

white**C**ryption®



# Secure Key Box 4.20 User Guide

The software referenced herein, this User Guide, and any associated documentation is provided to you pursuant to the agreement between your company, governmental body or other entity (“you”) and whiteCrypton Corporation (“whiteCrypton”) under which you have received a copy of Secure Key Box Licensed Technology and various related documentation, including this User Guide (such agreement, the “Agreement”). Defined terms not defined herein shall have the meanings ascribed to them in the Agreement. In the event of conflict between the terms of this User Guide and the terms of the Agreement, the terms of the Agreement shall prevail. Without limiting the generality of the remainder of this paragraph, (a) this User Guide is provided to you for informational purposes only, (b) your right to access, view, use, and copy this User Guide is limited to the rights and subject to the applicable requirements and limitations set forth in the Agreement, and (c) all of the content of this User Guide constitutes “Confidential Information” of whiteCrypton (or the equivalent term used in the Agreement) and is subject to all of the limitations and requirements pertinent to the use, disclosure and safeguarding of such information. Permitting anyone who is not directly involved in the authorized use of Secure Key Box Licensed Technology by your company or other entity to gain any access to this User Guide shall violate the Agreement and subject your company or other entity to liability therefor.

## Copyright and Trademark Information

Copyright © 2000-2015, whiteCrypton Corporation. All rights reserved.

Copyright © 2004-2015, Intertrust Technologies Corporation. All rights reserved.

whiteCrypton® and Cryptanium™ are either registered trademarks or trademarks of whiteCrypton Corporation in the United States and/or other countries.

Microsoft®, Visual Studio®, and Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

OS X® and Xcode® are trademarks of Apple Inc., registered in the United States and other countries.

IOS® is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Google is a registered trademark of Google Inc., used with permission.

Android™ is a trademark of Google Inc., registered in the United States and other countries.

PlayStation is a trademark or registered trademark of Sony Computer Entertainment Inc.

Sourcery™ CodeBench is a trademark of Mentor Graphics Corporation.

Broadcom® is a registered trademark of Broadcom Corporation.

## Disclaimer

The remainder of this User Guide notwithstanding, this User Guide is provided “as is”, without any warranty whatsoever (including that it is error-free or complete). This User Guide contains no express or implied warranties, covenants or grants of rights or licenses, and does not modify or supplement any express warranty, covenant or grant of rights or licenses that is set forth in the Agreement. This User Guide is current as of the date set forth in the header that appears above on this page, but may be modified at any time without prior notice. Future revisions and updates of this User Guide shall be distributed as part of Secure Key Box new releases. whiteCrypton shall under no circumstances bear any responsibility for your failure to operate Secure Key Box Licensed Technology in compliance with the then-current version of this User Guide. Your remedies with respect to your use of this User Guide, and whiteCrypton’s liability for your use of this User Guide

(including for any errors or inaccuracies that appear in this User Guide) are limited to those remedies expressly authorized by the Agreement (if any).

## Notice to U.S. Government End Users

This User Manual is a "Commercial Item," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States.

## Contact Information

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

[contact@whitecryption.com](mailto:contact@whitecryption.com)

[www.whitecryption.com](http://www.whitecryption.com)

# Table of Contents

---

1 Introduction .....	9
1.1 General Concepts.....	9
1.1.1 What Is SKB?.....	9
1.1.2 Nomenclature.....	10
1.1.3 Purpose of SKB.....	10
1.1.4 White-Box Cryptography.....	10
1.1.5 Secure Data Objects.....	11
1.1.6 Export Key.....	11
1.1.7 Loading the First Key.....	12
1.1.8 Diversification .....	13
1.1.9 Tamper Resistance .....	13
1.1.10 Evaluation and Production Packages.....	16
1.2 Supported Algorithms.....	16
1.3 Supported ECC Curves.....	19
1.4 Supported Target Platforms .....	20
1.5 Directory Structure and Contents .....	21
1.6 Limitations and Known Problems.....	23
2 Building Applications Protected by SKB.....	24
2.1 Building a Protected Application .....	24
2.2 Distributing a Protected Application.....	25
2.3 Building Examples, Tests, and Platform-Specific Library.....	25
2.3.1 Building for Windows API .....	26
2.3.2 Building for Windows Runtime and Windows Phone.....	26
2.3.3 Building for Linux.....	27
2.3.4 Building for Android .....	28
2.3.5 Building for OS X and iOS.....	29
2.3.6 Building for Google Native Client (NaCl).....	30
2.3.7 Building for PlayStation 3 .....	31
3 Cryptographic Operations .....	32
3.1 Loading Wrapped Keys .....	32
3.1.1 Unwrapping Keys Wrapped with the ElGamal ECC Algorithm .....	33
3.2 Loading Plain Keys .....	34
3.3 Wrapping Keys .....	34
3.4 Wrapping Plain Data .....	35
3.5 Exporting Keys.....	35
3.6 Importing Keys .....	36
3.7 Upgrading Exported Keys.....	36
3.8 Generating Keys.....	38
3.9 Deriving a Public Key from a Private Key.....	38
3.10 Deriving Keys.....	38

- 3.10.1 Deriving a Key as a Substring of Bytes of Another Key..... 38
- 3.10.2 Deriving a Key as Odd or Even Bytes of Another Key ..... 39
- 3.10.3 Deriving a Key by Encrypting or Decrypting an Existing Key ..... 39
- 3.10.4 Deriving a Key as a Protected Hash Value of Another Key ..... 39
- 3.10.5 Reversing the Order of Bytes of a Key..... 41
- 3.10.6 Using the NIST 800-108 Key Derivation Function ..... 41
- 3.10.7 Using KDF2 of the RSAES-KEM-KWS Scheme Defined in the OMA DRM Specification..... 42
- 3.10.8 Deriving a Key as Raw Bytes from a Private ECC Key ..... 42
- 3.10.9 Deriving a Key Using the CMLA Key Derivation Function..... 42
- 3.10.10 Deriving a Key By Encrypting Data Using 128-bit AES With a Concatenated Key ..... 42
- 3.11 Encrypting and Decrypting Data..... 44
  - 3.11.1 Encrypting Data..... 44
  - 3.11.2 Decrypting Data ..... 45
  - 3.11.3 Using the High-Speed AES ..... 45
- 3.12 Calculating a Digest ..... 45
- 3.13 Creating a Signature..... 46
- 3.14 Verifying a Signature..... 46
- 3.15 Executing a Key Agreement Algorithm..... 47
- 3.16 Binding Keys to a Specific Device ..... 48
- 3.17 Decrypting Encrypted PDF Documents ..... 48
- 4 Supporting Libraries..... 49
  - 4.1 Sensitive Operations Library..... 49
    - 4.1.1 Overview ..... 49
    - 4.1.2 Library Functions ..... 49
  - 4.2 Platform-Specific Library..... 55
    - 4.2.1 Overview ..... 55
    - 4.2.2 Library Functions ..... 55
    - 4.2.3 Key Caching..... 57
- 5 Utilities..... 60
  - 5.1 Custom ECC Tool..... 60
    - 5.1.1 Custom ECC Tool Overview ..... 60
    - 5.1.2 Parameter Size and Value Restrictions..... 61
    - 5.1.3 Running Custom ECC Tool..... 61
  - 5.2 Diffie-Hellman Tool..... 62
    - 5.2.1 Diffie-Hellman Tool Overview..... 62
    - 5.2.2 Running Diffie-Hellman Tool ..... 62
  - 5.3 Key Export Tool..... 64
    - 5.3.1 Key Export Tool Overview ..... 64
    - 5.3.2 Running Key Export Tool..... 64
  - 5.4 Binary Update Tool ..... 65
    - 5.4.1 Binary Update Tool Overview..... 66
    - 5.4.2 Running the Binary Update Tool ..... 66
- 6 Decrypting PDF Files..... 68

6.1 PDF Encryption Overview .....	68
6.2 PDF Requirements .....	68
6.3 Decrypting a PDF Document Using SKB .....	69
6.3.1 SKB_Pdf_AuthenticateUserPassword .....	69
6.3.2 SKB_Pdf_ComputeEncryptionKey .....	70
6.3.3 SKB_Pdf_CreateDecryptionContext .....	71
6.3.4 SKB_Pdf_DecryptionContext_ProcessBuffer .....	72
6.3.5 SKB_Pdf_DecryptionContext_Release .....	74
7 API Reference.....	75
7.1 API Overview.....	75
7.2 Obtaining Class Instances .....	75
7.3 Making Method Calls .....	75
7.4 Method Return Values .....	76
7.5 Object Lifecycle.....	77
7.6 Restrictions of Multithreading.....	77
7.7 Overriding Memory Allocation Operators .....	78
7.8 Classes .....	79
7.8.1 SKB_Engine.....	79
7.8.2 SKB_SecureData.....	79
7.8.3 SKB_Cipher .....	79
7.8.4 SKB_Transform .....	79
7.8.5 SKB_KeyAgreement .....	79
7.9 Methods .....	80
7.9.1 SKB_Engine_GetInstance.....	80
7.9.2 SKB_Engine_Release.....	80
7.9.3 SKB_Engine_SetDeviceld .....	80
7.9.4 SKB_Engine_GetInfo .....	81
7.9.5 SKB_Engine_CreateDataFromWrapped .....	82
7.9.6 SKB_Engine_CreateDataFromExported .....	84
7.9.7 SKB_Engine_WrapDataFromPlain.....	85
7.9.8 SKB_Engine_GenerateSecureData .....	86
7.9.9 SKB_Engine_CreateTransform .....	87
7.9.10 SKB_Engine_CreateCipher .....	88
7.9.11 SKB_Engine_CreateKeyAgreement .....	90
7.9.12 SKB_Engine_UpgradeExportedData.....	90
7.9.13 SKB_SecureData_Release .....	91
7.9.14 SKB_SecureData_GetInfo .....	92
7.9.15 SKB_SecureData_Export .....	92
7.9.16 SKB_SecureData_Wrap.....	93
7.9.17 SKB_SecureData_Derive.....	95
7.9.18 SKB_SecureData_GetPublicKey .....	97
7.9.19 SKB_Transform_Release.....	98
7.9.20 SKB_Transform_AddBytes .....	98

- 7.9.21 SKB\_Transform\_AddSecureData ..... 99
- 7.9.22 SKB\_Transform\_GetOutput ..... 99
- 7.9.23 SKB\_Cipher\_ProcessBuffer ..... 100
- 7.9.24 SKB\_Cipher\_Release ..... 102
- 7.9.25 SKB\_KeyAgreement\_GetPublicKey ..... 102
- 7.9.26 SKB\_KeyAgreement\_ComputeSecret ..... 103
- 7.9.27 SKB\_KeyAgreement\_Release ..... 104
- 7.10 Supporting Structures ..... 104
  - 7.10.1 SKB\_EngineProperty ..... 105
  - 7.10.2 SKB\_EngineInfo ..... 105
  - 7.10.3 SKB\_DataInfo ..... 106
  - 7.10.4 SKB\_CtrModeCipherParameters ..... 107
  - 7.10.5 SKB\_DigestTransformParameters ..... 107
  - 7.10.6 SKB\_SignTransformParameters ..... 107
  - 7.10.7 SKB\_SignTransformParametersEx ..... 108
  - 7.10.8 SKB\_VerifyTransformParameters ..... 109
  - 7.10.9 SKB\_SelectBytesDerivationParameters ..... 110
  - 7.10.10 SKB\_CipherDerivationParameters ..... 110
  - 7.10.11 SKB\_Sha1DerivationParameters ..... 111
  - 7.10.12 SKB\_Sha256DerivationParameters ..... 112
  - 7.10.13 SKB\_Nist800108CounterCmacAes128Parameters ..... 113
  - 7.10.14 SKB\_RawBytesFromEccPrivateDerivationParameters ..... 114
  - 7.10.15 SKB\_ShaAesDerivationParameters ..... 114
  - 7.10.16 SKB\_OmaDrmKdf2DerivationParameters ..... 115
  - 7.10.17 SKB\_SliceDerivationParameters ..... 115
  - 7.10.18 SKB\_EccDomainParameters ..... 116
  - 7.10.19 SKB\_AesWrapParameters ..... 117
  - 7.10.20 SKB\_AesUnwrapParameters ..... 117
  - 7.10.21 SKB\_RsaPssParameters ..... 117
  - 7.10.22 SKB\_EccParameters ..... 118
  - 7.10.23 SKB\_PrimeDhParameters ..... 119
  - 7.10.24 SKB\_RawBytesParameters ..... 120
- 7.11 Enumerations ..... 120
  - 7.11.1 SKB\_DataType ..... 120
  - 7.11.2 SKB\_DigestAlgorithm ..... 120
  - 7.11.3 SKB\_CipherAlgorithm ..... 120
  - 7.11.4 SKB\_SignatureAlgorithm ..... 123
  - 7.11.5 SKB\_DerivationAlgorithm ..... 124
  - 7.11.6 SKB\_CipherDirection ..... 126
  - 7.11.7 SKB\_DataFormat ..... 126
  - 7.11.8 SKB\_TransformType ..... 127
  - 7.11.9 SKB\_ExportTarget ..... 127
  - 7.11.10 SKB\_EccCurve ..... 128

- 7.11.11 SKB\_KeyAgreementAlgorithm .....129
- 7.11.12 SKB\_PrimeDhLength .....129
- 7.11.13 SKB\_CbcPadding.....129
- 7.11.14 SKB\_SelectBytesDerivationVariant .....130
- 8 Data Formats .....131
  - 8.1 Export Data Format .....131
  - 8.2 AES-Wrapped Data Buffer .....132
    - 8.2.1 ECB Mode .....132
    - 8.2.2 CTR Mode .....132
    - 8.2.3 CBC Mode.....133
  - 8.3 Key Format for the Triple DES Cipher .....134
  - 8.4 Input Buffer for the ElGamal ECC Cipher .....135
  - 8.5 Public ECC Key.....135
  - 8.6 Private ECC Key .....136
  - 8.7 AES-Wrapped Private ECC Key .....136
  - 8.8 ECDSA Output .....137



# 1 Introduction

---

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

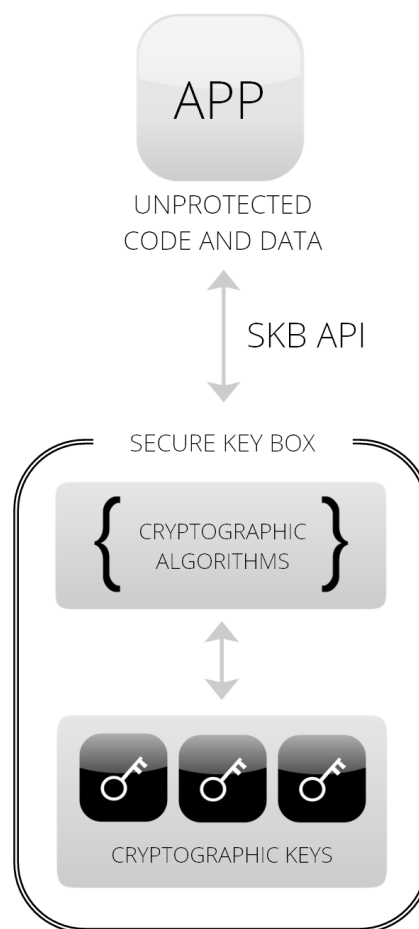
## 1.1 General Concepts

This section describes the general concepts that you should know before working with SKB.

### 1.1.1 What Is SKB?

SKB is a library that provides a set of high-level classes and methods for working with common cryptographic algorithms. The library's white-box implementation hides and protects cryptographic keys. In SKB, keys are encrypted and cryptographic algorithms operate directly with encrypted keys.

SKB exposes an API that provides access to a set of functions, which the calling application uses to implement various cryptographic operations.



## 1.1.2 Nomenclature

This User Guide uses the terms “secure”, “protect”, “white-box protected”, “safe”, “tamper resistance” and variations of each to convey very specific concepts — indeed, concepts that are far more specific and limited in their meanings than many meanings often associated with such terms in everyday usage. At the risk of stating the obvious, 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. Nor will it render any application or data absolutely tamper resistant. In addition, use of these terms is not intended to convey a promise or warranty that SKB will never contain a bug or error, or that SKB will always operate without error.

As used in this User Guide:

- “secure” and variations of “secure” refer to data objects, the values of which reside in a cryptographic container and are white-box protected, and that can be operated on by SKB functions despite the fact that they are not revealed in plain form.
- “protected”, “white-box protected” and variations of these terms mean that a value has been subjected to some cryptographic processing that has resulted in it being placed in a container that seeks to render the value inaccessible in plain form to the outside world.
- “safe”, “safely”, “safer” and variations of these terms refer to actions or objects that when processed in accordance with this User Guide will not compromise the security protections provided by SKB.
- “tamper resistance” and variations of this term refer to the application of whiteCryption’s Code Protection product to render it more difficult for unauthorized parties to engage in reverse engineering and code modification.

## 1.1.3 Purpose of SKB

Cryptographic algorithms and keys are used to protect sensitive data, authenticate communication partners, verify signatures, and implement various other security schemes. A common weak point of cryptographic algorithms in today’s open architectures, such as smartphones, tablets, and desktops, is that the cryptographic keys are revealed in the code or memory at some point. Hackers can monitor such devices with special tools and extract the secret cryptographic keys. Without an efficient protection of cryptographic keys, security features may be compromised.

SKB is designed to prevent such attacks by encrypting and hiding cryptographic keys in the code and memory.

## 1.1.4 White-Box Cryptography

The term “white-box cryptography” is used to describe a secure implementation of cryptographic algorithms in an execution environment that is fully observable and modifiable by an attacker, such as a desktop computer or a mobile device. It is different from black-box cryptography where the

algorithm's internal processing data is unavailable to the attacker. The white-box environment puts certain restrictions on implementations of the cryptographic algorithms. For instance, an encryption key may never appear in plain text; otherwise it can be retrieved by an attacker.

### 1.1.5 Secure Data Objects

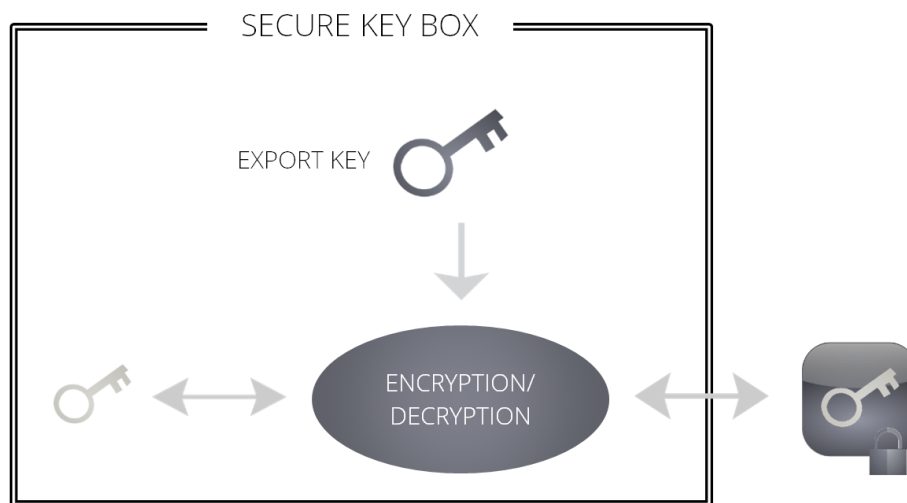
A secure data object (represented by the `SKB_SecureData` class in the API) is one of the basic concepts in the SKB protection scheme. It is a container of any sensitive data whose value is white-box protected and hidden from the outside world. Secure data objects can be operated on by SKB functions even though the contents of secure data objects are not revealed in plain form.

Because secure data objects usually hold cryptographic keys, in this document the terms "secure data object" and "cryptographic key" are used interchangeably.

### 1.1.6 Export Key

Secure data objects (cryptographic keys) are operated on in the device memory. Because the memory is not persistent, there needs to be a mechanism for safely storing secure data objects.

Each SKB package contains an embedded key called the export key. This export key is used for encrypting other cryptographic keys exported from the protected SKB domain into the unprotected outside world. The same export key is used for importing and decrypting the exported data back into SKB.



