to wrap a private ECC key, the buffer will be formatted as described in AES-Wrapped Private ECC Key.

- `buffer_size`

  Pointer to a variable that holds the size of the memory buffer in bytes where the output data is to be stored. For more details, see the description of the `buffer` parameter.

## 9.8.16 SKB_SecureData_Derive

This method creates a new SKB_SecureData object from another SKB_SecureData object using a particular derivation algorithm.

The method is declared as follows:

```
SKB_Result
SKB_SecureData_Derive(const SKB_SecureData*   self,
                      SKB_DerivationAlgorithm algorithm,
                      const void*             parameters,
                      SKB_SecureData**        data);
```

The following are the parameters used:

- `self`

  Pointer to the SKB_SecureData object from which a new SKB_SecureData object needs to be derived.

- `algorithm`

  Derivation algorithm to be used. The available algorithms are defined in the SKB_DerivationAlgorithm enumeration.

- `parameters`

  Pointer to a structure containing parameters for the derivation algorithm. For different algorithms, a different structure must be provided as follows:

  - If the SKB_DERIVATION_ALGORITHM_SLICE or SKB_DERIVATION_ALGORITHM_BLOCK_SLICE algorithm is used, this parameter must point to the SKB_SliceDerivationParameters structure.
  - If the SKB_DERIVATION_ALGORITHM_SELECT_BYTES algorithm is used, this parameter must point to the SKB_SelectBytesDerivationParameters structure.

- If the SKB_DERIVATION_ALGORITHM_CIPHER algorithm is used, this parameter must point to the SKB_CipherDerivationParameters structure.
- If the SKB_DERIVATION_ALGORITHM_SHA_1 algorithm is used, this parameter may point to the SKB_Sha1DerivationParameters structure, which specifies how many times the SHA-1 algorithm must be executed and how many bytes from the result must be derived. If the parameter is NULL, the SHA-1 algorithm will be executed once and all 20 bytes of the output will be derived as a new SKB_SecureData object.
- If the SKB_DERIVATION_ALGORITHM_SHA_256 or SKB_DERIVATION_ALGORITHM_SHA_512 algorithm is used, this parameter may point to the SKB_Sha2DerivationParameters structure, which provides plain buffers that should be prefixed and suffixed to the SKB_SecureData object processed. If the parameter is NULL, SKB will assume that there are no plain data buffers to be prefixed or suffixed.
- If the SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128, SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT, or SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_SPECK_L16BITLE algorithm is used, this parameter must point to the SKB_Nist800108KdfDerivationParameters structure.
- If the SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2 algorithm is used, this parameter must point to the SKB_OmaDrmKdf2DerivationParameters structure.
- If the SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE algorithm is used, this parameter may point to the SKB_RawBytesFromEccPrivateDerivationParameters structure, which specifies whether the output should be little-endian or big-endian. If the parameter is NULL, the output will be encoded as little-endian.
- If the SKB_DERIVATION_ALGORITHM_SHA_AES algorithm is used, this parameter must point to the SKB_ShaAesDerivationParameters structure.
- If the SKB_DERIVATION_ALGORITHM_HMAC_SHA256, SKB_DERIVATION_ALGORITHM_HMAC_SHA384, SKB_DERIVATION_ALGORITHM_HMAC_SHA512, SKB_DERIVATION_ALGORITHM_BLOCK_CONCATENATE, SKB_DERIVATION_ALGORITHM_XOR, or SKB_DERIVATION_ALGORITHM_DOUBLE_AES_ENCRYPT algorithm is used, this parameter must point to the SKB_GenericDerivationParameters structure.
- If the SKB_DERIVATION_ALGORITHM_HKDF_SHA256 algorithm is used, this parameter must point to the SKB_HkdfDerivationParameters structure.
- For all other key derivation algorithms, this parameter is not used and therefore should be NULL.

- `data`

  Address of a pointer that will point to the new derived SKB_SecureData object when this method is executed.

## 9.8.17 SKB_SecureData_GetPublicKey

This method derives a public key from the supplied private key.

The method is declared as follows:

```
SKB_Result
SKB_SecureData_GetPublicKey(const SKB_SecureData* self,
                            SKB_DataFormat        format,
                            const void*           parameters,
                            SKB_Byte*             output,
                            SKB_Size*             output_size);
```

The following are the parameters used:

- `self`

  Pointer to the SKB_SecureData object containing the private key. From this key, the public key will be derived.

- `format`

  Format in which the derived public key must be stored in the returned buffer of bytes. The available formats are defined in the SKB_DataFormat enumeration.

  For this method, only the following values are valid:

  - SKB_DATA_FORMAT_RAW (public DSA or ED25519 key)
  - SKB_DATA_FORMAT_PKCS1 (public RSA key)
  - SKB_DATA_FORMAT_ECC_BINARY (deprecated)
  - SKB_DATA_FORMAT_ECC_PUBLIC
  - SKB_DATA_FORMAT_ECC_PUBLIC_POINT
  - SKB_DATA_FORMAT_ECC_PUBLIC_X9_62

  **Important**

  The SKB_DATA_FORMAT_ECC_PUBLIC_X9_62 format is not supported if the private key is constructed using the SKB_ECC_CURVE_CUSTOM curve type of the SKB_EccCurve enumeration.

- `parameters`

  Pointer to a structure containing parameters necessary for deriving the public key.

  For different data formats, different structures must be provided as follows:

  - For SKB_DATA_FORMAT_RAW, this parameter must point to the SKB_DsaParameters structure, which specifies the parameters for a DSA public key. For an ED25519 public key this parameter must be NULL.
  - For SKB_DATA_FORMAT_PKCS1, this parameter may point to the SKB_RsaParameters structure, which specifies the public exponent. For RSA private keys whose public exponent is not known, NULL can be passed instead, in which case the public exponent will be calculated anew, if necessary.

  - For all the public ECC key related formats, this parameter must point to the SKB_EccParameters structure, which specifies the ECC curve type.

- `output`

  This parameter is either NULL or a pointer to the memory buffer where the output is to be stored.

  If this parameter is NULL, the method simply returns, in `output_size`, the number of bytes that would be sufficient to hold the output, and returns SKB_SUCCESS.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there, sets `output_size` to the exact number of bytes stored, and returns SKB_SUCCESS. If the buffer is not large enough, then the method sets `output_size` to the number of bytes that would be sufficient, and returns SKB_ERROR_BUFFER_TOO_SMALL.

  After successfully executing the method, the content of the `output` parameter will be a pointer to a buffer of bytes containing the public key, formatted according to the type specified in the `format` parameter. For details on how a particular type is formatted, see SKB_DataFormat.

- `output_size`

  Pointer to a variable that holds the size of the memory buffer in bytes where the public key is to be stored. For more details, see the description of the `output` parameter.

## 9.8.18 SKB_SecureData_Release

This method releases the specified SKB_SecureData object from memory. It must always be called when the object is no longer needed.

The method is declared as follows:

```
SKB_Result
SKB_SecureData_Release(SKB_SecureData* self);
```

The parameter `self` is a pointer to the SKB_SecureData object that must be released.

## 9.8.19 SKB_Cipher_ProcessBuffer

This method performs either data encryption or decryption depending on the parameters of the previously created SKB_Cipher object.

The method is declared as follows:

```
SKB_Result
SKB_Cipher_ProcessBuffer(SKB_Cipher*     self,
                         const SKB_Byte* in_buffer,
                         SKB_Size        in_buffer_size,
                         SKB_Byte*       out_buffer,
                         SKB_Size*       out_buffer_size,
```

```
                        const SKB_Byte* iv,
                        SKB_Size        iv_size);
```

The following are the parameters used:

- `self`

  Pointer to the previously created `SKB_Cipher` object, which contains all the necessary parameters.

- `in_buffer`

  Pointer to a buffer of data to be encrypted or decrypted. The following special notes apply to this parameter:

  - For block ciphers, this parameter must point to the beginning of a cipher block.
  - For the Speck cipher, input blocks are treated as two 4-byte integers in the little-endian format.
  - For the ElGamal ECC cipher, this parameter must point to a buffer of bytes described in Input Buffer for the ElGamal ECC Cipher.

- `in_buffer_size`

  Size in bytes of the data buffer to be encrypted or decrypted. The following special notes apply to this parameter:

  - For the DES and Triple DES cipher, this parameter must be a multiple of 8.
  - For the Speck cipher in the ECB or CBC mode, this parameter must be a multiple of 8.
  - For the AES cipher in the ECB or CBC mode, this parameter must be a multiple of 16.
  - For the RSA cipher, this parameter must be the size of the entire encrypted message, but not more than the length of the RSA key.

- `out_buffer`

  This parameter is either NULL or a pointer to the memory buffer where the output is to be stored.

  If this parameter is NULL, the method simply returns, in `out_buffer_size`, the number of bytes that would be sufficient to hold the output, and returns `SKB_SUCCESS`.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there, sets `out_buffer_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `out_buffer_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

  For the ElGamal ECC cipher, the output buffer will contain the X coordinate of the decrypted point in the big-endian format. It is the caller's responsibility to extract the decrypted message from this output according to the way the message was encrypted.

SKB supports in-place encryption and decryption, which means that the `out_buffer` and `in_buffer` parameters may point to the same memory location. Then, the output of this method will overwrite the input.

- `out_buffer_size`

  Pointer to a variable that holds the size of the memory buffer in bytes where the output data is to be stored. For more details, see the description of the `out_buffer` parameter.

- `iv`

  Pointer to the initialization vector, which must be provided only for the following ciphers:

  ◦ DES or Triple DES cipher in the CBC mode

  ◦ AES and Speck ciphers in the CBC or CTR mode

  The Speck cipher will interpret the passed initialization vector as two integers encoded using the little-endian byte order.

  ◦ AES cipher in the CCM mode, in which case the `iv` parameter must contain the value of the nonce

  For other cases, this parameter should contain NULL.

  For ciphers operating in the CBC and CTR modes, the initialization vector must be provided in the first call of this method. In subsequent calls, you may set the `iv` parameter to NULL, in which case, SKB will interpret the provided input buffer as continuation of the same message and will use the initialization vector that is internally preserved from the last method call (this approach is useful for processing very large data buffers that may not fit in the memory). In other words, if you provide the initialization vector, SKB will interpret the input buffer as a new message.

- `iv_size`

  Size in bytes of the `iv` parameter. It should be 0 if the `iv` parameter is NULL.

## 9.8.20 SKB_Cipher_ProcessAad

The purpose of this method is to provide the additional authenticated data to the `SKB_Cipher` object when using AES in the GCM or CCM mode. For information on the general procedure to be followed in this case, see Using Authenticated Encryption with Additional Data.

You may choose not to call this method at all, in which case the additional authenticated data will be assumed to be an empty string.

When this method is called, the following happens, depending on the mode in which the cipher is operating:

- In the GCM mode, the supplied input buffer is appended to the additional authenticated data that is processed by the cipher object.

- In the CCM mode, the supplied input buffer represents the whole additional authenticated data and replaces any previously supplied additional authenticated data.

**Important**

In the GCM mode, this method can only be called until the `SKB_Cipher_ProcessBuffer` or `SKB_Cipher_ProcessFinal` method is invoked for that cipher object. After that, invocation of the `SKB_Cipher_ProcessAad` method will return the `SKB_ERROR_INVALID_STATE` value.

This method is declared as follows:

```
SKB_Result
SKB_Cipher_ProcessAad(SKB_Cipher*     self,
                      const SKB_Byte* in_buffer,
                      SKB_Size        in_buffer_size);
```

The following are the parameters used:

- `self`

  Pointer to the previously created `SKB_Cipher` object.

- `in_buffer`

  Pointer to the data buffer containing additional authenticated data.

- `in_buffer_size`

  Size of `in_buffer` in bytes.

## 9.8.21 SKB_Cipher_ProcessFinal

This method is only used when performing AES encryption and decryption in the GCM mode. Its purpose is to either produce the authentication tag (in case of encryption), or verify the supplied authentication tag (in case of decryption). The method also sets the state of the `SKB_Cipher` object to finished, after which no further invocation of the `SKB_Cipher_ProcessAad` or `SKB_Cipher_ProcessBuffer` method with the same `SKB_Cipher` object can be performed. For information on the general procedure to be followed in this case, see Using Authenticated Encryption with Additional Data.

This method is declared as follows:

```
SKB_Result
SKB_Cipher_ProcessFinal(SKB_Cipher* self,
                        void*       parameters);
```

The following are the parameters used:

- `self`

Pointer to the previously created SKB_Cipher object.

- parameters

  Pointer to the SKB_AuthenticationParameters structure, which is used as follows:

  - In case of encryption, SKB will set the value of the authentication_tag parameter of the SKB_AuthenticationParameters structure to point to a data buffer containing the authentication tag calculated during encryption.
  - In case of decryption, you must set the value of the authentication_tag parameter of the SKB_AuthenticationParameters structure, which will be read and compared to the internally stored authentication tag calculated during decryption. If the tags do not match, the SKB_ERROR_AUTHENTICATION_FAILURE error will be returned.

## 9.8.22 SKB_Cipher_Release

This method releases an SKB_Cipher object from memory. It must always be called when the object is no longer needed.

The method is declared as follows:

```
SKB_Result
SKB_Cipher_Release(SKB_Cipher* self);
```

The parameter self is a pointer to the SKB_Cipher object to be released.

## 9.8.23 SKB_Transform_AddBytes

This method appends a plain buffer of bytes to a previously created SKB_Transform object. Data must be added to an SKB_Transform object before the actual transform algorithm (digest, signing, or verifying) is executed.

The method is declared as follows:

```
SKB_Result
SKB_Transform_AddBytes(SKB_Transform*  self,
                       const SKB_Byte* data,
                       SKB_Size        data_size);
```

The following are the parameters used:

- self

  Pointer to the previously created SKB_Transform object.

- data

  Pointer to the buffer of data to be appended to the SKB_Transform object.

If the SKB_SIGNATURE_ALGORITHM_RSA algorithm is selected, the input data must be properly formatted to be compatible with the standard RSA signing with PKCS#1 version 1.5 padding. Namely, the data must be formatted like the value T described in section 9.2 of RFC 8017.

- data_size

  Size of the data buffer in bytes.

  If data is used as input for a signature algorithm without a hash function, the buffer must contain an unencrypted message with the following size restrictions:

  - If the SKB_SIGNATURE_ALGORITHM_RSA algorithm is selected, SKB will automatically append padding bytes to data according to PKCS#1. This means that data_size must be equal or less than the private key size minus 11 bytes, which is the minimal possible padding size.
  - If the SKB_SIGNATURE_ALGORITHM_DSA algorithm is selected, data_size must be less than the size of the N parameter.
  - If the SKB_SIGNATURE_ALGORITHM_ECDSA algorithm is selected, data_size must be equal or less than 64.

## 9.8.24 SKB_Transform_AddSecureData

This method appends content of an SKB_SecureData object to a previously created SKB_Transform object. Data must be added to an SKB_Transform object before the actual transform algorithm (digest, signing, or verifying) is executed.

**Important**

This method cannot be used for the SKB_SIGNATURE_ALGORITHM_RSA, SKB_SIGNATURE_ALGORITHM_DSA, and SKB_SIGNATURE_ALGORITHM_ECDSA signing algorithms because they can operate only on plain input.

If you are using the ECDSA or RSA signing algorithms, this method is available only if the corresponding digest algorithm is also included in SKB. For instance, if you want to calculate a signature of a cryptographic key using the ECDSA signing algorithm with SHA-384 as the hash function, the SHA-384 digest algorithm must also be included in SKB.

Currently, this method does not support the ISO/IEC 9797-1 MAC algorithm 3 for DES (also known as Retail MAC) and the Speck-CMAC algorithm.

The method is declared as follows:

```
SKB_Result
SKB_Transform_AddSecureData(SKB_Transform*        self,
                            const SKB_SecureData* data);
```

The following are the parameters used:

- `self`

  Pointer to the previously created `SKB_Transform` object.

- `data`

  Pointer to the `SKB_SecureData` object whose content must be appended to the `SKB_Transform` object.

## 9.8.25 SKB_Transform_GetOutput

This method executes a transform algorithm using a particular `SKB_Transform` object. The transform algorithm is specified during the creation of the `SKB_Transform` object, and the input data is provided using the `SKB_Transform_AddBytes` and `SKB_Transform_AddSecureData` methods.

**Important**

After this method is called, executing the `SKB_Transform_AddBytes` or `SKB_Transform_AddSecureData` method on the same `SKB_Transform` object will produce an error. Similarly, calling the `SKB_Transform_GetOutput` method again will also result in an error.

The method is declared as follows:

```
SKB_Result
SKB_Transform_GetOutput(SKB_Transform* self,
                        SKB_Byte*      output,
                        SKB_Size*      output_size);
```

The following are the parameters used:

- `self`

  Pointer to the `SKB_Transform` object on which the transform algorithm must be executed.

- `output`

  This parameter is either NULL or a pointer to the memory buffer where the transform output will be stored.

  If this parameter is NULL, the method simply returns, in `output_size`, the number of bytes that would be sufficient to hold the output, and returns `SKB_SUCCESS`.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there, sets `output_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `output_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

  In the case of the `SKB_TRANSFORM_TYPE_VERIFY` transform, the output will be a single byte with the value 1 if the signature is valid, and 0 if it is not.

In the case of the ECDSA signature algorithm, the output will be a pointer to a buffer formatted as described in ECDSA Output.

- `output_size`

  Pointer to a variable that holds the size of the memory buffer in bytes where the transform output data is to be stored. For more details, see the description of the `output` parameter.

## 9.8.26 SKB_Transform_Release

This method releases the specified `SKB_Transform` object from memory.

The method is declared as follows:

```
SKB_Result
SKB_Transform_Release(SKB_Transform* self);
```

The parameter `self` is a pointer to the `SKB_Transform` object to be released.

## 9.8.27 SKB_KeyAgreement_GetPublicKey

This method creates a new public key that should be sent to the other party of the key agreement algorithm.

The method is declared as follows:

```
SKB_Result
SKB_KeyAgreement_GetPublicKey(SKB_KeyAgreement* self,
                              SKB_Byte*         public_key_buffer,
                              SKB_Size*         public_key_buffer_size);
```

The following are the parameters used:

- `self`

  Pointer to the previously created `SKB_KeyAgreement` object, which contains all the necessary parameters.

- `public_key_buffer`

  This parameter is either NULL or a pointer to the memory buffer where the public key will be stored.

  If this parameter is NULL, the method simply returns, in `public_key_buffer_size`, the number of bytes that would be sufficient to hold the output, and returns `SKB_SUCCESS`.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there, sets `public_key_buffer_size` to the exact number of bytes stored, and returns `SKB_SUCCESS`. If the buffer is not large enough, then the method sets `public_key_buffer_size` to the number of bytes that would be sufficient, and returns `SKB_ERROR_BUFFER_TOO_SMALL`.

For the SKB_KEY_AGREEMENT_ALGORITHM_ECDH and SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC algorithms, the public key is stored using the format described in Public ECC Key.

For the SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH algorithm, the buffer size is 128 bytes, and it stores the public value in the big-endian format.

- public_key_buffer_size

  Pointer to a variable that holds the size of the memory buffer in bytes where the public key is to be stored. For more details, see the description of the public_key_buffer parameter.

## 9.8.28 SKB_KeyAgreement_ComputeSecret

This method takes the public key received from the other party of the key agreement algorithm and computes the shared secret.

The method is declared as follows:

```
SKB_Result
SKB_KeyAgreement_ComputeSecret(SKB_KeyAgreement* self,
                               const SKB_Byte*   peer_public_key,
                               SKB_Size          peer_public_key_size,
                               SKB_Size          secret_size,
                               SKB_SecureData**  secret);
```

The following are the parameters used:

- self

  Pointer to the previously created SKB_KeyAgreement object, which contains all the necessary parameters.

- peer_public_key

  Pointer to the memory buffer where the public key received from the other party is stored.

  For the SKB_KEY_AGREEMENT_ALGORITHM_ECDH and SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC algorithms, the public key is expected to be stored using the format described in Public ECC Key.

  For the SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH algorithm, the buffer has to be 128 bytes long, and it must store the public value in the big-endian format.

- peer_public_key_size

  Size of the peer_public_key parameter in bytes. This size must be equal to the value returned by the SKB_KeyAgreement_GetPublicKey method used by the other key agreement party.

- secret_size

  Size of the desired shared secret data output.

  To select the largest possible shared secret size, pass the value SKB_KEY_AGREEMENT_MAXIMAL_SECRET_SIZE as an input for this parameter.

- `secret`

  Address of a pointer to the SKB_SecureData object containing the shared secret data that will be created by this method. The bytes will be ordered using the big-endian format.

### 9.8.29 SKB_KeyAgreement_Release

This method releases the specified SKB_KeyAgreement object from memory. It must always be called when the object is no longer needed.

The method is declared as follows:

```
SKB_Result
SKB_KeyAgreement_Release(SKB_KeyAgreement* self);
```

The parameter `self` is a pointer to the SKB_KeyAgreement object to be released.

## 9.9 Supporting Structures

This section describes various supporting structures used by the Native API.

### 9.9.1 SKB_EngineInfo

This structure is populated by the SKB_Engine_GetInfo method to provide information about a particular SKB_Engine instance.

> **Important**
>
> The content of a populated SKB_EngineInfo structure will not be valid after the corresponding SKB_Engine object is released from memory. During examination of the SKB_EngineInfo object, the SKB_Engine object must exist.

The SKB_EngineInfo structure is declared as follows:

```
typedef struct {
    struct {
        unsigned int major;
        unsigned int minor;
        unsigned int revision;
    } api_version;
    unsigned int        flags;
    unsigned int        property_count;
    SKB_EngineProperty* properties;
} SKB_EngineInfo;
```

The following are the properties used:

- `major`, `minor`, and `revision`

  Version numbers specified in the API header file.

- `flags`

  Configuration flags currently set in SKB. For information on the available flags and how to set them, see `SKB_SetFlags`.

- `property_count`

  Number of elements in the `properties` array.

- `properties`

  Array of engine properties where each property is an `SKB_EngineProperty` structure.

  The following properties are used:

  - `key_cache_max_items`: Maximum number of keys that can be cached in the memory.
  - `diversification_guid`: Unique diversification identifier consisting of 16 bytes in the hexadecimal format. SKB libraries with the same binary footprint will have the same identifier.
  - `export_guid`: Export key identifier consisting of 16 bytes in the hexadecimal format. SKB libraries with the same export key will have the same identifier. For information on the export key, see Persistent Exporting.
  - `export_key_version`: Current export key version in the one-way data upgrade scheme.

## 9.9.2 SKB_EngineProperty

This structure is a name-value pair representing a particular SKB_Engine property in the `SKB_EngineInfo` structure.

The SKB_EngineProperty structure is declared as follows:

```
typedef struct {
    const char* name;
    const char* value;
} SKB_EngineProperty;
```

## 9.9.3 SKB_DataInfo

This structure is used by the `SKB_SecureData_GetInfo` method to return the size and type of a particular SKB_SecureData object.

The structure is declared as follows:

```
```

```
typedef struct {
    SKB_DataType type;
    SKB_Size     size;
} SKB_DataInfo;
```

The following are the properties used:

- `type`

  Type of the data stored within the `SKB_SecureData` object. Available types are defined in the `SKB_DataType` enumeration.

- `size`

  Size of the `SKB_SecureData` object content in bytes.

  For the data type `SKB_DATA_TYPE_RSA_PRIVATE_KEY`, this value is the size of the modulus in bytes.

  Value 0 means that this information is not available.

## 9.9.4 SKB_AscX9Tr31WrapParameters

This structure specifies the key block header that must be provided to the `SKB_SecureData_Wrap` method in case the `SKB_CIPHER_ALGORITHM_ASC_X9_TR_31` wrapping algorithm is used.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* header;
    SKB_Size header_size;
} SKB_AscX9Tr31WrapParameters;
```

The following are the properties used:

- `header`

  Pointer to a buffer containing the key block header.

  The user is responsible for preparing the key header in the format described in section "A.2. Key Block Header (KBH)" of ASC X9 TR 31-2018. SKB only verifies if the format is as expected and extracts the needed information from the header data (the algorithm to use and length of the encrypted data), but it does not change the data. In particular, the key block length must be set properly, and the padding block must be added as the last optional block, if needed.

- `header_size`

  Size of the `header` buffer in bytes.

## 9.9.5 SKB_CtrModeCipherParameters

This structure provides an additional parameter for the SKB_Engine_CreateCipher method when the SKB_CIPHER_ALGORITHM_SPECK_64_128_CTR, SKB_CIPHER_ALGORITHM_AES_128_CTR, SKB_CIPHER_ALGORITHM_AES_192_CTR, or SKB_CIPHER_ALGORITHM_AES_256_CTR algorithms are used.

The structure is declared as follows:

```
typedef struct {
    SKB_Size counter_size;
} SKB_CtrModeCipherParameters;
```

The property counter_size specifies the counter size in bytes, which can be any value from 1 up to the block size.

## 9.9.6 SKB_DigestTransformParameters

This structure is used by the SKB_Engine_CreateTransform method if the SKB_TRANSFORM_TYPE_DIGEST transform is used. The purpose of this structure is to specify the digest algorithm.

The structure is declared as follows:

```
typedef struct {
    SKB_DigestAlgorithm algorithm;
} SKB_DigestTransformParameters;
```

The property algorithm specifies the digest algorithm to be used. The available algorithms are defined in the SKB_DigestAlgorithm enumeration.

## 9.9.7 SKB_SignTransformParameters

This structure is used by the SKB_Engine_CreateTransform method if the SKB_TRANSFORM_TYPE_SIGN transform is used. The purpose of this structure is to specify the signing algorithm and the signing key.

The structure is declared as follows:

```
typedef struct {
    SKB_SignatureAlgorithm algorithm;
    const SKB_SecureData*  key;
} SKB_SignTransformParameters;
```

The following are the properties used:

- algorithm

  Signing algorithm to be used. The available signing algorithms are defined in the SKB_SignatureAlgorithm enumeration.

- key

  Pointer to the SKB_SecureData object containing the signing key.

## 9.9.8 SKB_SignTransformParametersEx

This structure is an extension to the SKB_SignTransformParameters structure. It provides an additional ability to specify the ECC curve type in case the ECDSA signature algorithm is used, or salt and salt length in case the RSA signature algorithm based on the Probabilistic Signature Scheme is used.

The structure is declared as follows:

```
typedef struct {
    SKB_SignTransformParameters base;
    const void*                 extension;
} SKB_SignTransformParametersEx;
```

The following are the properties used:

- base

  SKB_SignTransformParameters structure that specifies the signature algorithm and the key to be used.

- extension

  If one of the following signature algorithms is used, this pointer must point to the SKB_EccParameters structure, which specifies the ECC curve type to be used:

  - SKB_SIGNATURE_ALGORITHM_ECDSA
  - SKB_SIGNATURE_ALGORITHM_ECDSA_MD5
  - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1
  - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA224
  - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256
  - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA384
  - SKB_SIGNATURE_ALGORITHM_ECDSA_SHA512

  If one of the following algorithms is used, this pointer must point to the SKB_RsaPssParameters structure, which specifies the salt and salt length:

  - SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX
  - SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX
  - SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX
  - SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX
  - SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384_EX

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512_EX

## 9.9.9 SKB_VerifyTransformParameters

This structure is used by the SKB_Engine_CreateTransform method if the SKB_TRANSFORM_TYPE_VERIFY transform is used. The purpose of this structure is to specify the verification algorithm, verification key, and the signature.

The structure is declared as follows:

```
typedef struct {
    SKB_SignatureAlgorithm algorithm;
    const SKB_SecureData*  key;
    const SKB_Byte*        signature;
    SKB_Size               signature_size;
} SKB_VerifyTransformParameters;
```

The following are the properties used:

- algorithm

  Verification algorithm to be used. The available verification algorithms are defined in the SKB_SignatureAlgorithm enumeration. Only the following algorithms are supported for verification:

  - SKB_SIGNATURE_ALGORITHM_AES_128_CMAC
  - SKB_SIGNATURE_ALGORITHM_AES_192_CMAC
  - SKB_SIGNATURE_ALGORITHM_AES_256_CMAC
  - SKB_SIGNATURE_ALGORITHM_SPECK_64_128_CMAC
  - SKB_SIGNATURE_ALGORITHM_HMAC_SHA1
  - SKB_SIGNATURE_ALGORITHM_HMAC_SHA224
  - SKB_SIGNATURE_ALGORITHM_HMAC_SHA256
  - SKB_SIGNATURE_ALGORITHM_HMAC_SHA384
  - SKB_SIGNATURE_ALGORITHM_HMAC_SHA512
  - SKB_SIGNATURE_ALGORITHM_HMAC_MD5
  - SKB_SIGNATURE_ALGORITHM_DES_RETAIL_MAC

- key

  Pointer to the SKB_SecureData object containing the verification key.

- signature

  Pointer to the data buffer containing the signature to be verified.

- `signature_size`

  Size of the signature buffer in bytes.

## 9.9.10 SKB_SliceDerivationParameters

This structure is used by the SKB_SecureData_Derive method if the SKB_DERIVATION_ALGORITHM_SLICE or SKB_DERIVATION_ALGORITHM_BLOCK_SLICE derivation algorithm is used. The purpose of this structure is to specify the range of bytes (first byte and the number of bytes) that must be derived from one SKB_SecureData object as another SKB_SecureData object.

The structure is declared as follows:

```
typedef struct {
    unsigned int first;
    unsigned int size;
} SKB_SliceDerivationParameters;
```

The following are the properties used:

- `first`

  Index of the first byte of the source SKB_SecureData object where the derived range starts. Bytes are numbered starting with 0.

  If you are using the SKB_DERIVATION_ALGORITHM_BLOCK_SLICE algorithm, the value must be a multiple of 8 or 16 depending on which block slicing algorithm variation is enabled in the SKB package.

- `size`

  Number of bytes to derive starting with the byte specified by the `first` parameter.

  If you are using the SKB_DERIVATION_ALGORITHM_BLOCK_SLICE algorithm, the value must be a multiple of 8 or 16 depending on which block slicing algorithm variation is enabled in the SKB package.

## 9.9.11 SKB_SelectBytesDerivationParameters

This structure is used by the SKB_SecureData_Derive method if the SKB_DERIVATION_ALGORITHM_SELECT_BYTES algorithm is used. It specifies whether odd or even bytes are to be copied from the input, and how many bytes must be copied.

The structure is declared as follows:

```
typedef struct {
    SKB_SelectBytesDerivationVariant variant;
    unsigned int                     output_size;
} SKB_SelectBytesDerivationParameters;
```

The following are the properties used:

- `variant`

  Reference to a value of the SKB_SelectBytesDerivationVariant enumeration, which tells whether odd or even bytes must be selected.

- `output_size`

  Size of the output in bytes, which is the number of bytes copied from the input.

## 9.9.12 SKB_CipherDerivationParameters

This structure is used by the SKB_SecureData_Derive method if the SKB_DERIVATION_ALGORITHM_CIPHER algorithm is used. The purpose of this structure is to specify all the necessary parameters to execute the derivation.

The structure is declared as follows:

```
typedef struct {
    SKB_CipherAlgorithm   cipher_algorithm;
    SKB_CipherDirection   cipher_direction;
    unsigned int          cipher_flags;
    const void*           cipher_parameters;
    const SKB_SecureData* cipher_key;
    const SKB_Byte*       iv;
    SKB_Size              iv_size;
} SKB_CipherDerivationParameters;
```

The following are the properties used:

- `cipher_algorithm`

  Cipher algorithm to be executed on the input data. This is a reference to the SKB_CipherAlgorithm enumeration.

  Currently, the SKB_DERIVATION_ALGORITHM_CIPHER algorithm supports only the following ciphers:

  - SKB_CIPHER_ALGORITHM_AES_128_ECB
  - SKB_CIPHER_ALGORITHM_AES_128_CBC
  - SKB_CIPHER_ALGORITHM_AES_192_ECB
  - SKB_CIPHER_ALGORITHM_AES_192_CBC
  - SKB_CIPHER_ALGORITHM_AES_256_ECB
  - SKB_CIPHER_ALGORITHM_AES_256_CBC
  - SKB_CIPHER_ALGORITHM_DES_ECB
  - SKB_CIPHER_ALGORITHM_DES_CBC

- SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB
- SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC

- `cipher_direction`

  Parameter that specifies whether the input data must be encrypted or decrypted. Available directions are defined in the SKB_CipherDirection enumeration.

- `cipher_flags`

  Optional flags for the cipher. Can be set only for the AES cipher when it is intended to be used with high throughput, for example media content decryption.

- `cipher_parameters`

  Pointer to a structure that provides additional parameters for the cipher.

  Currently, this parameter must always be NULL.

- `cipher_key`

  Pointer to the SKB_SecureData object containing the encryption or decryption key.

- `iv`

  Pointer to the initialization vector if you use the AES or DES cipher in the CBC mode. For other cases, this parameter should contain NULL.

- `iv_size`

  Initialization vector size in bytes.

## 9.9.13 SKB_Sha1DerivationParameters

This structure is used by the SKB_SecureData_Derive method if the SKB_DERIVATION_ALGORITHM_SHA_1 algorithm is used. For information on this algorithm, see Iterated SHA-1 Derivation. The purpose of this structure is to specify how many times the SHA-1 algorithm must be executed on the selected SKB_SecureData object and how many bytes must be derived from the final digest.

The structure is declared as follows:

```
typedef struct {
    unsigned int round_count;
    unsigned int output_size;
} SKB_Sha1DerivationParameters;
```

The following are the properties used:

- `round_count`

  How many times the SHA-1 algorithm must be executed in a sequence.

0 is also a valid value. In this case, the SHA-1 value will not be calculated; the derived SKB_SecureData object will simply contain the first output_size bytes of the source SKB_SecureData object.

- output_size

  Number of bytes to be derived from the final output of the SHA-1 algorithm. For example, if output_size is 4, the first four bytes of the hash value will be derived as a new SKB_SecureData object.

  The standard size of the SHA-1 output is 20 bytes. Hence, output_size must not exceed 20.

## 9.9.14 SKB_Sha2DerivationParameters

This structure is used by the SKB_SecureData_Derive method if the SKB_DERIVATION_ALGORITHM_SHA_256 or SKB_DERIVATION_ALGORITHM_SHA_512 algorithm is used. For information on these algorithms, see SHA Derivation with Plain Prefix and Suffix. The purpose of this structure is to provide the two plain data buffers that must be prefixed and suffixed to the source SKB_SecureData object before the SHA-256 or SHA-512 algorithm is executed.

This structure may be omitted (provided as NULL), in which case SKB will assume that there are no plain data buffers prefixed or suffixed to the source SKB_SecureData object.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* plain1;
    SKB_Size        plain1_size;
    const SKB_Byte* plain2;
    SKB_Size        plain2_size;
} SKB_Sha2DerivationParameters;
```

The following are the properties used:

- plain1

  Pointer to a buffer of bytes that must be prefixed to the source SKB_SecureData object before calculating the hash value.

  This property can be NULL, in which case there will be no plain data prefixed to the SKB_SecureData object.

- plain1_size

  Number of bytes in the plain1 buffer.

- plain2

  Pointer to a buffer of bytes that must be suffixed to the source SKB_SecureData object before calculating the hash value.

This property can be NULL, in which case there will be no plain data suffixed to the SKB_SecureData object.

- `plain2_size`

  Number of bytes in the `plain2` buffer.

## 9.9.15 SKB_Nist800108KdfDerivationParameters

This structure is used by the SKB_SecureData_Derive method if the SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128, SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT, or SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_SPECK_L16BITLE derivation algorithm is used. For more information on this derivation algorithm, see Using the NIST 800-108 Key Derivation Function.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* label;
    SKB_Size        label_size;
    const SKB_Byte* context;
    SKB_Size        context_size;
    SKB_Size        output_size;
} SKB_Nist800108KdfDerivationParameters;
```

The following are the properties used:

- `label`

  Pointer to the label, a binary buffer that identifies the purpose for the derived key, as defined by NIST 800-108.

- `label_size`

  Size of the label in bytes.

- `context`

  Pointer to the context, a binary buffer containing the information related to the derived key, as defined by NIST 800-108.

- `context_size`

  Size of the context in bytes.

- `output_size`

  Size of the derivation output in bytes.

It must not exceed 256 times the size of the block, and must be a multiple of the block size. This means that for AES-CMAC, it must not exceed 4096 bytes and must be a multiple of 16, and for Speck-CMAC it must not exceed 2048 bytes and must be a multiple of 8.

## 9.9.16 SKB_OmaDrmKdf2DerivationParameters

This structure is used by the SKB_SecureData_Derive method if the SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2 derivation algorithm is used. For more information on this derivation algorithm, see Using KDF2 of the RSAES-KEM-KWS Scheme Defined in the OMA DRM Specification.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* label;
    SKB_Size        label_size;
    SKB_Size        output_size;
} SKB_OmaDrmKdf2DerivationParameters;
```

The following are the properties used:

- label

  Pointer to the buffer containing the otherInfo parameter as defined in the OMA DRM specification.

- label_size

  Size of the label buffer in bytes.

- output_size

  Size of the derivation output in bytes.

## 9.9.17 SKB_RawBytesFromEccPrivateDerivationParameters

This structure may be used by the SKB_SecureData_Derive method to specify the endianness of the output if the SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE derivation algorithm is used. For more information on this derivation algorithm, see Deriving a Key as Raw Bytes from a Private ECC Key.

The structure is declared as follows:

```
typedef struct {
    unsigned int derivation_flags;
} SKB_RawBytesFromEccPrivateDerivationParameters;
```

If derivation_flags includes the SKB_DERIVATION_FLAG_OUTPUT_IN_BIG_ENDIAN flag, the output will be encoded using the big-endian format. Otherwise, the output will be little-endian.

## 9.9.18 SKB_ShaAesDerivationParameters

This structure is used by the SKB_SecureData_Derive method if the SKB_DERIVATION_ALGORITHM_SHA_AES derivation algorithm is used. For more information on this derivation algorithm, see Deriving a Key By Encrypting Data Using 128-bit AES With a Concatenated Key.

The structure is declared as follows:

```
typedef struct {
    const SKB_SecureData* secure_p;
    const SKB_Byte*       plain_1;
    SKB_Size              plain_1_size;
    const SKB_Byte*       plain_2;
} SKB_ShaAesDerivationParameters;
```

The following are the properties used:

- secure_p

  Pointer to the SKB_SecureData object containing the "secure_p" value.

- plain_1

  Pointer to the "plain_1" buffer.

  This property may be set to NULL. In that case, the simplified version of the derivation algorithm will be executed.

- plain_1_size

  Size of the "plain_1" buffer in bytes. It must be 0 if plain_1 is set to NULL.

- plain_2

  Pointer to the "plain_2" buffer, which must be 16 bytes long.

## 9.9.19 SKB_GenericDerivationParameters

This structure is used by the SKB_SecureData_Derive method for those derivation algorithms that require secure or plain data input. Currently, this structure is used by the following algorithms:

- SKB_DERIVATION_ALGORITHM_HMAC_SHA256, SKB_DERIVATION_ALGORITHM_HMAC_SHA384, or SKB_DERIVATION_ALGORITHM_HMAC_SHA512

  For these derivation algorithms, the structure provides either the secure data object or plain data (but not both at the same time) to be used as the derivation input.

- SKB_DERIVATION_ALGORITHM_BLOCK_CONCATENATE

For this derivation algorithm, the structure provides the second input secure data object that must be appended to the first input secure data object as described in Concatenating Two Keys. The plain input is not supported for this algorithm.

- `SKB_DERIVATION_ALGORITHM_XOR`

  For this derivation algorithm, the structure specifies either the plain data buffer or secure data buffer to be XOR-ed with the input key. You may use either the secure data input, or the plain data input, but not both at the same time.

- `SKB_DERIVATION_ALGORITHM_DOUBLE_AES_ENCRYPT`

  For this derivation algorithm, the structure is used to provide the 16-byte plain data buffer that must be encrypted two times. The secure input is not supported for this algorithm.

The structure is declared as follows:

```
typedef struct {
    const SKB_SecureData* secure_input;
    const SKB_Byte*       plain_input;
    SKB_Size              plain_input_size;
} SKB_GenericDerivationParameters;
```

The following are the properties used:

- `secure_input`

  Pointer to the `SKB_SecureData` object containing the secure data input. If this parameter is provided, the `plain_input` parameter must be NULL.

- `plain_input`

  Pointer to the plain data input. If this parameter is provided, the `secure_input` parameter must be NULL.

- `plain_input_size`

  Size of the `plain_input` buffer. It must be 0 if `plain_input` is set to NULL.

## 9.9.20 SKB_EccCurveParameters

This structure defines parameters for a custom ECC curve in protected form. It must be employed only when the `SKB_ECC_CURVE_CUSTOM` curve type of the SKB_EccCurve enumeration is used. To generate instances of the `SKB_EccCurveParameters` structure, you must use the Custom ECC Tool.

Currently, custom ECC curves are supported only for the ECDSA, ECDH, and ECC key generation algorithms. For all other cases, this structure cannot be used.

The structure is declared as follows:

```
typedef struct {
    SKB_Size        prime_bit_length;
```

```
    SKB_Size        order_bit_length;
    SKB_Size        ecc_instance_length;
    const SKB_Byte* context;
    SKB_Size        context_size;
} SKB_EccCurveParameters;
```

The following are the properties used:

- `prime_bit_length`

  Bit-length of the prime parameter.

- `order_bit_length`

  Bit-length of the order parameter.

- `ecc_instance_length`

  ECC runtime instance to be used. For more information on this parameter, see Custom ECC Tool.

- `context`

  Byte buffer containing curve parameters in protected form.

- `context_size`

  Size of the `context` buffer in bytes.

**Important**

Do not try to come up with values of these parameters yourself. To generate the entire definition of the `SKB_EccCurveParameters` instance, you must use the Custom ECC Tool.

## 9.9.21 SKB_BlockCipherWrapParameters

This structure provides a fixed initialization vector to the wrapping algorithm executed by the `SKB_SecureData_Wrap` method. If this structure is not provided, a random initialization vector will be generated.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* iv;
} SKB_BlockCipherWrapParameters;
```

The `iv` property is a pointer to the byte buffer containing the initialization vector.

## 9.9.22 SKB_BlockCipherUnwrapParameters

A pointer to this structure can be passed to the SKB_Engine_CreateDataFromWrapped method in case the ECB or CBC mode of the wrapping algorithm is used. This structure specifies the padding type to be used.

The structure is declared as follows:

```
typedef struct {
    SKB_PaddingType padding;
} SKB_BlockCipherUnwrapParameters;
```

The padding property specifies the padding type to be used. The available padding types are defined in the SKB_PaddingType enumeration.

## 9.9.23 SKB_GcmUnwrapParameters

A pointer to this structure must be passed to the SKB_Engine_CreateDataFromWrapped method in case the GCM mode of the AES algorithm is used.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* additional_authenticated_data;
    SKB_Size        additional_authenticated_data_size;
    const SKB_Byte* initialization_vector;
    SKB_Size        initialization_vector_size;
    const SKB_Byte* authentication_tag;
    SKB_Size        authentication_tag_size;
} SKB_GcmUnwrapParameters;
```

The following are the properties used:

- additional_authenticated_data

  Buffer containing the optional additional authenticated data.

- additional_authenticated_data_size

  Size of the additional_authenticated_data buffer in bytes.

- initialization_vector

  Buffer containing the initialization vector.

- initialization_vector_size

  Size of the initialization_vector buffer in bytes. The size must be 12 bytes.

- authentication_tag

  Buffer containing the authentication tag.

- `authentication_tag_size`

  Size of the `authentication_tag` buffer in bytes. The size must match the AES block size, which is 16 bytes.

## 9.9.24 SKB_GcmCipherParameters

This structure supplies the GCM initialization vector to the SKB_Engine_CreateCipher method when creating the AES cipher in the GCM mode.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* initialization_vector;
    SKB_Size        initialization_vector_size;
} SKB_GcmCipherParameters;
```

The following are the properties used:

- `initialization_vector`

  Data buffer containing the GCM initialization vector.

- `initialization_vector_size`

  Size of the `initialization_vector` buffer in bytes.

  **Important**

  The only supported size is 12 bytes.

## 9.9.25 SKB_AuthenticationParameters

When using the AES cipher in the GCM mode, this structure must be provided to the SKB_Cipher_ProcessFinal method. The structure serves one of the following purposes depending on the cipher direction:

- In case of encryption, this structure is used to hold a pointer to a data buffer that will contain the calculated authentication tag.
- In case of decryption, this structure is used to hold a pointer to the supplied authentication tag that must be compared to the authentication tag calculated during decryption.

When using the AES cipher in the CCM mode, this structure must be provided to the SKB_Engine_CreateCipher method; in this case, only the authentication tag size information is used.

The structure is declared as follows:

```
```

```
typedef struct {
    SKB_Byte* authentication_tag;
    SKB_Size  authentication_tag_size;
} SKB_AuthenticationParameters;
```

The following are the properties used:

- `authentication_tag`

  For the GCM mode, in case of encryption, this property will be set by SKB to point to a data buffer containing the calculated authentication tag; in case of decryption, you must set the authentication tag in this property, which will be then read and compared to the authentication tag calculated during decryption.

  For the CCM mode, this property must be NULL.

- `authentication_tag_size`

  For the GCM mode, this is the size of the `authentication_tag` buffer in bytes. The supported buffer sizes are 16, 15, 14, 13, 12, 8, and 4.

  For the CCM mode, this is the size of the authentication tag in bytes. The supported tag sizes are 16, 14, 12, 10, 8, 6, and 4. The authentication tag itself is not passed in this structure when operating in the CCM mode.

## 9.9.26 SKB_DsaParameters

This structure provides the necessary input parameters for the following tasks:

- calculating a DSA signature using the SKB_Engine_CreateTransform method
- generating a private DSA key using the SKB_Engine_GenerateSecureData method
- obtaining a public DSA key from a private DSA key using the SKB_SecureData_GetPublicKey method

The structure is declared as follows:

```
typedef struct {
    SKB_Size l_bit_length;
    SKB_Size n_bit_length;
    const SKB_Byte* p;
    const SKB_Byte* q;
    const SKB_Byte* g;
} SKB_DsaParameters;
```

The following are the properties used:

- `l_bit_length`

  Size of the L parameter in bits.

- `n_bit_length`

  Size of the N parameter in bits.

  Please note that only the following combinations of L and N lengths are supported:

  - L is between 512 and 1024 (multiple of 64) and N is 160
  - L is 2048 and N is 224
  - L is 2048 and N is 256
  - L is 3072 and N is 256

- `p`

  Pointer to a buffer containing the prime modulus "p" encoded using the big-endian byte order. The buffer size must be `l_bit_length` divided by 8.

- `q`

  Pointer to a buffer containing the prime divisor "q" encoded using the big-endian byte order. The buffer size must be `n_bit_length` divided by 8.

- `g`

  Pointer to a buffer containing the generator "g" encoded using the big-endian byte order. The buffer size must be `l_bit_length` divided by 8.

## 9.9.27 SKB_RsaParameters

This structure provides the necessary input parameters for the following tasks:

- generating a private RSA key using the SKB_Engine_GenerateSecureData method
- obtaining a public RSA key from a private RSA key using the SKB_SecureData_GetPublicKey method

The structure is declared as follows:

```
typedef struct {
    SKB_Size n_bit_length;
    SKB_Size e;
} SKB_RsaParameters;
```

The following are the properties used:

- `n_bit_length`

  Size of the key to be generated in bits.

  The value must be greater than or equal to 1024, and less than or equal to 4096.

- `e`

Public (or encryption) exponent "e".

## 9.9.28 SKB_RsaPssParameters

This structure provides an ability to specify the salt and salt length when one of the following signature algorithms is used:

- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_384_EX`
- `SKB_SIGNATURE_ALGORITHM_RSA_PSS_512_EX`

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* salt;
    SKB_Size        salt_length;
} SKB_RsaPssParameters;
```

The following are the properties used:

- `salt`

  Pointer to a byte buffer containing the salt value to be used. If `salt_length` is greater than 0, the value of this property must not be NULL.

- `salt_length`

  Length of the `salt` buffer in bytes.

  It must be less than or equal to the RSA key length (in bytes) minus the hash function block size minus 2.

## 9.9.29 SKB_EccParameters

This structure provides additional parameters when ECC functions are used.

The structure is declared as follows:

```
typedef struct {
    SKB_EccCurve            curve;
    SKB_EccCurveParameters* curve_parameters;
    const unsigned int*     random_value;
} SKB_EccParameters;
```

The following are the properties used:

- `curve`

  Specifies the ECC curve type to be used. The available curve types are defined in the SKB_EccCurve enumeration.

- `curve_parameters`

  Pointer to the SKB_EccCurveParameters structure containing parameters for a custom ECC curve in protected form.

  This parameter should be set only when the `SKB_ECC_CURVE_CUSTOM` curve type is used.

  Currently, custom ECC curves are supported only for the ECDSA, ECDH, and ECC key generation algorithms. For all other cases, there is no point setting this parameter.

- `random_value`

  The value of this property must always be NULL.

## 9.9.30 SKB_PrimeDhParameters

This structure is required by the SKB_Engine_CreateKeyAgreement method when the classical Diffie-Hellman algorithm (`SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH`) is selected.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* context;
    SKB_Size        context_size;
} SKB_PrimeDhParameters;
```

The following are the properties used:

- `context`

  Pointer to a data buffer containing a combination of the prime "p" and generator "g" in protected form to be used by the Diffie-Hellman algorithm. To obtain this protected data buffer, use the Diffie-Hellman Tool.

- `context_size`

  Length of the `context` buffer in bytes.

## 9.9.31 SKB_EcdhParameters

This structure is required by the SKB_Engine_CreateKeyAgreement method when the `SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC` algorithm is used. This structure allows you to provide a fixed private ECC key to the ECDH algorithm.

The structure is declared as follows:

```
typedef struct {
    SKB_EccCurve                  curve;
    const SKB_EccCurveParameters* curve_parameters;
    const SKB_SecureData*         private_key;
} SKB_EcdhParameters;
```

The following are the properties used:

- `curve`

  Specifies the ECC curve type to be used. The available curve types are defined in the SKB_EccCurve enumeration.

- `curve_parameters`

  Pointer to the SKB_EccCurveParameters structure, which provides parameters of a custom ECC curve.

  This parameter should be set only when the `SKB_ECC_CURVE_CUSTOM` curve type is used.

- `private_key`

  Pointer to the `SKB_SecureData` object that contains the fixed private ECC key that must be provided to the ECDH algorithm.

  The type of the `SKB_SecureData` object must be `SKB_DATA_TYPE_ECC_PRIVATE_KEY`.

## 9.9.32 SKB_EddhParameters

This structure may be passed to the SKB_Engine_CreateKeyAgreement method to provide a fixed private key when the `SKB_KEY_AGREEMENT_ALGORITHM_X25519` algorithm is used.

The structure is declared as follows:

```
typedef struct {
    const SKB_SecureData* private_key;
} SKB_EddhParameters;
```

The `private_key` parameter is a pointer to the `SKB_SecureData` object that specifies the fixed private key to be used. The key must be 32 bytes long, and its data type must be `SKB_DATA_TYPE_BYTES`. You may also set this parameter to be NULL, in which case a random private key will be generated.

## 9.9.33 SKB_RawBytesParameters

This structure is required by the SKB_Engine_GenerateSecureData method to generate an `SKB_SecureData` object containing a buffer of random raw bytes. The only purpose of this structure is to specify the number of bytes to generate.

The structure is declared as follows:

```
```

```
typedef struct {
    SKB_Size byte_count;
} SKB_RawBytesParameters;
```

The `byte_count` parameter specifies the number of bytes to be generated.

### 9.9.34 SKB_HkdfDerivationParameters

This structure is used by the SKB_SecureData_Derive method when the `SKB_DERIVATION_ALGORITHM_HKDF_SHA256` derivation algorithm is selected. For information on this algorithm, see Deriving a Key Using HKDF with SHA-256.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* info;
    SKB_Size        info_length;
    const SKB_Byte* salt;
    SKB_Size        salt_length;
    SKB_Size        L;
} SKB_HkdfDerivationParameters;
```

The following are the properties used:

- `info`

  Pointer to a data buffer containing the optional context and application specific information (parameter `info`).

- `info_length`

  Length of the `info` buffer in bytes. The length may be 0.

- `salt`

  Pointer to a data buffer containing the salt value (parameter `salt`).

- `salt_length`

  Length of the `salt` buffer in bytes. The length may be 0.

- L

  Length of the output keying material in bytes (parameter L). It must be a multiple of 16, and it must be less than or equal to 8160 (255 times the hash length, which is 32 for SHA-256).

## 9.10 Enumerations

This section describes various enumerations defined in the Native API.

## 9.10.1 SKB_DataType

This enumeration specifies the possible data types of the content encapsulated by an SKB_SecureData object.

The enumeration is defined as follows:

```
typedef enum {
    SKB_DATA_TYPE_BYTES,
    SKB_DATA_TYPE_UNWRAP_BYTES,
    SKB_DATA_TYPE_RSA_PRIVATE_KEY,
    SKB_DATA_TYPE_ECC_PRIVATE_KEY,
    SKB_DATA_TYPE_DSA_PRIVATE_KEY,
    SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT,
    SKB_DATA_TYPE_RSA_STATIC_UNWRAP_CONTEXT,
    SKB_DATA_TYPE_RSA_STATIC_SIGN_CONTEXT
} SKB_DataType;
```

The following are the values defined:

- SKB_DATA_TYPE_BYTES

  Buffer of bytes (typically, for symmetric-key algorithms).

  Due to security concerns, the recommendation is to use this data type for all symmetric-key algorithms, except those that are related to unwrapping. Although most unwrapping related algorithms support this data type as the unwrapping key, it is more secure to use the restricted SKB_DATA_TYPE_UNWRAP_BYTES for unwrapping instead. The risk of using SKB_DATA_TYPE_BYTES as the unwrapping key lies in the fact that an attacker might use it also in a decryption operation to obtain wrapped keys in plain.

- SKB_DATA_TYPE_UNWRAP_BYTES

  Similar to SKB_DATA_TYPE_BYTES, but, for security reasons, it can only be used as an unwrapping key in unwrapping algorithms, and as the encryption key and output of the SKB_DERIVATION_ALGORITHM_DOUBLE_AES_ENCRYPT derivation algorithm. Other cryptographic algorithms do not support this data type.

  Secure data objects of the SKB_DATA_TYPE_UNWRAP_BYTES type can only be obtained using the following methods:

  - create using the Key Export Tool
  - derive from a secure data object of type SKB_DATA_TYPE_BYTES by using the SKB_SecureData_Derive method with the SKB_DERIVATION_ALGORITHM_GET_RESTRICTED_USAGE_KEY algorithm
  - import a previously exported secure data object
  - create using the Sensitive Operations Library

- SKB_DATA_TYPE_RSA_PRIVATE_KEY

Private RSA key.

- `SKB_DATA_TYPE_ECC_PRIVATE_KEY`

  Private ECC key.

- `SKB_DATA_TYPE_DSA_PRIVATE_KEY`

  Private DSA key.

- `SKB_DATA_TYPE_RSA_PUBLIC_KEY_CONTEXT`

  Public RSA key.

  Currently, this data type is used only for wrapping raw bytes using the RSA algorithm as described in SKB_SecureData_Wrap.

- `SKB_DATA_TYPE_RSA_STATIC_UNWRAP_CONTEXT`

  Private RSA key that can only be used as an unwrapping key. An exported key of this type can only be prepared using the Key Export Tool with the `--static-key-unwrap` command-line parameter set.

  For security reasons, it is preferred to choose this type of key over the `SKB_DATA_TYPE_RSA_PRIVATE_KEY` for unwrapping, if the private RSA key is known at compile time.

- `SKB_DATA_TYPE_RSA_STATIC_SIGN_CONTEXT`

  Private RSA key that can only be used as a signing key. An exported key of this type can only be prepared using the Key Export Tool with the `--static-key-sign` command-line parameter set.

  For security reasons, it is preferred to choose this type of key over the `SKB_DATA_TYPE_RSA_PRIVATE_KEY` for signing, if the private RSA key is known at compile time.

## 9.10.2 SKB_DataFormat

This enumeration specifies the possible formats how a cryptographic key can be stored in a data buffer.

The enumeration is defined as follows:

```
typedef enum {
    SKB_DATA_FORMAT_RAW,
    SKB_DATA_FORMAT_PKCS8,
    SKB_DATA_FORMAT_PKCS1,
    SKB_DATA_FORMAT_ECC_BINARY,
    SKB_DATA_FORMAT_ECC_PRIVATE,
    SKB_DATA_FORMAT_ECC_PUBLIC,
    SKB_DATA_FORMAT_ECC_PUBLIC_POINT,
    SKB_DATA_FORMAT_ECC_PUBLIC_X9_62,
    SKB_DATA_FORMAT_CRT,
    SKB_DATA_FORMAT_CRT_EQUAL_LEN
} SKB_DataFormat;
```

The following are the values defined:

- `SKB_DATA_FORMAT_RAW`

  Buffer of raw bytes.