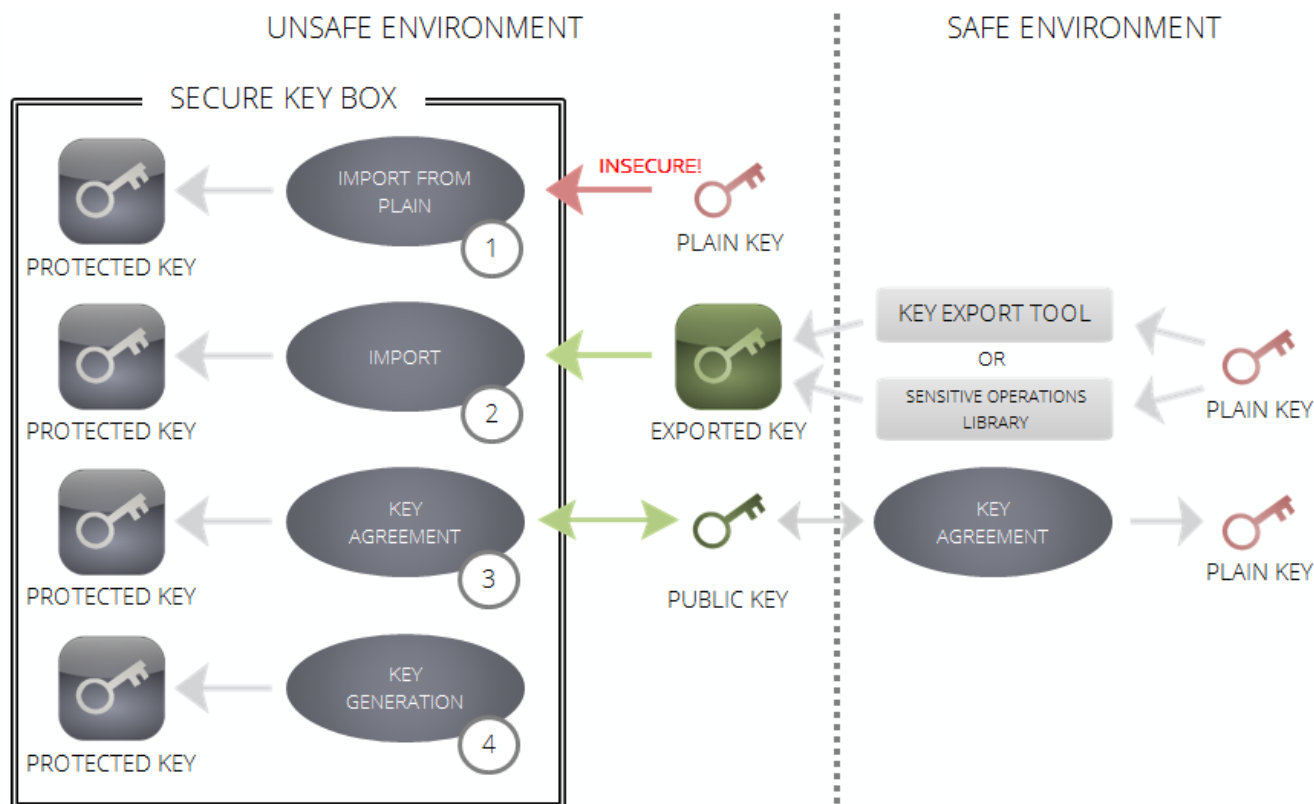
Using an export key

The export key is embedded inside the SKB package and is white-box protected, which means that it is not revealed in plain form in the program code or memory. Exported data can only be imported into an SKB instance that has the same export key.

### 1.1.7 Loading the First Key

Typically, if you need to load a new key, you encrypt it with another key, pass it to SKB, and then internally decrypt it as a new secure data object. However, at some point the first key needs to be loaded into SKB. The challenge is to safely deliver this key into SKB without revealing it in plain form.

The following diagram shows four ways how the first key can be obtained.





Four ways of loading the first cryptographic key into SKB

The diagram features two environments. The unsafe environment (on the left) is where potentially anyone can gain access to the hardware, code, and memory of the device (such as a desktop computer or mobile device). This is the environment where SKB is primarily intended to be used.

**⚠** As a general rule, secret cryptographic keys should never appear in unsafe environments in plain form.

The safe environment (on the right) is a place where cryptographic keys will be exposed in plain form and that you must maintain as safe as possible in terms of potential risks of breaking in, reverse engineering, and exposure to unwelcome parties (such as a closed-off facility or encrypted server with controlled access).

The following table explains the four methods of loading the first key into SKB, highlighted in the diagram.

Method	Description	Reference
1. Import from plain	<p>SKB directly loads a plain key within an insecure environment.</p> <div style="border: 1px solid black; padding: 5px;"> <p> This is an insecure approach for loading keys and should be avoided if possible. In normal circumstances, importing of plain keys is disabled in SKB.</p> </div>	§3.2
2. Import	<p>SKB loads a key in protected form (encrypted with the export key as described in §1.1.6).</p> <p>The protected form of a key can be obtained using either Key Export Tool (see §5.3) or Sensitive Operations Library (see §4.1).</p> <div style="border: 1px solid black; padding: 5px;"> <p> Key Export Tool and Sensitive Operations Library must never be delivered with the final protected application (see §2.2).</p> </div> <p>Note: The same protected form is used when SKB exports a key (see §3.5).</p>	§3.6
3. Key agreement	SKB exchanges public keys with another party and then internally generates a secret key.	§3.15
4. Key generation	SKB internally generates a new key.	§3.8

### 1.1.8 Diversification


A significant feature of SKB is code diversification. It means that each customer receives one or several packages whose binary code differs from other packages. SKB achieves this by generating unique representations of white-box algorithms individually for each customer. Although the API provided by each SKB instance is the same, the way the operations are physically implemented in the program code varies.

This feature improves security. For example, if an adversary manages to compromise a particular system that uses SKB, systems of other customers would not be directly affected.

### 1.1.9 Tamper Resistance

Tamper resistance is an optional feature that you can request for the SKB library delivered to you. Tamper resistance guards the library code against analysis and modification. Although this feature slightly reduces performance of the protected application, it significantly increases security against hacker attacks.

Note: SKB tamper resistance is implemented using Code Protection, a comprehensive tool for hardening software applications on multiple platforms. For information on Code Protection, see [www.whitecrypton.com/code-protection](http://www.whitecrypton.com/code-protection).

 If your SKB package has tamper resistance applied, you have to run the Binary Update Tool on your final application executable every time it is built. Otherwise, the application will crash at run time. For more information on running the Binary Update Tool, see §5.4.

### 1.1.9.1 Security Features

SKB's implementation of tamper resistance consists of a combination of the following security features:

Security feature	Description
Integrity protection	Hundreds of embedded overlapping checksums can prevent modifications of the binary code of the entire executable (not just SKB).
Code obfuscation	Library code is transformed to make it difficult to analyze and reverse engineer.
Anti-debug protection	Platform-specific anti-debug code adds protection against main-stream debuggers providing another barrier to code analysis.
iOS jailbreak detection	Normally, a cracked or modified iOS application can be run only on jailbroken iOS devices. iOS jailbreak detection protects the application from being executed on a jailbroken device.
Android rooting detection	Rooting is a security risk to Android applications that deal with sensitive data or enforce certain usage restrictions. Rooting detection will protect the application if a rooted device is detected.
Inlining of static void functions	Static void functions with simple declarations are inlined into the calling functions. Such operation increases the obfuscation level of the final protected code and makes it more difficult to trace.
String literal obfuscation	Large portion of string literals, or string constants, are 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.

Security feature	Description
Customizable defense action	Optionally, you can request a tamper resistance SKB library that is configured to execute specific callback functions depending on the type of attack it detects. Additionally, when requesting the SKB library, you can choose whether the program state should be corrupted or the application should be left running after a callback function is invoked. For more information on this feature, see §1.1.9.3.

### 1.1.9.2 Supported Platforms

Tamper resistant SKB libraries are currently available only for the following target platforms:

- Windows for Visual Studio 2010, 2012, and 2013 (x86 and x86\_64 architectures)
- GNU/Linux (x86 and x86\_64 architectures)
- OS X (x86 and x86\_64 architectures)
- iOS (ARMv7, ARMv7s, and ARMv8 architectures)
- Android (ARM and x86 architectures)

### 1.1.9.3 Callback Functions

When you request a tamper resistant edition of SKB, 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 (typically, resulting in a crash) whenever it detects a threat. Callback functions allow you to implement custom response to attacks. Additionally, when requesting an SKB library 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 library that you received is configured to use callbacks, you have to provide implementation for the callback functions in the source code. The following table describes the attack types for which callback functions are supported:

Attack	Callback function	Description
Debugger	<code>void SKB_Callback_AntiDebug()</code>	This function is called by SKB when it detects that the application is run under a debugger.
Rooting	<code>void SKB_Callback_Root()</code>	This function is called by SKB when it detects that the application is run on a rooted Android device.

Attack	Callback function	Description
Jailbreak	<code>void SKB_Callback_Jailbreak()</code>	This function is called by SKB when it detects that the application is run on a jailbroken iOS device.

Please note that you have to provide implementations for the above functions only if you specifically requested an SKB library that uses them.

### 1.1.10 Evaluation and Production Packages

Two editions of the SKB package are available to customers:

Edition	Description
Evaluation	<p>The evaluation package is free and is typically given to new customers who want to try out SKB before purchasing a license. You should not use an evaluation edition in a production environment for two reasons. Firstly, you do not have the right to do so. Secondly, all evaluation packages have the same export key (see §1.1.6). This means that all encrypted data that you export from SKB can be decrypted by other evaluation packages.</p> <p>Additionally, evaluation packages have an expiration date set. Once the date is reached, SKB will no longer be usable.</p>
Production	The production package is given to customers who have licensed SKB. Each production package has a unique export key and no expiration date.

## 1.2 Supported Algorithms

This section lists algorithms supported by SKB.

Note: Not all distributions of SKB include all the algorithms listed below.

Function	Algorithms
Encryption	<ul style="list-style-type: none"> <li>▪ DES in ECB mode (no padding)</li> <li>▪ Triple DES in ECB mode (no padding) with two keying options: <ul style="list-style-type: none"> <li>▪ All three keys are distinct.</li> <li>▪ Key 1 is the same as key 3.</li> </ul> </li> <li>▪ 128-bit, 196-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding), and CTR mode</li> </ul>



Function	Algorithms
Decryption	<ul style="list-style-type: none"> <li>▪ DES in ECB mode (no padding)</li> <li>▪ Triple DES in ECB mode (no padding) with two keying options: <ul style="list-style-type: none"> <li>▪ All three keys are distinct.</li> <li>▪ Key 1 is the same as key 3.</li> </ul> </li> <li>▪ 128-bit, 196-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding), and CTR mode</li> <li>▪ 1024-bit and 2048-bit RSA (no padding, PKCS#1 version 1.5 padding, and OAEP padding)</li> <li>▪ ElGamal ECC (for supported curve types, see §1.3)</li> </ul>
Signing	<ul style="list-style-type: none"> <li>▪ 128-bit AES-CMAC (based on OMAC1)</li> <li>▪ HMAC with up to 64-byte keys using SHA-1, SHA-256, SHA-384, or SHA-512 as the hash function</li> <li>▪ 1024-bit and 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 without a hash function</li> <li>▪ 1024-bit and 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using SHA-1 or SHA-256 as the hash function</li> <li>▪ 1024-bit and 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using SHA-1 or SHA-256 as the hash function</li> <li>▪ ECDSA without a hash function (for supported curve types, see §1.3)</li> <li>▪ ECDSA using SHA-1 or SHA-256 as the hash function (for supported curve types, see §1.3)</li> </ul>
Verification	<ul style="list-style-type: none"> <li>▪ 128-bit AES-CMAC (based on OMAC1)</li> <li>▪ HMAC with up to 64-byte keys using SHA-1, SHA-256, SHA-384, or SHA-512 as the hash function</li> </ul>
Key importing	<ul style="list-style-type: none"> <li>▪ plain bytes (for example, DES and AES keys)</li> <li>▪ plain RSA private keys</li> <li>▪ plain ECC private keys</li> </ul>

Function	Algorithms
Unwrapping	<ul style="list-style-type: none"> <li>▪ unwrapping raw bytes (for example, DES and AES keys) using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding), CBC mode (no padding or XML encryption padding), and CTR mode</li> <li>▪ unwrapping RSA keys using 128-bit, 192-bit, and 256-bit AES in CBC mode (XML encryption padding) and CTR mode</li> <li>▪ unwrapping ECC keys using 128-bit, 192-bit, and 256-bit AES in CBC mode (no padding or XML encryption padding) and CTR mode</li> <li>▪ unwrapping raw bytes (for example, DES and AES keys) using 1024-bit and 2048-bit RSA (no padding, PKCS#1 version 1.5 padding, or OAEP padding)</li> <li>▪ unwrapping raw bytes (for example, DES and AES keys) using ElGamal ECC (for supported curve types, see §1.3)</li> <li>▪ AES key unwrapping defined by NIST with 128-bit, 192-bit, and 256-bit AES keys</li> <li>▪ CMLA AES unwrapping defined by the <i>CMLA Technical Specification</i></li> <li>▪ CMLA RSA unwrapping defined by the <i>CMLA Technical Specification</i></li> <li>▪ unwrapping using XOR</li> </ul>
Wrapping	<ul style="list-style-type: none"> <li>▪ wrapping raw bytes (for example, DES and AES keys) using 128-bit, 192-bit, and 256-bit AES in CBC mode (XML encryption padding)</li> <li>▪ wrapping ECC keys using 128-bit, 192-bit, and 256-bit AES in CBC mode (XML encryption padding)</li> <li>▪ wrapping plain data using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding) and CBC mode (no padding)</li> <li>▪ wrapping using XOR</li> </ul>
Digests	<ul style="list-style-type: none"> <li>▪ SHA-1</li> <li>▪ SHA-256</li> <li>▪ SHA-384</li> <li>▪ SHA-512</li> </ul>
Key agreement	<ul style="list-style-type: none"> <li>▪ Classical Diffie-Hellman (DH) with up to 1024-bit prime P</li> <li>▪ Elliptic curve Diffie-Hellman (ECDH) (for supported curve types, see §1.3)</li> </ul>


Function	Algorithms
Key generation	<ul style="list-style-type: none"> <li>▪ random buffer of bytes (for example, DES and AES keys)</li> <li>▪ ECC key pairs (for supported curve types, see §1.3)</li> </ul>
Key derivation	<ul style="list-style-type: none"> <li>▪ slicing (selecting a substring of bytes from another key)</li> <li>▪ selecting odd or even bytes</li> <li>▪ encrypting/decrypting raw bytes (for example, DES and AES keys) using 128-bit, 192-bit, and 256-bit AES in ECB mode (no padding) and CBC mode (no padding)</li> <li>▪ iterated SHA-1</li> <li>▪ SHA-256 with plain prefix and suffix</li> <li>▪ SHA-384</li> <li>▪ byte reversing</li> <li>▪ NIST 800-108 key derivation with 128-bit AES-CMAC in counter mode</li> <li>▪ KDF2 used in the RSAES-KEM-KWS scheme of the Open Mobile Alliance (OMA) DRM specification</li> <li>▪ deriving raw bytes (for example, DES and AES keys) from an ECC private key</li> <li>▪ CMLA key derivation defined by the <i>CMLA Technical Specification</i></li> <li>▪ 128-bit AES encryption in ECB mode (no padding) with a concatenated key and optional SHA-1 function</li> </ul>
Miscellaneous	<ul style="list-style-type: none"> <li>▪ device binding</li> <li>▪ decryption of PDF files of version 1.6 and 1.7 using 128-bit AES in CBC mode</li> </ul>

## 1.3 Supported ECC Curves

SKB supports the following ECC curve types:

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

- 521-bit prime curve recommended by NIST (same as 521-bit SECG, SECP R1)
- 150-bit to 521-bit prime ECC curves with custom domain parameters

 ElGamal ECC decryption and ElGamal ECC key unwrapping support only 160-bit, 192-bit, 224-bit, and 256-bit prime curves. ECC key generation, ECDSA, and ECDH support all the listed ECC curve types, including ECC curves with custom domain parameters.

## 1.4 Supported Target Platforms

The following table lists operating systems and architectures supported by SKB, and build systems used to build and test the SKB library. Use of other platforms may require additional support from us.


Platform	Architectures	Build systems
Windows Vista/7/8/8.1 (Windows API)	x86, x86_64	Visual Studio 2010, 2012, and 2013
Windows 8/8.1 (Windows Runtime)	x86, x86_64	Visual Studio 2012 and 2013
OS X	x86, x86_64	Xcode 5.0
GNU/Linux	x86, x86_64	GCC 4.4.3
GNU/Linux	ARM	Sourcery CodeBench Lite 2012.03-57
GNU/Linux	MIPS	Sourcery CodeBench Lite 2014.05-27
Android	x86, x86_64, ARM (32-bit and 64-bit), MIPS (32-bit and 64-bit)	Android NDK r10d
iOS	ARM (32-bit and 64-bit)	Xcode 5.0
Windows Phone 8	ARM	Visual Studio 2012
Windows Phone 8.1	ARM	Visual Studio 2013
Google Native Client (NaCl)	X86, x86_64, ARM, PNaCl	Native Client SDK Pepper 39
PlayStation 3	Cell (PPU)	PlayStation 3 Programmer Tool Runtime Library 460.001

Platform	Architectures	Build systems
uClibc/Linux	MIPSEL	Broadcom CrossTools, GCC 4.2

## 1.5 Directory Structure and Contents

The following table describes the directory structure of the SKB package:

Directory or file	Description
<code>/Build/</code>	Contains the files needed to build SKB examples, tests, and Platform-Specific Library for various target operating systems. For information on building these files, see §2.3.
<code>/Build/Targets</code>	Contains several subdirectories for supported target operating systems and architectures. Each subdirectory contains build files required for building SKB examples, tests, and Platform-Specific Library.  For information on building SKB examples, tests, and Platform-Specific Library, see §2.3.
<code>/Documents/</code>	Contains SKB documentation.
<code>/Examples/</code>	Contains SKB examples. For information on building the examples, see §2.3.
<code>/Include/</code>	Contains the following header files: <ul style="list-style-type: none"> <li>▪ <code>skbConfiguration.h</code>: Tells which features are enabled and disabled in the current SKB package.</li> <li>▪ <code>skbExtensions.h</code>: Interface to functions that are considered to be an extension to the main SKB API. Currently, this interface only contains the functions for PDF file decryption (see §6).</li> <li>▪ <code>skbInternalExposed.h</code>: Interface that is internally used by unit tests and speed tests.</li> <li>▪ <code>skbInternalHelpers.h</code>: Interface of Sensitive Operations Library (see §4.1).</li> <li>▪ <code>skbPlatform.h</code>: Interface of Platform-Specific Library (see §4.2).</li> <li>▪ <code>skbSecureKeyBox.h</code>: Entire public interface of SKB (see §7). This is the main API that you use with SKB.</li> </ul>

Directory or file	Description
/Libraries/	<p>Contains the following binaries for different target platforms:</p> <ul style="list-style-type: none"> <li>▪ SKB static library</li> <li>▪ Sensitive Operations Library (see §4.1)</li> <li>▪ Platform-Specific Library (see §4.2)</li> <li>▪ Custom ECC Tool (see §5.1)</li> <li>▪ Diffie-Hellman Tool (see §5.2)</li> <li>▪ Key Export Tool (see §5.3)</li> <li>▪ SQLite library (see §4.2.3)</li> <li>▪ Binary Update Tool (see §5.4)</li> </ul> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p> If you have ordered a tamper resistant SKB library (see §1.1.9), you always have to run the Binary Update Tool on the final protected application once it is built as described in §5.4. Otherwise, the protected application will crash at run time.</p> </div>
/Source/	Contains the source files and configuration files used by SKB examples and tests.
/SpeedTests/	Contains speed tests for measuring the performance of various cryptographic algorithms. For information on building tests, see §2.3.
/Test/	Contains unit tests. You can compile and run them to verify that SKB is running correctly. For information on building tests, see §2.3.
/ThirdParty/	Contains third-party files needed to compile the source files delivered along with SKB.
/Tools/SkbPlatform	<p>Contains source code for Platform-Specific Library. This library is also available in a binary format, in the <code>Libraries</code> directory.</p> <p>For more information on Platform-Specific Library, see §4.2.</p>
/Tools/SkbUtils	Contains source code for several functions and variables used by the SKB tests.

Directory or file	Description
/export.id	<p>Text file containing the identifier of the export key (see §1.1.6) included in this SKB instance, as well as identifiers of all export keys whose exported data this SKB instance is capable of upgrading (see §3.7).</p> <p>The file structure is as follows:</p> <pre data-bbox="544 510 1455 682"> 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» </pre> <p>Legacy key specifies identifiers of export keys whose exported data can be upgraded by this SKB instance. Current key specifies the identifier of the export key that is used by this SKB instance to encrypt exported data.</p> <p>To find out which export key was used to export particular data, you can compare export key identifiers in this file to the export key identifier in the header of exported data as described in §8.1.</p>
/SConstruct	<p>This file is used by SCons to build the source files delivered along with SKB. For information on using SCons for building the files, see §2.3.</p>

## 1.6 Limitations and Known Problems

Please carefully review the following list of limitations and known problems before including SKB into your applications:

- Features utilizing RSA or ECC custom curve algorithms do not work on PlayStation 3.
- Before running SKB examples, speed tests, and unit tests on PlayStation 3, the `make_self` tool included in the PS3 SDK has to be run on the executables.
- When unwrapping raw bytes with RSA using OAEP padding, only the SHA-1 variant of OAEP padding is supported. SHA-256 is not supported.

## 2 Building Applications Protected by SKB

---


This chapter describes the recommended manner to build and deploy an application that is integrated with SKB. Following these steps is important to achieve the maximum security provided by SKB. It also provides instructions for building SKB examples, tests, and Platform-Specific Library (see §4.2) for different target platforms.

### 2.1 Building a 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. Link your application with the following libraries:
  - appropriate `secureKeyBox` library from the `Libraries` directory, depending on the target platform
  - Platform-Specific Library, which provides certain functions that have different implementations for different operating systems (see §4.2)
  - SQLite library if you are using SQLite-based key caching (see §4.2.3)

 Make sure you are not linking or distributing any of the unsafe SKB components listed in §2.2.

2. Build your application and make sure you use the following compiler and linker settings depending on your build system:

Build system	Settings to use
Visual Studio	<ul style="list-style-type: none"> <li>▪ enable references to remove unnecessary code (<code>/OPT:REF</code>)</li> <li>▪ enable COMDAT folding to remove duplicated code (<code>/OPT:ICF</code>)</li> </ul>
GCC	<ul style="list-style-type: none"> <li>▪ if compiling for OS X via the command line, use the <code>-w1, -dead_strip</code> option to remove unnecessary code sections</li> <li>▪ if compiling for Linux, use the <code>-w1, -gc-sections</code> option to remove unnecessary code sections</li> </ul>
Xcode	<ul style="list-style-type: none"> <li>▪ enable <b>Deployment Postprocessing</b> to remove information that can be used to reverse engineer the code</li> <li>▪ enable <b>Strip Linked Product</b> to remove information that can be used to reverse engineer the code</li> </ul>



3. To ensure that symbol information is correctly stripped from the executable, open the executable in a binary editor and search for a string “whitebox”.

The string should not be present in the code. If it is, ensure you have completed the items in step 2.

4. If the SKB package delivered to you has tamper resistance applied (see §1.1.9), run the Binary Update Tool on the final built application as described in §5.4.

## 2.2 Distributing a Protected Application

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 should 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 table shows groups of safe and unsafe SKB components.

Safe components	Unsafe components
<ul style="list-style-type: none"> <li>▪ SKB library (<code>SecureKeyBox</code>)</li> <li>▪ Platform-Specific Library (<code>SkbPlatform</code>)</li> <li>▪ examples</li> <li>▪ <code>SQLite</code> library</li> </ul>	<ul style="list-style-type: none"> <li>▪ Key Export Tool</li> <li>▪ Custom ECC Tool</li> <li>▪ Diffie-Hellman Tool</li> <li>▪ unit tests and speed tests</li> <li>▪ Sensitive Operations Library (<code>InternalHelpers</code>)</li> <li>▪ utilities (<code>SkbUtils</code>)</li> <li>▪ <code>LibTomCrypt</code> and <code>LibTomMath</code> libraries (only required by tests)</li> </ul>

Components in the left-hand column are self-sufficient and can be considered safe. Including these components in your application will not compromise the security provided by SKB. Components in the right-hand column, however, should be considered unsafe and must never be included in an application that is deployed in an open environment. They can only be used on a protected computer that is not accessible to end users.

## 2.3 Building Examples, Tests, and Platform-Specific Library

A number of additional C++ source files are delivered together with SKB. These files include examples, tests, and Platform-Specific Library (see §4.2). Platform-Specific Library is a mandatory component that is necessary for the execution of SKB. Other components are optional.

The following subsections describe the recommended and supported way of building these files for different targets.

### 2.3.1 Building for Windows API

Visual Studio is used to build SKB examples, tests, and Platform-Specific Library. This section describes how to set up the build environment and compile the source files.

#### 2.3.1.1 Prerequisites

To compile the source files, you will need a computer with the Windows operating system that has Visual Studio installed. For information on supported Visual Studio versions, see §1.4.

#### 2.3.1.2 Compiling

To compile the source files, proceed as follows:

1. Open the Visual Studio solution named `SecureKeyBox.sln` in one of the following directories, depending on your needs:
  - `Build/Targets/all-microsoft-win32-vs2010`: Visual Studio 2010 solution for Windows API.
  - `Build/Targets/all-microsoft-win32-vs2012`: Visual Studio 2012 solution for Windows API.
  - `Build/Targets/all-microsoft-win32-vs2013`: Visual Studio 2013 solution for Windows API.

This solution contains the following specific projects:

Project	Description
<code>SkbExamples</code>	Runs SKB examples
<code>SkbSpeedTests</code>	Runs SKB speed tests
<code>SkbTestSuite</code>	Runs SKB unit tests

2. Compile the solution.

### 2.3.2 Building for Windows Runtime and Windows Phone

Visual Studio is used to build SKB examples, tests, and Platform-Specific Library. This section describes how to set up the build environment and compile the source files.

### 2.3.2.1 Prerequisites

To compile the source files, you will need a computer with the Windows operating system that has Visual Studio installed. For information on supported Visual Studio versions, see §1.4.

### 2.3.2.2 Compiling

To compile the source files, proceed as follows:

1. Open the Visual Studio solution named `secureKeyBox.sln` in one of the following directories, depending on your needs:
  - `Build/Targets/all-microsoft-winrt-vs2012`: Visual Studio 2012 solution for Windows Runtime
  - `Build/Targets/all-microsoft-winrt-vs2013`: Visual Studio 2013 solution for Windows Runtime
  - `Build/Targets/arm-windows-phone-vs2012`: Visual Studio 2012 solution for Windows Phone 8.0
  - `Build/Targets/arm-windows-phone-vs2013`: Visual Studio 2013 solution for Windows Phone 8.1

The solution contains the following specific projects:

Project	Description
<code>SkbExamplesApp</code>	Runs SKB examples as a Microsoft design language app.
<code>SkbExamplesUnitTest</code>	Runs SKB examples as a Visual Studio unit test.
<code>SkbSpeedTestsApp</code>	Runs SKB speed tests as a Microsoft design language app.
<code>SkbSpeedTestsUnitTest</code>	Runs SKB speed tests as Visual Studio unit tests.
<code>SkbTestSuiteUnitTest</code>	Runs SKB unit tests as Visual Studio unit tests.

2. Compile the solution.

## 2.3.3 Building for Linux

The SCons build tool is used to build SKB examples, tests, and Platform-Specific Library. This section describes how to set up the build environment and compile the source code.

### 2.3.3.1 Prerequisites

The following prerequisites must be met before building the source files:

1. Download and install SCons 2.3.0.

2. Depending on your target architecture, download and install the necessary build system as listed in §1.4.

### 2.3.3.2 Compiling

To compile the source files, go to the root directory of the SKB package and execute the following command:

```
scons target=«your_target» build_config=«Debug | Release»
```

The following table describes parameters used in this command:

Parameter	Description
target	<p>Specifies the target platform, corresponding to an appropriate subdirectory in the <code>Build/Targets</code> directory:</p> <ul style="list-style-type: none"> <li>▪ <code>arm-unknown-linux</code>: GNU/Linux edition for the ARM architecture</li> <li>▪ <code>x86-unknown-linux</code>: GNU/Linux edition for the x86 architecture</li> <li>▪ <code>x86_64-unknown-linux</code>: GNU/Linux edition for the x86_64 architecture</li> <li>▪ <code>mips-unknown-linux</code>: GNU/Linux edition for the MIPS architecture</li> <li>▪ <code>mipsel-broadcom-linux</code>: GNU/Linux edition for the MIPSel architecture</li> </ul> <p>If the target is not specified, the source files will be built for the default target, which is your build machine.</p>
build_config	<p>Specifies whether the binaries should be compiled in release or debug mode. The following values can be set:</p> <ul style="list-style-type: none"> <li>▪ <code>Debug</code>: Produces binary files in debug mode and places them in the <code>Build/Targets/«<b>your_target</b>»/Debug</code> directory. This is the default value.</li> <li>▪ <code>Release</code>: Produces binary files in release mode and places them in the <code>Build/Targets/«<b>your_target</b>»/Release</code> directory.</li> </ul>

### 2.3.4 Building for Android

Android NDK is used to build SKB examples, tests, and Platform-Specific Library. This section describes how to set up the build environment, compile the source code, and run the compiled examples.

#### 2.3.4.1 Prerequisites

To build the source files for Android, you will need a computer with Android NDK installed. For information on Android NDK requirements, see §1.4.

 Make sure the Android NDK root directory is added to the system's `PATH` variable.

### 2.3.4.2 Compiling

To compile the source files, proceed as follows:

1. Go to one of the following directories, depending on the architecture used:
  - `Build/Targets/arm64-google-android`
  - `Build/Targets/arm-google-android`
  - `Build/Targets/mips64-google-android`
  - `Build/Targets/mips-google-android`
  - `Build/Targets/x86_64-google-android`
  - `Build/Targets/x86-google-android`
2. Execute the following command:

```
ndk-build APP_OPTIM=<<release|debug>>
```

`APP_OPTIM` specifies whether the binaries should be compiled in release or debug mode. The following values can be set:

- `release` (default value)
- `debug`

The compiled binary files will be placed in the `libs` directory.

### 2.3.4.3 Running

Once the files are compiled, you can transfer the files to the Android device via the Android Debug Bridge (ADB) tool, which is included in the Android SDK.

The following is a sample ADB script that copies compiled SKB examples to the `/data/local` directory on the connected Android device (this directory always allows executing files), makes them executable, and runs them:

```
adb shell rm /data/local/SkbExamples
adb push SkbExamples /data/local
adb shell chmod 777 /data/local/SkbExamples
adb shell "cd /data/local/ && ./SkbExamples"
```

## 2.3.5 Building for OS X and iOS

Xcode is used to build SKB examples, tests, and Platform-Specific Library. This section describes how to set up the build environment and compile the source code.

### 2.3.5.1 Prerequisites

To compile the source files, you will need a computer with the OS X system that has Xcode installed. For information on the supported Xcode version, see §1.4.

### 2.3.5.2 Compiling

To compile the source files, proceed as follows:

1. Go to the `Build/Targets/«your_target»` directory.
2. Open the Xcode project.
3. Select the required scheme.
4. Compile the project.

The location, where compiled files are placed, depends on the system:

System	Location of compiled files
OS X	One of the following directories depending on the compilation mode: <ul style="list-style-type: none"> <li>▪ <code>Build/Targets/universal-apple-macosx/build/Debug</code></li> <li>▪ <code>Build/Targets/universal-apple-macosx/build/Release</code></li> </ul>
iOS	One of the following directories depending on the compilation mode: <ul style="list-style-type: none"> <li>▪ <code>Build/Targets/arm-apple-ios/build/Debug-iphoneos</code></li> <li>▪ <code>Build/Targets/arm-apple-ios/build/Release-iphoneos</code></li> </ul>

## 2.3.6 Building for Google Native Client (NaCl)

The SCons build tool is used to build SKB examples, tests, and Platform-Specific Library. This section describes how to set up the build environment and compile the source code.

### 2.3.6.1 Prerequisites

The following prerequisites must be met before building the source files:


1. Download and install SCons 2.3.0.
2. Download and install Native Client SDK.

### 2.3.6.2 Compiling

Compiling for Google Native Client is performed the same way as described in §2.3.3.2, with the `target` parameter set to one of the following, depending on the necessary architecture:

- `nacl/arm`
- `nacl/pnacl`

- `nacl/x86_32`
- `nacl/x86_64`

 If your application is using SKB algorithms that depend on random generation, you must use the `SKB_InitRng` and `SKB_DestroyRng` functions of Platform-Specific Library as described in §4.2.2.

## 2.3.7 Building for PlayStation 3

The SCons build tool is used to build SKB examples, tests, and Platform-Specific Library. This section describes how to set up the build environment and compile the source code.

### 2.3.7.1 Prerequisites

The following prerequisites must be met before building the source files:

1. Download and install SCons 2.3.0.
2. Download and install PlayStation 3 Programmer Tool Runtime Library 460.001.

### 2.3.7.2 Compiling

Compiling for PlayStation 3 is performed the same way as described in §2.3.3.2, with the `target` parameter set to `ppu-playstation3`.

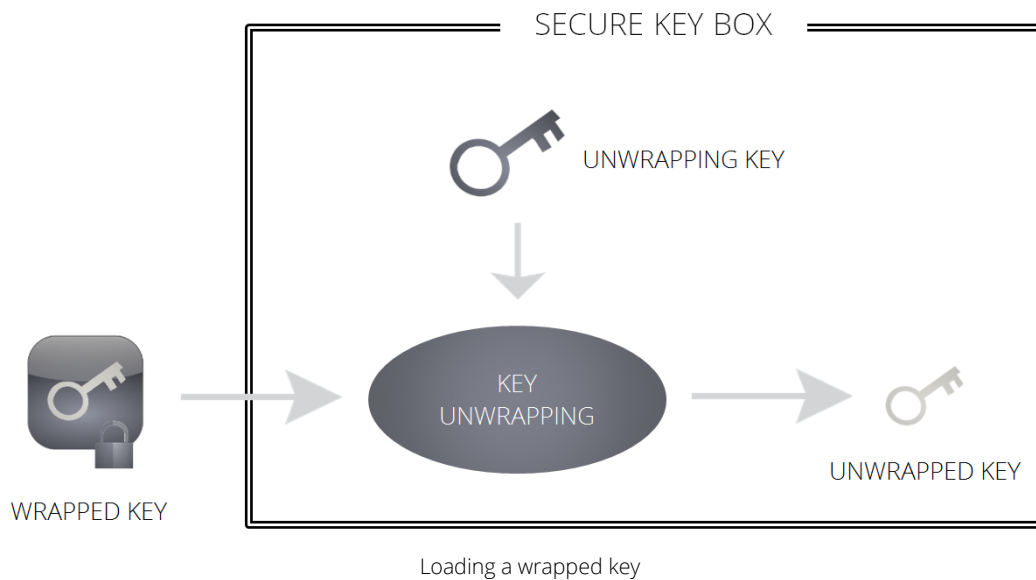
## 3 Cryptographic Operations

---

This chapter provides high-level task-based information about the main cryptographic operations that can be performed with SKB.

### 3.1 Loading Wrapped Keys

A wrapped key is a cryptographic key encrypted with another key. Loading wrapped keys (unwrapping) is a more secure way for importing keys into SKB. SKB loads a wrapped key and internally decrypts it using a pre-loaded unwrapping key.



Unwrapping is not the same operation as regular decryption, because regular decryption provides the output in plain form (see §3.11.2). The unwrapped key is directly transformed into a secure data object and is never exposed in plain form.

To unwrap a key, call the `SKB_Engine_CreateDataFromWrapped` method (see §7.9.5) and provide the necessary parameters, such as the following:

- wrapped key
- type of the wrapped key
- format of the wrapped key
- algorithm for unwrapping the data (for the special case of using the ElGamal ECC unwrapping algorithm, see §3.1.1)
- additional parameters for the unwrapping algorithm
- unwrapping key



### 3.1.1 Unwrapping Keys 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 (see §7.9.5) should contain two points on an ECC curve as described in §8.4.

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 notation. 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 (see §7.9.17) 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 NIST-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 should resemble the following:

```
SKB_SecureData* secret_key; // This will contain the unwrapped AES key
SKB_SecureData* temp_data;
SKB_SecureData* ecc_key = ...; // Previously obtained ECC private 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 encryption process, the
// whole 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);
```

## 3.2 Loading Plain Keys

SKB is designed to always work with keys in protected form. If a key needs to be loaded into SKB, normally you should delivered and load it in encrypted form. However, for very rare cases, SKB does support direct loading of plain keys.

 Loading a plain key is a very insecure operation and should be avoided if possible. There are better alternatives for achieving this as described in §1.1.7. Normally, importing of plain keys is disabled in SKB.

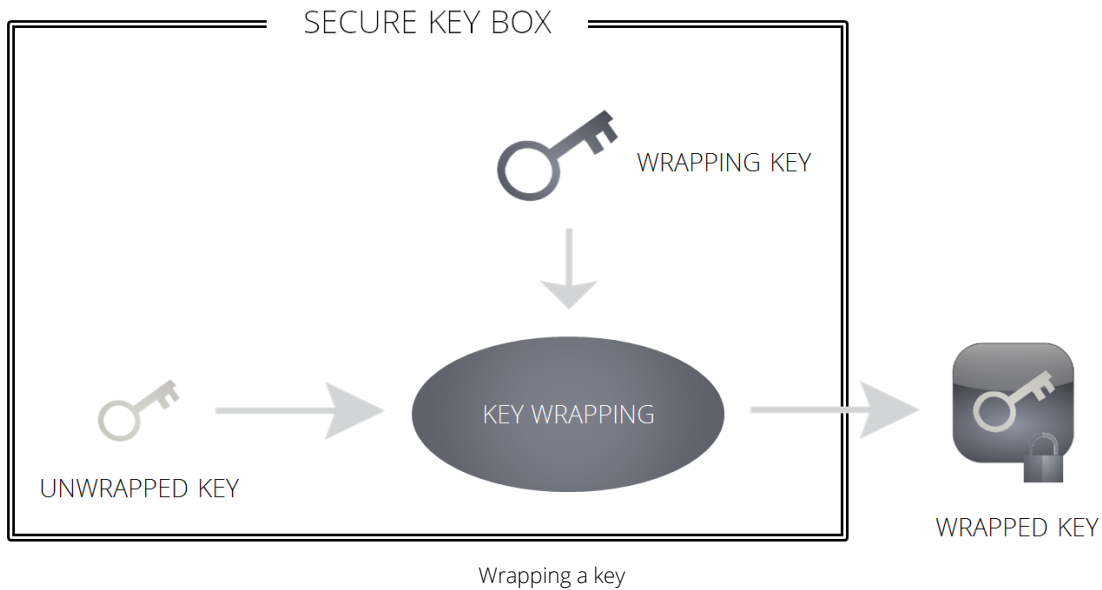
To directly load a plain key as a secure data object, call the `SKB_Engine_CreateDataFromWrapped` method (see §7.9.5) and specify the unwrapping algorithm `SKB_CIPHER_ALGORITHM_NULL`.

In this case, the unwrapping key and parameters do not have to be provided.

This operation will work only if importing of plain keys is enabled in SKB.

## 3.3 Wrapping Keys

A cryptographic key that is stored within a secure data object can be encrypted with another key. This process is called wrapping. The wrapped key can then be passed to any other cryptographic library (not necessarily SKB) where it can be unwrapped and used.



Wrapping is not the same operation as regular encryption, because regular encryption requires plain data as input (see §3.11.1). The wrapping operation takes a secure data object as an input, and therefore the wrapped key is never exposed in plain form.

Wrapping is also not the same operation as secure data exporting, because exporting encrypts keys with a hidden export key that is unique to the particular SKB instance (see §3.5). Wrapping allows using any arbitrary key as a wrapping key, provided that it is supported by the wrapping algorithm.

To wrap a key contained within a secure data object, call the `SKB_SecureData_Wrap` method, specify the secure data object, specify the wrapping algorithm and key, and provide the necessary parameters (see §7.9.16).

## 3.4 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 (see §7.9.7). The input is plain data, but the encryption key and the output of the method are secure data objects.

## 3.5 Exporting Keys

SKB operates on keys in memory. If you need some key to be persistent or if you want it to be available to multiple engines, you can request SKB to provide a protected form of the key, which can then be securely exported and stored.

When the exported key is needed again later, you can request SKB to import it as a new secure data object similar to the one whose data was initially exported. For information on importing exported keys, see §3.6.

When SKB is asked to export a key, it encrypts the actual contents of the referenced secure data object (not its binary representation) with the embedded export key (see §1.1.6). The export key of the exporting instance must match the export key of the importing instance.

To export a key, call the `SKB_SecureData_Export` method, specify the secure data object to be exported, and provide a memory buffer where the exported data should be written (see §7.9.15). For information on the format used to store exported keys, see §8.1.

## 3.6 Importing Keys

If you have a protected buffer of data containing a key previously exported from SKB (or obtained using Key Export Tool or Sensitive Operations Library), you can import it into SKB. The export key of the exporting instance must match the export key of the importing instance.