  In case of the public DSA key, the data is encoded using the big-endian byte order.

- `SKB_DATA_FORMAT_PKCS8`

  Private RSA or ECC key stored according to the PKCS#8 standard.

  In case of ECC, only the following curve types are supported for this format:

  - P-192
  - P-224
  - P-256
  - P-384
  - P-521
  - Curve25519
  - Ed25519

- `SKB_DATA_FORMAT_PKCS1`

  Public RSA key stored according to the format of the `SubjectPublicKeyInfo` structure as defined in RFC 5280.

- `SKB_DATA_FORMAT_ECC_BINARY`

  Either a private ECC key stored using the format described in Private ECC Key, or a public ECC key stored using the format described in Public ECC Key, depending on the context in which this data buffer is used.

  **Important**

  This enumeration value is deprecated. You should use `SKB_DATA_FORMAT_ECC_PRIVATE` or `SKB_DATA_FORMAT_ECC_PUBLIC` instead.

- `SKB_DATA_FORMAT_ECC_PRIVATE`

  Private ECC key stored using the format described in Private ECC Key.

- `SKB_DATA_FORMAT_ECC_PUBLIC`

  Public ECC key stored using the format described in Public ECC Key.

- `SKB_DATA_FORMAT_ECC_PUBLIC_POINT`

  Public ECC key stored using the uncompressed elliptic curve point format described in section 2.2 of RFC 5480.

- `SKB_DATA_FORMAT_ECC_PUBLIC_X9_62`

  Public ECC key stored using the format described in section 2 of RFC 5480.

- `SKB_DATA_FORMAT_CRT`

  AES-wrapped private RSA key consisting of variable-length components according to the principles of the Chinese Remainder Theorem (CRT). This format can only be used in the SKB_Engine_CreateDataFromWrapped method. For details on this format, see here.

- `SKB_DATA_FORMAT_CRT_EQUAL_LEN`

  Same as above except the format is slightly different as described here.

## 9.10.3 SKB_CipherAlgorithm

This enumeration specifies cryptographic algorithms that are used for encrypting and decrypting data, as well as wrapping and unwrapping secure data objects.

Please note that not all of the listed algorithms support encryption, decryption, wrapping, and unwrapping. For information on which operations support which ciphers, see Supported Algorithms.

The enumeration is defined as follows:

```
typedef enum {
    SKB_CIPHER_ALGORITHM_NULL,
    SKB_CIPHER_ALGORITHM_AES_128_ECB,
    SKB_CIPHER_ALGORITHM_AES_128_CBC,
    SKB_CIPHER_ALGORITHM_AES_128_CTR,
    SKB_CIPHER_ALGORITHM_AES_128_GCM,
    SKB_CIPHER_ALGORITHM_AES_128_CCM,
    SKB_CIPHER_ALGORITHM_RSA,
    SKB_CIPHER_ALGORITHM_RSA_1_5,
    SKB_CIPHER_ALGORITHM_RSA_OAEP,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA224,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA256,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA384,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA512,
    SKB_CIPHER_ALGORITHM_RSA_OAEP_MD5,
    SKB_CIPHER_ALGORITHM_ECC_ELGAMAL,
    SKB_CIPHER_ALGORITHM_AES_192_ECB,
    SKB_CIPHER_ALGORITHM_AES_192_CBC,
    SKB_CIPHER_ALGORITHM_AES_192_CTR,
    SKB_CIPHER_ALGORITHM_AES_192_GCM,
    SKB_CIPHER_ALGORITHM_AES_192_CCM,
    SKB_CIPHER_ALGORITHM_AES_256_ECB,
```

```
    SKB_CIPHER_ALGORITHM_AES_256_CBC,
    SKB_CIPHER_ALGORITHM_AES_256_CTR,
    SKB_CIPHER_ALGORITHM_AES_256_GCM,
    SKB_CIPHER_ALGORITHM_AES_256_CCM,
    SKB_CIPHER_ALGORITHM_DES_ECB,
    SKB_CIPHER_ALGORITHM_DES_CBC,
    SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB,
    SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC,
    SKB_CIPHER_ALGORITHM_NIST_AES,
    SKB_CIPHER_ALGORITHM_XOR,
    SKB_CIPHER_ALGORITHM_SPECK_64_128_ECB,
    SKB_CIPHER_ALGORITHM_SPECK_64_128_CBC,
    SKB_CIPHER_ALGORITHM_SPECK_64_128_CTR,
    SKB_CIPHER_ALGORITHM_ASC_X9_TR_31,
} SKB_CipherAlgorithm;
```

The following are the values defined:

- `SKB_CIPHER_ALGORITHM_NULL`

  Value that identifies that no algorithm was used, meaning that the corresponding data is not encrypted.

  This value is used by the `SKB_Engine_CreateDataFromWrapped` method to specify that the data to be loaded is in plain form as described in Loading Plain Keys.

- `SKB_CIPHER_ALGORITHM_AES_128_ECB`

  128-bit AES in the ECB mode

- `SKB_CIPHER_ALGORITHM_AES_128_CBC`

  128-bit AES in the CBC mode

- `SKB_CIPHER_ALGORITHM_AES_128_CTR`

  128-bit AES in the CTR mode

- `SKB_CIPHER_ALGORITHM_AES_128_GCM`

  128-bit AES in the GCM mode

- `SKB_CIPHER_ALGORITHM_AES_128_CCM`

  128-bit AES in the CCM mode

- `SKB_CIPHER_ALGORITHM_RSA`

  1024-bit to 4096-bit RSA with no padding.

- `SKB_CIPHER_ALGORITHM_RSA_1_5`

  1024-bit to 4096-bit RSA with PKCS#1 version 1.5 padding.

- SKB_CIPHER_ALGORITHM_RSA_OAEP

  1024-bit to 4096-bit RSA with OAEP padding using SHA-1 and MGF1 using SHA-1
- SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA224

  1024-bit to 4096-bit RSA with OAEP padding using SHA-224 and MGF1 using SHA-224
- SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA256

  1024-bit to 4096-bit RSA with OAEP padding using SHA-256 and MGF1 using SHA-256
- SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA384

  1024-bit to 4096-bit RSA with OAEP padding using SHA-384 and MGF1 using SHA-384
- SKB_CIPHER_ALGORITHM_RSA_OAEP_SHA512

  1024-bit to 4096-bit RSA with OAEP padding using SHA-512 and MGF1 using SHA-512
- SKB_CIPHER_ALGORITHM_RSA_OAEP_MD5

  1024-bit to 4096-bit RSA with OAEP padding using MD5 and MGF1 using MD5
- SKB_CIPHER_ALGORITHM_ECC_ELGAMAL

  ElGamal ECC
- SKB_CIPHER_ALGORITHM_AES_192_ECB

  192-bit AES in the ECB mode
- SKB_CIPHER_ALGORITHM_AES_192_CBC

  192-bit AES in the CBC mode
- SKB_CIPHER_ALGORITHM_AES_192_CTR

  192-bit AES in the CTR mode
- SKB_CIPHER_ALGORITHM_AES_192_GCM

  192-bit AES in the GCM mode
- SKB_CIPHER_ALGORITHM_AES_192_CCM

  192-bit AES in the CCM mode
- SKB_CIPHER_ALGORITHM_AES_256_ECB

  256-bit AES in the ECB mode
- SKB_CIPHER_ALGORITHM_AES_256_CBC

  256-bit AES in the CBC mode
- SKB_CIPHER_ALGORITHM_AES_256_CTR

  256-bit AES in the CTR mode

- `SKB_CIPHER_ALGORITHM_AES_256_GCM`

  256-bit AES in the GCM mode

- `SKB_CIPHER_ALGORITHM_AES_256_CCM`

  256-bit AES in the CCM mode

- `SKB_CIPHER_ALGORITHM_DES_ECB`

  DES in the ECB mode

- `SKB_CIPHER_ALGORITHM_DES_CBC`

  DES in the CBC mode

- `SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB`

  Triple DES in the ECB mode

- `SKB_CIPHER_ALGORITHM_TRIPLE_DES_CBC`

  Triple DES in the CBC mode

- `SKB_CIPHER_ALGORITHM_NIST_AES`

  AES key wrapping and unwrapping algorithm defined by NIST in the Special Publication 800-38F. Decryption using this cipher is also supported, in which case it is essentially the unwrapping algorithm applied to plain data.

- `SKB_CIPHER_ALGORITHM_XOR`

  Wrapping and unwrapping using XOR:

  - If the `SKB_SecureData_Wrap` function is used, the key to be wrapped is XOR-ed with the wrapping key.
  - If the `SKB_Engine_CreateDataFromWrapped` function is used, the wrapped buffer is XOR-ed with the unwrapping key.

  In both cases, the two XOR-ed buffers must be of equal size.

- `SKB_CIPHER_ALGORITHM_SPECK_64_128_ECB`

  Speck using 128-bit keys and 64-bit blocks in the ECB mode (no padding)

- `SKB_CIPHER_ALGORITHM_SPECK_64_128_CBC`

  Speck using 128-bit keys and 64-bit blocks in the CBC mode (no padding)

- `SKB_CIPHER_ALGORITHM_SPECK_64_128_CTR`

  Speck using 128-bit keys and 64-bit blocks in the CTR mode

- `SKB_CIPHER_ALGORITHM_ASC_X9_TR_31`

  Wrapping and unwrapping of raw bytes as specified by ASC X9 TR 31-2018 using either AES (128-bit, 192-bit, or 256-bit), or Triple DES (two-key or three-key)

## 9.10.4 SKB_CipherDirection

This enumeration specifies the possible directions (encryption or decryption) for the SKB_Engine_CreateCipher method and the SKB_DERIVATION_ALGORITHM_CIPHER derivation algorithm, which is described in SKB_CipherDerivationParameters.

The enumeration is defined as follows:

```
typedef enum {
    SKB_CIPHER_DIRECTION_ENCRYPT,
    SKB_CIPHER_DIRECTION_DECRYPT
} SKB_CipherDirection;
```

## 9.10.5 SKB_SignatureAlgorithm

This enumeration specifies the possible signing and verifying algorithms for the SKB_Transform object.

The enumeration is defined as follows:

```
typedef enum {
    SKB_SIGNATURE_ALGORITHM_AES_128_CMAC,
    SKB_SIGNATURE_ALGORITHM_AES_192_CMAC,
    SKB_SIGNATURE_ALGORITHM_AES_256_CMAC,
    SKB_SIGNATURE_ALGORITHM_SPECK_64_128_CMAC,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA1,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA224,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA256,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA384,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA512,
    SKB_SIGNATURE_ALGORITHM_HMAC_MD5,
    SKB_SIGNATURE_ALGORITHM_RSA,
    SKB_SIGNATURE_ALGORITHM_DSA,
    SKB_SIGNATURE_ALGORITHM_ECDSA,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA224,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA384,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA512,
    SKB_SIGNATURE_ALGORITHM_ECDSA_MD5,
    SKB_SIGNATURE_ALGORITHM_RSA_MD5,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA1,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA224,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA256,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA384,
```

```
    SKB_SIGNATURE_ALGORITHM_RSA_SHA512,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512_EX,
    SKB_SIGNATURE_ALGORITHM_DES_RETAIL_MAC,
    SKB_SIGNATURE_ALGORITHM_ED25519
} SKB_SignatureAlgorithm;
```

The following are the values defined:

- `SKB_SIGNATURE_ALGORITHM_AES_128_CMAC`

  128-bit AES-CMAC

- `SKB_SIGNATURE_ALGORITHM_AES_192_CMAC`

  192-bit AES-CMAC

- `SKB_SIGNATURE_ALGORITHM_AES_256_CMAC`

  256-bit AES-CMAC

- `SKB_SIGNATURE_ALGORITHM_SPECK_64_128_CMAC`

  Speck-CMAC using 128-bit keys and 64-bit blocks

- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA1`

  HMAC using SHA-1 as the hash function

- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA224`

  HMAC using SHA-224 as the hash function

- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA256`

  HMAC using SHA-256 as the hash function

- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA384`

  HMAC using SHA-384 as the hash function

- `SKB_SIGNATURE_ALGORITHM_HMAC_SHA512`

HMAC using SHA-512 as the hash function

- `SKB_SIGNATURE_ALGORITHM_HMAC_MD5`

  HMAC using MD5 as the hash function

- `SKB_SIGNATURE_ALGORITHM_RSA`

  1024-bit to 4096-bit RSA signing with PKCS#1 version 1.5 padding without a hash function. This algorithm can only be executed on short unencrypted messages prepared outside of SKB, which are formatted as described in SKB_Transform_AddBytes.

- `SKB_SIGNATURE_ALGORITHM_DSA`

  DSA specified by the FIPS 186-4 standard (can only be executed on a short unencrypted message prepared outside of SKB)

- `SKB_SIGNATURE_ALGORITHM_ECDSA`

  ECDSA with either standard or custom curves (can only be executed on a short unencrypted message prepared outside of SKB)

- `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1`

  ECDSA with either standard or custom curves using SHA-1 as the hash function

- `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA224`

  ECDSA with either standard or custom curves using SHA-224 as the hash function

- `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256`

  ECDSA with either standard or custom curves using SHA-256 as the hash function

- `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA384`

  ECDSA with either standard or custom curves using SHA-384 as the hash function

- `SKB_SIGNATURE_ALGORITHM_ECDSA_SHA512`

  ECDSA with either standard or custom curves using SHA-512 as the hash function

- `SKB_SIGNATURE_ALGORITHM_ECDSA_MD5`

  ECDSA with either standard or custom curves using MD5 as the hash function

- `SKB_SIGNATURE_ALGORITHM_RSA_MD5`

  1024-bit to 4096-bit RSA signing with PKCS#1 version 1.5 padding using MD5 as the hash function

- `SKB_SIGNATURE_ALGORITHM_RSA_SHA1`

  1024-bit to 4096-bit RSA signing with PKCS#1 version 1.5 padding using SHA-1 as the hash function

- `SKB_SIGNATURE_ALGORITHM_RSA_SHA224`

  1024-bit to 4096-bit RSA signing with PKCS#1 version 1.5 padding using SHA-224 as the hash function

- SKB_SIGNATURE_ALGORITHM_RSA_SHA256

  1024-bit to 4096-bit RSA signing with PKCS#1 version 1.5 padding using SHA-256 as the hash function

- SKB_SIGNATURE_ALGORITHM_RSA_SHA384

  1024-bit to 4096-bit RSA signing with PKCS#1 version 1.5 padding using SHA-384 as the hash function

- SKB_SIGNATURE_ALGORITHM_RSA_SHA512

  1024-bit to 4096-bit RSA signing with PKCS#1 version 1.5 padding using SHA-512 as the hash function

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5

  1024-bit to 4096-bit RSA signing with PSS padding using MD5 as the hash function. A random salt value of 16 bytes will be generated.

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1

  1024-bit to 4096-bit RSA signing with PSS padding using SHA-1 as the hash function. A random salt value of 20 bytes will be generated.

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224

  1024-bit to 4096-bit RSA signing with PSS padding using SHA-224 as the hash function. A random salt value of 28 bytes will be generated.

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256

  1024-bit to 4096-bit RSA signing with PSS padding using SHA-256 as the hash function. A random salt value of 32 bytes will be generated.

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384

  1024-bit to 4096-bit RSA signing with PSS padding using SHA-384 as the hash function. A random salt value of 48 bytes will be generated.

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512

  1024-bit to 4096-bit RSA signing with PSS padding using SHA-512 as the hash function. A random salt value of 64 bytes will be generated.

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5_EX

  Same as SKB_SIGNATURE_ALGORITHM_RSA_PSS_MD5 but allows specifying the salt value and length

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX

  Same as SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1 but allows specifying the salt value and length

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224_EX

  Same as SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA224 but allows specifying the salt value and length

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX

  Same as SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256 but allows specifying the salt value and length

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384_EX

  Same as SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA384 but allows specifying the salt value and length

- SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512_EX

  Same as SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA512 but allows specifying the salt value and length

- SKB_SIGNATURE_ALGORITHM_DES_RETAIL_MAC

  ISO/IEC 9797-1 MAC algorithm 3 for DES (also known as Retail MAC) without padding

- SKB_SIGNATURE_ALGORITHM_ED25519

  Elliptic curve signing algorithm EdDSA with the edwards25519 curve

## 9.10.6 SKB_DigestAlgorithm

This enumeration specifies the available digest algorithms, and is defined as follows:

```
typedef enum {
    SKB_DIGEST_ALGORITHM_SHA1,
    SKB_DIGEST_ALGORITHM_SHA224,
    SKB_DIGEST_ALGORITHM_SHA256,
    SKB_DIGEST_ALGORITHM_SHA384,
    SKB_DIGEST_ALGORITHM_SHA512,
    SKB_DIGEST_ALGORITHM_MD5,
} SKB_DigestAlgorithm;
```

## 9.10.7 SKB_DerivationAlgorithm

This enumeration specifies the possible algorithms that can be used for deriving one SKB_SecureData object from another using the SKB_SecureData_Derive method.

The enumeration is defined as follows:

```
typedef enum {
    SKB_DERIVATION_ALGORITHM_SLICE,
    SKB_DERIVATION_ALGORITHM_BLOCK_SLICE,
    SKB_DERIVATION_ALGORITHM_BLOCK_CONCATENATE,
    SKB_DERIVATION_ALGORITHM_SELECT_BYTES,
    SKB_DERIVATION_ALGORITHM_CIPHER,
    SKB_DERIVATION_ALGORITHM_SHA_1,
    SKB_DERIVATION_ALGORITHM_SHA_256,
    SKB_DERIVATION_ALGORITHM_SHA_384,
    SKB_DERIVATION_ALGORITHM_SHA_512,
    SKB_DERIVATION_ALGORITHM_HMAC_SHA256,
    SKB_DERIVATION_ALGORITHM_HMAC_SHA384,
```

```
    SKB_DERIVATION_ALGORITHM_HMAC_SHA512,
    SKB_DERIVATION_ALGORITHM_REVERSE_BYTES,
    SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128,
    SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT,
    SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_SPECK_L16BITLE,
    SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2,
    SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE,
    SKB_DERIVATION_ALGORITHM_SHA_AES,
    SKB_DERIVATION_ALGORITHM_XOR,
    SKB_DERIVATION_ALGORITHM_DOUBLE_AES_ENCRYPT,
    SKB_DERIVATION_ALGORITHM_GET_RESTRICTED_USAGE_KEY,
    SKB_DERIVATION_ALGORITHM_HKDF_SHA256,
} SKB_DerivationAlgorithm;
```

The following are the values defined:

- `SKB_DERIVATION_ALGORITHM_SLICE`

  Derives a new `SKB_SecureData` object as a substring of bytes of another `SKB_SecureData` object

- `SKB_DERIVATION_ALGORITHM_BLOCK_SLICE`

  Same as `SKB_DERIVATION_ALGORITHM_SLICE`, but it requires the index of the first byte and the number of bytes in the substring to be multiples of 8 or 16 depending on which block slicing algorithm variation is enabled in the SKB package

- `SKB_DERIVATION_ALGORITHM_BLOCK_CONCATENATE`

  Derives a new `SKB_SecureData` object by concatenating two existing `SKB_SecureData` objects as described in Concatenating Two Keys

- `SKB_DERIVATION_ALGORITHM_SELECT_BYTES`

  Derives a new `SKB_SecureData` object from the input `SKB_SecureData` object by copying only odd or even bytes from it as described in Deriving a Key as Odd or Even Bytes of Another Key

- `SKB_DERIVATION_ALGORITHM_CIPHER`

  Derives a new `SKB_SecureData` object from the input `SKB_SecureData` object by encrypting or decrypting it with another key

- `SKB_DERIVATION_ALGORITHM_SHA_1`

  Obtains a digest from the input `SKB_SecureData` object by executing SHA-1 one or several times, and stores the specified substring of bytes from the output as a new `SKB_SecureData` object as described in Iterated SHA-1 Derivation

- `SKB_DERIVATION_ALGORITHM_SHA_256`

Obtains a SHA-256 digest from a buffer that contains an SKB_SecureData object, prefixed and suffixed with plain data, and stores the output as a new SKB_SecureData object as described in SHA Derivation with Plain Prefix and Suffix

- SKB_DERIVATION_ALGORITHM_SHA_384

  Obtains a digest from the input SKB_SecureData object by executing SHA-384 and storing the entire 48-byte output as a new SKB_SecureData object as described in SHA-384 Derivation

- SKB_DERIVATION_ALGORITHM_SHA_512

  Same as SKB_DERIVATION_ALGORITHM_SHA_256, but the SHA-512 algorithm is used instead

- SKB_DERIVATION_ALGORITHM_HMAC_SHA256

  Executes HMAC-SHA-256 on the input buffer (either a SKB_SecureData object or plain data) to derive a new 32-byte SKB_SecureData object as described in HMAC Derivation

- SKB_DERIVATION_ALGORITHM_HMAC_SHA384

  Executes HMAC-SHA-384 on the input buffer (either a SKB_SecureData object or plain data) to derive a new 48-byte SKB_SecureData object as described in HMAC Derivation

- SKB_DERIVATION_ALGORITHM_HMAC_SHA512

  Executes HMAC-SHA-512 on the input buffer (either a SKB_SecureData object or plain data) to derive a new 64-byte SKB_SecureData object as described in HMAC Derivation

- SKB_DERIVATION_ALGORITHM_REVERSE_BYTES

  Derives a new SKB_SecureData object where the order of bytes is reversed as described in Reversing the Order of Bytes of a Key. You can use this derivation type to convert keys encoded in little-endian to big-endian and vice versa.

- SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128

  Derives a new SKB_SecureData object according to the key derivation function specified in NIST 800-108, using 128-bit AES-CMAC as the pseudorandom function in the counter mode. For more information on this algorithm, see Using the NIST 800-108 Key Derivation Function. This algorithm version uses the 32-bit parameter "L".

- SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128_L16BIT

  Same as SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128, but it uses the 16-bit parameter "L".

- SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_SPECK_L16BITLE

  Derives a new SKB_SecureData object according to the key derivation function specified in NIST 800-108, using Speck-CMAC as the pseudorandom function in the counter mode. For more information on this algorithm, see Using the NIST 800-108 Key Derivation Function. In this algorithm, the "L" parameter is 16 bits long encoded using the little-endian format.

- SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2

  Derives a new SKB_SecureData object according to KDF2 used in the RSAES-KEM-KWS scheme of the OMA DRM Specification

- SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE

  Derives a new SKB_SecureData object with the type SKB_DATA_TYPE_BYTES from another SKB_SecureData object whose type is SKB_DATA_TYPE_ECC_PRIVATE_KEY as described in Deriving a Key as Raw Bytes from a Private ECC Key

- SKB_DERIVATION_ALGORITHM_SHA_AES

  Derives a new SKB_SecureData object using the algorithm described in Deriving a Key By Encrypting Data Using 128-bit AES With a Concatenated Key

- SKB_DERIVATION_ALGORITHM_XOR

  Derives a new SKB_SecureData object by XOR-ing a key with plain data or another key

- SKB_DERIVATION_ALGORITHM_DOUBLE_AES_ENCRYPT

  Derives a new SKB_SecureData object by encrypting a buffer of 16 bytes with another SKB_SecureData object two times using the AES cipher as described in Deriving a Key By Encrypting a Plain Buffer Two Times. This algorithm requires that the data type of the input SKB_SecureData object is SKB_DATA_TYPE_UNWRAP_BYTES.

- SKB_DERIVATION_ALGORITHM_GET_RESTRICTED_USAGE_KEY

  Takes an SKB_SecureData object of type SKB_DATA_TYPE_BYTES as input and derives a new SKB_SecureData object with the same content, but with type SKB_DATA_TYPE_UNWRAP_BYTES

- SKB_DERIVATION_ALGORITHM_HKDF_SHA256

  HMAC-based Extract-and-Expand Key Derivation Function (HKDF) with SHA-256 as defined by RFC 5869

## 9.10.8 SKB_TransformType

This enumeration specifies the available transform types used by the SKB_Engine_CreateTransform method when creating an SKB_Transform object.

The enumeration is defined as follows:

```
typedef enum {
    SKB_TRANSFORM_TYPE_DIGEST,
    SKB_TRANSFORM_TYPE_SIGN,
    SKB_TRANSFORM_TYPE_VERIFY
} SKB_TransformType;
```

The following are the values defined:

- SKB_TRANSFORM_TYPE_DIGEST

Transform for calculating a digest (hash value)

- `SKB_TRANSFORM_TYPE_SIGN`

  Transform for creating a signature

- `SKB_TRANSFORM_TYPE_VERIFY`

  Transform for verifying a signature.

## 9.10.9 SKB_ExportTarget

This enumeration specifies the various export types supported by the `SKB_SecureData_Export` method and the `SKB_DukptState_Export` method.

The enumeration is defined as follows:

```
typedef enum {
    SKB_EXPORT_TARGET_CLEARTEXT,
    SKB_EXPORT_TARGET_PERSISTENT,
    SKB_EXPORT_TARGET_CROSS_ENGINE,
    SKB_EXPORT_TARGET_CUSTOM
} SKB_ExportTarget;
```

The following are the values defined:

- `SKB_EXPORT_TARGET_CLEARTEXT`

  Not used by SKB.

- `SKB_EXPORT_TARGET_PERSISTENT`

  The object to be exported is encrypted with the export key before it leaves the protected SKB domain. For information on this export type, see Persistent Exporting.

- `SKB_EXPORT_TARGET_CROSS_ENGINE`

  The object to be exported is passed to the unprotected outside world in the exact form in which the object exists in memory. For information on this export type, see Cross-Engine Exporting.

- `SKB_EXPORT_TARGET_CUSTOM`

  Not used by SKB.

## 9.10.10 SKB_EccCurve

This enumeration specifies the available ECC curve types.

This enumeration is defined as follows:

```
typedef enum {
    SKB_ECC_CURVE_SECP_R1_160,
```

```
    SKB_ECC_CURVE_NIST_192,
    SKB_ECC_CURVE_NIST_224,
    SKB_ECC_CURVE_NIST_256,
    SKB_ECC_CURVE_NIST_384,
    SKB_ECC_CURVE_NIST_521,
    SKB_ECC_CURVE_CUSTOM
} SKB_EccCurve;
```

The following are the values defined:

- `SKB_ECC_CURVE_SECP_R1_160`

  secp160r1, which is the 160-bit prime curve recommended by SECG.

- `SKB_ECC_CURVE_NIST_192`

  P-192, which is the 192-bit prime curve recommended by NIST (also known as secp192r1 and prime192v1).

- `SKB_ECC_CURVE_NIST_224`

  P-224, which is the 224-bit prime curve recommended by NIST (also known as secp224r1).

- `SKB_ECC_CURVE_NIST_256`

  P-256, which is the 256-bit prime curve recommended by NIST (also known as secp256r1 and prime256v1).

- `SKB_ECC_CURVE_NIST_384`

  P-384, which is the 384-bit prime curve recommended by NIST (also known as secp384r1).

  Currently, this curve type is supported only for ECDSA, ECDH, and key generation, but not for decrypting and unwrapping.

- `SKB_ECC_CURVE_NIST_521`

  P-521, which is the 521-bit prime curve recommended by NIST (also known as secp521r1).

  Currently, this curve type is supported only for ECDSA, ECDH, and key generation, but not for decrypting and unwrapping.

- `SKB_ECC_CURVE_CUSTOM`

  Prime ECC curve with custom parameters. If you use this curve type, you have to generate the specific curve definition using the Custom ECC Tool.

  Currently, this curve type is supported only for ECDSA, ECDH, and key generation, but not for decrypting and unwrapping.

## 9.10.11 SKB_KeyAgreementAlgorithm

This enumeration specifies the available key agreement algorithms used for the following purposes:

- creating an SKB_KeyAgreement object by the SKB_Engine_CreateKeyAgreement method
- establishing a TLS connection by the SKB_TLS_OpenConnection method or the SKB_TLS_OpenConnectionOnSocket method

The enumeration is defined as follows:

```
typedef enum {
    SKB_KEY_AGREEMENT_ALGORITHM_ECDH,
    SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH,
    SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC,
    SKB_KEY_AGREEMENT_ALGORITHM_X25519
} SKB_KeyAgreementAlgorithm;
```

The following are the values defined:

- SKB_KEY_AGREEMENT_ALGORITHM_ECDH

  Elliptic curve Diffie-Hellman with a randomly generated private ECC key

- SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH

  Classical Diffie-Hellman with the protected prime "p" and generator "g". If you use this algorithm, you have to generate protected values of "p" and "g" using the Diffie-Hellman Tool.

- SKB_KEY_AGREEMENT_ALGORITHM_ECDH_STATIC

  Elliptic curve Diffie-Hellman with a fixed private ECC key, which is provided as a secure data object

- SKB_KEY_AGREEMENT_ALGORITHM_X25519

  Montgomery-X-coordinate DH function using Curve25519

## 9.10.12 SKB_PaddingType

This enumeration specifies the ECB and CBC mode padding types that can be referenced by the SKB_BlockCipherUnwrapParameters structure.

The enumeration is defined as follows:

```
typedef enum {
    SKB_PADDING_TYPE_NONE,
    SKB_PADDING_TYPE_XMLENC
} SKB_PaddingType;
```

The following are the values defined:

- SKB_PADDING_TYPE_NONE

  No padding

- SKB_PADDING_TYPE_XMLENC

  XML encryption padding

## 9.10.13 SKB_SelectBytesDerivationVariant

This enumeration is used by the SKB_SelectBytesDerivationParameters structure to specify whether odd or even bytes must be selected.

The enumeration is defined as follows:

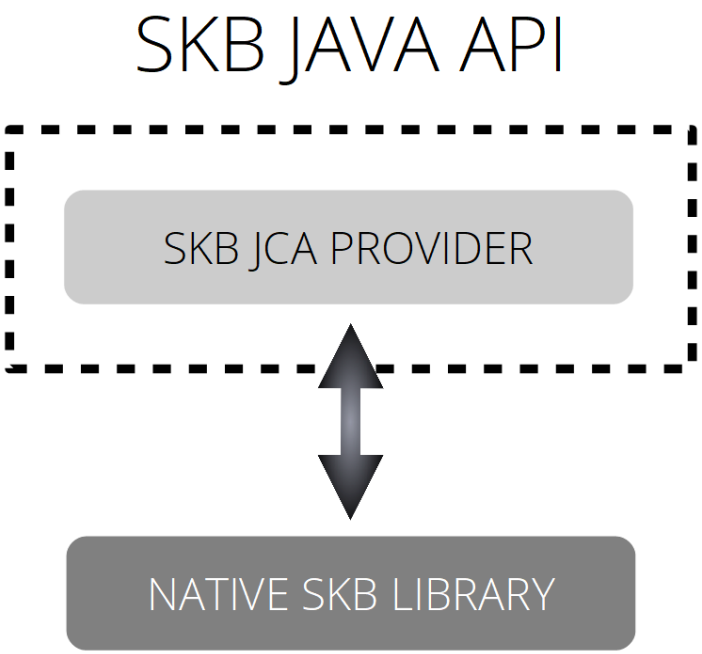
```
typedef enum {
    SKB_SELECT_BYTES_DERIVATION_ODD_BYTES,
    SKB_SELECT_BYTES_DERIVATION_EVEN_BYTES,
} SKB_SelectBytesDerivationVariant;
```

# 10. Java API

This chapter describes how to use the SKB Java library.

## 10.1 Overview of the SKB Java Library

The purpose of the SKB Java library is to provide an API that serves as a bridge between the native SKB library, which is written in C, and the Java programming language. In simple words, the SKB Java API is a Java wrapper around the native SKB classes and methods.




The SKB JCA Provider is an implementation of the Java Cryptography Architecture (JCA) API, which 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 the Java programming language.

**Important**

This User Guide assumes you have good knowledge of the general JCA concepts and API.

## 10.2 Adding the SKB Java Library to Your Application

To use the SKB Java API in your application, you must execute the following main steps:

1. Include the `skb.jar` file into your application.

This JAR file is provided in the `lib` directory, and it contains all Java classes of the SKB Java API. The JCA-related SKB classes belong to the `com.whitecryption.skb.provider` package.

2. Include the native shared SKB library that is called by the SKB Java API.

   Depending on your platform, it is one of the following files:

   - `SecureKeyBoxJava.dll` (for Windows)
   - `libSecureKeyBoxJava.dylib` (for macOS)
   - `libSecureKeyBoxJava.so` (for other platforms)

   For every target platform, a separate edition of this library is provided in the `lib` directory.

3. Optionally, before using the SKB Java API, register one or both SKB-specific implementations of the JCA Provider using the `Security.addProvider()` method, for example as follows:

```
Security.addProvider(new SkbProvider());
```

   For information on the two available Provider implementations, see Implementation Details.

   Alternatively, instead of registering the Provider globally, you may choose to create a Provider object and then supply it to individual JCA methods directly, for example as follows:

```
SkbProvider skb = new SkbProvider();
KeyFactory skbKeyFactory = KeyFactory.getInstance("RSA", skb);
```

## 10.3 Limitations

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

- Only a subset of target platforms is supported, as indicated in Supported Platforms.
- The SKB JCA Provider does not support the following algorithms:
  - Speck cipher
  - authenticated encryption with additional data (AEAD) modes of AES (GCM and CCM)
  - ISO/IEC 9797-1 MAC algorithm 3 for DES (also known as Retail MAC)
  - Speck-CMAC
  - DSA
  - generating RSA keys
  - 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
  - all key derivation algorithms
  - device binding
- Multi-threading is not supported.
- Custom ECC curve types are not supported.

If you require any of the unsupported features above, there is an undocumented internal mechanism in place that allows calling the native SKB API directly from Java. While this is not the recommended way of using SKB in Java, in some cases it might satisfy an immediate need. If you are interested in this approach, please contact Zimperium.

## 10.4 Implementation Details

Zimperium 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.

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

- The SKB JCA Provider implementation supports Java versions 6 and later.
- The SKB JCA Provider classes belong to the `com.whitecryption.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-specific JCA Provider implementation classes are available, as shown in the following table.

| Class | Description |
|---|---|
| `SkbProvider` | This Provider contains the full scope of SKB algorithms (except those listed in the previous section), but it does not support the `SKB_CIPHER_FLAG_HIGH_SPEED` AES implementation. When referenced by name, this Provider is called "whiteCryption". |
| `SkbHighSpeedAesProvider` | This Provider is specifically intended for the `SKB_CIPHER_FLAG_HIGH_SPEED` AES implementation. It does not support any other ciphers and algorithms. When referenced by name, this Provider is called "whiteCryptionHighSpeedAes". |

## 10.5 Algorithms Supported by the SKB JCA Provider

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

# 10.5.1 Cipher

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

## Encryption

`Cipher.ENCRYPT_MODE` supports the following algorithms:

- `DES/ECB/NoPadding`
- `DES/ECB/PKCS5Padding`
- `DES/CBC/NoPadding`
- `DES/CBC/PKCS5Padding`
- `DESede/ECB/NoPadding`
- `DESede/ECB/PKCS5Padding`
- `DESede/CBC/NoPadding`
- `DESede/CBC/PKCS5Padding`
- `AES/ECB/NoPadding` (128-bit, 192-bit, and 256-bit keys)
- `AES/ECB/PKCS5Padding` (128-bit, 192-bit, and 256-bit keys)
- `AES/CBC/NoPadding` (128-bit, 192-bit, and 256-bit keys)
- `AES/CBC/PKCS5Padding` (128-bit, 192-bit, and 256-bit keys)
- `AES/CTR/NoPadding` (128-bit, 192-bit, and 256-bit keys)
- `AES/CTR/PKCS5Padding` (128-bit, 192-bit, and 256-bit keys)

## Decryption

`Cipher.DECRYPT_MODE` supports the following algorithms:

- `DES/ECB/NoPadding`
- `DES/ECB/PKCS5Padding`
- `DES/CBC/NoPadding`
- `DES/CBC/PKCS5Padding`
- `DESede/ECB/NoPadding`
- `DESede/ECB/PKCS5Padding`
- `DESede/CBC/NoPadding`
- `DESede/CBC/PKCS5Padding`

- AES/ECB/NoPadding (128-bit, 192-bit, and 256-bit keys)
- AES/ECB/PKCS5Padding (128-bit, 192-bit, and 256-bit keys)
- AES/CBC/NoPadding (128-bit, 192-bit, and 256-bit keys)
- AES/CBC/PKCS5Padding (128-bit, 192-bit, and 256-bit keys)
- AES/CTR/NoPadding (128-bit, 192-bit, and 256-bit keys)
- AES/CTR/PKCS5Padding (128-bit, 192-bit, and 256-bit keys)
- RSA/NONE/NoPadding (1024-bit to 4096-bit keys)
- RSA/NONE/PKCS1Padding (1024-bit to 4096-bit keys)
- RSA/NONE/OAEPWithMD5AndMGF1Padding (1024-bit to 4096-bit keys)
- RSA/NONE/OAEPWithSHA1AndMGF1Padding (1024-bit to 4096-bit keys)
- RSA/NONE/OAEPWithSHA224AndMGF1Padding (1024-bit to 4096-bit keys)
- RSA/NONE/OAEPWithSHA256AndMGF1Padding (1024-bit to 4096-bit keys)
- RSA/NONE/OAEPWithSHA384AndMGF1Padding (1024-bit to 4096-bit keys)
- RSA/NONE/OAEPWithSHA512AndMGF1Padding (1024-bit to 4096-bit keys)
- ECCElGamal/NONE/NoPadding (only with the standard 160-bit, 192-bit, 224-bit, and 256-bit curves)
- AesWrap/NONE/NoPadding (unwrapping defined by NIST in the Special Publication 800-38F can decrypt AES keys

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

## Wrapping

Cipher.WRAP_MODE supports the following algorithms:

- DESede/ECB/NoPadding
- DESede/CBC/ISO10126Padding
- AES/ECB/NoPadding (128-bit, 192-bit, and 256-bit keys) can wrap DES, AES, and ECC keys
- AES/CBC/ISO10126Padding (128-bit, 192-bit, and 256-bit keys) can wrap DES, AES, and ECC keys
- AES/CTR/NoPadding (128-bit, 192-bit, and 256-bit keys) can wrap DES, AES, and ECC keys
- AesWrap/NONE/NoPadding (wrapping defined by NIST in the Special Publication 800-38F can wrap AES keys

## Unwrapping

Cipher.UNWRAP_MODE supports the following algorithms:

- DESede/ECB/NoPadding
- DESede/ECB/PKCS5Padding
- DESede/ECB/ISO10126Padding
- DESede/CBC/NoPadding
- DESede/CBC/PKCS5Padding
- DESede/CBC/ISO10126Padding
- AES/ECB/NoPadding (128-bit, 192-bit, and 256-bit keys) can unwrap DES, AES, and ECC keys
- AES/ECB/PKCS5Padding (128-bit, 192-bit, and 256-bit keys) can unwrap DES, AES, RSA, and ECC keys
- AES/ECB/ISO10126Padding (128-bit, 192-bit, and 256-bit keys) can unwrap DES, AES, RSA, and ECC keys
- AES/CBC/NoPadding (128-bit, 192-bit, and 256-bit keys) can unwrap DES, AES, and ECC keys
- AES/CBC/PKCS5Padding (128-bit, 192-bit, and 256-bit keys) can unwrap DES, AES, RSA, 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 4096-bit keys) can unwrap DES and AES keys
- RSA/NONE/PKCS1Padding (1024-bit to 4096-bit keys) can unwrap DES and AES keys
- RSA/NONE/OAEPWithMD5AndMGF1Padding (1024-bit to 4096-bit keys) can unwrap DES and AES keys
- RSA/NONE/OAEPWithSHA1AndMGF1Padding (1024-bit to 4096-bit keys) can unwrap DES and AES keys
- RSA/NONE/OAEPWithSHA256AndMGF1Padding (1024-bit to 4096-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 (unwrapping defined by NIST in the Special Publication 800-38F can unwrap AES keys

**Important**

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.

When unwrapping DESede keys, the largest possible DESede key (192-bit or 128-bit) will be be returned. If you specifically need the two-key (128-bit) Triple DES keying option, specify the wrapped key algorithm as "DESede2".

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

### 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.

### 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.

### Default Values

The following are the default values for all cipher modes:

- `DES` defaults to `DES/ECB/PKCS5Padding`
- `DESede` defaults to `DESede/ECB/PKCS5Padding`
- `AES` defaults to `AES/ECB/PKCS5Padding`
- `RSA` defaults to `RSA/NONE/Pkcs1Padding`
- `ECCElGamal` defaults to `EccElGamal/NONE/NoPadding`
- `AesWrap` defaults to `AesWrap/NONE/NoPadding`

## 10.5.2 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.

### SkbEcParameterSpec

This class is used to provide initialization parameters for ECC keys. Its constructor is defined as follows:

```
SkbEcParameterSpec(int keySize)
```

where `keySize` is one of the following:

- 160 for secp160r1, which is the 160-bit prime curve recommended by SECG
- 192 for P-192, which is the 192-bit prime curve recommended by NIST (also known as secp192r1 and prime192v1)
- 224 for P-224, which is the 224-bit prime curve recommended by NIST (also known as secp224r1)

- 256 for P-256, which is the 256-bit prime curve recommended by NIST (also known as secp256r1 and prime256v1)
- 384 for P-384, which is the 384-bit prime curve recommended by NIST (also known as secp384r1)
- 521 for P-521, which is the 521-bit prime curve recommended by NIST (also known as secp521r1)

Custom ECC curves are not supported by the SKB JCA Provider.

### 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.

## 10.5.3 KeyStore

When working with the SKB-specific implementation of the `KeyStore` class, you should keep in mind the following details:

- When initializing the `KeyStore` object, the `SkbProvider.KEYSTORE_TYPE` constant should be used as the type.

  The following is the recommended way for initializing the object:

  ```
  KeyStore keyStore = KeyStore.getInstance(SkbProvider.KEYSTORE_TYPE,
                                           SkbProvider.PROVIDER_NAME);
  ```

- Within the `KeyStore` object, all keys are always stored in the SKB-protected format (the format used for keys exported out of SKB), and, additionally, they are encrypted with a password.
- When a key is retrieved from the `KeyStore` object, the key is always provided in the SKB-protected format.
- If importing of plain keys is enabled in your SKB library, you can also store plain keys in the `KeyStore` object.

  Within the `KeyStore` object, plain keys are always converted to the SKB-protected format, and you can never retrieve them back out of the `KeyStore` object in the original plain format.

  If importing of plain keys is not enabled, an exception will be thrown if you attempt to store a plain key.

- The SKB-specific KeyStore implementation supports storing certificates, however, it relies on the default Provider (for CertificateFactory) to serialize and deserialize them.

  If the Provider is not available, an exception will be thrown if you attempt to store a certificate.

## 10.5.4 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)
  ```

  where 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 SkbEcParameterSpec.

  ecPrivateKey must be the SkbEcPrivateKey object generated using either KeyFactory or KeyPairGenerator.

- PKCS8EncodedKeySpec is used to load plain RSA keys (encoded using 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.

**Important**

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

## 10.5.5 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)
  ```

  where 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.

**Important**

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

## 10.5.6 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

## 10.5.7 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.

For the DH algorithm, you have to use the SkbDhParameterSpec object to provide input parameters.

**Important**

EC and ECDH are not compatible formats.

## 10.5.8 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.

**Important**

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

## 10.5.9 Mac

Mac supports the following algorithms:

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

## 10.5.10 KeyAgreement

KeyAgreement supports the ECDH and DH algorithms. For these algorithms, you have to use the ECDH or DH keys generated by KeyPairGenerator.

**Important**

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.

ECDH format is not compatible with the EC format.

### 10.5.11 MessageDigest

`MessageDigest` supports the following algorithms:

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

## 10.6 Java API Example

There are SKB Java API example projects provided in the `Examples/JCA` directory of the SKB package. For information on building and running these projects, please read the accompanying `README.txt` files.

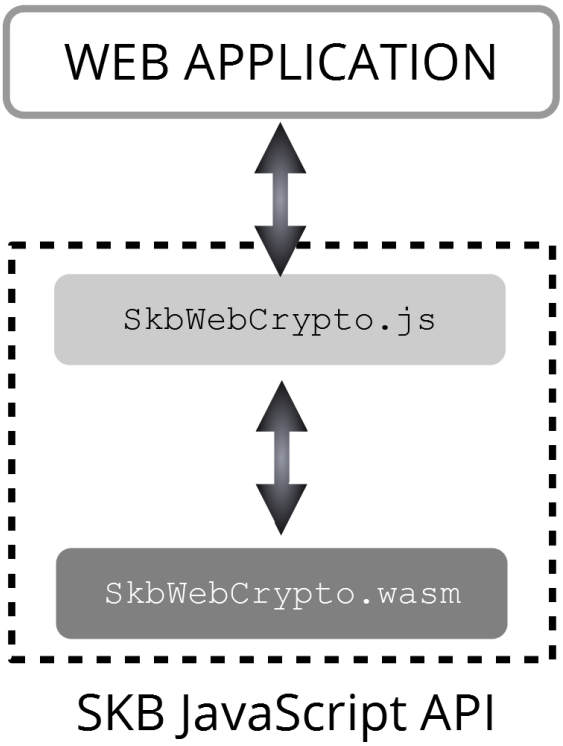
# 11. JavaScript API

This chapter describes how to use the SKB JavaScript library.

## 11.1 Overview of the SKB JavaScript Library

The SKB JavaScript library is an SKB-based implementation of the Web Cryptography API specification, which is built into most modern browsers.

The SKB JavaScript library is a natural means for using SKB-protected cryptographic algorithms and functions within a Web application. The library is accessible through a JavaScript API exposed in the `SkbWebCrypto.js` file, which internally employs a WebAssembly module contained within the `SkbWebCrypto.wasm` file.




The `SkbWebCrypto.wasm` file contains all the low-level cryptographic functions and data structures that have the secure properties of SKB. All keys processed by the SKB JavaScript library are never revealed in cleartext. The `SkbWebCrypto.wasm` file is not intended to be used directly by your Web application; instead, all interaction with the SKB JavaScript API must be done through the `SkbWebCrypto.js` interface.

These files will be present in the SKB package only if you requested the WebAssembly target platform when ordering SKB.

**Important**

This chapter assumes you have good knowledge about the standard Web Cryptography API.

## 11.2 Adding the SKB JavaScript Library to Your Application

To use the SKB JavaScript API in your application, you must execute the following main steps:

1. Copy the `SkbWebCrypto.js` and `SkbWebCrypto.wasm` files from the `lib/wasm32-unknown-unknown` directory of the SKB installation to the appropriate place where they will be accessible by your Web application.

   Both files must be placed in the same directory.

2. In those HTML pages where you need to execute any of the cryptographic functions provided by the SKB JavaScript API, include the `SkbWebCrypto.js` file, for example as follows:

   ```
   <script src="SkbWebCrypto.js"></script>
   ```

3. Within the JavaScript code, before using any of the SKB JavaScript API methods, initialize the `SkbCrypto` object, for example as follows:

   ```
   window.skb_crypto = new SkbCrypto();
   ```

   The interface of the `SkbCrypto` class is the same as the one of the standard Web Cryptography API, which is described here:

   https://www.w3.org/TR/WebCryptoAPI/#crypto-interface

   This means that the experience of using the SKB JavaScript API is essentially the same as using the Web Cryptography API provided by the browser. In connection with this, it is recommended to set the initialized `SkbCrypto` object as a property of the `windows` object (although it is not mandatory), because it minimizes usage differences between the SKB JavaScript API and the standard Web Cryptography API.

## 11.3 Technical Details

The SKB JavaScript API works almost exactly like the standard Web Cryptography API. However, there are a few differences:

- You must invoke SKB JavaScript API functions via the SKB-specific `SkbCrypto` object instead of the standard `window.crypto` interface.

  For example, generating an AES key using the standard Web Cryptography API would look something like this:

  ```
  let key = await window.crypto.subtle.generateKey(
      {name: "AES-GCM", length: 128},
      false,
  ```

```
    ["encrypt", "decrypt"]
);
```

With the SKB JavaScript API, the same operation would look like this:

```
let key = await window.skb_crypto.subtle.generateKey(
    {name: "AES-GCM", length: 128},
    false,
    ["encrypt", "decrypt"]
);
```

- The SKB JavaScript library uses a different set of key formats (values of the `KeyFormat` enumeration).

  The standard Web Cryptography API recognizes the following key format values:

  - `raw`
  - `pkcs8`
  - `spki`
  - `jwk`

  However, the SKB JavaScript API uses the following values:

  - `skb-key`: AES or HMAC key in the SKB-protected format
  - `skb-public`: Public RSA key in the SKB-protected format
  - `skb-private`: Private RSA key in the SKB-protected format

- If you SKB package contains the Key Embedding Tool, you can set the export key using the `setExportKey` method as follows:

  ```
  window.skb_crypto.subtle.setExportKey(data)
  ```

  The input data must be an instance of `ArrayBuffer` or `ArrayBufferView`. You may use the Key Embedding Tool to get the input data in the right JavaScript format (the contents of the `.js` output file). The method will return a `Promise` object. If an error occurs during execution, the method will throw `OperationError`.

- The `exportKey` method exports keys in the SKB-protected format using the persistent export type. If your SKB package contains the Key Embedding Tool, the method `setExportKey` must be called first.

- The `importKey` method imports keys in the SKB-protected format, which can be obtained using the following methods:

  - exporting a key using the `exportKey` method
  - preparing a key using the Key Export Tool

- To set the device ID, call the `setDeviceId` method as follows:

  ```
  ```

```
window.skb_crypto.subtle.setDeviceId(data)
```

Similarly to the `setExportKey` method, the input must be an instance of `ArrayBuffer` or `ArrayBufferView`, the method will return a `Promise` object, and, in case of an error, it will throw `OperationError`.

- In addition to the default (high-security) AES implementation, the "AES-CTR", "AES-CBC", and "AES-GCM" encryption and decryption algorithms also support the high-speed AES implementations, which can be configured as follows:
  - To enable the `SKB_CIPHER_FLAG_HIGH_SPEED` AES implementation, in the corresponding `Algorithm` dictionary, set the `high_speed` member to `true`, for example as follows:

```
let AES_CBC = {
    name: "AES-CBC",
    iv: new Uint8Array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]),
    high_speed: true,
}
window.skb_crypto.subtle.encrypt(AES_CBC, key, data);
```

  - To enable the `SKB_CIPHER_FLAG_BALANCED` AES implementation, in the corresponding `Algorithm` dictionary, set the `balanced` member to `true`, for example as follows:

```
let AES_CBC = {
    name: "AES-CBC",
    iv: new Uint8Array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]),
    balanced: true,
}
window.skb_crypto.subtle.encrypt(AES_CBC, key, data);
```

**Important**

You must not set the `high_speed` and `balanced` members to `true` at the same time.

## 11.4 Algorithms Supported by the SKB JavaScript Library

The following table shows which algorithms and methods are currently supported by the SKB JavaScript API.

| | encrypt | decrypt | sign | verify | digest | generateKey | deriveKey | deriveBits | importKey | exportKey | wrapKey | unwrapKey |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RSASSA-PKCS1-v1_5 | | | ✔ | ✔ | | ✔ | | | ✔ | ✔ | | |
| RSA-PSS | | | ✔ | ✔ | | ✔ | | | ✔ | ✔ | | |
| RSA-OAEP | ✔ | ✔ | | | | ✔ | | | ✔ | ✔ | ✔ | ✔ |
| ECDSA | | | ✔ | ✔ | | ✔ | | | ✔ | ✔ | | |
| ECDH | | | | | | | ✔ | ✔ | ✔ | ✔ | | |
| AES-CTR | ✔ | ✔ | | | | ✔ | | | ✔ | ✔ | ✔ | ✔ |
| AES-CBC | ✔ | ✔ | | | | ✔ | | | ✔ | ✔ | ✔ | ✔ |
| AES-GCM | ✔ | ✔ | | | | ✔ | | | ✔ | ✔ | ✔ | ✔ |
| AES-KW | | | | | | ✔ | | | ✔ | ✔ | ✔ | ✔ |
| HMAC | | | ✔ | ✔ | | ✔ | | | ✔ | ✔ | | |
| SHA-1 | | | | | ✔ | | | | | | | |
| SHA-256 | | | | | ✔ | | | | | | | |
| SHA-384 | | | | | ✔ | | | | | | | |
| SHA-512 | | | | | ✔ | | | | | | | |
| HKDF | | | | | | | ✔ | ✔ | ✔ | | | |
| PBKDF2 | | | | | | | ✔ | ✔ | ✔ | | | |

Cells containing a check mark on a green background indicate algorithms and methods supported by the SKB JavaScript API; cells containing a check mark on a white background indicate algorithms and methods that are supported by the standard Web Cryptography API, but not the SKB JavaScript API.

Please refer to https://www.w3.org/TR/WebCryptoAPI/#algorithm-overview for technical details on the data displayed in the table above.

## 11.5 Limitations

The SKB JavaScript library has the following limitations when compared to the standard Web Cryptography API:

- When using AES in the CTR mode, the counter length must be a multiple of 8.
- When using AES in the GCM mode, the initialization vector must be 12 bytes long.
- When using HMAC key generation, the key length must be a multiple of 8.
- When an RSA key is imported using the `importKey` method, the resulting `CryptoKey` object will not contain the public exponent in plain. This applies to all RSA algorithms supported by the SKB JavaScript library.
- RSA wrapping and unwrapping with OAEP padding only support SHA-1 and SHA-256.
- The `label` property of the `RsaOaepParams` dictionary is not supported (it will be ignored).

  For information on the `label` property, see https://www.w3.org/TR/WebCryptoAPI/#dfn-RsaOaepParams-label.
- The `getRandomValues` method is not supported.

## 11.6 JavaScript API Example

There is an SKB JavaScript API example project provided in the `Examples/WebCrypto` directory of the SKB package. For information on building and running this example, please read the accompanying `README.txt` file.

# 12. TLS Support

This chapter describes how to implement the SKB Transport Layer Security (TLS) protocol using SKB.

## 12.1 Available Extensions

SKB provides the following two alternative extensions for implementing the TLS protocol:

Here are the two available extensions:

- SKB provider for OpenSSL 3

  The SKB provider will protect the private keys used for peer authentication, and it is the recommended implementation for ensuring secure TLS communication. To use this implementation, you have to be familiar with OpenSSL 3.

- TLS Library that exposes a few high-level methods that take care of the entire TLS communication process and protects all keys involved in the protocol.

  **Important**

  The TLS Library is deprecated and will be removed in future releases of SKB.

In both cases, the underlying design is based on the secure properties of SKB.

## 12.2 Using the SKB Provider for OpenSSL

OpenSSL 3 has a concept of providers, which are units of code that provide implementations for the operations offered by the OpenSSL API. SKB takes advantage of this design feature to provide a secure white-box cryptography implementation for some of the algorithms involved in the TLS protocol. More specifically, the SKB implementation protects the private client and/or server signing keys at all times, even when in use.

Once the SKB provider is initialized, the subsequent invocations of the OpenSSL API will use the secure implementation provided by SKB where applicable. The protected keys will never be revealed in the clear. For example, even if you call the `PEM_write_PrivateKey` function to write a private key to a file, the file will contain the key in protected form that can only be loaded by the SKB provider.

The SKB provider is delivered in two forms:

| Form | Purpose |
|---|---|
| Static library | This form of the library is intended to be linked with the final application and deployed on open and potentially insecure target platforms. The only dependencies of the library are OpenSSL, SKB, and the C++ Standard Library. The library file is named `libSkb0penSslProvider.a` or `Skb0penSslProvider.lib` (depending on the target platform), and it is available in the `lib` directory of the SKB package. |
| Dynamic library | This form of the library is intended to be used with the `openssl` command-line utility on your development platforms. It can be used for testing purposes and for preparation of keys. The library file is named `Skb0penSslProvider.dll` (on Windows), `libSkb0penSslProvider.so` (on Linux), and `libSkb0penSslProvider.dylib` on (macOS), and it is available in the `bin` directory of the SKB package. |

## 12.2.1 Supported Cryptographic Operations

Currently, the SKB provider supports the following operations of OpenSSL:

- key generation for ECDSA (standard NIST curves) and Ed25519 keys
- Creating ECDSA, Ed25519, and RSA signatures
- authentication to the peer in TLS communication using an SKB-protected private key

## 12.2.2 Platform Support

The static library of the SKB provider is available for the following target platforms:

- Android (x86, x86_64, 32-bit/64-bit ARM) built with Android NDK r23
- iOS (32-bit/64-bit ARM) built with Xcode 12.4
- glibc/Linux (x86_64) built with GCC 7.5
- Windows (x86_64) built separately with Visual Studio 2015 and 2017
- macOS (x86_64, 64-bit ARM) built with Xcode 12.4

The dynamic library of the SKB provider is available for the following development platforms:

- glibc/Linux (x86_64) built with GCC 7.5
- Windows (x86_64) built with Visual Studio 2017
- macOS (x86_64) built with Xcode 12.4

**Important**

Support of the following platforms is experimental and may not fully work as intended:

- iOS for 32-bit ARM
- macOS for 64-bit ARM

## 12.2.3 Working with the SKB Provider

To use the SKB provider, proceed as follows:

- To use the static library of the SKB provider, construct your code according to the following pattern:

```
#include <openssl/provider.h>
#include "SkbOpenSslProvider.h"

// ...

// Load the SKB provider
OSSL_PROVIDER_add_builtin(NULL, "skbOpenSslProvider", SKB_OSSL_provider_init);
OSSL_PROVIDER* skb_provider = OSSL_PROVIDER_load(NULL, "skbOpenSslProvider");

// Afterwards, load also the default provider, which is required by the SKB provider
OSSL_PROVIDER* default_provider = OSSL_PROVIDER_load(NULL, "default");

// ...

// Work with OpenSSL in the usual manner

// ...

// When no longer needed, unload the SKB provider and the default provider
OSSL_PROVIDER_unload(skb_provider);
OSSL_PROVIDER_unload(default_provider);
```

- To use the dynamic library of the SKB provider, add the following options when running the `openssl` command-line utility:
  - `-provider-path «path to the library»`
  - `-provider SkbOpenSslProvider`
  - `-provider default`

Here is an example of the `openssl` invocation

```
openssl genpkey -algorithm ED25519 -out server.key
-provider-path bin\Windows -provider SkbOpenSslProvider -provider default
```

The `default` provider must be loaded as well, because the SKB provider depends on it.