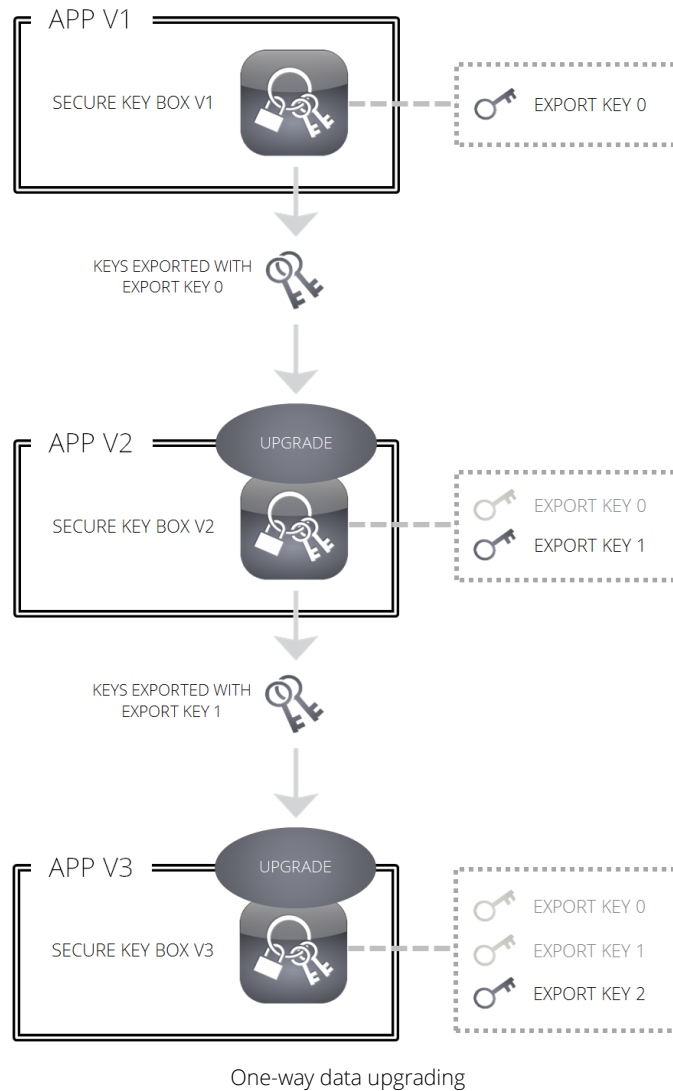
To import a key, call the `SKB_Engine_CreateDataFromExported` method and provide the exported data buffer (see §7.9.6). This method will create a new secure data object containing the imported key.

## 3.7 Upgrading Exported Keys

SKB supports one-way data upgrade deployments. This means that you have an option to request multiple SKB packages, where each package is assigned a version number, starting with version 1. These packages are configured so that an SKB instance with a greater version number can upgrade and import keys exported by all SKB instances 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 read data 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 that application.

The one-way data key upgrade mechanism is implemented by embedding into each SKB package all export keys of its previous versions as shown in the following diagram.




To implement the one-way data upgrading process, proceed as follows:

1. Order multiple versioned SKB packages.
2. Integrate SKB package with version 1 into your application.
3. When creating an updated version of your application, execute the following steps:
  - In the application code, replace the SKB library 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 as described in §7.9.12.

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 Key Export Tool as described in §5.3.

 Key upgrading should be performed only once. To avoid security risks, do not perform key upgrading and importing every time SKB is run. After the upgrade, make sure all keys of previous versions are permanently deleted.

## 3.8 Generating Keys

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 (for Windows, the CryptoAPI is used; for other systems, the `/dev/random` device is used). If necessary, you can create a custom implementation for the random generator function as described in §4.

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 (see §7.9.8).

With the help of this method, you can generate:

- random buffer of bytes (for example, a DES or AES key)
- private ECC keys

Currently, SKB does not support generating private RSA keys.

## 3.9 Deriving a Public Key from a Private Key

As described in §3.8, SKB can generate new random private ECC keys. 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 (see §7.9.18). This method will return a buffer of bytes containing the corresponding public key.

Currently, this operation is supported only for ECC keys, but not for RSA keys.

## 3.10 Deriving Keys

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

### 3.10.1 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 (see §7.9.17).

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 16.

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 §3.1.1.

This operation can only be performed on secure data objects that contain raw bytes (for example, a DES or AES key), not an RSA or ECC private key.

### 3.10.2 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 create a new key as odd or even bytes of another key, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_SELECT_BYTES` algorithm, and specify the necessary parameters (see §7.9.17).

This operation can only be performed on secure data objects that contain raw bytes (for example, a DES or AES key), but not an RSA or ECC private key.

### 3.10.3 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/decrypting key, and the output key have to be secure data objects. SKB supplies a special derivation algorithm for this purpose.

To create a new key as a result of encrypting another key, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_CIPHER` algorithm, and specify the necessary parameters (see §7.9.17).

This operation can only be performed on secure data objects that contain raw bytes (for example, a DES or AES key), but not an RSA or ECC private key.

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

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

- iterated SHA-1 derivation (see §3.10.4.1)
- SHA-256 derivation with plain prefix and suffix (see §3.10.4.2)
- SHA-384 derivation (see §3.10.4.3)

The main difference from 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.

#### 3.10.4.1 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 create a new key as a substring of bytes of a SHA-1 hash value of another key, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_SHA_1` algorithm, and specify the necessary parameters (see §7.9.17).

This operation can only be performed on secure data objects that contain raw bytes (for example, a DES or AES key), but not an RSA or ECC private key.

#### 3.10.4.2 SHA-256 Derivation with Plain Prefix and Suffix

This derivation algorithm creates a 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 create a new key using this SHA-256 derivation, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_SHA_256` algorithm, and specify the plain prefix and suffix buffers (see §7.9.17).

This operation can only be performed with secure data objects that contain raw bytes (for example, a DES or AES key), but not an RSA or ECC private key.



### 3.10.4.3 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 create a new key as a SHA-384 hash value of another key, call the `SKB_SecureData_Derive` method and select the `SKB_DERIVATION_ALGORITHM_SHA_384` algorithm (see §7.9.17).

This operation can only be performed on secure data objects that contain raw bytes (for example, a DES or AES key), but not an RSA or ECC private key.

### 3.10.5 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 little-endian data buffer to big-endian and vice versa.

To create a new secure data object with a reversed order of bytes, call the `SKB_SecureData_Derive` method and select the `SKB_DERIVATION_ALGORITHM_REVERSE_BYTES` algorithm (see §7.9.17).

This operation can only be performed on secure data objects that contain raw bytes (for example, a DES or AES key), but not an RSA or ECC private key.

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

SKB provides a derivation algorithm that is based on the *NIST Special Publication 800-108*, which is available here:

<http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf>

The following special notes apply to the SKB implementation:

- 128-bit AES-CMAC is used as the pseudorandom 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.
- The size of the integer specifying the length of the derived key (parameter “L”) is 32 bits and is encoded using the big-endian notation.

To execute this derivation algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMACE128` algorithm, and specify the necessary parameters (see §7.9.17).

This operation can only be performed on secure data objects that contain raw bytes (for example, a DES or AES key), but not an RSA or ECC private key.

### 3.10.7 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 execute this derivation algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2` algorithm, and specify the necessary parameters (see §7.9.17).

This operation can only be performed on secure data objects that contain raw bytes (for example, a DES or AES key), but not an RSA or ECC private key.

### 3.10.8 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 (for example, a DES or AES key) from an ECC private key.

To execute this derivation algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE` algorithm, and specify the necessary parameters (see §7.9.17).

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

This operation can only be performed on secure data objects that contain an ECC private key.

### 3.10.9 Deriving a Key Using the CMLA Key Derivation Function

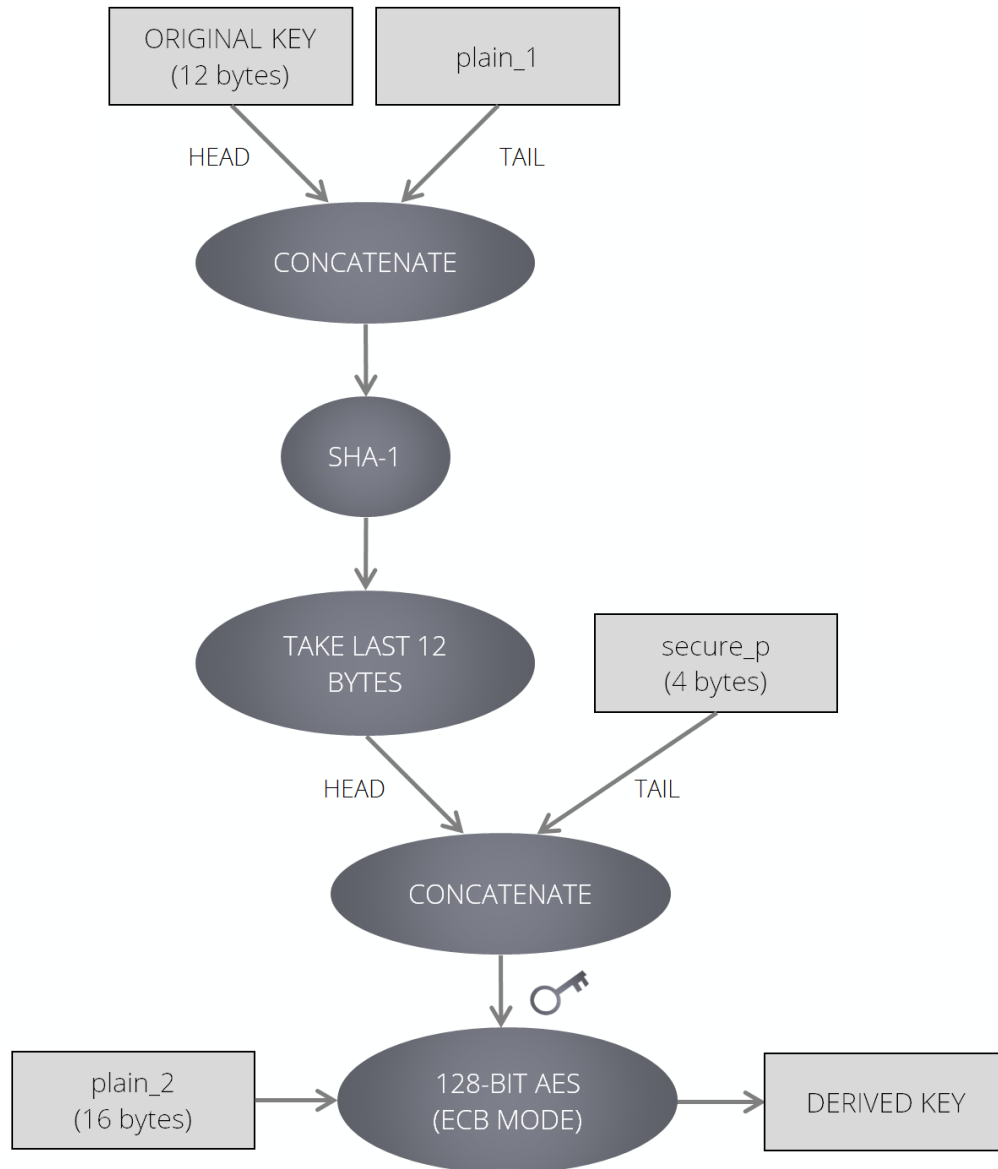
SKB provides a derivation algorithm that is based on the key derivation function specified in the *CMLA Technical Specification*.

To execute this derivation algorithm, call the `SKB_SecureData_Derive` method, select the `SKB_DERIVATION_ALGORITHM_CMLA_KDF` algorithm, and specify the necessary parameters (see §7.9.17).

This operation can only be performed on secure data objects that contain raw bytes (for example, a DES or AES key), but not an RSA or ECC private key.

### 3.10.10 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:

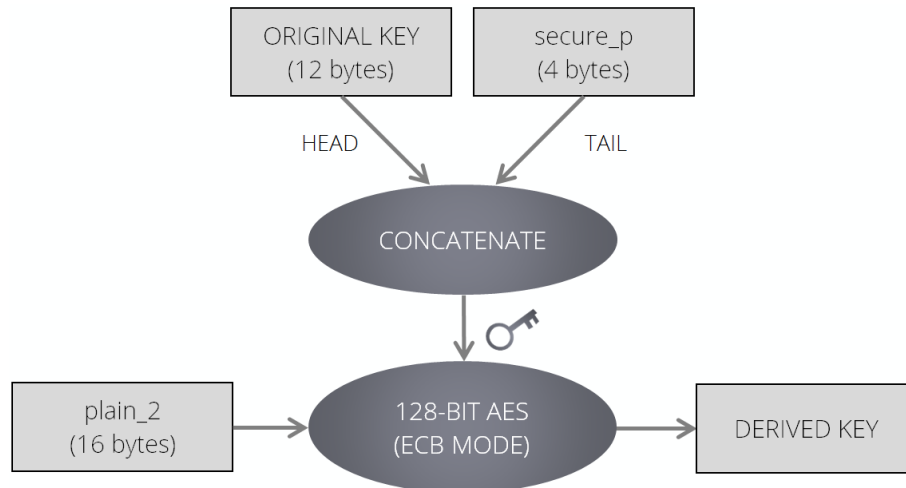


Key derivation based on 128-bit AES encryption with a concatenated key and SHA-1

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 provided in the output.

Additionally, this derivation algorithm supports a simplified mode of operation when plain\_1 is not provided (is `NULL`). Then the algorithm is executed as follows:



Key derivation based on 128-bit AES encryption with a concatenated key without SHA-1

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

To execute this derivation algorithm, call the `SKB_SecureData_Derive` method (see §7.9.17), select the `SKB_DERIVATION_ALGORITHM_SHA_AES` algorithm, and supply `SKB_ShaAesDerivationParameters` as the parameters structure (see §7.10.15).

This operation can only be performed on secure data objects that contain raw bytes and are 12 bytes long.

## 3.11 Encrypting and Decrypting Data

This section describes operations related to encrypting and decrypting data.

### 3.11.1 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, and AES are the only supported encryption ciphers. The main reason for this is that encryption for asymmetric key ciphers (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 encryption algorithm and key, specify the direction `SKB_CIPHER_DIRECTION_ENCRYPT`, and provide the necessary parameters as described in §7.9.10.
2. Encrypt the input buffer by calling the `SKB_Cipher_ProcessBuffer` method as described in §7.9.23.

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 as described in §7.9.24.

### 3.11.2 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 decryption algorithm and key, specify the direction `SKB_CIPHER_DIRECTION_DECRYPT`, and provide the necessary parameters as described in §7.9.10.
2. Decrypt the input buffer by calling the `SKB_Cipher_ProcessBuffer` method as described in §7.9.23.

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 as described in §7.9.24.

### 3.11.3 Using the High-Speed AES

SKB provides an alternative high-speed implementation of AES, which is intended for encrypting and decrypting high-volume data, such as a video stream. High-speed AES performance is very close to the performance of unprotected AES.

To use high-speed AES, specify the `SKB_CIPHER_FLAG_HIGH_SPEED` flag when creating the `SKB_Cipher` object as described in §7.9.10.

## 3.12 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 the necessary parameters as described in §7.9.9.
2. Supply a buffer of plain or secure data as an input to the transform object by calling the `SKB_Transform_AddBytes` method (§7.9.20) and `SKB_Transform_AddSecureData` method (§7.9.21).

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

3. To calculate the digest, call the `SKB_Transform_GetOutput` function as described in §7.9.22.

4. Release the transform object when it is no longer needed by calling the `SKB_Transform_Release` method as described in §7.9.19.

## 3.13 Creating 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. Obtain a secure data object containing the signing key.
2. Create a transform object by calling the `SKB_Engine_CreateTransform` method, select the `SKB_TRANSFORM_TYPE_SIGN` type, and specify the necessary parameters as described in §7.9.9.
3. Supply a buffer of plain or secure data as an input to the transform object by calling the `SKB_Transform_AddBytes` method (§7.9.20) and `SKB_Transform_AddSecureData` method (§7.9.21).

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` and `SKB_SIGNATURE_ALGORITHM_ECDSA`). These algorithms assume that the input is already a message digest calculated using an arbitrary hash function. Therefore, these algorithms 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. Since these signing algorithms operate only on plain data, they are significantly faster than other algorithms that employ a hash function.

4. To calculate the signature, call the `SKB_Transform_GetOutput` function as described in §7.9.22.
5. Release the transform object when it is no longer needed by calling the `SKB_Transform_Release` method as described in §7.9.19.

## 3.14 Verifying a Signature

Verifying a signature involves executing the verification algorithm on a signature buffer using a particular verification key. The output is 1 if the signature is verified and 0 if it is not.

To verify a signature, proceed as follows:

1. Obtain a secure data object containing the verification key.
2. Create a transform object by calling the `SKB_Engine_CreateTransform` method, select the `SKB_TRANSFORM_TYPE_VERIFY` type, and specify the necessary parameters including the verification key, as described in §7.9.9.
3. Supply a buffer of plain or secure data as an input to the transform object by calling the `SKB_Transform_AddBytes` method (§7.9.20) and `SKB_Transform_AddSecureData` method (§7.9.21).

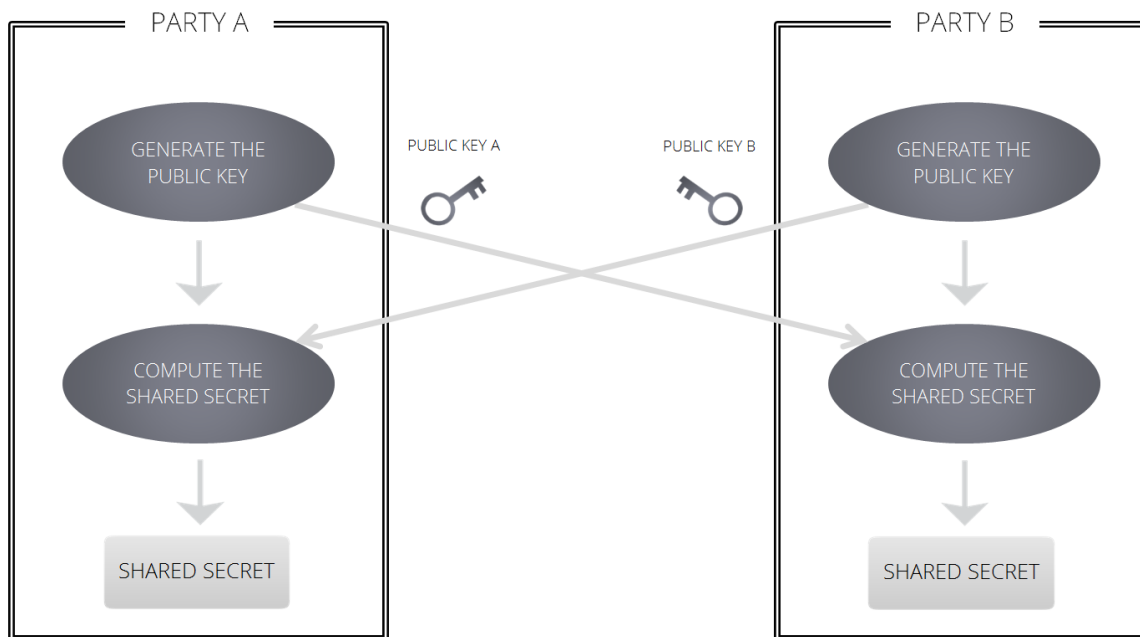
You can call these methods more than once to pass a large buffer of input data consisting of several smaller data chunks.

4. To verify the signature against the supplied data buffer, call the `SKB_Transform_GetOutput` function as described in §7.9.22.
5. Release the transform object when it is no longer needed by calling the `SKB_Transform_Release` method as described in §7.9.19.

## 3.15 Executing a Key Agreement Algorithm

The key agreement algorithm involves two parties that want to obtain a shared secret (usually a cryptographic key) that is 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:



Key agreement algorithm

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 as described in §7.9.11.
2. Generate a public key by calling the `SKB_KeyAgreement_GetPublicKey` method as described in §7.9.25.
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 as described in §7.9.26.
5. Release the key agreement object when it is no longer needed by calling the `SKB_KeyAgreement_Release` method as described in §7.9.27.

## 3.16 Binding Keys to a Specific Device

Normally, keys exported by SKB can be imported by any other SKB instance that has the same export key (see §1.1.6), regardless of the device it is run on. In some cases, you may want to bind exported keys to a specific device, so that they cannot be imported on any other device.

SKB provides device binding via the method `SKB_Engine_SetDeviceId` (see §7.9.3), which can be called after initializing the engine. 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 combined with SKB export key to create a unique format for exported keys. The SKB instance that imports keys must have the same export key and same device ID set as the instance that exports the keys.

When the device ID is no longer needed, you can restore the default export format that depends only on the export key.

## 3.17 Decrypting Encrypted PDF Documents

SKB provides several functions for safely decrypting contents of encrypted PDF files without revealing the user password and the derived encryption key. A typical PDF decryption process involves the following steps:

1. Obtain a user password.
2. Authenticate the user password to verify that the password is valid.
3. Derive the encryption key from the user password, which is then used in the actual decryption process.
4. Decrypt parts of encrypted PDF objects with the derived encryption key.

For detailed instructions on how to perform PDF decryption using SKB, see §6.



## 4 Supporting Libraries

---

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

The following supporting libraries are available:

Library	Description
Sensitive Operations Library	Contains functions for importing plain keys into SKB. For information on this library, see §4.1.
Platform-Specific Library	Contains functions that may be implemented differently on the same architecture. For information on this library, see §4.2.