**Important**

If your SKB package contains the Key Embedding Tool, the SKB provider will only support keys exported using the cross-engine export format, and the provider itself will save generated keys in this format.

## 12.2.4 SKB Provider Functions

This section describes the functions exported by the SKB provider. These functions are defined in the `SkbOpenSslProvider.h` file, which is located in the `Include` directory of the SKB package.

### SKB_OSSL_provider_init

This is the SKB-provided function of type `OSSL_provider_init_fn` that must be passed to the `OSSL_PROVIDER_add_builtin` function to register the SKB provider in OpenSSL.

### SKB_OSSL_provider_export

This function exports a private key from the EVP_PKEY object using the SKB exported data format, which can then be loaded as an `SKB_SecureData` object by the `SKB_Engine_CreateDataFromExported` method. This may be necessary if you intend to work with the protected OpenSSL keys in the native SKB API.

The possible return values of this function are of the same type as used by the methods of the Native API.

The function is declared as follows:

```
SKB_Result
SKB_OSSL_provider_export(const EVP_PKEY*  pkey,
                         SKB_ExportTarget target,
                         SKB_Byte**       exported
                         SKB_Size*        exported_size);
```

The following are the parameters used:

- pkey

  The EVP_PKEY object to be exported.

- `target`

Export type to be used. In most cases, this must be SKB_EXPORT_TARGET_PERSISTENT. The only exception is when your SKB package contains the Key Embedding Tool, in which case the value must be SKB_EXPORT_TARGET_CROSS_ENGINE.

- exported

  Address of the pointer that will be set by the SKB provider to point to a newly allocated memory buffer containing the exported data. This buffer must be released by calling the OPENSSL_free function when no longer needed.

- exported_size

  Pointer to a variable that will hold the size of the allocated exported buffer in bytes.

## SKB_OSSL_provider_import

This function imports data that was previously exported using the SKB_OSSL_provider_export function, which is described in the previous section, or using the SKB_SecureData_Export method. This may be necessary if you intend to use the protected SKB keys within OpenSSL.

The possible return values of this function are of the same type as used by the methods of the Native API.

The function is declared as follows:

```
SKB_Result
SKB_OSSL_provider_import(const SKB_Byte* exported,
                         SKB_Size        exported_size,
                         const char*     key_type_name,
                         EVP_PKEY**      pkey);
```

The following are the parameters used:

- exported

  Pointer to the memory buffer containing the exported data.

- exported_size

  Size of the exported buffer in bytes.

- key_type_name

  String that will be passed to OpenSSL to specify the key type.

  Currently, this must be either "EC" or "ED25519", depending on the type of the key you exported.

- pkey

  Address of a pointer to the EVP_PKEY object that will be created by this function from the imported data. This object must be released by calling the EVP_PKEY_free function when no longer needed.

### 12.2.5 Provider Examples

There are several example projects in the `Examples/SkbOpenSslProvider` directory of the SKB package, which demonstrate the main features of the SKB provider for OpenSSL. For information on building and running these projects, please read the accompanying `README.txt` files.

## 12.3 Using the TLS Library

The purpose of the TLS Library is to protect keys used for TLS communication. The underlying implementation of TLS methods and algorithms is based on the secure properties of SKB, which ensure that the private key that is used for authentication, as well as the session key used for traffic encryption are protected against key extraction and tampering attacks.

The SKB TLS is a static library that must be linked with the final application. The library is named `libSkbTls.a` or `SkbTls.lib` depending on the target platform, and it is located in the `lib` directory of the SKB package. The API of the TLS Library is defined in the `SkbTls.h` file, which is provided in the `Include` directory of the SKB package. The API provides a small set of high-level methods that take care of the entire TLS communication process, such as opening and closing a connection, and reading and writing data over the TLS protocol. The `SkbTls.h` header includes the `SkbSecureKeyBox.h` header, which contains the main Native API.

**Important**

The TLS Library is deprecated and will be removed in future releases of SKB.

### 12.3.1 Supported TLS Version and Cipher Suites

The TLS Library only supports TLS 1.2 with the following cipher suites:

- `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256` (0xc02b)

  For this cipher suite, only the following variations are supported:

  - ECDHE with the P-256 curve and ECDSA with the P-256 curve
  - ECDHE with Curve25519 (also known as X25519) and Ed25519
- `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384` (0xc02c)
- `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` (0xc02f)
- `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384` (0xc030)

## 12.3.2 Dependencies

The TLS Library depends on the underlying operating system to provide the basic networking (sockets) functionality. On Windows platforms, you must link the `ws2_32` library into your program; on other platforms, the sockets functionality is part of the C standard library. Applications using the TLS Library on Linux must be explicitly linked with the `pthread` library.

## 12.3.3 Methods

This section describes the methods provided by the SKB TLS API.

### SKB_TLS_OpenConnection

This method opens a connection to the server (in case of a client) or starts accepting connections from clients (in case of a server) based on TLS connection parameters provided as input. When the communication channel is no longer needed, it must be closed by calling the SKB_TLS_CloseConnection method.

The method is declared as follows:

```
SKB_Result
SKB_TLS_OpenConnection(const SKB_TLS_ConnectionParameters* parameters,
                       SKB_TLS_ConnectionType              connection_type,
                       SKB_TLS_Connection**                connection);
```

The following are the parameters used:

- `parameters`

  Pointer to the SKB_TLS_ConnectionParameters structure that contains connection details.

- `connection_type`

  Specifies whether a client-side or server-side connection must be established. Available connection types are defined in the SKB_TLS_ConnectionType enumeration.

- `connection`

  Address of a pointer to the `SKB_TLS_Connection` object that will be created by this method. You must pass this object to all subsequent calls of TLS API methods.

### SKB_TLS_OpenConnectionOnSocket

Same as the SKB_TLS_OpenConnection method, except communication will happen through an existing socket descriptor, which is passed as a parameter. This method is intended for advanced users only.

The method is declared as follows:

```
```

```
SKB_Result
SKB_TLS_OpenConnectionOnSocket(const SKB_TLS_ConnectionParameters* parameters,
                               SKB_TLS_ConnectionType              connection_type,
                               int                                 socket_fd,
                               SKB_TLS_Connection**                connection);
```

The following are the parameters used:

- `parameters`

  Pointer to the SKB_TLS_ConnectionParameters structure that contains connection details.

- `connection_type`

  Specifies whether a client-side or server-side connection must be established. Available connection types are defined in the SKB_TLS_ConnectionType enumeration.

- `socket_fd`

  Existing socket descriptor, which was created elsewhere and through which communication will take place.

- `connection`

  Address of a pointer to the SKB_TLS_Connection object that will be created by this method. You must pass this object to all subsequent calls of TLS API methods.

## SKB_TLS_Read

This method reads a buffer of data from the peer through an open TLS connection. The return value of the method is the number of bytes actually received, or 0 if an error occurred.

The method is declared as follows:

```
SKB_Size
SKB_TLS_Read(SKB_TLS_Connection* connection,
             SKB_Byte*           buffer,
             SKB_Size            size,
             SKB_Result*         result);
```

The following are the parameters used:

- `connection`

  Address of a pointer to the SKB_TLS_Connection object created using the SKB_TLS_OpenConnection method.

- `buffer`

  Pointer to a buffer where the read data must be written.

- `size`

  Size of `buffer` in bytes.

  If the total size of incoming data is larger than `buffer`, only the number of bytes specified in the `size` parameter will be read. The value of the `size` parameter may be greater than the total amount of data sent by the peer.

- `result`

  Pointer to an `SKB_Result` variable where the method will store the result code. The parameter may also be NULL, in which case the method will not set the return code.

### SKB_TLS_Write

This method sends a buffer of data to the peer through an open TLS connection. The return value of the method is the number of bytes actually sent, or 0 if an error occurred.

The method is declared as follows:

```
SKB_Size
SKB_TLS_Write(SKB_TLS_Connection* connection,
              const SKB_Byte*     buffer,
              SKB_Size            size,
              SKB_Result*         result);
```

The following are the parameters used:

- `connection`

  Address of a pointer to the `SKB_TLS_Connection` object created using the SKB_TLS_OpenConnection method.

- `buffer`

  Pointer to a buffer of data to be sent to the peer.

- `size`

  Number of bytes to be sent to the peer. If the return value is less than this number, it means that not all data has been sent. In this case, you should call `SKB_TLS_Write` again and pass the pointer to the remaining data along with the size of the remaining data until all data is sent.

- `result`

  Pointer to an `SKB_Result` variable where the method will store the result code. The parameter may also be NULL, in which case the method will not set the return code.

## SKB_TLS_GetPeerCertASN1

This method returns a pointer to the DER-encoded ASN.1 peer certificate and its size. In case of a server connection, the returned value can be NULL if the client is not configured to send the certificate during the handshake operation.

The method is declared as follows:

```
SKB_Size
SKB_TLS_GetPeerCertASN1(SKB_TLS_Connection* connection,
                        const SKB_Byte**    cert_ptr,
                        SKB_Size*           cert_size);
```

The following are the parameters used:

- `connection`

  Address of a pointer to the `SKB_TLS_Connection` object created using the `SKB_TLS_OpenConnection` method.

- `cert_ptr`

  Address of a pointer to where the pointer to the peer certificate data buffer will be stored.

- `size`

  Pointer to an `SKB_Size` variable where the method will store the size of the peer certificate data buffer.

## SKB_TLS_CloseConnection

This method sends the `close_notify` alert to the peer and closes the TLS connection. You must call it when communication with the peer is complete.

The method is declared as follows:

```
SKB_Result
SKB_TLS_CloseConnection(SKB_TLS_Connection* connection);
```

The `connection` parameter is a pointer to the `SKB_TLS_Connection` object created using the `SKB_TLS_OpenConnection` method. Once the connection is closed, the pointer will become invalid.

## SKB_TLS_CreateSelfSignedCertificate

This method creates a self-signed certificate using a protected private key.

The method is declared as follows:

```
SKB_Result
SKB_TLS_CreateSelfSignedCertificate(
    SKB_SignatureAlgorithm                      signature_algorithm,
```

```
    SKB_SecureData*                                private_key,
    const SKB_TLS_CertificateSigningRequestInfo* certificate_signing_request_info,
    SKB_Size                                       expiration_days,
    int                                            start_date_offset,
    SKB_TLS_SubjectType                            subject_type,
    SKB_Byte*                                      certificate,
    SKB_Size*                                      certificate_size
);
```

The following are the parameters used:

- `signature_algorithm`

  Algorithm to be used for signing the certificate. Available algorithms are defined in the SKB_SignatureAlgorithm enumeration.

  Currently, the only supported algorithm for this method is `SKB_SIGNATURE_ALGORITHM_ED25519`.

- `private_key`

  The private key to be used for signing the certificate, provided as the `SKB_SecureData` object. Its type must match the specified signature algorithm.

- `certificate_signing_request_info`

  Pointer to the SKB_TLS_CertificateSigningRequestInfo structure containing the necessary certificate signing request data.

- `expiration_days`

  Number of days the certificate must be valid for.

  The value must be such that the current date and time plus `expiration_days` must be before the end of the year 2049.

- `start_date_offset`

  Offset in seconds that will be added to the current date and time to obtain the start of the certificate validity period.

  This value may be positive, negative, or zero.

- `subject_type`

  Specifies whether a Certificate Authority (CA) certificate or an end-entity certificate must be created. The available values are defined in the SKB_TLS_SubjectType enumeration.

- `certificate`

  This parameter is either NULL or a pointer to the memory buffer where the generated certificate is to be stored.

If this parameter is NULL, the method simply returns, in `certificate_size`, the number of bytes that would be sufficient to hold the output, and returns SKB_TLS_SUCCESS.

If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there (in DER-encoded format as defined in RFC 5280), sets `certificate_size` to the exact number of bytes stored, and returns SKB_TLS_SUCCESS. If the buffer is not large enough, then the method sets `certificate_size` to the number of bytes that would be sufficient, and returns SKB_TLS_ERROR_BUFFER_TOO_SMALL.

- `certificate_size`

  Pointer to a variable that holds the size of the memory buffer in bytes where the output data is to be stored. For more details, see the description of the `certificate` parameter.

## 12.3.4 Structures

This section describes the structures used by the SKB TLS API.

### SKB_TLS_ConnectionParameters

This structure is used to store various parameters required to establish a TLS connection. A pointer to this structure must be passed to the SKB_TLS_OpenConnection method.

The structure is declared as follows:

```
typedef struct {
    const char*                hostname;
    SKB_Size                   port;
    SKB_KeyAgreementAlgorithm  key_agreement_algorithm;
    SKB_TLS_CertificatePinType peer_certificate_pin_type;
    const SKB_Byte*            peer_certificate_pin;
    SKB_Size                   peer_certificate_pin_size;
    SKB_SignatureAlgorithm     signature_algorithm;
    SKB_SecureData*            private_key;
    const SKB_Byte*            certificate;
    SKB_Size                   certificate_size;
    SKB_CipherAlgorithm        session_encryption_algorithm;
    SKB_DigestAlgorithm        hash_function;
    SKB_Size                   connection_timeout;
    SKB_Size                   read_write_timeout;
} SKB_TLS_ConnectionParameters;
```

The following are the properties used:

- `hostname`

In case of the SKB_TLS_CLIENT connection type, this property must hold the name or IP address of the server, such as "example.com" or "192.168.1.1".

In case of the SKB_TLS_SERVER type, the value of this property must be NULL.

- `port`

  Connection port to be used.

- `key_agreement_algorithm`

  Key agreement algorithm to be used in TLS communication. Available algorithms are defined in the SKB_KeyAgeementAlgorithm enumeration.

  Please note the following limitations:

  - The only supported key agreement algorithms for TLS communication are SKB_KEY_AGREEMENT_ALGORITHM_X25519 and SKB_KEY_AGREEMENT_ALGORITHM_ECDH.
  - The SKB_KEY_AGREEMENT_ALGORITHM_X25519 algorithm can only be used together with the SKB_SIGNATURE_ALGORITHM_ED25519 signature algorithm.
  - The SKB_KEY_AGREEMENT_ALGORITHM_ECDH algorithm can only be used together with the SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256 and SKB_SIGNATURE_ALGORITHM_RSA_SHA256 signature algorithms.

- `peer_certificate_pin_type`

  Certificate pinning method used for verifying authenticity of the peer. Available pinning methods are defined in the SKB_TLS_CertificatePinType enumeration.

  Currently, only the following methods are supported:

  - SKB_TLS_CERTIFICATE_PIN_HASH_CERTIFICATE
  - SKB_TLS_CERTIFICATE_PIN_HASH_TBS_CERTIFICATE
  - SKB_TLS_CERTIFICATE_PIN_NONE

- `peer_certificate_pin`

  Pointer to a buffer containing data used for the selected certificate pinning method:

  - For the SKB_TLS_CERTIFICATE_PIN_HASH_CERTIFICATE method, this buffer must contain a hash value of the whole trusted certificate.

    The hash function used to calculate the hash value must match the one specified in the `hash_function` parameter.

  - For the SKB_TLS_CERTIFICATE_PIN_HASH_TBS_CERTIFICATE method, this buffer must contain a hash value of the TBS`Certificate` part of a trusted certificate.

    The hash function used to calculate the hash value must match the one specified in the `hash_function` parameter.

- For the SKB_TLS_CERTIFICATE_PIN_NONE method, this buffer is not used.

For more information on certificate pinning methods, see SKB_TLS_CertificatePinType.

- peer_certificate_pin_size

  Size of the peer_certificate_pin buffer in bytes.

  For the SKB_TLS_CERTIFICATE_PIN_NONE method, this parameter is not used.

- signature_algorithm

  Signature algorithm to be used in TLS communication. Available algorithms are defined in the SKB_SignatureAlgorithm enumeration.

  Please note the following limitations:

  - The only supported signature algorithms for TLS communication are SKB_SIGNATURE_ALGORITHM_ED25519, SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256, and SKB_SIGNATURE_ALGORITHM_RSA_SHA256.
  - The SKB_SIGNATURE_ALGORITHM_ED25519 algorithm can only be used together with the SKB_KEY_AGREEMENT_ALGORITHM_X25519 key agreement algorithm.
  - The SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256 and SKB_SIGNATURE_ALGORITHM_RSA_SHA256 algorithms can only be used together with the SKB_KEY_AGREEMENT_ALGORITHM_ECDH key agreement algorithm.

- private_key

  Caller's private key represented as the SKB_SecureData object. The key must be suitable for authentication using the algorithm specified in the signature_algorithm parameter. In case of creating a client connection, this parameter (as well as the certificate and certificate_size parameters) can be NULL.

- certificate

  Pointer to a buffer containing a certificate or certificate chain for the private_key to be supplied to the peer for authentication. In case of creating a client connection, this parameter (as well as the private_key and certificate_size parameters) can be NULL.

- certificate_size

  Size of the certificate buffer in bytes. If the certificate parameter is not NULL, the size must be greater than 0.

- session_encryption_algorithm

  Session encryption algorithm to be used in TLS communication. Available algorithms are defined in the SKB_CipherAlgorithm enumeration.

  Currently, only the following encryption algorithms are supported for TLS communication:

  - SKB_CIPHER_ALGORITHM_AES_128_GCM

- SKB_CIPHER_ALGORITHM_AES_256_GCM

- hash_function

  Hash function to be used in TLS communication. Available functions are defined in the SKB_DigestAlgorithm enumeration.

  Please note the following limitations:

  - The only supported algorithms for TLS communication are SKB_DIGEST_ALGORITHM_SHA256 and SKB_DIGEST_ALGORITHM_SHA384.
  - The SKB_DIGEST_ALGORITHM_SHA256 algorithm can only be used in conjuncture with the SKB_CIPHER_ALGORITHM_AES_128_GCM encryption algorithm.
  - The SKB_DIGEST_ALGORITHM_SHA384 algorithm can only be used in conjuncture with the SKB_CIPHER_ALGORITHM_AES_256_GCM encryption algorithm.

- connection_timeout

  Timeout in milliseconds for establishing the connection.

- read_write_timeout

  Timeout in milliseconds for performing read and write operations.

  The value of the read_write_timeout parameter is shared by all opened connections, and the value used for establishing the last connection is used for all subsequent read and write calls on all connections.

## SKB_TLS_CertificateSigningRequestInfo

This structure contains certificate signing request data provided as input to the SKB_TLS_CreateSelfSignedCertificate method to create a self-signed certificate.

The structure is declared as follows:

```
typedef struct {
    const char* country;
    const char* state_or_province_name;
    const char* locality;
    const char* organization;
    const char* organizational_unit;
    const char* common_name;
} SKB_TLS_CertificateSigningRequestInfo;
```

The following are the properties used:

- country

  The two-letter ISO code for the country where your organization is located.

This is an optional property.

- `state_or_province_name`

  The state, province, or region where your organization is located.

  This is an optional property.

- `locality`

  The city where your organization is located.

  This is an optional property.

- `organization`

  The legal name of your organization.

  This is an optional property.

- `organizational_unit`

  The division of your organization handling the certificate.

  This is an optional property.

- `common_name`

  A fully qualified domain name of your server, device identifier, or other type of data that identifies the entity associated with the certificate.

  This is the only mandatory property.

## 12.3.5 Enumerations

This section describes the enumerations used by the SKB TLS API.

### SKB_TLS_ConnectionType

This enumeration holds connection types that can be passed to the SKB_TLS_OpenConnection method.

The enumeration is declared as follows:

```
typedef enum {
    SKB_TLS_CLIENT,
    SKB_TLS_SERVER
} SKB_TLS_ConnectionType;
```

The following are the values defined:

- `SKB_TLS_CLIENT`

  Creates a client-side connection, which attempts to connect to a server.

- SKB_TLS_SERVER

  Creates a server-side connection, which starts to accept connections from clients.

## SKB_TLS_CertificatePinType

This enumeration specifies the possible certificate pinning methods for verifying authenticity of the peer.

The enumeration is defined as follows:

```
typedef enum {
    SKB_TLS_CERTIFICATE_PIN_HASH_CERTIFICATE,
    SKB_TLS_CERTIFICATE_PIN_HASH_TBS_CERTIFICATE,
    SKB_TLS_CERTIFICATE_PIN_PUBLIC_KEY,
    SKB_TLS_CERTIFICATE_PIN_CERTIFICATE,
    SKB_TLS_CERTIFICATE_PIN_NONE
} SKB_TLS_CertificatePinType;
```

The following are the values defined:

- SKB_TLS_CERTIFICATE_PIN_HASH_CERTIFICATE

  With this method, a hash value of the whole peer certificate must be provided to the SKB_TLS_OpenConnection method. When establishing a connection, the method will traverse the peer-provided certificate chain until a matching value is found. The TLS connection will be opened only if a match is found and the certificate chain had valid signatures and expiration periods up until the matching value.

- SKB_TLS_CERTIFICATE_PIN_HASH_TBS_CERTIFICATE

  With this method, a hash value of the TBSCertificate part of the peer certificate must be provided to the SKB_TLS_OpenConnection method. When establishing a connection, the method will traverse the peer-provided certificate chain until a matching hash value is found. The TLS connection will be opened only if a match is found and the certificate chain had valid signatures and expiration periods up until the matching value.

- SKB_TLS_CERTIFICATE_PIN_PUBLIC_KEY

  This method is not yet implemented.

- SKB_TLS_CERTIFICATE_PIN_CERTIFICATE

  This method is not yet implemented.

- SKB_TLS_CERTIFICATE_PIN_NONE

  With this method, no pinning is used.

### SKB_TLS_SubjectType

This enumeration specifies connection types that can be passed to the SKB_TLS_CreateSelfSignedCertificate method.

The enumeration is declared as follows:

```
typedef enum {
    SKB_TLS_END_ENTITY,
    SKB_TLS_CA
} SKB_TLS_SubjectType;
```

The following are the values defined:

- SKB_TLS_END_ENTITY

  End-entity certificate, in which the basic constraints extension is omitted.

- SKB_TLS_CA

  Certificate Authority (CA) certificate, in which the basic constraints extension is added and the CA flag is set to TRUE.

## 12.3.6 Return Values

All SKB TLS API methods return a value of type SKB_Result to tell what the outcome of the execution was. All possible return values are declared using #define macros in the SkbTls.h file.

When a method completes successfully, the return value is SKB_TLS_SUCCESS; in all other cases the returned value signifies an error. The following return values are used by the SKB TLS API:

- SKB_TLS_SUCCESS (0)

  The method completed successfully.

- SKB_TLS_ERROR_INTERNAL (-90001)

  An internal error occurred. Please consult with Zimperium for assistance.

- SKB_TLS_ERROR_INVALID_PARAMETERS (-90002)

  Invalid or insufficient data was passed to the method.

- SKB_TLS_ERROR_NOT_SUPPORTED (-90003)

  The specified algorithms are not supported by the SKB TLS API.

- SKB_TLS_ERROR_OUT_OF_RESOURCES (-90004)

  The device has run out of memory.

- SKB_TLS_ERROR_BUFFER_TOO_SMALL (-90005)

  The provided memory buffer was not large enough to contain the output.

- SKB_TLS_ERROR_UNSPECIFIED (-90006)

  A general error occurred.

- SKB_TLS_ERROR_PROTOCOL (-90007)

  An error with the TLS protocol, such as the handshake process, occurred.

- SKB_TLS_ERROR_CERTIFICATE (-90008)

  An error with parsing or validation of a certificate occurred.

- SKB_TLS_ERROR_CONNECTION (-90009)

  A general error related to the network connection occurred.

- SKB_TLS_ERROR_CONNECTION_TIMEOUT (-90010)

  Opening a connection or read request timed out.

  This is a more specific error than SKB_TLS_ERROR_CONNECTION.

- SKB_TLS_ERROR_CONNECTION_RESET (-90011)

  The connection was reset.

  This is a more specific error than SKB_TLS_ERROR_CONNECTION.

- SKB_TLS_ERROR_CONNECTION_PEER_SHUTDOWN (-90012)

  An error while reading data occurred because the peer has shut down.

  This is a more specific error than SKB_TLS_ERROR_CONNECTION.

- SKB_TLS_ERROR_CONNECTION_ZERO_RETURN (-90013)

  The SKB_TLS_Read method was called after receiving the `close_notify` alert from the peer.

  This is a more specific error than SKB_TLS_ERROR_CONNECTION.

## 12.3.7 TLS API Example

There is an SKB TLS API example project provided in the Examples/SkbTls directory of the SKB package. For information on building and running this project, please read the accompanying README.txt file.

# 13. Secure Database Library

This chapter describes how to use the Secure Database Library provided by SKB.

## 13.1 Overview of the Secure Database Library

The Secure Database Library is a static library that serves as a secure database engine built upon SQLite 3.37.2. Its main purpose is to provide the ability to create and use encrypted databases. All files produced by this library (such as the database files and SQLite temporary files) are encrypted using the AES cipher. This requires that whenever you initialize the Secure Database Library, you must provide a specially prepared database encryption key that is used for encrypting and decrypting the database.

In general, using the Secure Database Library is very similar to using the SQLite library, and all the files produced by the Secure Database Library are essentially SQLite files. The only main difference is that you have to use additional initialization and release functions before and after working with the Secure Database Library.

The Secure Database Library also provides a command-line utility named `SkbSecureDatabaseShell`, which is the SKB equivalent of the `sqlite3` utility. You may use it to manually access encrypted databases from the command line.

**Note**

The Secure Database Library includes parts of the SQLite code, which is released as public domain software.

## 13.2 Security Features

The Secure Database Library is designed to withstand reverse-engineering and key-extraction attacks in open execution environments, such as on mobile devices, tablets, and desktop computers. Its security is based on the following principles:

- The database is stored and used in encrypted form at all times.
- The AES key that is used for encrypting and decrypting the database is provided and used in encrypted form. Unless specifically configured otherwise, this form is unique to each SKB package, which means that hackers cannot simply replace the AES key of one SKB package with that of another package.
- The Secure Database Library uses the `SKB_CIPHER_FLAG_BALANCED` implementation of the AES cipher.

## 13.3 Supported Platforms

The Secure Database Library is available for the following target platforms:

- Android (x86, x86_64, 32-bit/64-bit ARM) built with Android NDK r23
- iOS/iPadOS (32-bit/64-bit ARM) built with Xcode 12.4

- macOS (x86_64, 64-bit ARM) built with Xcode 12.4
- tvOS (64-bit ARM) built with Xcode 12.4
- Windows (x86, x86_64) built separately with Visual Studio 2013, 2015, and 2017
- glibc/Linux (x86_64) built with GCC 7.5
- MinGW (x86, x86_64) built with MSYS2 Rev2 GCC 7.3 (x86) and Rev3 GCC 8.2 (x86_64)

The `SkbSecureDatabaseShell` utility is available for the following platforms:

- glibc/Linux (x86_64) built with GCC 7.5
- Windows (x86_64) built with Visual Studio 2017
- macOS (x86_64) built with Xcode 12.4

## 13.4 Limitations

The Secure Database Library has the following limitations:

- The write-ahead logging functionality of SQLite is not supported.
- Only the default page size is supported. For example, executing the following command will create a corrupted database file:

```
PRAGMA page_size = 2048;
```

## 13.5 Preparing the Database Encryption Key

The Secure Database Library uses a 128-bit AES key to encrypt and decrypt the database. This key must be provided in the SKB-protected form prior to using the API. To obtain the protected form of the database encryption key, you must use either the Key Export Tool or the key export function of SKB.