### 4.1 Sensitive Operations Library

This section describes Sensitive Operations Library, internally called `skbInternalHelpers`.

#### 4.1.1 Overview

Sensitive Operations Library is used to perform the following operations:

- load plain keys as secure data objects
- export secure data objects as plain keys

Since importing and exporting plain keys are very insecure operations, this library is separated from the main API and there is no dependency from one to another. For example, you may want to use Sensitive Operations Library on a secure server that operates with plain keys, but you will definitely want to exclude this library from a client application that is exposed to attacks (see 1.1.7).

If importing of plain data is disabled in SKB, the `SKB_Engine_CreateDataFromWrapped` method will not allow loading plain keys (see §7.9.5). This however will not affect how Sensitive Operations Library works.

Note: Sensitive Operations Library is required to run unit tests and Key Export Tool.

#### 4.1.2 Library Functions

Sensitive Operations Library has its own interface defined in the `Include/SkbInternalHelpers.h` file. This section describes the functions declared in this interface.

#### 4.1.2.1 SKB\_CreateRawBytesFromPlain

This function creates an `SKB_SecureData` object from a plain data buffer.

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 table explains the parameters:

Parameter	Description
<code>engine</code>	Pointer to the pre-initialized engine.
<code>plain</code>	Pointer to the data buffer containing the plain key.
<code>plain_size</code>	Size of the buffer in bytes.
<code>data</code>	Address of a pointer to the <code>SKB_SecureData</code> that will contain the loaded key after this function is executed.

#### 4.1.2.2 SKB\_CreatePlainFromRawBytes

This function returns a plain data buffer from an `SKB_SecureData` object.

The function is declared as follows:

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

The following table explains the parameters:

Parameter	Description
<code>data</code>	Pointer to the <code>SKB_SecureData</code> from which the plain data buffer must be created.

Parameter	Description
<code>plain</code>	<p>This parameter is either <code>NULL</code> or a pointer to the memory buffer where the plain key is to be written.</p> <p>If this parameter is <code>NULL</code>, the method simply returns, in <code>plain_size</code>, a number of bytes that would be sufficient to hold the plain key, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and the buffer size is large enough to hold the plain key, the method stores the plain key there, sets <code>plain_size</code> to the exact number of bytes stored, and returns <code>SKB_SUCCESS</code>. If the buffer is not large enough, then the method sets <code>plain_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p>
<code>plain_size</code>	Pointer to the size of the plain data buffer in bytes.


#### 4.1.2.3 SKB\_CreateEccPrivateFromPlain

This function creates an `SKB_SecureData` object from a plain ECC private key.

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 table explains the parameters:

Parameter	Description
<code>engine</code>	Pointer to the pre-initialized engine.
<code>plain</code>	<p>Pointer to the data buffer containing the ECC private key. For information on the ECC key format, see §7.11.7.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">  The data must be provided in big-endian encoding. </div>
<code>plain_size</code>	Size of the buffer in bytes.
<code>data</code>	Address of a pointer to the <code>SKB_SecureData</code> that will contain the loaded key after this function is executed.


#### 4.1.2.4 SKB\_CreatePlainFromEccPrivate

This function derives a plain ECC private key from an `SKB_SecureData` object.

The function is declared as follows:

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

The following table explains the parameters:

Parameter	Description
data	Pointer to the <code>SKB_SecureData</code> from which the plain ECC private key must be created.
plain	<p>This parameter is either <code>NULL</code> or a pointer to the memory buffer where the plain key is to be written.</p> <p>If this parameter is <code>NULL</code>, the method simply returns, in <code>plain_size</code>, a number of bytes that would be sufficient to hold the plain key, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and the buffer size is large enough to hold the plain key, the method stores the plain key there, sets <code>plain_size</code> to the exact number of bytes stored, and returns <code>SKB_SUCCESS</code>. If the buffer is not large enough, then the method sets <code>plain_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">  The data will be provided in big-endian encoding.         </div>
plain_size	Pointer to the size of the plain data buffer in bytes.

#### 4.1.2.5 SKB\_CreateRsaPrivateFromPlainPKCS8

This function creates an `SKB_SecureData` object from a plain RSA private key stored according to the PKCS#8 standard.

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 table explains the parameters:

Parameter	Description
engine	Pointer to the pre-initialized engine.

Parameter	Description
plain	Pointer to the data buffer containing the RSA private key stored according to the PKCS#8 standard.
plain_size	Size of the buffer in bytes.
data	Address of a pointer to the <code>SKB_SecureData</code> that will contain the loaded key after this function is executed.

#### 4.1.2.6 SKB\_CreateRsaPrivateFromPlain

This function creates an `SKB_SecureData` object from a plain RSA private key defined as a set of key components.

 The input parameters must be provided in big-endian encoding.

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 table explains the parameters:

Parameter	Description
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 <code>SKB_SecureData</code> that will contain the loaded key after this function is executed.

## 4.1.2.7 SKB\_CreatePlainFromRsaPrivate

This function derives plain RSA private key components from an `SKB_SecureData` object.

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

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 table explains the parameters:

Parameter	Description
<code>data</code>	Pointer to the <code>SKB_SecureData</code> from which the plain RSA private key components must be derived.
<code>plain_p</code>	This parameter is either <code>NULL</code> or a pointer to the memory buffer where the prime number “p” is to be written.  If this parameter is <code>NULL</code> , the method simply returns, in <code>key_size</code> , a number of bytes that would be sufficient to hold the prime number “p”, and returns <code>SKB_SUCCESS</code> .  If this parameter points to a memory buffer (it is not <code>NULL</code> ), and the buffer size is large enough to hold the prime number “p”, the method stores the value there, sets <code>key_size</code> to the exact number of bytes stored, and returns <code>SKB_SUCCESS</code> . If the buffer is not large enough, then the method sets <code>key_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code> .
<code>plain_q</code>	Pointer to the prime number “q”.  This parameter works similar to <code>plain_p</code> and will have the same size.
<code>plain_d</code>	Pointer to the decryption exponent “d”.  This parameter works similar to <code>plain_p</code> and will have the same size.
<code>plain_n</code>	Pointer to the modulus “n”.  This parameter works similar to <code>plain_p</code> and will have the same size.
<code>key_size</code>	Pointer to the size of the prime number “p”, prime number “q”, decryption exponent “d”, and modulus “n”.

## 4.2 Platform-Specific Library

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

### 4.2.1 Overview

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 Platform-Specific Library. This library is available as source code in the `Tools/SkbPlatform` directory, and as a precompiled binary in the `Libraries` directory.

Platform-Specific Library has its own interface defined in the `Include/SkbPlatform.h` file. You can use the provided implementation as is or create your own custom implementation of library functions to suit your specific needs, for example to run SKB on an operating system that is not directly supported. All the necessary implementation information is provided in the comments of the `SkbPlatform.h` file.

For information on compiling Platform-Specific Library, see §2.3.

### 4.2.2 Library Functions

The functions in Platform-Specific Library can be grouped according to their logical purpose:

Purpose	Description
Key caching	<p>Key caching speeds up operations with RSA keys. For details on this component, see §4.2.3.</p> <p>The following functions are related to key caching:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_KeyCache_Create</code></li> <li>▪ <code>SKB_KeyCache_Destroy</code></li> <li>▪ <code>SKB_KeyCache_GetInfo</code></li> <li>▪ <code>SKB_KeyCache_SetGUID</code></li> <li>▪ <code>SKB_KeyCache_GetGUID</code></li> <li>▪ <code>SKB_KeyCache_ClearData</code></li> <li>▪ <code>SKB_KeyCache_SetData</code></li> <li>▪ <code>SKB_KeyCache_GetData</code></li> </ul>

Purpose	Description
Random generation	<p>The function <code>SKB_GetRandomBytes</code> is used to generate a buffer of random bytes of a specific size.</p> <p>There are two additional random generator related functions that are intended for the Google Native Client (NaCl) target only:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_InitRng</code></li> <li>▪ <code>SKB_DestroyRng</code></li> </ul> <p>If you are building an application for the Google Native Client target, and this application uses SKB algorithms that depend on random generation (including but not limited to key generation, key exporting, ECDSA, and ECDH), you must call the <code>SKB_InitRng</code> function before the first instance of random generation. Otherwise, SKB will return the <code>SKB_ERROR_INVALID_STATE</code> (-80008) error code. The <code>SKB_DestroyRng</code> function must be called after the last instance of random generation. We recommend calling the <code>SKB_InitRng</code> function before the first invocation of the <code>SKB_Engine_GetInstance</code> function (see §7.9.1). In a similar manner, we recommend calling the <code>SKB_DestroyRng</code> function after the last invocation of the <code>SKB_Engine_Release</code> function (see §7.9.2).</p> <p>For details on these functions, see the comments in the <code>skbPlatform.h</code> file.</p>
Mutex handling	<p>The purpose of mutexes is to avoid the simultaneous use of common resources. SKB uses mutex functions to ensure the correct use of threads.</p> <p>The following functions are related to mutex handling:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_Mutex_Create</code></li> <li>▪ <code>SKB_Mutex_LockAutoCreate</code></li> <li>▪ <code>SKB_Mutex_Lock</code></li> <li>▪ <code>SKB_Mutex_Unlock</code></li> <li>▪ <code>SKB_Mutex_Destroy</code></li> </ul>
Logging	<p>SKB uses the function <code>SKB_LogMessage</code> to write log messages to a particular output. Logging is only used in the debug mode.</p>
Debugging	<p>SKB calls the function <code>SKB_StopInDebugger</code> when an exception occurs at run time. It is only used in the debug mode.</p>

For details on each function, see the comments in the `skbPlatform.h` file.



### 4.2.3 Key Caching

In SKB, RSA operations might take a significant amount of time. To address this problem, SKB provides functionality called RSA key caching. It speeds up operations with RSA keys by caching them after their initialization. RSA-related algorithms are the only ones that use key caching.

Before compiling Platform-Specific Library, you can set a particular key caching mode as described in the following table:

Mode	Description
SQLite	<p>The SQLite implementation of key caching is used. If the SKB library delivered to you includes any RSA features, this is the default mode for all targets, except Google Native Client (NaCl) and PlayStation 3.</p> <p>In this mode, the key caching data is stored in an SQLite database named <code>skb.db</code> in protected form. The implementation of this key caching mode is defined in the <code>SkbProtectedKeyCacheSQLite.cpp</code> file, which is located in the <code>Tools/SkbPlatform/KeyCacheImpl</code> directory.</p> <p>You can either use this implementation in your application as is or treat it as an example implementation for key caching. If you use this implementation without modification, make sure that different applications are not accessing the same <code>skb.db</code> file. The path to this file varies for different operating systems. You can adjust the path by modifying the <code>skb&lt;&lt;target&gt;&gt;KeyCacheFilePath.cpp</code> file, located in the <code>Tools/SkbPlatform/&lt;target&gt;</code> directory. For instance, if you are protecting an Android application, the file name is <code>SkbAndroidKeyCacheFilePath.cpp</code>, and it is located in the <code>Tools/SkbPlatform/Android</code> directory.</p> <p>To use SQLite-based key caching, you will need an SQLite static library (version 3.7.14 or later) to be included in your project. You can obtain the library in the <code>Libraries</code> directory.</p>
In-memory	<p>An internal in-memory map-like data structure is used for key caching. If the SKB library delivered to you includes any RSA features, this is the default mode for the Google Native Client and PlayStation 3 targets.</p> <p>The implementation of this key caching mode is defined in the <code>SkbProtectedKeyCacheInMemory.cpp</code> file, which is located in the <code>Tools/SkbPlatform/KeyCacheImpl</code> directory.</p> <p>By default, only the last 10 keys are cached. If you want to change the number of cached keys, define the <code>SKB_KEY_CACHE_MAX_IN_MEMORY_ITEMS</code> preprocessor definition as the required number, for example as follows:</p> <pre>#define SKB_KEY_CACHE_MAX_IN_MEMORY_ITEMS 20</pre>

Mode	Description
Custom	SKB uses your own custom implementation of key caching. For information on creating your own implementation, see §4.2.3.5.
None	Key caching is not used at all. This is the default mode if there are no RSA features included in the SKB library delivered to you.

#### 4.2.3.1 Configuring Key Caching Using Visual Studio

In Visual Studio, the key caching mode to be used is set using a specific preprocessor definition in the `skbPlatform` project properties. The following preprocessor definitions can be set, each corresponding to a particular key caching mode:

- `SKB_USE_KEY_CACHE_SQLITE`
- `SKB_USE_KEY_CACHE_IN_MEMORY`
- `SKB_USE_KEY_CACHE_CUSTOM`
- `SKB_USE_KEY_CACHE_NONE`

#### 4.2.3.2 Configuring Key Caching Using SCons

For SCons, the key caching mode is set by passing the input parameter `skb_key_cache` to the SCons script. The input parameter can have the following values, each corresponding to a particular key caching mode:

- `sqlite`
- `inmem`
- `custom`
- `none`

For more information on running the SCons build script, see §2.3.3.2.

#### 4.2.3.3 Configuring Key Caching Using Android NDK

For Android NDK, the key caching mode is set by passing the input parameter `SKB_KEY_CACHE` to the `ndk-build` command. The input parameter can have the following values, each corresponding to a particular key caching mode:

- `SKB_USE_KEY_CACHE_SQLITE`
- `SKB_USE_KEY_CACHE_IN_MEMORY`
- `SKB_USE_KEY_CACHE_CUSTOM`
- `SKB_USE_KEY_CACHE_NONE`

#### 4.2.3.4 Configuring Key Caching Using Xcode

In Xcode, the key caching mode to be used is set using a specific preprocessor macro:

- For OS X, the macro is set in the `secureKeyBox` project properties.
- For iOS, the macro is set in the `skbPlatform` target properties.

The following preprocessor macros can be set, each corresponding to a particular key caching mode:

- `SKB_USE_KEY_CACHE_SQLITE`
- `SKB_USE_KEY_CACHE_IN_MEMORY`
- `SKB_USE_KEY_CACHE_CUSTOM`
- `SKB_USE_KEY_CACHE_NONE`

#### 4.2.3.5 Creating a Custom Key Caching Implementation

In some cases, you might want to create your own implementation of key caching, for example to avoid including the additional SQLite code in your application. In such cases, the key cache API must be reimplemented according to the API description in the `skbPlatform.h` file.

## 5 Utilities

---

This chapter describes the command-line utilities provided together with the SKB package.

The following utilities are available:

Utility	Description	Section
Custom ECC Tool	Generates protected forms of ECC domain parameters, which are used for defining custom curves.	§5.1
Diffie-Hellman Tool	Generates protected forms of parameters for the DH key agreement algorithm.	§5.2
Key Export Tool	Creates a white-box protected exported form of an <code>SKB_SecureData</code> object from plain input data, and upgrades previously exported data.	§5.3
Binary Update Tool	Adjusts the final application executable if the tamper-resistant SKB library is used.	§5.4

### 5.1 Custom ECC Tool

Custom ECC Tool generates protected forms of ECC domain parameters, which are used for defining custom curves. The generated protected forms of custom ECC domain parameters must then be specified in the `SKB_EccDomainParameters` structure (see §7.10.18) when you use custom ECC curves.

#### 5.1.1 Custom ECC Tool Overview

Custom ECC Tool can generate protected forms for the following ECC domain parameters:

- constant “a” in the curve equation
- prime “p” of the elliptic curve
- order “n” of the base point
- X coordinate of the base point
- Y coordinate of the base point
- fixed random value to be passed to the ECDSA and ECDH algorithms (see §7.10.22)

To generate a protected form of any of these parameters, simply run Custom ECC Tool at command prompt and specify the type of the parameter and the input value. The utility will write the protected binary form of the input parameter to the standard output.

Custom ECC Tool generates only one parameter at a time. To generate multiple parameters, run the utility multiple times, specifying a different parameter each time.

### 5.1.2 Parameter Size and Value Restrictions

The size of all input parameters must be between 150 and 521 bits, and none of the parameters should have an equal or greater value than the order of the base point.

Note that SKB contains two run-time instances of ECC. One instance corresponds to 150 to 256 bit curves, and the other corresponds to 257 to 521 bit curves. The 150 to 256 bit ECC instance is faster than the 257 to 521 bit ECC instance. If the size of the order of the base point is greater than 256 bits, at run time, SKB will use the 257 to 521 bit ECC instance, which is slower.

### 5.1.3 Running Custom ECC Tool

Custom ECC Tool is located in the `Libraries` directory along with the precompiled SKB library. You can run the utility by simply executing it at the command line and passing several parameters to it.

The following is the pattern to be used to run Custom ECC Tool:

```
CustomEccTool <parameter_type> <parameter_value>
```

The following table explains the input parameters:

Parameter	Description
<code>&lt;parameter_type&gt;</code>	Type of the ECC domain parameter for which the protected form must be generated. The following types are available: <ul style="list-style-type: none"> <li>▪ <code>-a</code>: constant "a" in the curve equation</li> <li>▪ <code>-p</code>: prime "p" of the elliptic curve</li> <li>▪ <code>-n</code>: order "n" of the base point</li> <li>▪ <code>-x</code>: X coordinate of the base point</li> <li>▪ <code>-y</code>: Y coordinate of the base point</li> <li>▪ <code>-r</code>: fixed random value to be passed to the ECDSA and ECDH algorithms</li> </ul>
<code>&lt;parameter_value&gt;</code>	Plain parameter value, which must be specified as an unsigned integer. If you are passing the parameter "a" and it is a negative number, it must be provided as "p-a" where "p" is the prime of the elliptic curve.

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

The `skbEccCustomDomainParameters.h` file, located in the `Examples` directory, contains examples of protected ECC domain parameters for different curve types.

## 5.2 Diffie-Hellman Tool

The SKB implementation of DH key agreement algorithm operates on encrypted parameters. Diffie-Hellman Tool is used to generate the protected forms of these parameters, which must then be provided to the `SKB_PrimeDhParameters` structure (see §7.10.23) when you use DH key agreement.

### 5.2.1 Diffie-Hellman Tool Overview

The DH algorithm requires the following basic input parameters:

- prime P
- generator G
- random value X

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.

To generate a protected form of any of these parameters, simply run Diffie-Hellman Tool at command prompt and specify the type of the parameter and the input value. The utility will write the protected form of the input parameter either to the standard output or to a binary file, depending on your choice.

Diffie-Hellman Tool generates only one output value at a time. To generate multiple values, run the utility multiple times, specifying a different parameter each time.

### 5.2.2 Running Diffie-Hellman Tool

Diffie-Hellman Tool is located in the `Libraries` directory along with the precompiled SKB library. You can run the utility by simply executing it at the command line and passing several parameters to it.

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

```
PrimeDHTool <<arguments>>
```

The following table explains the input arguments:

Argument	Description
<code>-s &lt;value&gt;</code>	Maximum bit-length of P. This argument is mandatory.

Argument	Description
<code>-p «value»</code>	<p>Prime P as an unsigned integer.</p> <p>If this parameter is specified, the generator G (argument <code>-g</code>) must also be provided. The output will be a single protected buffer containing both the P and G parameters.</p> <p>The greatest common divisor of P and G must be 1.</p>
<code>-g «value»</code>	<p>Generator G as an unsigned integer.</p> <p>If this parameter is specified, the prime P (argument <code>-p</code>) must also be provided. The output will be a single protected buffer containing both the P and G parameters.</p> <p>The value of this parameter must be less than P, and the greatest common divisor of P and G must be 1.</p>
<code>-x «value»</code>	<p>Random value X as an unsigned integer.</p> <p>The output of this parameter can be used to provide a fixed random value to the DH algorithm as described in §7.10.23.</p> <p>If this parameter is provided, the parameters <code>-p</code> and <code>-g</code> must not be supplied.</p> <p>The output will be a buffer containing the protected random value.</p>
<code>--output_format «value»</code>	<p>Specifies the format of the output. Possible values are the following:</p> <ul style="list-style-type: none"> <li>▪ <code>binary</code>: Output is a binary file containing a buffer of bytes.</li> <li>▪ <code>source</code>: Output is a definition of a C array, which you can then copy directly into your source code.</li> </ul> <p>Optionally, you can use the command-line argument <code>-i «value»</code> to specify how many elements should be displayed on each line. The default value is 8.</p> <ul style="list-style-type: none"> <li>▪ <code>hex</code>: Output is a string containing the exported data in hexadecimal format.</li> </ul>
<code>--output «value»</code>	<p>File name of the output file generated by Diffie-Hellman Tool.</p> <p>If this argument is not provided, Diffie-Hellman Tool writes the result to the standard output.</p>
<code>--help</code>	<p>Displays the help.</p>

## 5.3 Key Export Tool

Key Export Tool is used for the following purposes:

- creating a protected exported form of an `SKB_SecureData` object from plain input data  
The input data can be raw bytes (for example, a DES or AES key), an RSA private key, or an ECC private key.
- upgrading previously exported data to the current version in the one-way data upgrade scheme (see §3.7)

 Key Export Tool must always be used in a secure environment (see §1.1.7).

### 5.3.1 Key Export Tool Overview

Key Export Tool performs the following actions:

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

The output format can be either binary data or C code, 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 (see §7.9.6). This operation can be performed only if the importing SKB instance has the same export key as the one that exported the data.

### 5.3.2 Running Key Export Tool

Key Export Tool is located in the `Libraries` directory along with the precompiled SKB library.

To run the utility, simply 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> --output <output_file> [--device-id <file_with_device_id>]
```

The following table explains the input parameters:



Parameter	Description
<code>--input-format</code>	<p>Specifies the format of the input file. Possible values are the following:</p> <ul style="list-style-type: none"> <li>▪ <code>bytes</code>: raw bytes in plain (for example, a DES or AES key) If you are importing a key for the Triple DES algorithm, make sure the input corresponds to the format described in §8.3.</li> <li>▪ <code>rsa</code>: plain RSA private key in the PKCS#8 format</li> <li>▪ <code>ecc</code>: plain ECC private key in the format that corresponds to the format described in §8.6</li> <li>▪ <code>upgrade</code>: previously exported data that needs to be upgraded to the current version (see §3.7)</li> </ul>
<code>--output-format</code>	<p>Specifies the format of the output file. Possible values are the following:</p> <ul style="list-style-type: none"> <li>▪ <code>binary</code>: Output is a binary file containing a buffer of bytes.</li> <li>▪ <code>source</code>: Output is a definition of a C array, which you can then copy directly into your source code.</li> </ul>
<code>--input</code>	File name of the input file.
<code>--output</code>	File name of the output file generated by Key Export Tool.
<code>--device-id</code>	<p>Optional parameter that allows you to set the device ID. Device ID is combined with the export key to create a unique format for exported keys as described in §3.16. If this parameter is not specified, the export format depends only on the export key.</p> <p>If you want to set the device ID, this parameter must contain a path to a binary file that contains the device ID.</p> <p><b>Note:</b> This parameter is available only if the SKB package you requested has the device binding feature enabled.</p>

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

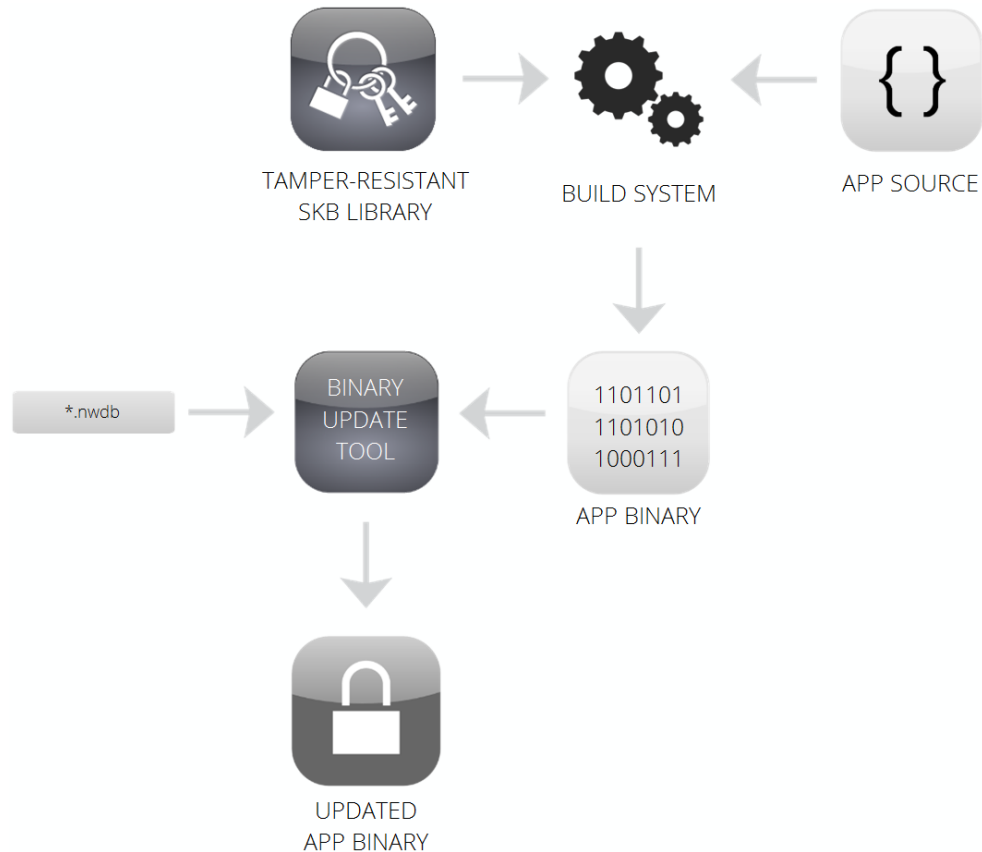
## 5.4 Binary Update Tool

If the SKB library that you link with your application has tamper resistance applied (see §1.1.9), 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 a command-line utility called Binary Update Tool, which is included in the SKB package.

**⚠** If the binary update process is not executed on a tamper resistant SKB library, the built application will crash at run time.

### 5.4.1 Binary Update Tool Overview

The binary code must be adjusted using the Binary Update Tool after every build of the final application. As an input, the Binary Update Tool requires the binary executable of the protected application and the **«target»**.nwdb file that is delivered together with SKB.



Building an application that uses a tamper-resistant SKB library

**⚠** OS X and iOS 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 from the command line as described here:

<https://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/codesign.1.html>

### 5.4.2 Running the Binary Update Tool

To process an executable with the Binary Update Tool, execute the following command:

```
scp-update-binary --binary=«compiled executable» «"*.nwdb" file»
```

The `scp-update-binary` file and the `.nwdb` file are located in the `Libraries` directory. Each target platform, for which tamper resistance is supported, has a separate `.nwdb` file.

The `--binary` parameter specifies a path to the application executable that contains the SKB library. As the final parameter, you must provide the `.nwdb` file of the particular target platform.

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

## 6 Decrypting PDF Files

---

SKB provides functions specifically dedicated to decrypting encrypted PDF files without revealing the user password and the derived encryption key. This chapter describes how to use these functions.

### 6.1 PDF Encryption Overview

A PDF document can be encrypted to protect its contents from unauthorized access. Encryption applies only to string and stream objects in the PDF file. Other objects, such as integers and Boolean values, which are used primarily to convey information about the document's structure rather than its content, are left unencrypted. In an encrypted PDF file, every string and stream object is encrypted with a different key. All these keys are derived from one primary encryption key, which in turn is derived from the user password.

Encryption-related information is stored in the document's encryption dictionary, which itself is stored in the **Encrypt** entry of the document's trailer dictionary. The trailer dictionary is a collection of key and value pairs in the very end of the PDF file. The absence of the **Encrypt** entry means that the document is not encrypted.

The encryption dictionary is also a collection of key and value pairs describing all necessary parameters of the particular encryption used.

### 6.2 PDF Requirements

SKB supports only encrypted PDF files that match the following criteria:

- PDF version is either 1.6 or 1.7.
- The following values are set in the encryption dictionary:

Key	Value
<b>Filter</b>	<i>Standard</i>
<b>Length</b>	<i>128</i>
<b>R</b>	<i>3</i>
<b>V</b>	<i>2</i>

These restrictions ensure that 128-bit AES in CBC mode is used for content encryption and that the proper key handling algorithms are chosen. Optionally, the **R** and **V** values can both be set to *4*, but then the following additional rules must be met:

- All crypt filters in the crypt filter dictionary must use *AESV2* as the value for the **CFM** parameter, and *16* for the **Length** parameter.

- The **EncryptMetadata** parameter in the encryption dictionary must be absent or set to *true*.

Note: PDF versions 1.5 and earlier use RC4 for content decryption. PDF versions above 1.7 use AES-256. These encryption algorithms are currently not supported by SKB.

## 6.3 Decrypting a PDF Document Using SKB

SKB provides the necessary algorithms to perform the following typical PDF decryption process:

1. Obtain the user password in secure format.

To ensure security, SKB requires that the user password is delivered as an `SKB_SecureData` object containing the password as raw bytes (data type is `SKB_DATA_TYPE_BYTES`). It is up to you to decide how to import the user password as an `SKB_SecureData` object. For information on `SKB_SecureData` objects and how they can be obtained, see §7.8.2.

2. Authenticate the user password using the `SKB_Pdf_AuthenticateUserPassword` function (see §6.3.1).

This algorithm verifies that the user password can actually decrypt the document. Authentication is done only once, typically when the PDF document is opened. The user password is always handled as a secure data object in protected form.

3. Derive the encryption key from the user password using the `SKB_Pdf_ComputeEncryptionKey` function (see §6.3.2).

Since the user password is not directly used for data decryption, an encryption key needs to be derived. This is done only once before any decryption is performed.

4. Prepare a PDF decryption context (represented by the `SKB_Pdf_DecryptionContext` object) using the `SKB_Pdf_CreateDecryptionContext` function (see §6.3.3).

A PDF decryption context must be created for every PDF object whose data you want to decrypt. The context is passed to the decryption function. The context holds a PDF object decryption key and optionally the initialization vector.

5. Decrypt a buffer from a PDF object using the `SKB_Pdf_DecryptionContext_ProcessBuffer` function (see §6.3.4).

You can call this function as many times as necessary to decrypt the required parts of a PDF object for which the PDF decryption context was created.

6. When the PDF decryption context is no longer needed, release it from the memory using the `SKB_Pdf_DecryptionContext_Release` function (see §6.3.5).

The functions mentioned above are not considered part of the main SKB API. Therefore, they are defined in a separate header file `skbExtensions.h`, which is located in the `Include` directory.

### 6.3.1 `SKB_Pdf_AuthenticateUserPassword`

This function verifies if the provided user password is valid.

The function is declared as follows:

```
SKB_Result
SKB_Pdf_AuthenticateUserPassword(const SKB_SecureData* password,
                                const SKB_Byte*      o,
                                SKB_Size             o_size,
                                int                  p,
                                const SKB_Byte*     file_id,
                                SKB_Size             file_id_size,
                                const SKB_Byte*     u,
                                SKB_Size             u_size,
                                SKB_Byte*           is_user_password_valid);
```

The following table describes the parameters:

Parameter	Description
password	Pointer to an <code>SKB_SecureData</code> object containing the user password.
o	Pointer to the value of parameter <b>O</b> in the encryption dictionary of the PDF file.
o_size	Size of the o value.
p	Value of parameter <b>P</b> in the encryption dictionary of the PDF file.
file_id	Pointer to the first element of the file identifier array. This array is the value of the <b>ID</b> entry in the document's trailer dictionary.
file_id_size	Size of the <code>file_id</code> value.
u	Pointer to the value of parameter <b>U</b> in the encryption dictionary of the PDF file.
u_size	Size of the u value.
is_user_password_valid	Pointer to an <code>SKB_Byte</code> variable that will be set to 1 if the user password is correct and 0 otherwise.

### 6.3.2 SKB\_Pdf\_ComputeEncryptionKey

This function derives the encryption key from the user password.

The function is declared as follows:

```
SKB_Result
SKB_Pdf_ComputeEncryptionKey(const SKB_SecureData* password,
                             const SKB_Byte*      o,
                             SKB_Size             o_size,
```

```

int
const SKB_Byte*
SKB_Size
SKB_SecureData**
p,
file_id,
file_id_size,
encryption_key);

```

The following table describes the parameters:

Parameter	Description
password	Pointer to an <code>SKB_SecureData</code> object containing the user password.
o	Pointer to the value of parameter <b>O</b> in the encryption dictionary of the PDF file.
o_size	Size of the <code>o</code> value.
p	Value of parameter <b>P</b> in the encryption dictionary of the PDF file.
file_id	Pointer to the first element of the file identifier array. This array is the value of the <b>ID</b> entry in the document's trailer dictionary.
file_id_size	Size of the <code>file_id</code> value.
encryption_key	Address of a pointer to the <code>SKB_SecureData</code> that will contain the derived encryption key after this function is executed.

### 6.3.3 SKB\_Pdf\_CreateDecryptionContext

This function prepares a PDF decryption context object that is later used in the `SKB_Pdf_DecryptionContext_ProcessBuffer` function (see §6.3.4).

The function is declared as follows:

```

SKB_Result
SKB_Pdf_CreateDecryptionContext(const SKB_SecureData* encryption_key,
                               int object_number,
                               int generation_number,
                               const SKB_Byte* iv
                               SKB_Pdf_DecryptionContext** ctx);

```

The following table describes the parameters:

Parameter	Description
<code>encryption_key</code>	Pointer to an <code>SKB_SecureData</code> object containing the encryption key, which you can obtain using the <code>SKB_Pdf_ComputeEncryptionKey</code> function (see §6.3.2).  The encryption key is combined with metadata of the particular PDF object to calculate a decryption key, which is then stored in the PDF decryption context object.
<code>object_number</code>	Object number in the PDF file.
<code>generation_number</code>	Generation number of the object.
<code>iv</code>	Pointer to an initialization vector to be stored in the PDF decryption context.  You can set this parameter to <code>NULL</code> , in which case the initialization vector must be passed in the first call of the <code>SKB_Pdf_DecryptionContext_ProcessBuffer</code> function (see §6.3.4).
<code>ctx</code>	Address of a pointer to the <code>SKB_Pdf_DecryptionContext</code> object that will contain the PDF decryption context after this function is executed.

### 6.3.4 `SKB_Pdf_DecryptionContext_ProcessBuffer`

This function decrypts a part of a particular encrypted object in the PDF file.

The function is declared as follows:

```
SKB_Result
SKB_Pdf_DecryptionContext_ProcessBuffer(
    SKB_Pdf_DecryptionContext* ctx,
    const SKB_Byte*          in_buffer,
    SKB_Size                 in_buffer_size,
    const SKB_Byte*          iv,
    SKB_Byte                 is_last_chunk,
    SKB_Byte*                out_buffer,
    SKB_Size*                out_buffer_size);
```

The following table describes the parameters:

Parameter	Description
<code>ctx</code>	Pointer to the <code>SKB_Pdf_DecryptionContext</code> object, which you prepared before using the <code>SKB_Pdf_CreateDecryptionContext</code> function (see §6.3.3).




Parameter	Description
<code>in_buffer</code>	Pointer to an input buffer containing a part of the encrypted PDF object data to be decrypted.
<code>in_buffer_size</code>	Size of the input buffer.
<code>iv</code>	<p>Pointer to the initialization vector.</p> <p>This parameter may be <code>NULL</code> in which case the initialization vector will be taken from the PDF decryption context. The last block of the processed buffer will be stored as the initialization vector in the PDF decryption context after this function is executed.</p>
<code>is_last_chunk</code>	<p>Parameter that should be set to 1 if this is the last part of the encrypted object data, and 0 otherwise.</p> <p>This information is used to process the CBC mode padding in the encrypted data and calculate the precise decrypted content length.</p> <p>If this parameter is 1 then the PDF decryption context will no longer contain an initialization vector after the function is executed, and the next call of the <code>SKB_Pdf_DecryptionContext_ProcessBuffer</code> function using the same PDF decryption context has to provide an initialization vector.</p>
<code>out_buffer</code>	<p>This parameter is either <code>NULL</code> or a pointer to the memory buffer where the decrypted content is to be written.</p> <p>If this parameter is <code>NULL</code>, the function simply returns, in <code>out_buffer_size</code>, a number of bytes that would be sufficient to hold the output, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and the buffer size is large enough to hold the output, the method stores the output there, sets <code>out_buffer_size</code> to the exact number of bytes stored, and returns <code>SKB_SUCCESS</code>. If the buffer is not large enough, then the method sets <code>out_buffer_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p>
<code>out_buffer_size</code>	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 <code>out_buffer</code> parameter.

You can actually point the `in_buffer` and `out_buffer` to the same memory buffer, in which case the encrypted input data will be overwritten with the decrypted content.

### 6.3.5 SKB\_Pdf\_DecryptionContext\_Release

This function releases a PDF decryption context object from the memory.

 You must always call this function when you have completed decrypting PDF object data and no longer need the PDF decryption context.

The function is declared as follows:

```
SKB_Result  
SKB_Pdf_DecryptionContext_Release(SKB_Pdf_DecryptionContext* ctx);
```

`ctx` is a pointer to the `SKB_Pdf_DecryptionContext` object to be released.

## 7 API Reference

---

This chapter provides full reference information about the SKB API.

### 7.1 API Overview

The SKB 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.

A concrete example is that 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.

### 7.2 Obtaining Class Instances

An instance of a class is obtained by declaring a pointer to an object for the class and passing the address of that pointer to a particular method. The method creates the instance 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 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 the new instance. Here is a sample call:

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

### 7.3 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` instance 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);
```

## 7.4 Method Return Values

All methods return an integer value of type `SKB_Result`. When a method call succeeds, the return value is `SKB_SUCCESS`. Otherwise, it is a negative number, as defined by constants in the header file.

The following table lists the defined return value constants:

Constant	Value	Description
<code>SKB_SUCCESS</code>	0	The called method was successfully executed.
<code>SKB_FAILURE</code>	-1	The method failed to perform the requested operation.
<code>SKB_ERROR_INTERNAL</code>	-80001	An internal SKB error occurred.
<code>SKB_ERROR_INVALID_PARAMETERS</code>	-80002	Invalid parameters were supplied to the method.
<code>SKB_ERROR_NOT_SUPPORTED</code>	-80003	The configuration provided to the method is not supported by SKB.
<code>SKB_ERROR_OUT_OF_RESOURCES</code>	-80004	The method failed to allocate the required amount of memory on the heap.
<code>SKB_ERROR_BUFFER_TOO_SMALL</code>	-80005	The provided memory buffer was not large enough to contain the output.
<code>SKB_ERROR_INVALID_FORMAT</code>	-80006	The format of the input buffer is invalid.
<code>SKB_ERROR_ILLEGAL_OPERATION</code>	-80007	The method was requested to perform an operation that it cannot.
<code>SKB_ERROR_INVALID_STATE</code>	-80008	You are trying to release the <code>SKB_Engine</code> object while it is still being referenced from other SKB objects.

Constant	Value	Description
<code>SKB_ERROR_OUT_OF_RANGE</code>	-80009	The specified offset or index of the input buffer is out of range.
<code>SKB_ERROR_EVALUATION_EXPIRED</code>	-80101	The evaluation period of the current SKB package has expired.
<code>SKB_ERROR_KEY_CACHE_FAILED</code>	-80102	A key cache operation failed. Typically this occurs when the key cache database is not available or is write-protected.
<code>SKB_ERROR_INVALID_EXPORT_KEY_VERSION</code>	-80103	Either you are trying to upgrade a key whose version number is equal or greater than that of the current SKB instance (see §3.7), or you are trying to import a key whose version is not equal to that of the current SKB instance.
<code>SKB_ERROR_INVALID_EXPORT_KEY</code>	-80104	The export key of the current SKB instance does not match the export key that was used for exporting the data that you are trying to import or upgrade.

## 7.5 Object Lifecycle

To avoid exceptions and correctly release memory, you have to follow certain rules regarding the lifecycle of SKB objects:

- All SKB objects must be released when they are no longer needed by calling the corresponding release methods.
- `SKB_Engine` (see §7.8.1) is the first SKB object to be created and the last one to be released. All other objects created via the `SKB_Engine` object must be released before it.
- `SKB_SecureData` (see §7.8.2) cannot be released while it is being used as a key in other objects, such as `SKB_Cipher`, `SKB_Transform`, and `SKB_KeyAgreement`.

## 7.6 Restrictions of Multithreading

As a general rule, SKB methods and objects are not synchronized and therefore they should 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 (see §7.9.1). This method will always return the same `SKB_Engine` instance.

- Since the `SKB_secureData` object is immutable, it can also be shared between multiple threads.

## 7.7 Overriding Memory Allocation Operators

You may want to override the `new` and `delete` operators to implement custom memory allocation for your application. To successfully achieve this, you must take into account that SKB uses the non-throwing `new` operator for memory allocation.

Assume you have the following code for overriding the `new` and `delete` operators:

```
void* operator new (size_t size)
{
    // your implementation
}

void* operator new[] (size_t size)
{
    // your implementation
}

void operator delete (void* ptr)
{
    // your implementation
}

void operator delete[] (void* ptr)
{
    // your implementation
}
```

SKB requires that you also provide the following implementations for the non-throwing operators:

```
void* operator new (size_t size, const std::nothrow_t&)
{
    return operator new (size);
}

void* operator new[] (size_t size, const std::nothrow_t&)
{
    return operator new[] (size);
}

void operator delete (void* ptr, const std::nothrow_t&)
{
    return operator delete (ptr);
}

void operator delete[] (void* ptr, const std::nothrow_t&)
{
    return operator delete[] (ptr);
}
```

## 7.8 Classes

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

### 7.8.1 SKB\_Engine

`SKB_Engine` is the first object that you create before using the API. It is used to initialize other API objects.