When using the Key Export Tool, the `--input-format` parameter must always be `bytes`, and the `--output-format` parameter must be either `binary` (for use at runtime) or `source` (for use at compile time). Please note that the cross-engine export format (the `--cross-engine` parameter) is also supported by the Secure Database Library.

**Important**

If your SKB package contains the Key Embedding Tool, the cross-engine export format is the only format supported by the Secure Database Library.

## 13.6 Secure Database Library API

This section describes how the Secure Database Library API works and the functions it provides.

## 13.6.1 API Overview

In general, when working with the Secure Database Library you simply use the standard functions, structures and other elements of the SQLite API. There are a few additional requirements from the API point of view:

- To use the Secure Database Library API, you must include the `SkbSecureDatabase.h` file, which is located in the `Include` directory of the SKB package.
- You must link the following static libraries, which are located in the `lib` directory of the SKB package:
  - `SecureKeyBox.lib` or `libSecureKeyBox.a` depending on the target platform
  - `SkbSecureDatabase.lib` or `libSkbSecureDatabase.a` depending on the target platform
  - `sqlite3.lib` or `libsqlite3.a` depending on the target platform
- Before you start working with the Secure Database Library API, you must call a special initialization function. This is where you also provide the AES key used for encrypting and decrypting the database.
- After you finish working with the Secure Database Library API, you must call a special release function.

Please see the example project for details on how these points work in practice.

## 13.6.2 Initialization Functions

This section describes the available initialization functions of the Secure Database Library API. Depending on your needs, you have to use one of these functions before you start working with the API.

### sqlite3_crypto_open

This function initializes the cryptographic engine and opens the database connection in encrypted mode. In case of an error, this function returns the SQLite error code.

The function is declared as follows:

```
int sqlite3_crypto_open(const char           *filename,
                        sqlite3**            db,
                        const unsigned char* aes_key,
                        unsigned int         aes_key_size,
                        sqlite3_vfs**        vfs);
```

The following are the parameters used:

- `filename`

  Pointer to the database file name. UTF-8 encoding is supported.

- `db`

Address of the pointer to the SQLite database handle, which will be created or opened after this function is executed.

- `aes_key`

  Pointer to the byte buffer where the database encryption key is stored. The key must be prepared in the special protected format.

- `aes_key_size`

  Size of the aes_key buffer in bytes.

- `vfs`

  Address of the pointer to the virtual file system handle, which will be created after this function is executed.

### sqlite3_crypto_create

This function initializes the cryptographic engine but does not open the database connection. The function returns the virtual file system handle. You have to explicitly connect to the encrypted database later using the `sqlite3_open_v2` function.

The function is declared as follows:

```
sqlite3_vfs* sqlite3_crypto_create(const unsigned char* aes_key,
                                   unsigned int         aes_key_size);
```

The following are the parameters used:

- `aes_key`

  Pointer to the byte buffer where the database encryption key is stored. The key must be prepared in the special protected format.

- `aes_key_size`

  Size of the aes_key buffer in bytes.

## 13.6.3 Release Functions

This section describes the special release functions of the Secure Database Library API. Depending on your needs, you have to use one of these functions after you finish working with the API.

### sqlite3_crypto_close

This function closes the database connection and releases the cryptographic engine. In case of an error, this function returns the SQLite error code.

The function is declared as follows:

```
int sqlite3_crypto_close(sqlite3*     db,
                         sqlite3_vfs* vfs);
```

The following are the parameters used:

- `db`

  Pointer to the SQLite database handle.

- `vfs`

  Pointer to the virtual file system handle.

### sqlite3_crypto_release

This function releases the cryptographic engine but does not close the database connection. The function returns the virtual file system handle. You have to explicitly close the database connection using the `sqlite3_close` function either before or after calling the `sqlite3_crypto_release` function.

The function is declared as follows:

```
void sqlite3_crypto_release(sqlite3_vfs* vfs);
```

`vfs` is a pointer to the virtual file system handle.

## 13.7 Secure Database Shell Tool

The Secure Database Shell Tool is a command-line utility that is the equivalent of the `sqlite3` executable. This tool allows you to manually enter and execute SQL statements against the encrypted database.

The Secure Database Shell Tool is provided in the `bin` directory of the SKB package, and its executable file is named `SkbSecureDatabaseShell` or `SkbSecureDatabaseShell.exe` depending on your development platform.

You use the Secure Database Shell Tool just like you would use the `sqlite3` utility with one additional requirement; when launching the `SecureDatabaseShell` executable, you must provide the additional parameter `-key` as follows:

```
SkbSecureDatabaseShell -key «AES key»
```

«AES key» is a HEX string representing the binary form of the protected database encryption key. If the `-key` argument is not provided, all databases created by the Secure Database Shell Tool will be unencrypted, and all encrypted databases will be inaccessible.

# 13.8 Secure Database Library Example

There is a Secure Database Library example project provided in the `Examples/SkbSecureDatabase` directory of the SKB package. For information on building and running this example, please read the accompanying `README.txt` file.

# 14. DUKPT API

This chapter describes how to use the Derived Unique Key Per Transaction (DUKPT) API provided by SKB.

## 14.1 Overview of the DUKPT API

The DUKPT API is a higher-level API built on top of the core functions of SKB, and its purpose is to provide a set of simple methods and data structures for implementing working key derivation algorithms on the originating Secure Cryptographic Device (SCD) according to the DUKPT key management scheme using the secure properties of SKB.

The DUKPT key management scheme is specified by ANSI X9.24-3-2017.

## 14.2 Implementation Details and Limitations

The following implementation details apply specifically to the DUKPT API provided by SKB:

- The working keys derived through the DUKPT API will be encapsulated as secure data objects, which can then be used by the standard methods of the Native API.
- The DUKPT API supports only the originating (encrypting) end of the cryptographic communication.
- Initial Key, which is derived from the Base Derivation Key (BDK), and the Initial Key ID must already be available before using the DUKPT API. The Initial Key could, for example, be sent from a server in a wrapped format and then unwrapped using the native API of SKB.
- The Key Serial Number (KSN) compatibility mode is not supported.
- Although, according to the DUKPT scheme, the working keys are derived for specific purposes, such as encryption of PINs, message authentication and so on, SKB does not put any restrictions on the way these keys are later used by the native API of SKB.
- Implementation of the DUKPT API is part of the main SKB library (`libSecureKeyBox.a` or `SecureKeyBox.lib`, depending on the target platform). Therefore, to use the DUKPT API you have to link this library into your application.
- The DUKPT API is defined in the `SkbDukpt.h` file, which is provided in the `Include` directory of the SKB package. The `SkbDukpt.h` header includes the `SkbSecureKeyBox.h` header, which contains the main SKB Native API.
- Methods of the DUKPT API return the same return values as those of the native API, which are described in Method Return Values.
- The DUKPT API does not support the "Update Initial Key" procedure.

## 14.3 General Steps of Using the DUKPT API

The DUKPT API is designed to be used only on the originating SCD. The general sequence of steps is as follows:

1. Initialize the DUKPT state using one of the following methods:
   - To start a completely new DUKPT session, obtain the Initial Key derived from the BDK and the Initial Key ID, and create a new DUKPT state using the `SKB_DukptState_Create` method.
   - To resume working with an existing DUKPT session, import the previously exported DUKPT state using the `SKB_Engine_ImportDukptState` method.

   The output of both methods is a DUKPT state object `SKB_DukptState`, which must be passed to all other methods of the DUKPT API.
2. Derive a new working key or several working keys with a specific type and purpose using the `SKB_DukptState_Derive` method.

   Please note that this operation does not automatically increment the transaction counter.
3. Increment the transaction counter using the `SKB_DukptState_Increment` method.
4. To derive more working keys, go to step 2.
5. If the DUKPT state must be retained for future use, export it using the `SKB_DukptState_Export` method.
6. If there is a need to update the Initial Key, use the `SKB_DukptState_CalculateDukptUpdateKey` method to perform the "Calculate DUKPT Update Key" procedure, which is described in section 6.5.3 of *ANSI X9.24-3-2017*.

   Please note that zKeyBox does not provide the means for performing the "Update Initial Key" procedure. But you can achieve the same result by creating a new `SKB_DukptState` object from a new Initial Key (delivered by the server) that is wrapped with the Update Key derived by the `SKB_DukptState_CalculateDukptUpdateKey` method. When you receive the new wrapped Initial Key from the server, you must unwrap it in zKeyBox (for example, by unwrapping the TR-31 key block as suggested by *ANSI X9.24-3-2017*) and then pass it to the `SKB_DukptState_Create` method to create a new `SKB_DukptState` object, which will be equivalent to the state that would be obtained after performing the "Update Initial Key" procedure.
7. Release the DUKPT state from memory using the `SKB_DukptState_Release` method.

## 14.4 Methods

This section describes the methods provided by the DUKPT API.

## 14.4.1 SKB_DukptState_Create

This method creates a new SKB_DukptState object, which must then be passed to all other methods of the DUKPT API.

The method is declared as follows:

```
SKB_Result
SKB_DukptState_Create(const SKB_SecureData*       initial_key,
                      const SKB_Byte*             initial_key_id,
                      SKB_Size                    initial_key_id_size,
                      SKB_DukptDerivationFunction derivation_function,
                      SKB_DukptKsnCompatibility   ksn_compatibility,
                      SKB_DukptState**            dukpt_state);
```

The following are the parameters used:

- `initial_key`

  Pointer to the SKB_SecureData object containing the Initial Key derived from the BDK.

- `initial_key_id`

  Pointer to the buffer containing the Initial Key ID.

- `initial_key_id_size`

  Size of the `initial_key_id` buffer in bytes.

- `derivation_function`

  Type of the derivation function to be used. The available types are defined in the SKB_DukptDerivationFunction enumeration.

- `ksn_compatibility`

  Determines whether the KSN compatibility mode must be used. The available values are defined in the SKB_DukptKsnCompatibility enumeration.

  Currently, the only supported value is SKB_DUKPT_KSN_AES_MODE, which means that the KSN compatibility mode is not used.

- `dukpt_state`

  Address of a pointer to the SKB_DukptState object that will contain the DUKPT state object after this method is executed.

## 14.4.2 SKB_DukptState_Derive

This method derives a new working key of a specific type and purpose. The derived key will be returned as the SKB_SecureData object. Please note that this method does not automatically increment the transaction counter.

The method is declared as follows:

```
SKB_Result
SKB_DukptState_Derive(const SKB_DukptState*    dukpt_state,
                      SKB_DukptWorkingKeyUsage working_key_usage,
                      SKB_DukptWorkingKeyType  working_key_type,
                      SKB_Byte*                ksn,
                      SKB_Size*                ksn_size,
                      SKB_SecureData**         working_key);
```

The following are the parameters used:

- `dukpt_state`

  Pointer to the pre-initialized SKB_DukptState object.

- `working_key_usage`

  Usage type of the derived key. The available usage types are defined in the SKB_DukptWorkingKeyUsage enumeration.

- `working_key_type`

  Type of the key being derived. The available key types are defined in the SKB_DukptWorkingKeyType enumeration.

  Please note that the working key must be the same strength or weaker than the Initial Key that was used to create the SKB_DukptState object. For example, Triple DES working keys can always be derived, but a 256-bit AES working key cannot be derived if the Initial Key was a 192-bit AES key. Please see ANSI X9.24-3-2017 for complete information on which working key types can be derived from which Initial Key types.

- `ksn`

  This parameter is either NULL or a pointer to the memory buffer where the KSN (concatenation of the Initial Key ID and the transaction counter) will be written after this method is executed.

  If this parameter is NULL, the method simply returns, in `ksn_size`, the number of bytes that would be sufficient to hold the KSN, and returns SKB_SUCCESS.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the KSN, the method stores the KSN there, sets `ksn_size` to the exact number of bytes stored, and returns SKB_SUCCESS. If the buffer is not large enough, then the method sets `ksn_size` to the number of bytes that would be sufficient, and returns SKB_ERROR_BUFFER_TOO_SMALL.

- `ksn_size`

  Size of the `ksn` buffer in bytes.

  Because currently the KSN compatibility mode is not supported (`SKB_DUKPT_KSN_AES_MODE` is the only supported mode), the KSN size is always 12 bytes.

- `working_key`

  Address of a pointer that will point to the new derived `SKB_SecureData` object when this method is executed. The `SKB_SecureData` object will contain the derived working key.

## 14.4.3 SKB_DukptState_Increment

This method increments the transaction counter stored inside a DUKPT state object.

The method is declared as follows:

```
SKB_Result
SKB_DukptState_Increment(SKB_DukptState* dukpt_state);
```

`dukpt_state` is a pointer to the pre-initialized `SKB_DukptState` object.

## 14.4.4 SKB_DukptState_Export

This method returns a protected form of a particular `SKB_DukptState` object. This protected data is intended for saving the state in a persistent storage. Later the exported data can be imported back using the `SKB_Engine_ImportDukptState` method.

The method is declared as follows:

```
SKB_Result
SKB_DukptState_Export(const SKB_DukptState* dukpt_state,
                      SKB_ExportTarget      target,
                      SKB_Byte*             buffer,
                      SKB_Size*             buffer_size);
```

The following are the parameters used:

- `dukpt_state`

  Pointer to the `SKB_DukptState` object to be exported.

- `target`

  Export type to be used. Available export types are defined in the `SKB_ExportTarget` enumeration.

  The same principles that apply to exporting and importing of keys are also applicable to exporting and importing of the DUKPT state.

- `buffer`

This parameter is either NULL or a pointer to the memory buffer where the exported data is to be written.

If this parameter is NULL, the method simply returns, in `buffer_size`, the number of bytes that would be sufficient to hold the exported data, and returns SKB_SUCCESS.

If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the exported data, the method stores the exported data there, sets `buffer_size` to the exact number of bytes stored, and returns SKB_SUCCESS. If the buffer is not large enough, then the method sets `buffer_size` to the number of bytes that would be sufficient, and returns SKB_ERROR_BUFFER_TOO_SMALL.

- `buffer_size`

  Pointer to a variable that holds the size of the memory buffer in bytes where the exported data is to be stored.

## 14.4.5 SKB_Engine_ImportDukptState

This method imports the SKB_DukptState object that was previously exported using the SKB_DukptState_Export method.

The method is declared as follows:

```
SKB_Result
SKB_Engine_ImportDukptState(SKB_Engine*      self,
                            const SKB_Byte*  buffer,
                            SKB_Size         buffer_size,
                            SKB_DukptState** dukpt_state);
```

The following are the parameters used:

- `self`

  Pointer to the pre-initialized SKB engine.

- `buffer`

  Pointer to the memory buffer containing the exported data.

- `buffer_size`

  Size of `buffer` in bytes.

- `dukpt_state`

  Address of a pointer to the SKB_DukptState object that will be created by this method. This object will contain the imported DUKPT state.

## 14.4.6 SKB_DukptState_CalculateDukptUpdateKey

This method derives the Update Key that will be used to deliver a new Initial Key from the server. The implementation of this method follows the "Calculate DUKPT Update Key" algorithm, which is described in section 6.5.3 of ANSI X9.24-3-2017.

**Important**

After this method successfully derives the Update Key, the transaction counter will be set to 0xFFFFFFFF and all DUKPT API methods associated with the same SKB_DukptState object (except SKB_DukptState_Release) will start failing with the SKB_ERROR_INVALID_STATE error code. Therefore, you will have to release the current SKB_DukptState object and create a new one based on the new Initial Key.

The method is declared as follows:

```
SKB_Result
SKB_DukptState_CalculateDukptUpdateKey(SKB_DukptState*         dukpt_state,
                                       SKB_DukptWorkingKeyType key_encryption_key_type,
                                       SKB_SecureData**        key_encryption_key);
```

The following are the parameters used:

- dukpt_state

  Pointer to the pre-initialized SKB_DukptState object.

- key_encryption_key_type

  Type of the Update Key you want to derive. The possible values are defined in the SKB_DukptWorkingKeyType enumeration, but only the following key types are supported by this method:

  - SKB_DUKPT_WORKING_KEY_TYPE_AES128
  - SKB_DUKPT_WORKING_KEY_TYPE_AES192
  - SKB_DUKPT_WORKING_KEY_TYPE_AES256

  **Important**

  The type you choose must not be stronger than the type of the Inital Key you provided to the SKB_DukptState_Create method. For example, if the Initial Key was a 192-bit AES key, you may only choose SKB_DUKPT_WORKING_KEY_TYPE_AES128 or SKB_DUKPT_WORKING_KEY_TYPE_AES192.

- key_encryption_key

Address of a pointer to the SKB_SecureData object that will contain the Update Key after this method is executed. The server will derive the same key and use it to wrap a new Initial Key. When the wrapped Initial Key is delivered from the server, use the Update Key to unwrap the Initial Key and create a new DUKPT instance with it.

### 14.4.7 SKB_DukptState_Release

This method releases the specified SKB_DukptState object from memory. It must always be called when the object is no longer needed.

The method is declared as follows:

```
SKB_Result
SKB_DukptState_Release(SKB_DukptState* dukpt_state);
```

The parameter dukpt_state is a pointer to the SKB_DukptState object that must be released.

## 14.5 Enumerations

This section describes the enumerations provided by the DUKPT API.

### 14.5.1 SKB_DukptDerivationFunction

This enumeration specifies the possible derivation functions supported by the SKB_DukptState_Create method.

The enumeration is defined as follows:

```
typedef enum {
    SKB_DUKPT_DERIVATION_FUNCTION_AES128,
    SKB_DUKPT_DERIVATION_FUNCTION_AES192,
    SKB_DUKPT_DERIVATION_FUNCTION_AES256
} SKB_DukptDerivationFunction;
```

As can be determined from the names, each value represents a particular key length of the AES cipher to be used.

### 14.5.2 SKB_DukptKsnCompatibility

This enumeration provides values that determine whether the KSN compatibility mode must be enabled when creating a new DUKPT state using the SKB_DukptState_Create method.

The enumeration is defined as follows:

```
typedef enum {
    SKB_DUKPT_KSN_AES_MODE,
```

```
    SKB_DUKPT_KSN_COMPATIBILITY_MODE
} SKB_DukptKsnCompatibility;
```

The following are the values defined:

- `SKB_DUKPT_KSN_AES_MODE`

  Enables the new KSN format, which is based on AES.

  Currently, this is the only value supported by the DUKPT API.

- `SKB_DUKPT_KSN_COMPATIBILITY_MODE`

  Enables the KSN compatibility mode.

  **Important**

  Currently, this value is not supported.

## 14.5.3 SKB_DukptWorkingKeyUsage

This enumeration defines working key usage types supported by the SKB_DukptState_Derive method.

The enumeration is defined as follows:

```
typedef enum {
    SKB_DUKPT_WORKING_KEY_USAGE_KEY_ENCRYPTION_KEY,
    SKB_DUKPT_WORKING_KEY_USAGE_PIN_ENCRYPTION,
    SKB_DUKPT_WORKING_KEY_USAGE_MESSAGE_AUTHENTICATION_GENERATION,
    SKB_DUKPT_WORKING_KEY_USAGE_MESSAGE_AUTHENTICATION_VERIFICATION,
    SKB_DUKPT_WORKING_KEY_USAGE_MESSAGE_AUTHENTICATION_BOTHWAYS,
    SKB_DUKPT_WORKING_KEY_USAGE_DATA_ENCRYPTION_ENCRYPT,
    SKB_DUKPT_WORKING_KEY_USAGE_DATA_ENCRYPTION_DECRYPT,
    SKB_DUKPT_WORKING_KEY_USAGE_DATA_ENCRYPTION_BOTHWAYS,
    SKB_DUKPT_WORKING_KEY_USAGE_KEY_DERIVATION
} SKB_DukptWorkingKeyUsage;
```

Except for differences in the naming pattern, these values correspond to the values of the KeyUsage enumeration defined in ANSI X9.24-3-2017.

## 14.5.4 SKB_DukptWorkingKeyType

This enumeration defines the types of keys that can be derived using the SKB_DukptState_Derive and SKB_DukptState_CalculateDukptUpdateKey methods.

The enumeration is defined as follows:

```
```

```
typedef enum {
    SKB_DUKPT_WORKING_KEY_TYPE_AES128,
    SKB_DUKPT_WORKING_KEY_TYPE_AES192,
    SKB_DUKPT_WORKING_KEY_TYPE_AES256,
    SKB_DUKPT_WORKING_KEY_TYPE_TDEA2,
    SKB_DUKPT_WORKING_KEY_TYPE_TDEA3,
    SKB_DUKPT_WORKING_KEY_TYPE_HMAC128,
    SKB_DUKPT_WORKING_KEY_TYPE_HMAC192,
    SKB_DUKPT_WORKING_KEY_TYPE_HMAC256
} SKB_DukptWorkingKeyType;
```

The following are the values defined:

- `SKB_DUKPT_WORKING_KEY_TYPE_AES128`

  128-bit AES key

- `SKB_DUKPT_WORKING_KEY_TYPE_AES192`

  192-bit AES key

- `SKB_DUKPT_WORKING_KEY_TYPE_AES256`

  256-bit AES key

- `SKB_DUKPT_WORKING_KEY_TYPE_TDEA2`

  Triple DES key with two distinct keys

- `SKB_DUKPT_WORKING_KEY_TYPE_TDEA3`

  Triple DES key with three distinct keys

- `SKB_DUKPT_WORKING_KEY_TYPE_HMAC128`

  128-bit HMAC key

- `SKB_DUKPT_WORKING_KEY_TYPE_HMAC192`

  192-bit HMAC key

- `SKB_DUKPT_WORKING_KEY_TYPE_HMAC256`

  256-bit HMAC key

## 14.6 DUKPT API Usage Example

There is a DUKPT API example `ExampleDukpt.cpp` provided in the `Examples/Cpp/src` directory of the SKB package. For information on building and running this example, please read the code comments and the accompanying `README.txt` file.

# 15. Secure PIN Entry

This chapter describes how to implement functionality that allows users to securely enter a personal identification number (PIN) in your Android application.

## 15.1 Secure PIN Overview

Secure PIN is a set of tools that allow you to implement secure graphical user interface (GUI) based PIN entry in Android applications.

**Note**

Secure PIN is only available, if you specifically selected it when requesting the SKB package.

### 15.1.1 Security

The core principle of Secure PIN is to ensure protection of the entry and processing of PINs. Secure PIN is designed to satisfy dozens of requirements put forth by various standards associated with PIN entry on commercial off-the-shelf (COTS) devices. Some of the main requirements fulfilled by Secure PIN are:

- The entered PIN digits and entire numbers are never revealed in cleartext.
- The PIN encryption key is a secure data object.
- PIN entry is aborted in case of potential threats, such as when the application is modified, the entry pad loses focus, a debugger is detected, and so on.

  **Important**

  Because of this security feature, the developer mode must be off when running the production edition of your Android application containing Secure PIN.

- PIN entry does not rely on the system's default keyboard.

**Note**

For additional security, you may consider using the SKB implementation of DUKPT API to derive the PIN encryption key.

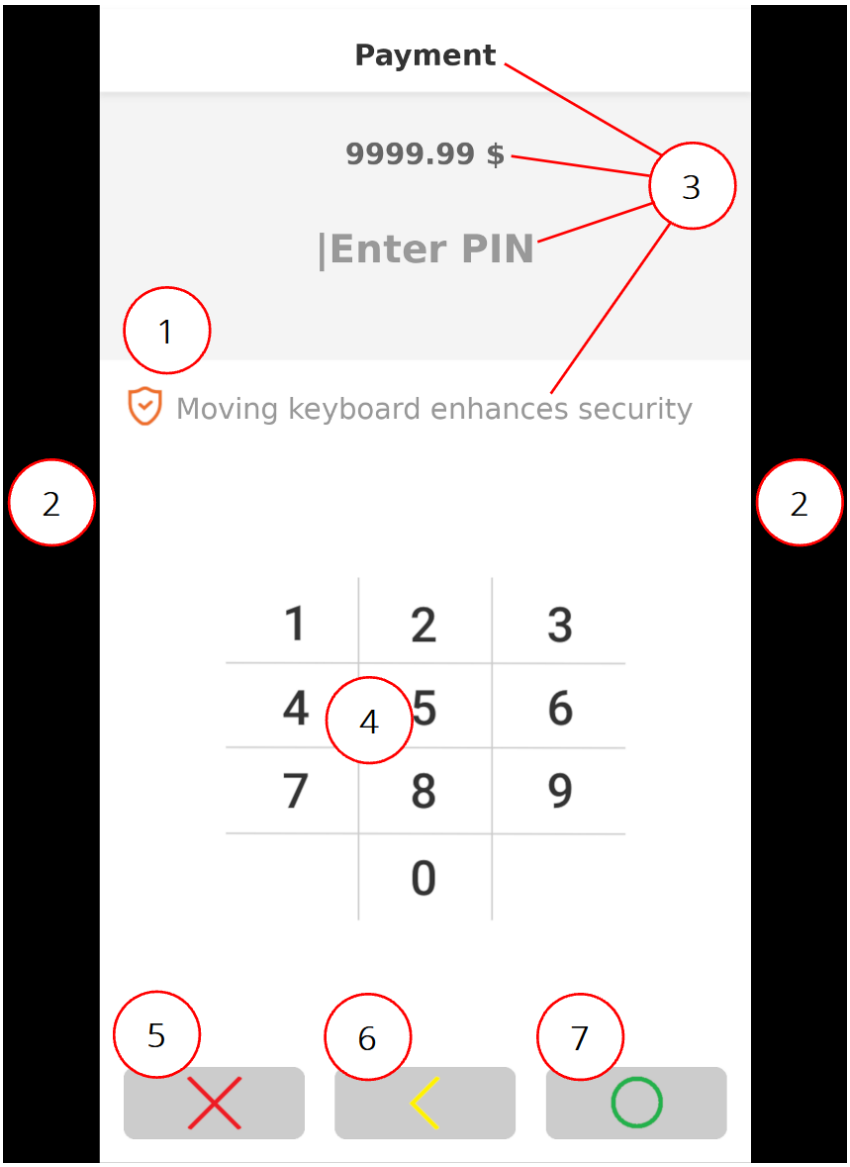
### 15.1.2 Editions of Secure PIN

There are two editions of Secure PIN.

| Edition | Description |
| --- | --- |
| Development | This edition is provided in SKB packages that are not protected with zShield. It allows running the application on Android emulators (it supports the x86 and x86-64 architectures) as well as devices that have the developer mode enabled, and has many of the runtime protection features disabled. Therefore, this edition is intended only for development of your application, but must never be included in production. |
| Production | This edition is provided in zShield-protected SKB packages. It includes the complete set of security features, and supports only the 32-bit and 64-bit ARM architectures. This is the edition you must use to comply with the industry security standards. For details on using this edition, see Working with the Production Edition of Secure PIN. |

## 15.1.3 PIN Entry Pad

The main visual component of Secure PIN is a graphical layout that represents the PIN entry pad. It is comprised of a number of graphical elements as shown in the following diagram.




The following list explains each of the diagram elements marked with a number:

1. Background image that fills the entire layout
2. Color of areas not covered by the background image
3. Four separate lines of centered text you can display at arbitrary vertical positions (each text can have an individual font, color, and size)

   One of these lines will display asterisks representing the entered digits.
4. Image representing the PIN pad digit buttons
5. Image representing the cancel button
6. Image representing the clear button

7. Image representing the confirm button

The resources of the PIN entry pad, such as the layout properties, fonts, and images, must be prepared using a special script. You must then provide the resulting configuration when invoking the PIN pad in code. You may generate multiple configurations and dynamically choose which one to load at runtime.

### 15.1.4 Architecture

The main part of Secure PIN is implemented as a native static library `libSecurePin.a`, which is included in the `securepin.aar` file. Therefore, to use Secure PIN, you have to create your own native library that is linked with `libSecurePin.a`, and then integrate your library into the Android application. There are two editions of the `libSecurePin.a` library available; one is for 32-bit ARM, and the other for 64-bit ARM.

The interface of the `libSecurePin.a` library is defined in the `SecurePin.h` header, which is also included in the `securepin.aar` file. The interface provides only one function; by calling this function you will invoke the secure PIN entry pad.

**Note**

The `SecurePin.h` header includes `SkbSecureKeyBox.h`, which is the main native API of SKB.

## 15.2 Working with the Production Edition of Secure PIN

As explained in Editions of Secure PIN, the production edition of Secure PIN is the one that is included in a zShield-protected SKB package. This is the edition you can use in production.

When using the production edition of Secure PIN, there is an additional point you must take care of. Namely, when processing the final Android application with the Binary Update Tool, you must also supply the `libSecurePin.nwdb` file as its input in addition to the SKB `.nwdb` file.

**Note**

For details and tips on embedding a zShield-protected Secure PIN into your application, explore the `CMakeLists.txt` file in the Secure PIN example application.