### 7.8.2 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:

- loading encrypted or plain 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 using the `SKB_SecureData_Derive` method
- wrapping plain keys

### 7.8.3 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.

### 7.8.4 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 a plain buffer of data.

### 7.8.5 SKB\_KeyAgreement


`SKB_KeyAgreement` is an object used to perform the key agreement algorithm.

## 7.9 Methods

This section describes all the methods provided by the API.

### 7.9.1 SKB\_Engine\_GetInstance

This method creates an `SKB_Engine` instance (see §7.8.1). This instance is the first object that you must obtain before using the API.

 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.

### 7.9.2 SKB\_Engine\_Release

This method releases an `SKB_Engine` instance once it is no longer needed.

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

 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 that should be released.

### 7.9.3 SKB\_Engine\_SetDeviceId

This method sets the device ID, which is a byte array of arbitrary length that will be combined with the export key to form a unique format for exported keys. This method enables you to bind exported keys to a specific device (see §3.16). By default, when an engine is initialized, there is no device ID set and the export format depends only on the export key.

The method is declared as follows:




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

The following table describes the parameters:

Parameter	Description
<code>self</code>	Pointer to the pre-initialized engine.
<code>id</code>	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.
<code>size</code>	Number of bytes in the <code>id</code> 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.

#### 7.9.4 SKB\_Engine\_GetInfo

This method populates an `SKB_EngineInfo` structure (see §7.10.2) with the generic information about an initialized engine.

 The contents 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 method is declared as follows:

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

The following table describes the parameters:

Parameter	Description
<code>self</code>	Pointer to the pre-initialized engine that you want to get the information about.
<code>info</code>	Pointer to the <code>SKB_EngineInfo</code> structure to be populated with the engine information (see §7.10.2).

## 7.9.5 SKB\_Engine\_CreateDataFromWrapped

This method creates a new `SKB_SecureData` object from a wrapped buffer of data (usually a cryptographic key) by unwrapping it with a previously loaded key. The unwrapped data is never exposed in plain form. For more information on using this method, see §3.1.

As a special case of calling this method, you can also load a plain buffer of data as an `SKB_SecureData` object (see §3.2). In this case, the unwrapping algorithm and the decryption 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. The loading of plain keys can be executed only if loading of plain data is enabled in SKB (see §4.1).

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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the pre-initialized engine.
<code>wrapped</code>	Pointer to the buffer of encrypted data (cryptographic key) to be unwrapped.  If you are unwrapping a key for the Triple DES algorithm, make sure the input corresponds to the format described in §8.3.  If you are unwrapping an AES-wrapped ECC private key, for information on how the input buffer must be formatted, see §8.7.  For other cases of AES-wrapped data, see §8.2.
<code>wrapped_size</code>	Size of the buffer of encrypted data in bytes.
<code>wrapped_type</code>	Type of the wrapped key. The available types are defined in the <code>SKB_DataType</code> enumeration (see §7.11.1).
<code>wrapped_format</code>	Format how the wrapped key is stored in the input data buffer. The available formats are defined in the <code>SKB_DataFormat</code> enumeration (see §7.11.7).

Parameter	Description
wrapping_algorithm	<p>Cryptographic algorithm to be used for decrypting the data. The available algorithms are defined in the <code>SKB_CipherAlgorithm</code> enumeration (see §7.11.3). For information on algorithms that support key unwrapping, see §1.2.</p> <p>The following algorithms only support unwrapping of raw bytes, meaning that the <code>wrapped_type</code> parameter should always be <code>SKB_DATA_TYPE_BYTES</code>, and <code>wrapped_format</code> should always be <code>SKB_DATA_FORMAT_RAW</code>:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_CIPHER_ALGORITHM_ECC_ELGAMAL</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_RSA</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_RSA_1_5</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_RSA_OAEP</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_NIST_AES</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_CMLA</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_RSA_CMLA</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_XOR</code></li> </ul> <p>If the <code>SKB_CIPHER_ALGORITHM_NIST_AES</code> algorithm is used, in the case of integrity check failure this method will return the <code>SKB_ERROR_INVALID_FORMAT</code> error.</p> <p>If the <code>SKB_CIPHER_ALGORITHM_NULL</code> 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.</p> <p>If the <code>SKB_CIPHER_ALGORITHM_ECC_ELGAMAL</code> algorithm is used, see the special instructions described in §3.1.1.</p>

Parameter	Description
<code>wrapping_parameters</code>	<p>Additional parameters for the unwrapping algorithm.</p> <p>You can optionally point this parameter to the <code>SKB_AesUnwrapParameters</code> structure (see §7.10.20) to specify the CBC padding type, if you are using one of the following algorithms:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_128_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_192_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_256_CBC</code></li> </ul> <p>If you use any of the algorithms above and set the <code>wrapping_parameters</code> value to <code>NULL</code>, CBC mode with the XML encryption padding will be used by default, which is the equivalent of the <code>SKB_CBC_PADDING_TYPE_XMLENC</code> value of the <code>SKB_CbcPadding</code> enumeration (see §7.11.13).</p> <p>If you are using the <code>SKB_CIPHER_ALGORITHM_ECC_ELGAMAL</code> algorithm, this parameter must be a pointer to the <code>SKB_EccParameters</code> structure (see §7.10.22). For special instructions for using the ElGamal ECC unwrapping algorithm, see §3.1.1.</p> <p>For all other cases, set this parameter to <code>NULL</code>.</p>
<code>unwrapping_key</code>	<code>SKB_SecureData</code> object containing the key needed to decrypt the data.
<code>data</code>	Address of a pointer to the <code>SKB_SecureData</code> that will contain the unwrapped key after this method is executed.

## 7.9.6 SKB\_Engine\_CreateDataFromExported

This method imports data that was previously exported using the `SKB_SecureData_Export` method (see §7.9.15). This operation can be performed only if the importing SKB instance has the same export key as the one that exported the data (see §1.1.6).

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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the pre-initialized engine.
<code>exported</code>	Pointer to the memory buffer containing the exported data.
<code>exported_size</code>	Size of the memory buffer.
<code>data</code>	Address of a pointer to the <code>SKB_SecureData</code> object that will be created by this method. This object will contain the imported data.

### 7.9.7 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 §3.4.

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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the pre-initialized engine.
<code>plain</code>	Pointer to the memory buffer where the plain input data is stored.
<code>plain_size</code>	Pointer to a variable that holds the size of the input data in bytes.
<code>data_type</code>	Type of data stored in the input buffer. The available types are defined in the <code>SKB_DataType</code> enumeration (see §7.11.1).  Currently, this method supports only the <code>SKB_DATA_TYPE_BYTES</code> data type.

Parameter	Description
<code>plain_format</code>	<p>Format how the plain data is stored in the input buffer. The available formats are defined in the <code>SKB_DataFormat</code> enumeration (see §7.11.7).</p> <p>Currently, this method supports only the <code>SKB_DATA_FORMAT_RAW</code> data type.</p>
<code>algorithm</code>	<p>Algorithm to be used for encrypting the input data. Available algorithms are defined in the <code>SKB_CipherAlgorithm</code> enumeration (see §7.11.3).</p> <p>Currently, this method supports only the following algorithms:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_128_ECB</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_128_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_192_ECB</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_192_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_256_ECB</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_256_CBC</code></li> </ul>
<code>encryption_parameters</code>	<p>Pointer to a structure that provides additional parameters for the cipher.</p> <p>Currently, this parameter must always be <code>NULL</code>.</p>
<code>encryption_key</code>	Pointer to the <code>SKB_SecureData</code> object containing the encryption key.
<code>iv</code>	<p>Pointer to the initialization vector if the encryption algorithm used is AES in CBC mode.</p> <p>If the initialization vector is not used or if it is all zeros the value of this parameter should be <code>NULL</code>.</p>
<code>iv_size</code>	<p>Size of the initialization vector in bytes.</p> <p>If the value of the <code>iv</code> parameter is <code>NULL</code>, this parameter should be 0.</p>
<code>data</code>	Address of a pointer to the <code>SKB_SecureData</code> object that will contain the output when this method is executed.

### 7.9.8 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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the pre-initialized engine.
<code>data_type</code>	Type of data to be generated. The available types are defined in the <code>SKB_DataType</code> enumeration (see §7.11.1).  Currently, the <code>SKB_DATA_TYPE_RSA_PRIVATE_KEY</code> type is not supported for generating secure data, meaning that SKB cannot generate private RSA keys.
<code>generate_parameters</code>	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: <ul style="list-style-type: none"> <li>For <code>SKB_DATA_TYPE_BYTES</code>, this parameter must point to the <code>SKB_RawBytesParameters</code> structure (see §7.10.24), which specifies the number of bytes to be generated.</li> <li>For <code>SKB_DATA_TYPE_ECC_PRIVATE_KEY</code>, this parameter must point to the <code>SKB_EccParameters</code> structure (see §7.10.22), which specifies the ECC curve type to be used.</li> </ul>
<code>data</code>	Address of a pointer to the <code>SKB_SecureData</code> object that will be created by this method. This object will contain the generated data.

### 7.9.9 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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the pre-initialized engine.
<code>transform_type</code>	Transform type to be created. Available transform types are defined in the <code>SKB_TransformType</code> enumeration (see §7.11.8).
<code>transform_parameters</code>	<p>Pointer to a structure that provides the necessary parameters for the transform. For different transform types, a different structure must be provided:</p> <p>For the <code>SKB_TRANSFORM_TYPE_DIGEST</code> transform, this parameter must point to the <code>SKB_DigestTransformParameters</code> structure (see §7.10.5).</p> <p>For the <code>SKB_TRANSFORM_TYPE_SIGN</code> transform, this parameter must point to one of the following structures:</p> <ul style="list-style-type: none"> <li>▪ If one of the following algorithms is to be used, this parameter must point to the <code>SKB_SignTransformParametersEx</code> structure (see §7.10.7): <ul style="list-style-type: none"> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_ECDSA</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256</code></li> </ul> </li> <li>▪ For all other algorithms, this parameter must point to the <code>SKB_SignTransformParameters</code> structure (see §7.10.6).</li> </ul> <p>For the <code>SKB_TRANSFORM_TYPE_VERIFY</code> transform, this parameter must point to the <code>SKB_VerifyTransformParameters</code> structure (see §7.10.8).</p>
<code>transform</code>	Address of a pointer to the <code>SKB_Transform</code> object that will be created by this method.

### 7.9.10 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 table explains the parameters:

Parameter	Description
self	Pointer to the pre-initialized engine.
cipher_algorithm	Algorithm to be used for encrypting or decrypting data. Available algorithms are defined in the <code>SKB_CipherAlgorithm</code> enumeration (see §7.11.3).
cipher_direction	Parameter that specifies whether the provided data should be encrypted or decrypted. Available directions are defined in the <code>SKB_CipherDirection</code> enumeration (see §7.11.6).  Encryption is supported only for the DES, Triple DES, and AES ciphers.
cipher_flags	Optional flags for the cipher.  Currently, the only defined flag is <code>SKB_CIPHER_FLAG_HIGH_SPEED</code> . This flag can be used only for the AES cipher when it is intended to be used with high throughput, for example when used for media content decryption.
cipher_parameters	Pointer to a structure that provides additional parameters for the cipher.  For the <code>SKB_CIPHER_ALGORITHM_AES_128_CTR</code> , <code>SKB_CIPHER_ALGORITHM_AES_192_CTR</code> , and <code>SKB_CIPHER_ALGORITHM_AES_256_CTR</code> ciphers, it must point to the <code>SKB_CtrModeCipherParameters</code> structure (see §7.10.4), or <code>NULL</code> for the default counter size of 16.  For the <code>SKB_CIPHER_ALGORITHM_ECC_ELGAMAL</code> cipher, it must point to the <code>SKB_EccParameters</code> structure, which specifies the curve type (see §7.10.22).  For all other ciphers, this parameter must be <code>NULL</code> .
cipher_key	Pointer to the <code>SKB_SecureData</code> object containing the encryption or decryption key.

Parameter	Description
<code>cipher</code>	Address of a pointer to the <code>SKB_cipher</code> object which will be created by this method.

### 7.9.11 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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the pre-initialized engine.
<code>key_agreement_algorithm</code>	Key agreement algorithm to be used. Available algorithms are defined in the <code>SKB_KeyAgreementAlgorithm</code> enumeration (see §7.11.11).
<code>key_agreement_parameters</code>	Pointer to a structure providing the necessary parameters for the key agreement algorithm.  For the <code>SKB_KEY_AGREEMENT_ALGORITHM_ECDH</code> algorithm, this parameter must point to an <code>SKB_EccParameters</code> structure (see §7.10.22).  For the <code>SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH</code> algorithm, this parameter must point to an <code>SKB_PrimeDhParameters</code> structure (see §7.10.23).
<code>key_agreement</code>	Address of a pointer to the <code>SKB_KeyAgreement</code> object which will be created by this method.

### 7.9.12 SKB\_Engine\_UpgradeExportedData

This method upgrades an exported `SKB_secureData` object to the latest version as described in §3.7.

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 table explains the parameters:

Parameter	Description
<code>engine</code>	Pointer to the pre-initialized engine.
<code>input</code>	Input data buffer containing the previously exported <code>SKB_SecureData</code> object that needs to be upgraded to the latest export format.
<code>input_size</code>	Size of the input data buffer in bytes.
<code>buffer</code>	<p>This parameter is either <code>NULL</code> or a pointer to the memory buffer where the upgraded data is to be written.</p> <p>If this parameter is <code>NULL</code>, the method simply returns, in <code>buffer_size</code>, a number of bytes that would be sufficient to hold the output, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and the buffer size is large enough to hold the output, the method stores the output there, sets <code>buffer_size</code> to the exact number of bytes stored, and returns <code>SKB_SUCCESS</code>. If the buffer is not large enough, then the method sets <code>buffer_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p>
<code>buffer_size</code>	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 <code>buffer</code> parameter.

### 7.9.13 SKB\_SecureData\_Release

This method releases an `SKB_SecureData` object. It should 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 should be released.

### 7.9.14 SKB\_SecureData\_GetInfo

This method provides 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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the <code>SKB_SecureData</code> object whose size and type you want to know.
<code>info</code>	Pointer to the <code>SKB_DataInfo</code> structure, which will be populated by this method to return the characteristics of the <code>SKB_SecureData</code> object (see §7.10.3).

### 7.9.15 SKB\_SecureData\_Export

This method returns a protected form of the contents of a particular `SKB_SecureData` object. This protected data is intended for exporting to a persistent storage. Later the exported data can be imported back into the same SKB instance (with the same export key) using the `SKB_Engine_CreateDataFromExported` method (see §7.9.6).

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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the <code>SKB_SecureData</code> object to be exported.
<code>target</code>	Export type to be used. Available export types are defined in the <code>SKB_ExportTarget</code> enumeration (see §7.11.9).
<code>target_parameters</code>	Currently, this parameter is not used.

Parameter	Description
<code>buffer</code>	<p>This parameter is either <code>NULL</code> or a pointer to the memory buffer where the exported data is to be written.</p> <p>If this parameter is <code>NULL</code>, the method simply returns, in <code>buffer_size</code>, a number of bytes that would be sufficient to hold the exported data, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and the buffer size is large enough to hold the exported data, the method stores the exported data there, sets <code>buffer_size</code> to the exact number of bytes stored, and returns <code>SKB_SUCCESS</code>. If the buffer is not large enough, then the method sets <code>buffer_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p> <p>For information on the exported data format, see §8.1.</p>
<code>buffer_size</code>	<p>Pointer to a variable that holds the size of the memory buffer in bytes where the exported data is to be stored. For more details, see the description of the <code>buffer</code> parameter.</p>

### 7.9.16 SKB\_SecureData\_Wrap

This method wraps (encrypts) the contents of a particular `SKB_SecureData` object using a specified cipher and wrapping key. For more information on wrapping secure data, see §3.3.

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 table explains the parameters:

Parameter	Description
<code>self</code>	<p>Pointer to the <code>SKB_SecureData</code> object whose contents need to be wrapped.</p>

Parameter	Description
wrapping_algorithm	<p>Wrapping algorithm to be used. The available algorithms are defined in the <code>SKB_CipherAlgorithm</code> enumeration (see §7.11.3).</p> <p>Currently, only the following algorithms are supported for wrapping:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_128_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_192_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_256_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_XOR</code></li> </ul> <p>The AES-based algorithms can only be used on <code>SKB_SecureData</code> objects whose data type is <code>SKB_DATA_TYPE_BYTES</code> or <code>SKB_DATA_TYPE_ECC_PRIVATE_KEY</code> (see §7.11.1).</p> <p>The <code>SKB_CIPHER_ALGORITHM_XOR</code> algorithm can only be used on <code>SKB_SecureData</code> objects whose data type is <code>SKB_DATA_TYPE_BYTES</code>.</p>
wrapping_parameters	<p>Pointer to a structure that provides additional parameters for the wrapping algorithm.</p> <p>This parameter is applicable only if you use one of the following algorithms:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_128_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_192_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_256_CBC</code></li> </ul> <p>Then this parameter can be used to provide a specific initialization vector to the AES wrapping algorithm. In that case, you should point this parameter to the <code>SKB_AesWrapParameters</code> structure (see §7.10.19) where the initialization vector is specified. If this structure is not provided (<code>wrapping_parameters</code> is <code>NULL</code>), the AES algorithm generates a random initialization vector.</p>
wrapping_key	<p>Pointer to the <code>SKB_SecureData</code> object containing the wrapping key.</p>

Parameter	Description
<code>buffer</code>	<p>This parameter is either <code>NULL</code> or a pointer to the memory buffer where the output is to be stored.</p> <p>If this parameter is <code>NULL</code>, the call is simply a request to find out how many bytes are needed to store the output, so the method returns, in <code>buffer_size</code>, a number indicating how many bytes would be sufficient to hold the output, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and <code>buffer_size</code> is large enough to hold the output, the method places the output there and sets <code>buffer_size</code> to the exact number of bytes stored. If the buffer is not large enough, then the method sets <code>buffer_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p> <p>For information on the way the output buffer is formatted in case you use the AES-based algorithms, see §8.2.3.</p>
<code>buffer_size</code>	<p>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 <code>buffer</code> parameter.</p>

### 7.9.17 SKB\_SecureData\_Derive

This method creates a new `SKB_SecureData` object from another `SKB_SecureData` object using a particular derivation algorithm. This method can only be used on `SKB_SecureData` objects whose data type is `SKB_DATA_TYPE_BYTES` (see §7.11.1).

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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the <code>SKB_SecureData</code> object from which a new <code>SKB_SecureData</code> object needs to be derived.
<code>algorithm</code>	Derivation algorithm to be used. The available algorithms are defined in the <code>SKB_DerivationAlgorithm</code> enumeration (see §7.11.5).

Parameter	Description
parameters	<p>Pointer to a structure containing parameters for the derivation algorithm. For different algorithms, a different structure must be provided:</p> <ul style="list-style-type: none"> <li>▪ If the <code>SKB_DERIVATION_ALGORITHM_SLICE</code> or <code>SKB_DERIVATION_ALGORITHM_BLOCK_SLICE</code> algorithm is used, this parameter must point to the <code>SKB_SliceDerivationParameters</code> structure, which defines the range of bytes to be derived as a new <code>SKB_SecureData</code> object (see §7.10.17).</li> <li>▪ If the <code>SKB_DERIVATION_ALGORITHM_SELECT_BYTES</code> algorithm is used, this parameter must point to the <code>SKB_SelectBytesDerivationParameters</code> structure, which provides the necessary input parameters (see §7.10.9).</li> <li>▪ If the <code>SKB_DERIVATION_ALGORITHM_CIPHER</code> algorithm is used, this parameter must point to the <code>SKB_CipherDerivationParameters</code> object (see §7.10.10).</li> <li>▪ If the <code>SKB_DERIVATION_ALGORITHM_SHA_1</code> algorithm is used, this parameter may point to the <code>SKB_Sha1DerivationParameters</code> structure, which specifies how many times the SHA-1 algorithm should be executed and how many bytes from the result should be derived (see §7.10.11). If the parameter is <code>NULL</code>, the SHA-1 algorithm will be executed once and the whole output of 20 bytes will be returned as a new <code>SKB_SecureData</code> object.</li> <li>▪ If the <code>SKB_DERIVATION_ALGORITHM_SHA_256</code> algorithm is used, this parameter may point to the <code>SKB_Sha256DerivationParameters</code> structure, which provides the plain buffers that should be prepended and appended to the <code>SKB_SecureData</code> object processed (see §7.10.12). If the parameter is <code>NULL</code>, SKB will assume that there are no plain data buffers to be prepended or appended.</li> <li>▪ If the <code>SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128</code> algorithm is used, this parameter must point to the <code>SKB_Nist800108CounterCmacAes128Parameters</code> structure, which provides the necessary input parameters (see §7.10.13).</li> <li>▪ If the <code>SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2</code> algorithm is used, this parameter must point to the <code>SKB_OmaDrmKdf2DerivationParameters</code> structure, which provides the necessary input parameters (see §7.10.16).</li> <li>▪ If the <code>SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE</code> algorithm is used, this parameter may point to the <code>SKB_RawBytesFromEccPrivateDerivationParameters</code> structure, which specifies whether the output should be encoded in little-endian or big-endian (see §7.10.14). If the parameter is <code>NULL</code>, the output will be encoded in little-endian.</li> </ul>



Parameter	Description
	<ul style="list-style-type: none"> <li>If the <code>SKB_DERIVATION_ALGORITHM_SHA_AES</code> algorithm is used, this parameter must point to the <code>SKB_ShaAesDerivationParameters</code> structure, which provides the necessary input parameters (see §7.10.15).</li> <li>For all other key derivation algorithms, this parameter is not used and therefore should be <code>NULL</code>.</li> </ul>
<code>data</code>	Address of a pointer that will point to the new derived <code>SKB_SecureData</code> object when this method is executed.