## 15.3 Adding Secure PIN to Your Application

Here are the main steps for adding Secure PIN to an Android application:

1. Generate Secure PIN configuration to describe your PIN entry pad.
2. Include the `securepin.aar` file into your application by adding it to the `dependencies` section of the `build.gradle` file.

   The `securepin.aar` file is provided in the `lib/google-android` directory. It contains the essential parts of the Secure PIN code.
3. Create a native library linked with the `libSecurePin.a` file, which is included in the `securepin.aar` file.

The `securepin.aar` archive also contains the `SecurePin.h` header, which is the interface to the `libSecurePin.a` library.

For your convenience, the `securepin.aar` file is configured to support Prefab. This means that you can load the `libSecurePin.a` library as a Prefab package named `securepin`. For example, if you use CMake, the library can be loaded using the following commands in the `CMakeLists.txt` file:

```
find_package(securepin REQUIRED CONFIG)

...

target_link_libraries( ...

                       securepin::SecurePin

                       ...
)
```

4. At some point, before calling the `SPIN_GetPin` function, make sure the following call is made:

```
com.whitecryption.securepin.SecurePinAssets.enable(getApplicationContext());
```

This is required to pass the application context to Secure PIN. It needs the context to load the assets and access the Java environment.

**Note**

Alternatively, the same can be achieved directly in the native code by making this call using Java reflection. Both approaches are demonstrated in the Secure PIN example project.

5. To display the PIN entry pad, call the `SPIN_GetPin` function, which is declared in the `SecurePin.h` header.

   When calling this function, you will have to provide the configuration you created in step 1. When the PIN entry pad is closed, the function will return the results of the operation.

6. If you are working with a zShield-protected SKB package, make sure you also supply the `libSecurePin.nwdb` file when running the Binary Update Tool.

   The `libSecurePin.nwdb` file is located in the `lib/google-android` directory.

# 15.4 Generating the Secure PIN Configuration

Secure PIN configuration holds information about all the settings and resources required to display the PIN entry pad. When calling the function that displays the entry pad, you have to supply an instance of the `SPIN_LayoutConfig` structure, which represents the Secure PIN configuration. Secure PIN configuration can be generated using a special Python script called `securepin_generate_layout.py`, which is available in the `src/SecurePIN` directory of the SKB package.

To better understand the way Secure PIN configuration is defined and generated, refer to the Secure PIN example project.

## 15.4.1 Script Requirements

The `securepin_generate_layout.py` script requires Python 3 and the following libraries:

- `jinja2`
- `xmlschema`
- `freetype-py`
- `pillow`

## 15.4.2 Running the Generation Script

To generate Secure PIN configuration, you must execute the `securepin_generate_layout.py` script with the following parameters:

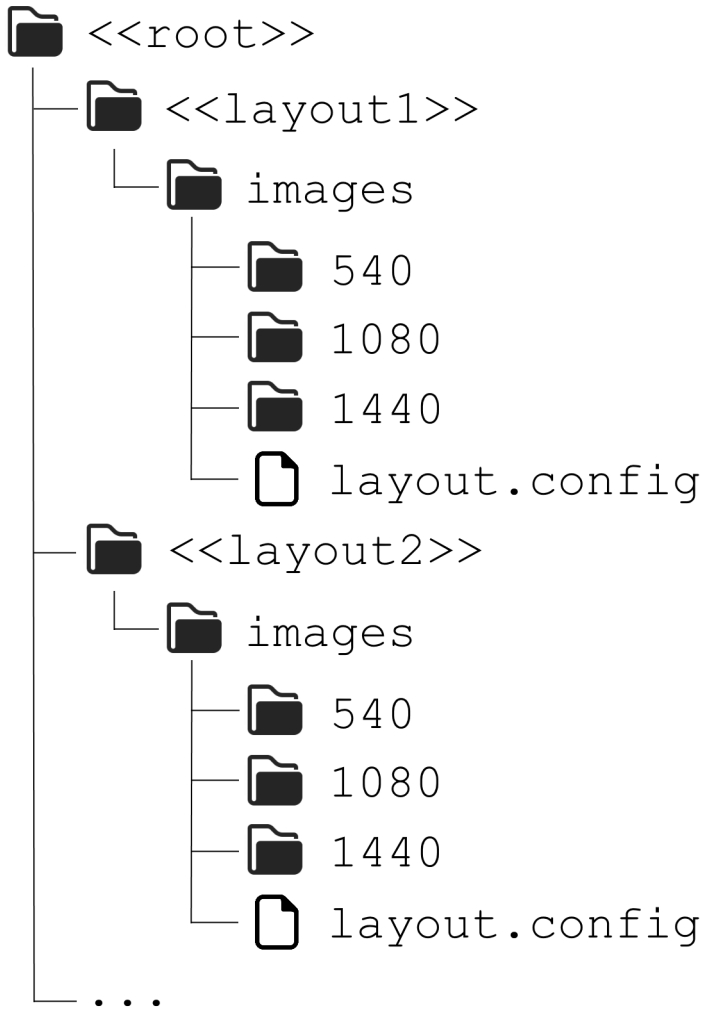
- `--layouts`: Points to the directory containing input parameters for one or several Secure PIN configurations. For details on how this directory must be structured, see Defining the Layouts.
- `--configs`: Specifies the directory where the script will generate two files:
  - `SecurePinLayoutConfig.cpp`: Source file containing generated definitions of the `SPIN_LayoutConfig` structure, which must be supplied to the `SPIN_GetPin` function.

  As described in Defining the Layouts, multiple such configurations can be generated. All of their definitions will be placed in the `SecurePinLayoutConfig.cpp` file. The names of the structure instances will match the names of the corresponding layout directories with the suffix `_config`.
  - `SecurePinLayoutConfig.h`: Header file corresponding to the `SecurePinLayoutConfig.cpp` file.

  You must either include both of these files in your native library or copy the corresponding contents into your code.
- `--assets`: Specifies the directory where the script will place image and font resources compiled from the input parameters.

## 15.4.3 Defining the Layouts

The layouts for Secure PIN are defined using a special structure of files and directories, which is shown in the following diagram.




«`root`» is the root directory that you must specify in the `--layouts` argument of the `securepin_generate_layout.py` script.

Each subdirectory holds resources for one particular layout. You may define as many layout subdirectories as needed.

In each layout subdirectory, you must create following elements:

- `images` directory, which holds image files used to construct the PIN entry pad.

  Because of the different sizes and resolutions of Android device screens, the images are organized into subdirectories, such as `540`, `1080`, and `1440` (or any other number), which represent the screen width in pixels. When the application is run, the directory that is the closest match to the screen resolution will be selected and, if necessary, additionally scaled to fit the resolution.

In at least one of the subdirectories of the `images` directory, you must create all the following images with those exact file names:

- `background.png`: Image displayed as the background of the entire PIN entry pad.

  The image width should match the corresponding resolution width specified in the subdirectory name, otherwise the image will either not cover the whole screen or will be cropped.

- `bt_confirm.png`: Image that represents the PIN entry confirmation button.
- `bt_clear.png`: Image that represents the button that deletes the entered PIN digits.
- `bt_cancel.png`: Image that represents the cancel button, which closes the PIN entry pad without confirming the PIN.
- `pinpad.png`: Image that represents the buttons of the PIN pad.

  The PIN pad will be interpreted as consisting of 3 by 4 equally sized tiles. Therefore, the arrangement of digits must be exactly as shown in the following diagram.


  You cannot change the order of digits.

**Note**

You may also create empty layout subdirectories, in which case their assets will be automatically generated by scaling images from the non-empty subdirectory of the closest resolution width.

- `layout.config` file, which contains various settings for the particular layout.

  For details on this file, see Layout Settings.

## 15.4.4 Layout Settings

The `layout.config` file is a plain text file containing key and value pairs in the form "«key»=«value»". Each pair must be written on a separate line. String values must be enclosed in either single or double quotes. By defining these pairs you assign values to constants used to display and position the graphical elements of the PIN entry pad. The `securepin_generate_layout.py` script will convert this data into a definition of the `SPIN_LayoutConfig` structure.

Here are the keys you must define in the `layout.config` file:

- `design_width` and `design_height`: The dimensions of the entire PIN entry form in virtual design points. These are relative units that do not necessarily equal pixels on the screen. If the actual screen resolution in pixels does not exactly match these dimensions, all form elements will be scaled accordingly. In other words, the sizes and positions of images and all other elements set in the configuration will be interpreted as being proportional to these two values.

  A good practice is to ensure `design_width` and `design_height` match the pixel dimensions of the largest `background.png` image.

- `bg_color`: Color of the areas not covered by the background image. The color must be specified using HEX code without the alpha parameter, for example "#FF5926".

- `pinpad_boundary_x`, `pinpad_boundary_y`, `pinpad_boundary_width`, and `pinpad_boundary_height`: Top left coordinates and the dimensions of the rectangle that represents the space within which the image of the PIN pad buttons (`pinpad.png`) will be positioned.

  Depending on the value of the `randomize_position` argument passed to the `SPIN_GetPin` function, the PIN pad buttons will be either centered within the rectangle both vertically and horizontally at a fixed position, or placed at a different position at random within the specified rectangle after every touch of the pad.

  **Note**

  Coordinates (0,0) correspond to the upper left corner of the PIN entry form.

- `pinpad_boundary_color`: Color of the rectangle defined by the previous set of keys. A visible color should only be used for debugging; the rectangle should be invisible in production. The color is specified using HEX code with the alpha (opacity) parameter, for example "#77FFFFFF", where "77" is the alpha value.

- `bt_cancel_x` and `bt_cancel_y`: Top left coordinates of the cancel button (`bt_cancel.png`).

- `bt_cancel_hidden`: "true" or "false" key that determines whether the cancel button is visible.

- `bt_cancel_bound_to_pad`: "true" or "false" key that determines whether the position of the cancel button moves together with the PIN pad buttons image (when the `randomize_position` argument of the `SPIN_GetPin` function is `true`).

If "true", you must calculate the `bt_cancel_x` and `bt_cancel_y` coordinates as if the upper left corner of the PIN pad buttons image is positioned at the upper left corner of the boundary rectangle (at coordinates `pinpad_boundary_x` and `pinpad_boundary_y`).

- `bt_clear_x` and `bt_clear_y`: Top left coordinates of the clear button (`bt_clear.png`).
- `bt_clear_hidden`: "true" or "false" key that determines whether the clear button is visible.
- `bt_clear_bound_to_pad`: "true" or "false" key that determines whether the position of the clear button moves together with the PIN pad buttons image (when the `randomize_position` argument of the `SPIN_GetPin` function is `true`).

  If "true", you must calculate the `bt_clear_x` and `bt_clear_y` coordinates as if the upper left corner of the PIN pad buttons image is positioned at the upper left corner of the boundary rectangle (at coordinates `pinpad_boundary_x` and `pinpad_boundary_y`).
- `bt_confirm_x` and `bt_confirm_y`: Top left coordinates of the confirmation button (`bt_confirm.png`).
- `bt_confirm_hidden`: "true" or "false" key that determines whether the confirmation button is visible.
- `bt_confirm_bound_to_pad`: "true" or "false" key that determines whether the position of the confirmation button moves together with the PIN pad buttons image (when the `randomize_position` argument of the `SPIN_GetPin` function is `true`).

  If "true", you must calculate the `bt_confirm_x` and `bt_confirm_y` coordinates as if the upper left corner of the PIN pad buttons image is positioned at the upper left corner of the boundary rectangle (at coordinates `pinpad_boundary_x` and `pinpad_boundary_y`).
- `text_line_0_font_name`, `text_line_1_font_name`, `text_line_2_font_name`, and `text_line_3_font_name`: Names of fonts used for the four lines of text you can display at arbitrary vertical positions.

  The specified fonts must be installed on the computer where you run the `securepin_generate_layout.py` script. To determine the names of available fonts, execute one of the following commands depending on your operating system:

  ◦ On Windows, execute the command:

  ```
  reg query "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts"
  ```

  ◦ On macOS, execute the command:

  ```
  system_profiler SPFontsDataType
  ```

  ◦ On Linux, execute the command:

  ```
  fc-list
  ```

  You must then specify the chosen font file name without the extension, for example "DejaVuSans-Bold".

**Important**

The third of the four text lines (index 2) will display the asterisks representing the entered PIN digits.

- `text_line_0_font_color`, `text_line_1_font_color`, `text_line_2_font_color`, and `text_line_3_font_color`: Color of the four lines of text. The color must be specified using HEX code without the alpha (opacity) parameter.
- `text_line_0_font_size`, `text_line_1_font_size`, `text_line_2_font_size`, and `text_line_3_font_size`: Font size of the four lines of text. The size is relative to the `design_width` and `design_height` values, and will scale accordingly.
- `text_line_0_y`, `text_line_1_y`, `text_line_2_y`, and `text_line_3_y`: Y coordinate of the top border of the rectangles containing each of the four lines of text. The X coordinate is not set because all four text lines will always be centered horizontally.
- `text_line_0_chars`, `text_line_1_chars`, `text_line_2_chars`, and `text_line_3_chars`: Allowed characters for each of the four lines of text. The values must be enclosed by single or double quotes and they may contain a combination of:
  - Specific single characters, such as "0", "A", or "*", in UTF-8 encoding
  - Ranges of characters or their 16-bit Unicode values, such as "a-z", "0-9", or "2013-204A"

  To include the minus sign, you should add it at the beginning or end of the string; otherwise, it may be interpreted as a separator in a range definition.

  Here is an example that allows entering digits, the "-", "'", and "*" characters, and a set of Cyrillic letters:

  ```
  text_line_0_chars = "-0-9*'0410-044F"
  ```

  If, at runtime, any of the four text lines contains a character that does not belong to the defined set, the PIN entry form will either automatically close itself or display the missing characters as rectangles, depending on the value of the `ignore_missing_chars` argument provided to the SPIN_GetPin function.

  **Note**

  `text_line_2_chars` always includes the asterisk character.

## 15.5 Secure PIN API

This section describes the main Secure PIN structures declared in the `SecurePin.h` header.

### 15.5.1 SPIN_LayoutConfig

The `SPIN_LayoutConfig` structure is used to hold the PIN entry form layout configuration provided to the SPIN_GetPin function.

**Important**

To obtain instances of this structure, you must run the securepin_generate_layout.py script.

The structure is declared as follows:

```
struct SPIN_LayoutConfig {
    unsigned int          bg_color;
    short                 pinpad_boundary_x;
    short                 pinpad_boundary_y;
    short                 pinpad_boundary_width;
    short                 pinpad_boundary_height;
    unsigned int          pinpad_boundary_color;
    short                 bt_cancel_x;
    short                 bt_cancel_y;
    bool                  bt_cancel_hidden;
    bool                  bt_cancel_bound_to_pad;
    short                 bt_clear_x;
    short                 bt_clear_y;
    bool                  bt_clear_hidden;
    bool                  bt_clear_bound_to_pad;
    short                 bt_confirm_x;
    short                 bt_confirm_y;
    bool                  bt_confirm_hidden;
    bool                  bt_confirm_bound_to_pad;
    unsigned int          text_line_0_font_color;
    unsigned int          text_line_1_font_color;
    unsigned int          text_line_2_font_color;
    unsigned int          text_line_3_font_color;
    short                 text_line_0_y;
    short                 text_line_1_y;
    short                 text_line_2_y;
    short                 text_line_3_y;
    int                   design_width;
    int                   design_height;
    float                 text_line_0_font_size;
    float                 text_line_1_font_size;
    float                 text_line_2_font_size;
    float                 text_line_3_font_size;
    const char*           font_files;
    const char*           image_files;
    const unsigned int    assets_len;
```

```
    const unsigned int*   assets_resolutions;
    const unsigned char** fontassets_hashes;
    const unsigned char** imgassets_hashes;
};
```

For information on individual parameters, see Layout Settings where most of them are described. The default values for these parameters will be set based on the input layout configuration, but you may manually modify them if needed.

The following additional parameters (not present in the input layout configuration) are generated by the `securepin_generate_layout.py` script to hold internal technical information, which you must not modify:

- `font_files`
- `image_files`
- `assets_len`
- `assets_resolutions`
- `fontassets_hashes`
- `imgassets_hashes`

## 15.5.2 SPIN_Result

The `SPIN_Result` enumeration represents all the possible outcome states of the PIN entry process, which will be returned by the `SPIN_GetPin` function.

The enumeration is defined as follows:

```
enum SPIN_Result {
    SPIN_SUCCESS,
    SPIN_BYPASS,
    SPIN_CANCELED,
    SPIN_TIMEOUT,
    SPIN_INVALID_STATE,
    SPIN_INVALID_PARAMETERS,
    SPIN_INSECURE_CONDITION,
    SPIN_MISSING_CHAR,
    SPIN_GENERIC_ERROR,
};
```

The following values are available:

- `SPIN_SUCCESS`

  PIN entry was successful.

- `SPIN_BYPASS`

  The user pressed the confirmation button without entering any PIN digits. This is only possible if the bypass is enabled when calling the `SPIN_GetPin` function.

- `SPIN_CANCELED`

  The user canceled the PIN entry process by pressing the cancel button.

- `SPIN_TIMEOUT`

  The user failed to enter a PIN within the specified timeout period, which is set by the `timeout_ms` argument of the `SPIN_GetPin` function.

- `SPIN_INVALID_STATE`

  Either the `SPIN_GetPin` function was called while the previous secure PIN entry process was still in progress, or the `SecurePinAssets.enable` function was not executed in the Java code before calling the `SPIN_GetPin` function.

- `SPIN_INVALID_PARAMETERS`

  Invalid parameters were provided to the `SPIN_GetPin` function.

- `SPIN_INSECURE_CONDITION`

  The PIN entry form encountered an insecure condition that the user can fix. For example, the form lost focus or the developer mode is active.

  Please note that if a security threat is detected (debugger, rooted device, or code tampering) the application will crash immediately.

- `SPIN_MISSING_CHAR`

  One of the text lines supplied to the PIN entry form contained characters that do not belong to the range of allowed characters, which are specified in the corresponding `layout.config` file using the `text_line_*_chars` keys.

- `SPIN_GENERIC_ERROR`

  Unknown error occurred in the Secure PIN code. In this case, please contact Zimperium.

## 15.5.3 SPIN_GetPin

The `SPIN_GetPin` is the only function exposed by the Secure PIN API. By calling this function you will initialize and open the secure PIN entry form. When the PIN entry pad is closed, this function will return a value of the `SPIN_Result` enumeration representing the outcome state.

The function is declared as follows:

```
SPIN_Result SPIN_GetPin(
    const SPIN_LayoutConfig* view_config,
```

```
    bool                    show_upside_down,
    bool                    randomize_position,
    unsigned int            timeout_ms,
    unsigned int            allowed_pin_lengths,
    const void*             messages[],
    size_t                  message_count,
    bool                    ignore_missing_chars,
    const                   SKB_SecureData* pin_block_encryption_key,
    void*                   pin_block_buffer
);
```

The following are the function arguments:

- `view_config`

  Pointer to the SPIN_LayoutConfig structure, which contains configuration for the PIN entry form layout.

- `show_upside_down`

  Boolean variable that determines whether the PIN entry form should be displayed upside down. This is a convenience feature that might be useful to some users accepting payments.

- `randomize_position`

  Boolean variable that determines whether the PIN entry buttons are placed at a fixed position (`false`), or they randomly move around the pad after each touch of the button (`true`).

  The size and position of the boundary rectangle is set in the `SPIN_LayoutConfig` structure.

- `timeout_ms`

  Number of milliseconds during which the user must enter the PIN. If the specified time elapses, the PIN entry form will automatically close.

- `allowed_pin_lengths`

  Bit flag integer that determines the allowed PIN lengths. Each bit in the integer controls one particular PIN length and corresponds to one of the following `#define` macros:

```
#define SPIN_BYPASS_ALLOWED (1u << 0 )
#define SPIN_PIN_LENGTH_4   (1u << 4 )
#define SPIN_PIN_LENGTH_5   (1u << 5 )
#define SPIN_PIN_LENGTH_6   (1u << 6 )
#define SPIN_PIN_LENGTH_7   (1u << 7 )
#define SPIN_PIN_LENGTH_8   (1u << 8 )
#define SPIN_PIN_LENGTH_9   (1u << 9 )
#define SPIN_PIN_LENGTH_10  (1u << 10)
#define SPIN_PIN_LENGTH_11  (1u << 11)
#define SPIN_PIN_LENGTH_12  (1u << 12)
```

As can be seen, the supported lengths are from 4 to 12 digits. SPIN_BYPASS_ALLOWED is a special bit flag that, if set to 1, allows the user to confirm the PIN (press the confirmation button) without entering any PIN digits, which will produce the SPIN_BYPASS return value.

Here is an example of a variable that can be supplied to the allowed_pin_lengths argument to set lengths 4 and 8:

```
unsigned int allowed_lengths = SPIN_PIN_LENGTH_4 | SPIN_PIN_LENGTH_8;
```

- messages

  Pointer to an array of char strings in UTF-8 encoding that represent the four text lines displayed on the layout.

  The array may contain up to four strings corresponding to the four text lines. If less than four strings are provided, the empty string character will be used as the default value.

  The third of the four strings (index 2) will display the asterisks representing the entered PIN digits. Therefore, if the third string contains any text, it will be overwritten with asterisks once the user presses any of the digit buttons.

- message_count

  Number of strings in the char array referred by the messages argument.

- ignore_missing_chars

  Boolean variable that determines whether the PIN entry form should permit any of the four messages to contain characters that do not belong to the range of allowed characters. The allowed characters are specified in the corresponding layout.config file using the text_line_*_chars keys.

  If this argument is set to false, the PIN entry form will automatically close if any of the four messages contains missing characters. Otherwise, all missing characters will be displayed as rectangles.

- pin_block_encryption_key

  Pointer to the SKB_SecureData object holding the 128-bit AES key used to encrypt the PIN.

- pin_block_buffer

  Pointer to a buffer of at least 16 bytes where the partial ISO Format 4 PIN block (Intermediate Block A) will be written after successful PIN entry. For details on this format, see the following document:

  https://listings.pcisecuritystandards.org/documents/Implementing_ISO_Format_4_PIN_Blocks_Information_Supplement.pdf

## 15.6 Secure PIN Example

There is a Secure PIN example project provided in the Examples/AndroidPinPad directory of the SKB package. For information on building and running this project and re-generating the Secure PIN configuration, please read the accompanying README.txt file.

# 16. Accessing Trusted Storage Data and Keys

This chapter describes the library provided by SKB for accessing generic data and keys from the Trusted Storage of Trustonic Application Protection (TAP) SDK.

## 16.1 Trusted Storage Overview

The Trusted Storage is a component of TAP SDK that provides a secure mechanism for storing cryptographic keys and generic data within a file. Customers who use TAP SDK and want to integrate with SKB or migrate to SKB need a way to read contents from the Trusted Storage. For this purpose, SKB offers a library named `SkbTrustedStorage`. This library provides read-only access to a Trusted Storage file and allows you to retrieve generic data as cleartext, and keys as secure data objects.

**Important**

Currently, the `SkbTrustedStorage` library is supported only for iOS and Android with the ARM architecture.

## 16.2 Overview of the "SkbTrustedStorage" Library

`SkbTrustedStorage` is a static library that must be linked with your application. The SKB package provides these files:

- `libSkbTrustedStorage.a`: The prebuilt library located in the `lib/google-android/arm64-v8a`, `lib/google-android/armeabi-v7a`, or `lib/apple/iphoneos` directory, depending on the target platform.
- `Include/SkbTrustedStorage.h`: The header file for accessing the library API.

The `SkbTrustedStorage` library depends on the main native SKB library, so it also must be linked.

To enable the `SkbTrustedStorage` library to read the Trusted Storage file, your SKB package must have the same export key as was used for the TAP package. Also, the device ID set in the SKB must be the same as the one that was set when the Trusted Storage was created by TAP.

## 16.3 Library API

This section describes the main structures and functions declared in the `SkbTrustedStorage.h` header.

### 16.3.1 SKB_TS_UUID

This structure contains data for deriving the Trusted Application Universally Unique Identifier (UUID) according to the RFC 4122. The UUID is one of the input parameters for the `SKB_TS_Init` function.

The structure is declared as follows:

```
typedef struct {
    uint32_t timeLow;
    uint16_t timeMid;
    uint16_t timeHiAndVersion;
    uint8_t clockSeqAndNode[8];
} SKB_TS_UUID;
```

The following are the properties used:

- `timeLow`

  Integer containing the low 32 bits of the time.

- `timeMid`

  Integer containing the middle 16 bits of the time.

- `timeHiAndVersion`

  Integer containing the 4-bit version in the most significant bits, followed by the high 12 bits of the time.

- `clockSeqAndNode`

  Integer containing the following data, starting with the most significant bits:

  ◦ 1 to 3-bit variant

  ◦ 13 to 15-bit clock sequence

  ◦ 48-bit node ID

## 16.3.2 SKB_TS_Parameters

This structure is used to provide input parameters for the SKB_TS_Init function.

The structure is declared as follows:

```
typedef struct {
    char*              dir_path;
    SKB_Size           dir_path_len;
    SKB_TS_UUID UUID;
} SKB_TS_Parameters;
```

The following are the properties used:

- `dir_path`

  Path to the directory containing the Trusted Storage file (extension `.tf`) whose contents you want to access. Usually, it should be set to the value returned by the `getFilesDir()` Java method on Android, or the `/Library/Persistent/SFS` path in the iOS application's sandbox directory. The full path to the `/Library` directory for an iOS application can be acquired with the following Objective-C code:

```
NSArray *dirPaths = NSSearchPathForDirectoriesInDomains(NSLibraryDirectory,
NSUserDomainMask, YES);
NSString *libDir = [dirPaths objectAtIndex:0];
```

- `dir_path_len`

  Length of the `dir_path` value in bytes.

- `UUID`

  Instance of the SKB_TS_UUID structure, which represents your Trusted Application UUID.

## 16.3.3 SKB_TS_KeyInfo

This structure contains information about a key extracted from the Trusted Storage by the SKB_TS_ReadKey function.

The structure is declared as follows:

```
typedef struct {
    SKB_Size       size;
    SKB_TS_KeyType type;
} SKB_TS_KeyInfo;
```

The following are the properties used:

- `size`

  Size of the key in bytes.

- `type`

  Key type, which is specified by one of the values of the SKB_TS_KeyType enumeration.

## 16.3.4 SKB_TS_KeyType

This enumeration contains types of keys that can be extracted from the Trusted Storage. These values are used in the SKB_TS_KeyInfo structure to specify the type of the key extracted by the SKB_TS_ReadKey function.

The enumeration is defined as follows:

```
typedef enum {
    SKB_TS_KEY_TYPE_AES,
    SKB_TS_KEY_TYPE_DES,
    SKB_TS_KEY_TYPE_DES3,
    SKB_TS_KEY_TYPE_HMAC_MD5,
    SKB_TS_KEY_TYPE_HMAC_SHA1,
    SKB_TS_KEY_TYPE_HMAC_SHA224,
```

```
    SKB_TS_KEY_TYPE_HMAC_SHA256,
    SKB_TS_KEY_TYPE_HMAC_SHA384,
    SKB_TS_KEY_TYPE_HMAC_SHA512,
    SKB_TS_KEY_TYPE_RSA_KEYPAIR,
    SKB_TS_KEY_TYPE_ECDSA_KEYPAIR,
    SKB_TS_KEY_TYPE_ECDH_KEYPAIR,
    SKB_TS_KEY_TYPE_DSA_KEYPAIR,
    SKB_TS_KEY_TYPE_GENERIC_SECRET,
} SKB_TS_KeyType;
```

These enumeration values are similar to the values defined in the `tee_internal_api.h` file. Other TAP key types are not supported.

## 16.3.5 SKB_TS_Result

This enumeration represents all the possible outcome states of calling the `SkbTrustedStorage` functions.

The enumeration is defined as follows:

```
typedef enum {
  SKB_TS_SUCCESS,
  SKB_TS_ERROR_ITEM_NOT_FOUND,
  SKB_TS_ERROR_CORRUPT_OBJECT,
  SKB_TS_ERROR_INVALID_PARAMETERS,
  SKB_TS_ERROR_BUFFER_TOO_SMALL,
  SKB_TS_ERROR_NOT_SUPPORTED,
  SKB_TS_ERROR_GENERIC,
} SKB_TS_Result;
```

The following values are available:

- `SKB_TS_SUCCESS`

  The function was executed successfully.

- `SKB_TS_ERROR_ITEM_NOT_FOUND`

  The specified object ID was not found in the Trusted Storage file.

- `SKB_TS_ERROR_CORRUPT_OBJECT`

  The object in the Trusted Storage file was invalid.

- `SKB_TS_ERROR_INVALID_PARAMETERS`

  Incorrect parameters were provided to the function.

- `SKB_TS_ERROR_BUFFER_TOO_SMALL`

The provided memory buffer was not large enough to contain the output.

- `SKB_TS_ERROR_NOT_SUPPORTED`

  An SKB feature required by the `SkbTrustedStorage` library is not available.

- `SKB_TS_ERROR_GENERIC`

  Unknown error occurred. In this case, please contact Zimperium.

## 16.3.6 SKB_TS_Init

This is the first function you must call when using the `SkbTrustedStorage` library. It creates the `SKB_TrustedStorage` object, which must then be passed to all the other functions of the API.

The function is declared as follows:

```
SKB_TS_Result SKB_TS_Init(
    SKB_Engine*                    engine,
    const SKB_TS_Parameters* params,
    SKB_TrustedStorage**           trusted_storage
);
```

The following are the function arguments:

- `engine`

  Pointer to the pre-initialized `SKB_Engine` object.

- `params`

  Pointer to the `SKB_TS_Parameters` structure, which provides the input parameters.

- `trusted_storage`

  Address of a pointer to the `SKB_TrustedStorage` object that will be created by this function. You must pass this object to the other library functions. When you are done working with the `SkbTrustedStorage` library, the object must be released using the `SKB_TS_Release` function.

**Note**

`SKB_TrustedStorage` is a singleton object, which means that it is not possible to hold multiple objects initialized with different parameters simultaneously. The `SKB_TS_Init` function can be called again with different parameters only after releasing the previous instance by calling the `SKB_TS_Release` function.

## 16.3.7 SKB_TS_ReadKey

This function extracts a specified key as a secure data object from the Trusted Storage.

The function is declared as follows:

```
SKB_TS_Result SKB_TS_ReadKey(
    const SKB_TrustedStorage* trusted_storage,
    uint32_t                  storage_id,
    const void*               object_id,
    size_t                    object_id_len,
    SKB_SecureData**          extracted_key,
    SKB_TS_KeyInfo*           key_info
);
```

The following are the function arguments:

- `trusted_storage`

  The `SKB_TrustedStorage` object created using the `SKB_TS_Init` function.

- `storage_id`

  Storage ID, which must always be the constant `TS_STORAGE_PRIVATE` or `1`.

- `object_id`

  Object ID of the key you want to extract.

- `object_id_len`

  Size of the `object_id` value in bytes.

- `extracted_key`

  Address of a pointer to the `SKB_SecureData` object that will contain the extracted key after the function is executed. This object must be released by calling the `SKB_SecureData_Release` method when no longer needed.

- `key_info`

  Pointer to the `SKB_TS_KeyInfo` structure that will be populated by this function to provide information about the extracted key.

## 16.3.8 SKB_TS_ReadData

This function extracts generic data as cleartext from the Trusted Storage.

The function is declared as follows:

```
SKB_TS_Result SKB_TS_ReadData(
   const SKB_TrustedStorage* trusted_storage,
   uint32_t                  storage_id,
   const void*               object_id,
   size_t                    object_id_len,
   void*                     decrypted_data,
```

```
    size_t*                  decrypted_data_len
);
```

The following are the function arguments:

- `trusted_storage`

  The SKB_TrustedStorage object created using the SKB_TS_Init function.

- `storage_id`

  Storage ID, which must always be the constant TS_STORAGE_PRIVATE or 1.

- `object_id`

  Object ID of the data you want to extract.

- `object_id_len`

  Size of the `object_id` value in bytes.

- `decrypted_data`

  This parameter is either NULL or a pointer to the memory buffer where the extracted data is to be written.

  If this parameter is NULL, the method simply returns, in `decrypted_data_len`, the number of bytes that would be sufficient to hold the output, and returns SKB_TS_SUCCESS.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there, sets `decrypted_data_len` to the exact number of bytes stored, and returns SKB_TS_SUCCESS. If the buffer is not large enough, then the method sets `decrypted_data_len` to the number of bytes that would be sufficient, and returns SKB_TS_ERROR_BUFFER_TOO_SMALL.

- `decrypted_data_len`

  Pointer to a variable that holds the size of the memory buffer in bytes where the extracted data is to be stored. For more details, see the description of the `decrypted_data` parameter.

## 16.3.9 SKB_TS_ReadObjectIds

This function extracts object IDs from the Trusted Storage.

The function is declared as follows:

```
SKB_TS_Result SKB_TS_ReadObjectIds(
   const SKB_TrustedStorage* trusted_storage,
   uint32_t                  storage_id,
   void*                     object_id_list,
```

```
    size_t*                 object_id_list_len
);
```

The following are the function arguments:

- `trusted_storage`

  The SKB_TrustedStorage object created using the SKB_TS_Init function.

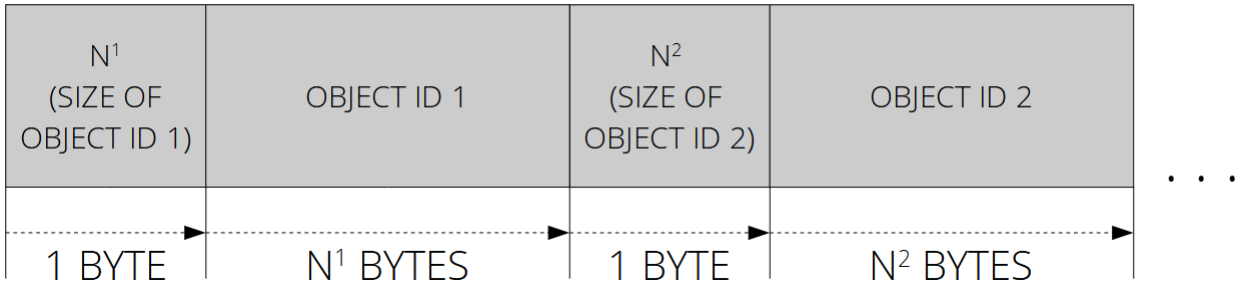
- `storage_id`

  Storage ID, which must always be the constant TS_STORAGE_PRIVATE or 1.

- `object_id_list`

  This parameter is either NULL or a pointer to the memory buffer where the extracted object IDs are to be written.

  The format of the returned object IDs is depicted in the following diagram.




  The object ID size is always stored in one byte. Each object ID is the verbatim copy of the object ID with exactly the size listed in the preceding object ID size field.

  If this parameter is NULL, the method simply returns, in `object_id_list_len`, the number of bytes that would be sufficient to hold the output, and returns SKB_TS_SUCCESS.

  If this parameter points to a memory buffer (it is not NULL), and the buffer size is large enough to hold the output, the method stores the output there, sets `object_id_list_len` to the exact number of bytes stored, and returns SKB_TS_SUCCESS. If the buffer is not large enough, then the method sets `object_id_list_len` to the number of bytes that would be sufficient, and returns SKB_TS_ERROR_BUFFER_TOO_SMALL.

- `object_id_list_len`

  Size of the `object_id_list` value in bytes.

## 16.3.10 SKB_TS_Release

This function releases the SKB_TrustedStorage object from the memory. You must call this function when you are done working with the SkbTrustedStorage library.

The function is declared as follows:

```
SKB_TS_Result SKB_TS_Release(SKB_TrustedStorage* trusted_storage);
```

The parameter `trusted_storage` is a pointer to the `SKB_TrustedStorage` object to be released.

## 16.4 Trusted Storage Example

There is an example project in the `Examples/SkbTrustedStorage` directory of the SKB package, which demonstrates the features of the `SkbTrustedStorage` library. For information on building and running this project, please read the accompanying `README.txt` file.

# 17. Data Formats

This is a reference chapter describing various data formats used in SKB.

## 17.1 Wrapped Data Buffer

This section describes the format of the encrypted data buffer, such as raw bytes, private RSA key, private DSA key, or the encrypted part of a wrapped private ECC key, that is either to be passed to the Triple DES or AES unwrapping algorithm, or is the output of the Triple DES or AES wrapping algorithm. Different block cipher modes of operation and padding schemes are described in separate subsections. In all cases, the big-endian encoding is used to represent numbers.

**Note**

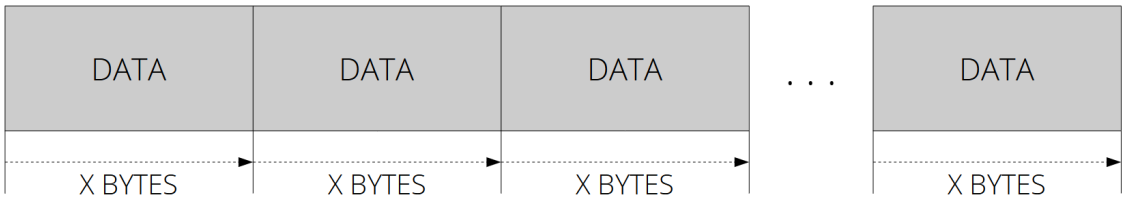
Not all cipher modes and padding schemes are supported for all cipher algorithms. For full information on which combinations are supported, see Supported Algorithms.

### 17.1.1 ECB Mode

In ECB mode, the size of the wrapped data buffer is an exact multiple of the block size. Two padding schemes are supported in this mode.

#### No Padding

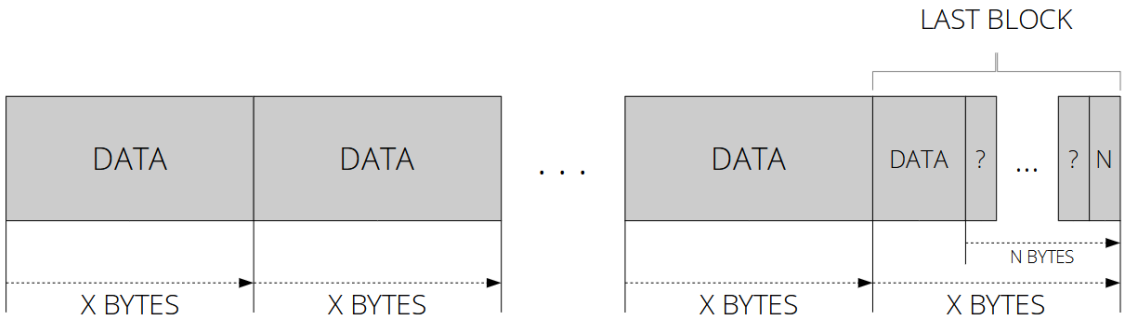
If no padding is used, it is assumed that the size of the encrypted message is an exact multiple of the block size, and nothing is suffixed to the end of the buffer.




X is the block size in bytes, which is 8 for Triple DES and 16 for AES.

#### XML Encryption Padding

SKB supports the ECB mode with padding conventions of the standard XML encryption, which are described at https://www.w3.org/TR/xmlenc-core/#sec-Padding. This means that if the size of the encrypted message within the data buffer is not an exact multiple of the block size, the last block must be padded by suffixing additional bytes to the data buffer to reach a multiple of the block size. The last byte in the last block must contain a number that specifies how many bytes must be stripped from the end of the decrypted data. Other added bytes are arbitrary.




X is the block size in bytes, which is 8 for Triple DES and 16 for AES. N is the number of bytes added to the last block. If the message size happens to be an exact multiple of the block size, an additional block is added, in which the contents are arbitrary, but the last byte contains the number 8 or 16, depending on the block size.

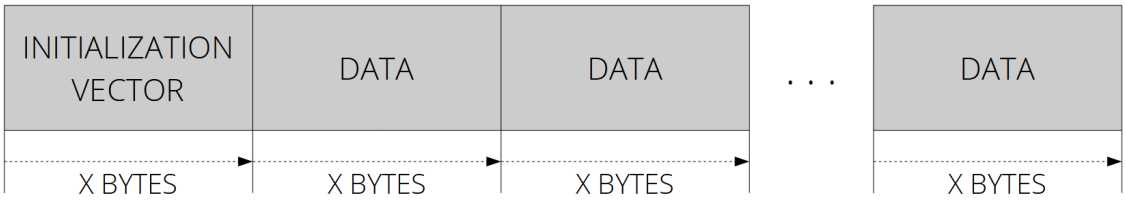
## 17.1.2 CBC Mode

In SKB, two padding schemes are supported for the CBC mode. In both cases, the wrapped data buffer begins with the initialization vector whose size is the same as the block size, followed by a data buffer that is a multiple of the block size.

The following subsections describe the two supported padding schemes.

### No Padding

If no padding is used, it is assumed that the size of the encrypted message is an exact multiple of the block size, and nothing is suffixed to the end of the buffer.

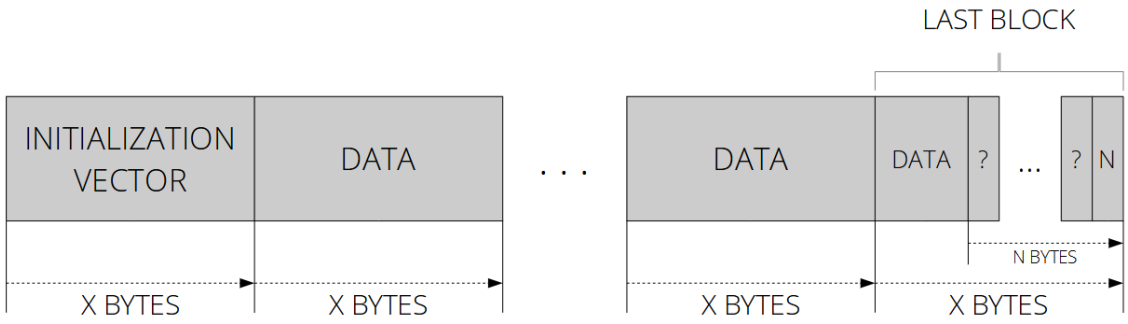



X is the block size in bytes, which is 8 for Triple DES and 16 for AES.

### XML Encryption Padding

SKB supports the CBC mode with padding conventions of the standard XML encryption, which are described at https://www.w3.org/TR/xmlenc-core/#sec-Padding. This means that if the size of the encrypted message within the data buffer is not an exact multiple of the block size, the last block must be padded by suffixing additional bytes to the data buffer to reach a multiple of the block size. The last byte in the last block must contain a number that specifies how many bytes must be stripped from the end of the decrypted data. Other added bytes are arbitrary.

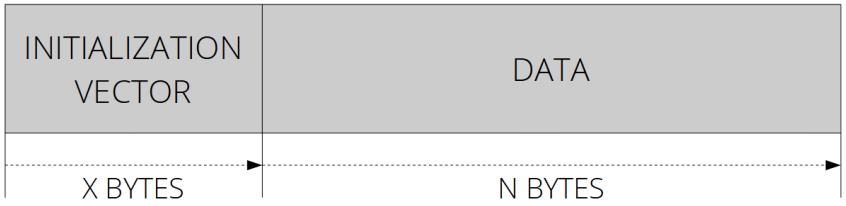
 




X is the block size in bytes, which is 8 for Triple DES and 16 for AES. N is the number of bytes added to the last block. If the message size happens to be an exact multiple of the block size, an additional block is added, in which the contents are arbitrary, but the last byte contains the number 8 or 16, depending on the block size.

### 17.1.3 CTR Mode

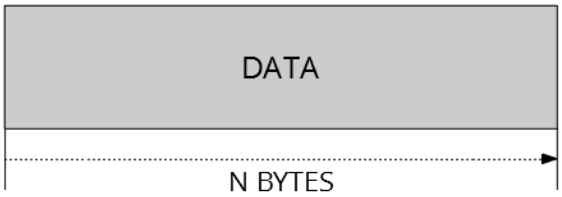
In CTR mode, the wrapped data buffer begins with the initialization vector whose size is the same as the block size (16 bytes for AES), followed by a data buffer of N bytes. N is an arbitrary number, not necessarily a multiple of the block size.




X is the block size in bytes, which is 16 for AES.

### 17.1.4 GCM Mode

In GCM mode, the wrapped data is a simple buffer of arbitrary length. No additional data or padding need to be accounted for.




## 17.2 Key Format for the Triple DES Cipher

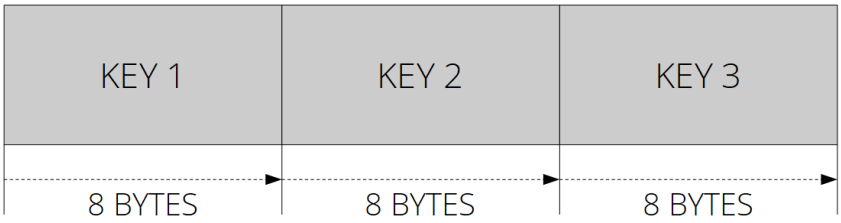
SKB supports two keying options for the Triple DES cipher:

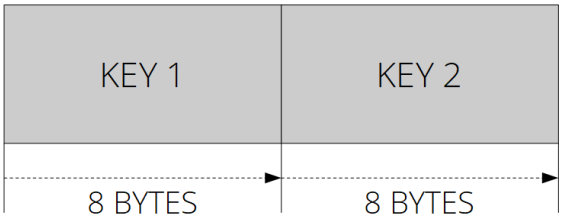
- All three keys are distinct.
- Key 1 and key 2 are distinct, but key 3 is identical to key 1.

In both cases, the keys have to be provided as one buffer of bytes. SKB determines the keying option to be used based on the buffer size.

If the buffer is 192 bits long, SKB assumes the keys are provided in the following format.




If the buffer is 128 bits long, SKB assumes the keys are provided in the following format.




In the latter case, it is assumed that key 3 is identical to key 1.

## 17.3 Input Buffer for the ElGamal ECC Cipher

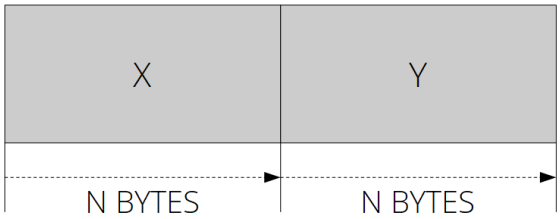
The buffer that is passed as an input to the ElGamal ECC decryption and unwrapping algorithms must contain coordinates of two points on an ECC curve. The two points correspond to the two arguments $c_1$ and $c_2$ of the ElGamal algorithm such that $(c_1, c_2) = (g^y, m' \cdot h^y)$, where g is the generator, y is the random value, m' is the message, h is $g^x$, and x is the private key.

The input buffer must correspond to the following format.


$X_1$ and $Y_1$ are the X and Y coordinates of $c_1$, and $X_2$ and $Y_2$ are the X and Y coordinates of $c_2$ encoded using the big-endian format. N is the number of bytes used to store each coordinate, which is calculated as N = (L+7) / 8, where L is the length of the curve in bits.

## 17.4 Public ECC Key

When you extract the public ECC key from a private ECC key using `SKB_DATA_FORMAT_ECC_PUBLIC` or `SKB_DATA_FORMAT_ECC_BINARY` data format, or when you obtain a public ECC key as part of the ECDH algorithm, the key will be stored in a data buffer formatted as described in the following diagram. Similarly, SKB expects you to use the same format when supplying peer's public ECC key to the shared secret computation function of the ECDH algorithm.

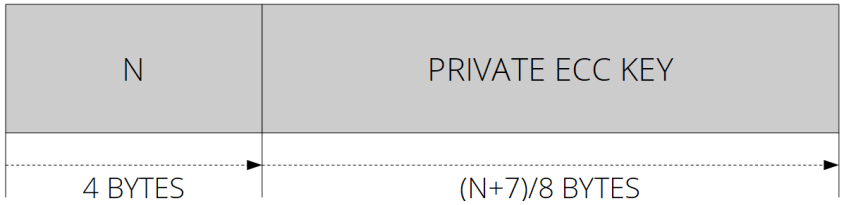



X and Y are the coordinates of the public key encoded using the big-endian format, and N is the number of bytes used to store each coordinate. N depends on the ECC curve as shown in the following table.

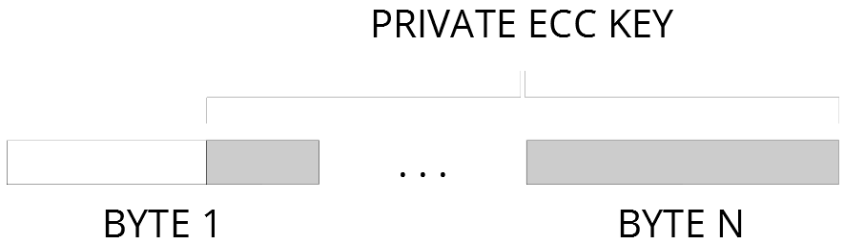
| **Curve Type** | **N (Bytes)** |
|---|---|
| `SKB_ECC_CURVE_SECP_R1_160` | 20 |
| `SKB_ECC_CURVE_NIST_192` | 24 |
| `SKB_ECC_CURVE_NIST_224` | 28 |
| `SKB_ECC_CURVE_NIST_256` | 32 |
| `SKB_ECC_CURVE_NIST_384` | 48 |
| `SKB_ECC_CURVE_NIST_521` | 66 |
| `SKB_ECC_CURVE_CUSTOM` | (prime domain parameter bit-length + 7) / 8 |

## 17.5 Private ECC Key

In case you are loading a plain private ECC key using the Key Export Tool or the SKB_CreateEccPrivateFromPlain function of the Sensitive Operations Library, SKB expects you to provide the private ECC key using the following format.

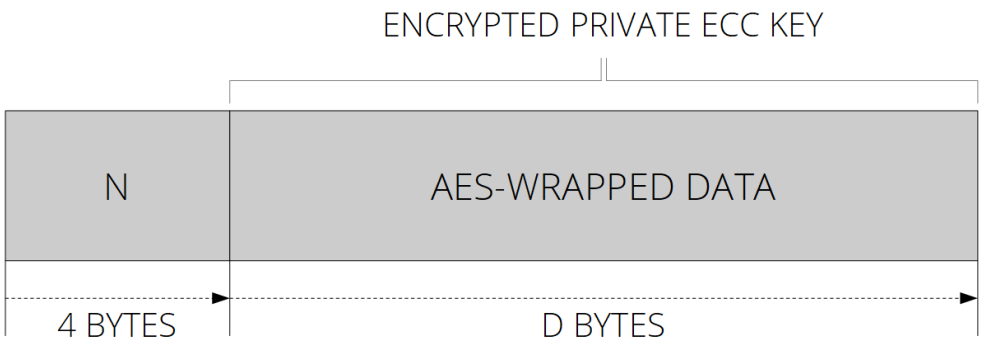



N, which is stored in the first four bytes, is the bit-length of the ECC curve. The rest of the buffer is the actual private ECC key. Both parameters use the big-endian format. The private ECC key is encoded such that the least significant bit of the last byte of the encoded number contains the least significant bit of the number as shown in the following diagram.




## 17.6 AES-Wrapped Private ECC Key

If you are unwrapping an AES-wrapped private ECC key, the input buffer for the unwrapping algorithm must correspond to the following format. The same format is used by SKB when producing the output buffer by wrapping a private ECC key using AES.

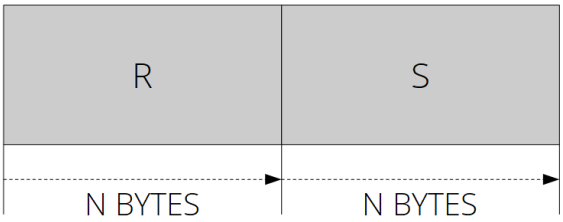



The first 4 bytes must contain the number N in plain (encoded in the big-endian format), which is the bit-length of the ECC curve used. The rest of the buffer is AES-wrapped data, containing the private ECC key. Once the AES-wrapped data is decrypted, the first (N+7)/8 bytes are taken as the actual unwrapped private ECC key.

The private ECC key will be encoded (in case you use SKB for wrapping) or must be encoded (in case you use SKB for unwrapping) such that the least significant bit of the last byte of the encoded number contains the least significant bit of the number as described in Private ECC Key.

## 17.7 ECDSA Output

The format of the output of the ECDSA algorithm is shown in the following diagram.




R and S are the two parameters of the signature used in the ECDSA algorithm, encoded using the big-endian format. N depends on the ECC curve used as described in the following table.

| **Curve Type** | **N (Bytes)** |
|---|---|
| SKB_ECC_CURVE_SECP_R1_160 | 21 |
| SKB_ECC_CURVE_NIST_192 | 24 |
| SKB_ECC_CURVE_NIST_224 | 28 |
| SKB_ECC_CURVE_NIST_256 | 32 |
| SKB_ECC_CURVE_NIST_384 | 48 |
| SKB_ECC_CURVE_NIST_521 | 66 |
| SKB_ECC_CURVE_CUSTOM | (order domain parameter bit-length) + 7 / 8 |

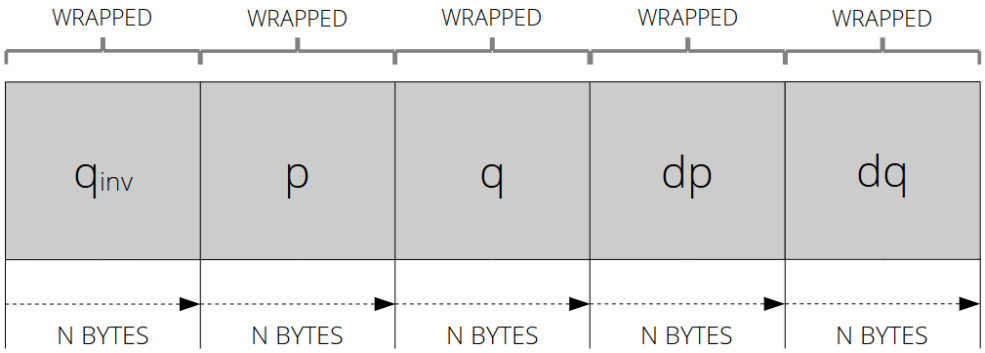
# 17.8 AES-Wrapped Private RSA Key in the CRT Format

SKB supports a special type of AES-based unwrapping of private RSA keys that follows the optimization principles of the Chinese Remainder Theorem (CRT). For this algorithm to work correctly, you must ensure the input buffer containing the wrapped private RSA key is formatted as described in this section.

The subsequent sections describe two flavors of this algorithm. In one of them, the individual components of the wrapped key are of equal size; in the other, the components can have different sizes.

## 17.8.1 Equal Length

The following diagram describes what the input buffer must be when you are working with the `SKB_DATA_FORMAT_CRT_EQUAL_LEN` format.

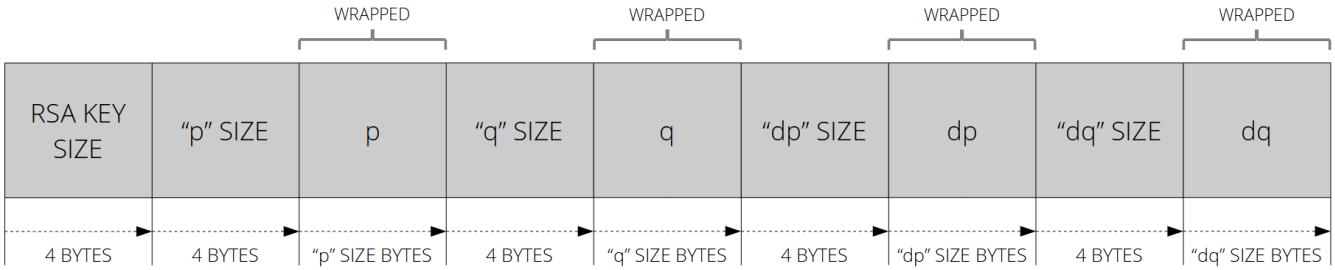



In this case, each of the equal-length blocks is the big-endian format of the corresponding key component with the ISO/IEC 7816-4 padding applied, individually wrapped with the chosen version of AES. N must be a multiple of 16, and it must be one fifth of the overall buffer size. If the CBC mode is used, each component must be wrapped with the initialization vector set to 0.

Note that, unlike the variable length version, this version will also compute the public component e, and if this component does not fit in 8 bytes, the resulting RSA key will be corrupted.

## 17.8.2 Variable Length

The following diagram describes what the input buffer must be when you are working with the `SKB_DATA_FORMAT_CRT` format.




In this case, the buffer starts with four bytes containing the big-endian representation of the RSA key size in bytes, followed by four pairs corresponding to the four components p, q, dp, and dq respectively. Each pair consists of:

- Four bytes containing the big-endian representation of the size of the corresponding component in bytes
- Actual component in big-endian with the ISO/IEC 7816-4 padding applied, individually wrapped with the chosen version of AES.

If the CBC mode is used, each component must be wrapped with the initialization vector set to 0. The size of each wrapped component must be a multiple of 16.