### 7.9.18 SKB\_SecureData\_GetPublicKey

This method returns a public key that corresponds to the supplied private key.

Currently, this method supports only ECC keys, but not RSA.

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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the <code>SKB_SecureData</code> object containing the private key. From this key, the public key will be derived.
<code>format</code>	<p>Format in which the derived public key should be stored in the returned buffer of bytes. The available formats are defined in the <code>SKB_DataFormat</code> enumeration (see §7.11.7).</p> <p>Currently, the only valid value is <code>SKB_DATA_FORMAT_ECC_BINARY</code>.</p>
<code>parameters</code>	<p>Pointer to a structure containing parameters necessary for the deriving of the public key.</p> <p>Since SKB supports only ECC key generation, this parameter should point to the <code>SKB_EccParameters</code> structure, which specifies the ECC curve type (see §7.10.22).</p>

Parameter	Description
<code>output</code>	<p>If this parameter is <code>NULL</code>, the call is simply a request to find out how many bytes are needed to store the public key. Then the method returns, in <code>output_size</code>, a number indicating how many bytes would be sufficient to hold the output, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and the buffer size (<code>output_size</code>) is large enough to hold the public key output, the method places the output there and sets <code>output_size</code> to the exact number of bytes stored. If the buffer is not large enough, then the method sets <code>output_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p> <p>After successfully executing the method, the content of the <code>output</code> parameter will be a pointer to a buffer of bytes containing the public key. For information on the format used, see §8.5.</p>
<code>output_size</code>	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 <code>output</code> parameter.

### 7.9.19 SKB\_Transform\_Release

This method releases the specified `SKB_Transform` object.

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.

### 7.9.20 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) can be 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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the previously created <code>SKB_Transform</code> object.
<code>data</code>	Pointer to the buffer of data to be appended to the <code>SKB_Transform</code> object.
<code>data_size</code>	Size of the data buffer in bytes.

### 7.9.21 `SKB_Transform_AddSecureData`

This method appends the contents 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) can be executed.

**⚠** This method cannot be used for the `SKB_SIGNATURE_ALGORITHM_RSA` and `SKB_SIGNATURE_ALGORITHM_ECDSA` signing algorithms because they can operate only on plain input.

The method is declared as follows:

```
SKB_Result
SKB_Transform_AddSecureData (SKB_Transform*      self,
                             const SKB_SecureData* data);
```

The following table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the previously created <code>SKB_Transform</code> object.
<code>data</code>	Pointer to the <code>SKB_SecureData</code> object whose contents must be appended to the <code>SKB_Transform</code> object.

### 7.9.22 `SKB_Transform_GetOutput`

This method executes a transform algorithm on a particular `SKB_Transform` object. The transform algorithm is specified during the creation of the `SKB_Transform` object, and the input data is then provided using the `SKB_Transform_AddBytes` and `SKB_Transform_AddSecureData` methods.

The `SKB_Transform_GetOutput` method is declared as follows:

```
SKB_Result
SKB_Transform_GetOutput (SKB_Transform* self,
                        SKB_Byte*      output,
                        SKB_Size*      output_size);
```

The following table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the <code>SKB_Transform</code> object on which the transform algorithm must be executed.
<code>output</code>	<p>This parameter is either <code>NULL</code> or a pointer to the memory buffer where the transform output will be stored.</p> <p>If this parameter is <code>NULL</code>, the method returns, in <code>output_size</code>, a number of bytes sufficient to hold the output, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and the buffer size (<code>output_size</code>) is large enough to hold the output, the method stores the output there and sets <code>output_size</code> to the exact number of bytes stored. If the buffer is not large enough, then the method sets <code>output_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p> <p>In the case of the <code>SKB_TRANSFORM_TYPE_VERIFY</code> transform, the output will be a single byte with the value 1 if the signature is verified and 0 if it is not.</p> <p>In the case of the ECDSA signature algorithm, the output will be a pointer to a buffer with a format described in §8.8.</p>
<code>output_size</code>	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 <code>output</code> parameter.

### 7.9.23 SKB\_Cipher\_ProcessBuffer

This method performs either data encryption or decryption depending on the previously created `SKB_Cipher` object (see §7.8.3).

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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the previously created <code>SKB_Cipher</code> object, which contains all the necessary parameters.
<code>in_buffer</code>	<p>Pointer to a buffer of data to be encrypted or decrypted.</p> <p>For block ciphers, this parameter must point to the beginning of a cipher block.</p> <p>For the ElGamal ECC cipher, this parameter must be a pointer to a buffer of bytes described in §8.4.</p>
<code>in_buffer_size</code>	<p>Size in bytes of the data buffer to be encrypted or decrypted.</p> <p>For the DES and Triple DES cipher in the ECB mode, this parameter must be a multiple of the cipher block size, which is 8 bytes.</p> <p>For the AES cipher in the ECB or CBC mode, this parameter must be a multiple of the cipher block size, which is 16 bytes.</p> <p>For the RSA cipher, this parameter must be the size of the entire encrypted message, but no more than the length of the RSA key.</p>
<code>out_buffer</code>	<p>This parameter is either <code>NULL</code> or a pointer to the memory buffer where the output is to be stored.</p> <p>If this parameter is <code>NULL</code>, the call is simply a request to find out how many bytes are needed for the cipher output, so the method returns, in <code>out_buffer_size</code>, a number indicating how many bytes would be sufficient to hold the output, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and the buffer size (<code>out_buffer_size</code>) is large enough to hold the cipher output, the method places the output there and sets <code>out_buffer_size</code> to the exact number of bytes stored. If the buffer is not large enough, then the method sets <code>out_buffer_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p> <p>For the ElGamal ECC cipher, the output buffer contains the X coordinate of the decrypted point in big-endian notation. It is caller's responsibility to extract the decrypted message from this output according to the way the message was encrypted.</p> <p>SKB supports in-place encryption and decryption, which means that the <code>out_buffer</code> parameter may be the same as the <code>in_buffer</code> parameter. Then, the output of this method will overwrite the input.</p>

Parameter	Description
<code>out_buffer_size</code>	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 <code>out_buffer</code> parameter.
<code>iv</code>	<p>Pointer to the initialization vector if you use the AES cipher in the CBC or CTR mode, or <code>NULL</code> in other cases.</p> <p>The initialization vector must be provided in the first call of this method. In subsequent calls, you may set the <code>iv</code> parameter to <code>NULL</code>, 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 interprets the input buffer as a new message.</p>
<code>iv_size</code>	Size in bytes of the initialization vector. It should be 0 if the <code>iv</code> parameter is <code>NULL</code> .

### 7.9.24 SKB\_Cipher\_Release

This method releases an `SKB_Cipher` object. It should 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 that should be released.

### 7.9.25 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 table explains the parameters:

Parameter	Description
-----------	-------------

Parameter	Description
<code>self</code>	Pointer to the previously created <code>SKB_KeyAgreement</code> object, which contains all the necessary parameters.
<code>public_key_buffer</code>	<p>This parameter is either <code>NULL</code> or a pointer to the memory buffer where the public key will be stored.</p> <p>If this parameter is <code>NULL</code>, the method returns, in <code>public_key_buffer_size</code>, a number of bytes sufficient to hold the public key, and returns <code>SKB_SUCCESS</code>.</p> <p>If this parameter points to a memory buffer (it is not <code>NULL</code>), and the buffer size <code>public_key_buffer_size</code> is large enough to hold the public key, the method stores the output there and sets <code>public_key_buffer_size</code> to the exact number of bytes stored. If the buffer is not large enough, the method sets <code>public_key_buffer_size</code> to a number of bytes that would be sufficient, and returns <code>SKB_ERROR_BUFFER_TOO_SMALL</code>.</p> <p>For the <code>SKB_KEY_AGREEMENT_ALGORITHM_ECDH</code> algorithm, the public key is stored using the format described in §8.5.</p> <p>For the <code>SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH</code> algorithm, the buffer size is 128 bytes, and it stores the public value encoded in big-endian.</p>
<code>public_key_buffer_size</code>	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 <code>public_key_buffer</code> parameter.

### 7.9.26 `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 table explains the parameters:

Parameter	Description
<code>self</code>	Pointer to the previously created <code>SKB_KeyAgreement</code> object, which contains all the necessary parameters.
<code>peer_public_key</code>	Pointer to the memory buffer where the public key received from the other party is stored.  For the <code>SKB_KEY_AGREEMENT_ALGORITHM_ECDH</code> algorithm, the public key is expected to be stored using the format described in §8.5.  For the <code>SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH</code> algorithm, the buffer has to be 128 bytes long, and it should store the public value encoded in big-endian.
<code>peer_public_key_size</code>	Size of the <code>peer_public_key</code> parameter in bytes. This size must be equal to the value returned by the <code>SKB_KeyAgreement_GetPublicKey</code> method used by the other key agreement party.
<code>secret_size</code>	Size of the desired shared secret data output.  To select the largest possible shared secret size, the value <code>SKB_KEY_AGREEMENT_MAXIMAL_SECRET_SIZE</code> should be passed as an input for this parameter.
<code>secret</code>	Address of a pointer to the <code>SKB_SecureData</code> object containing the shared secret data that will be created by this method. The bytes are ordered using the big-endian notation.

### 7.9.27 SKB\_KeyAgreement\_Release

This method releases an `SKB_KeyAgreement` object. It should 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 that should be released.

## 7.10 Supporting Structures

This section describes various supporting structures used by the API.



### 7.10.1 SKB\_EngineProperty

`SKB_EngineProperty` is a name-value pair representing a particular `SKB_Engine` property in the `SKB_EngineInfo` structure (see §7.10.2).


The `SKB_EngineProperty` structure is declared as follows:

```
typedef struct {
    const char* name;
    const char* value;
} SKB_EngineProperty;
```

For information on available properties, see §7.10.2.

### 7.10.2 SKB\_EngineInfo

`SKB_EngineInfo` is a structure that is populated by the `SKB_Engine_GetInfo` method (see §7.9.4) to provide information about a particular `SKB_Engine` instance.

 The contents 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 table describes the properties:

Property	Description
<code>major</code> , <code>minor</code> , <code>revision</code>	Version numbers specified in the API header file.
<code>flags</code>	Currently, this property is not used because there are no engine-specific flags defined.
<code>property_count</code>	Number of elements in the <code>properties</code> array.

Property	Description
properties	<p>Array of engine properties with <code>property_count</code> elements, where each property specified is an <code>SKB_EngineProperty</code> structure (see §7.10.1).</p> <p>The following properties are used:</p> <ul style="list-style-type: none"> <li>▪ <code>implementation</code>: Cryptographic technique used by SKB. Available values are the following: <ul style="list-style-type: none"> <li>▪ <code>v</code>: identifies an implementation based on composite automata</li> <li>▪ <code>p</code>: identifies an implementation based on polynomial encryption</li> </ul> </li> <li>▪ <code>key_cache</code>: Key caching mechanism used by SKB. Available values are <code>sqlite</code>, <code>memory</code>, and <code>custom</code>. For information on key caching and its modes, see §4.2.3.</li> <li>▪ <code>key_cache_max_items</code>: Maximum number of keys that can be cached in the memory. This property is available only if the <code>memory</code> key caching mechanism is used.</li> <li>▪ <code>diversification_guid</code>: Unique diversification identifier consisting of 16 bytes in the hexadecimal format. SKB packages with the same binary implementation will have the same identifier.</li> <li>▪ <code>export_guid</code>: Export key identifier consisting of 16 bytes in the hexadecimal format. SKB packages with the same export key will have the same identifier.</li> <li>▪ <code>export_key_version</code>: Current export key version in the one-way data upgrade scheme (see §3.7).</li> </ul>

### 7.10.3 SKB\_DataInfo

This structure is used by the `SKB_SecureData_GetInfo` method to return the size and type of a particular `SKB_SecureData` object (see §7.9.14).

The structure is declared as follows:

```
typedef struct {
    SKB_DataType type;
    SKB_Size     size;
} SKB_DataInfo;
```

The following table explains the properties:

Property	Description
type	Type of the data stored within the <code>SKB_SecureData</code> object. Available types are defined in the <code>SKB_DataType</code> enumeration (see §7.11.1).
size	Size of the contents in bytes. Value 0 means that the information is not available. For the data type <code>SKB_DATA_TYPE_RSA_PRIVATE_KEY</code> , this value is the modulus in bytes.

#### 7.10.4 SKB\_CtrModeCipherParameters

This structure provides an additional parameter for the `SKB_Engine_CreateCipher` method when the `SKB_CIPHER_ALGORITHM_AES_128_CTR`, `SKB_CIPHER_ALGORITHM_AES_192_CTR`, and `SKB_CIPHER_ALGORITHM_AES_256_CTR` algorithms are used (see §7.9.10).

The structure is declared as follows:

```
typedef struct {
    SKB_Size counter_size;
} SKB_CtrModeCipherParameters;
```

The property `counter_size` specifies the counter size in bytes.

#### 7.10.5 SKB\_DigestTransformParameters

This structure is used by the `SKB_Engine_CreateTransform` method if the `SKB_TRANSFORM_TYPE_DIGEST` transform is used (see §7.9.9). 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 (see §7.11.2).

#### 7.10.6 SKB\_SignTransformParameters

This structure is used by the `SKB_Engine_CreateTransform` method if the `SKB_TRANSFORM_TYPE_SIGN` transform is used (see §7.9.9). 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 table explains the properties:

Property	Description
algorithm	Signing algorithm to be used. The available signing algorithms are defined in the <code>SKB_SignatureAlgorithm</code> enumeration (see §7.11.4).
key	Pointer to the <code>SKB_SecureData</code> object, which contains the signing key.  This key must not be released before the <code>SKB_Transform</code> object that uses it is released.

### 7.10.7 SKB\_SignTransformParametersEx

This structure is an extension to the `SKB_SignTransformParameters` structure. It provides the 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 table explains the properties:

Property	Description
base	<code>SKB_SignTransformParameters</code> structure that specifies the signature algorithm and the key to be used (see §7.10.6).

Property	Description
extension	<p>If one of the following signature algorithms is used, this pointer must point to the <code>SKB_EccParameters</code> structure, which specifies the ECC curve type to be used (see §7.10.22):</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_ECDSA</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256</code></li> </ul> <p>If the <code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX</code> OR <code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX</code> signature algorithm is used, this pointer must point to the <code>SKB_RsaPssParameters</code> structure, which specifies the salt and salt length (see §7.10.21).</p>

### 7.10.8 SKB\_VerifyTransformParameters

This structure is used by the `SKB_Engine_CreateTransform` method if the `SKB_TRANSFORM_TYPE_VERIFY` transform is used (see §7.9.9). 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 table explains the properties:

Property	Description
algorithm	<p>Verification algorithm to be used. The available verification algorithms are defined in the <code>SKB_SignatureAlgorithm</code> enumeration (see §7.11.4).</p> <p>Only the following algorithms are supported for verification:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_AES_128_CMAC</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_HMAC_SHA1</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_HMAC_SHA256</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_HMAC_SHA384</code></li> <li>▪ <code>SKB_SIGNATURE_ALGORITHM_HMAC_SHA512</code></li> </ul>

Property	Description
key	Pointer to the <code>SKB_SecureData</code> object, which contains the verification key. This key must not be released before the <code>SKB_Transform</code> object that uses it is released.
signature	Pointer to the data buffer containing the signature to be verified.
signature_size	Size of the signature in bytes.

### 7.10.9 SKB\_SelectBytesDerivationParameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_SELECT_BYTES` algorithm is used (see §7.9.17). It specifies whether odd or even bytes should be copied from the input, and how many bytes to copy.

The structure is declared as follows:

```
typedef struct {
    SKB_SelectBytesDerivationVariant variant;
    unsigned int output_size;
} SKB_SelectBytesDerivationParameters;
```

The following table explains the properties:

Property	Description
variant	Reference to a value of the <code>SKB_SelectBytesDerivationVariant</code> enumeration (see §7.11.14), which tells whether odd or even bytes should be selected.
output_size	Size of the output in bytes, which is the number of bytes copied from the input.

### 7.10.10 SKB\_CipherDerivationParameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_CIPHER` algorithm is used (see §7.9.17). 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 table explains the properties:

Property	Description
<code>cipher_algorithm</code>	<p>Cipher algorithm to be executed on the input data. This is a reference to the <code>SKB_CipherAlgorithm</code> enumeration (see §7.11.3).</p> <p>Currently, the <code>SKB_DERIVATION_ALGORITHM_CIPHER</code> algorithm supports only the following ciphers:</p> <ul style="list-style-type: none"> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_128_ECB</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_128_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_192_ECB</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_192_CBC</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_256_ECB</code></li> <li>▪ <code>SKB_CIPHER_ALGORITHM_AES_256_CBC</code></li> </ul>
<code>cipher_direction</code>	<p>Parameter that specifies whether the input data should be encrypted or decrypted. Available directions are defined in the <code>SKB_CipherDirection</code> enumeration (see §7.11.6).</p>
<code>cipher_flags</code>	<p>Optional flags for the cipher.</p> <p>Currently, the only defined flag is <code>SKB_CIPHER_FLAG_HIGH_SPEED</code>. This flag can be used only for the AES cipher when it is intended to be used with high throughput, for example when used for media content decryption.</p>
<code>cipher_parameters</code>	<p>Pointer to a structure that provides additional parameters for the cipher.</p> <p>Currently, this parameter must always be <code>NULL</code>.</p>
<code>cipher_key</code>	<p>Pointer to the <code>SKB_SecureData</code> object containing the encryption or decryption key.</p>
<code>iv</code>	<p>Pointer to the initialization vector.</p>
<code>iv_size</code>	<p>Size in bytes of the initialization vector.</p>

### 7.10.11 SKB\_Sha1DerivationParameters

This structure is used by the `SKB_SecureData_Derive` method (see §7.9.17) if the `SKB_DERIVATION_ALGORITHM_SHA_1` algorithm is used (see §3.10.4.1). The purpose of this structure is to specify how many times the SHA-1 algorithm should be executed on the source

`SKB_SecureData` object and how many bytes should be derived from the final hash value as a new `SKB_SecureData` object.

The structure is declared as follows:

```
typedef struct {
    unsigned int round_count;
    unsigned int output_size;
} SKB_Sha1DerivationParameters;
```

The following table explains the properties:

Property	Description
<code>round_count</code>	How many times the SHA-1 algorithm should be executed in a sequence.  0 is also a valid value. In this case, the SHA-1 value will not be calculated; the derived <code>SKB_SecureData</code> object will simply contain the first <code>output_size</code> bytes of the source <code>SKB_SecureData</code> object.
<code>output_size</code>	Number of bytes to be derived from the final output of the SHA-1 algorithm. For example, if <code>output_size</code> is 4, the first four bytes of the hash value will be derived as a new <code>SKB_SecureData</code> object.  The standard size of the SHA-1 output is 20 bytes. Hence, <code>output_size</code> cannot exceed 20.

### 7.10.12 SKB\_Sha256DerivationParameters

This structure is used by the `SKB_SecureData_Derive` method (see §7.9.17) if the `SKB_DERIVATION_ALGORITHM_SHA_256` algorithm is used (see §3.10.4.2). The purpose of this structure is to provide the two plain data buffers that should be prepended and appended to the source `SKB_SecureData` object before the SHA-256 algorithm is executed.

This structure may be omitted (provided as `NULL`). In that case, SKB will assume that there are no plain data buffers prepended or appended 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_Sha256DerivationParameters;
```

The following table explains the properties:



Property	Description
<code>plain1</code>	Pointer to a buffer of bytes that should be prepended to the source <code>SKB_SecureData</code> object before calculating the SHA-256 hash value.  This property can be <code>NULL</code> , in which case there will be no plain data prepended to the <code>SKB_SecureData</code> object.
<code>plain1_size</code>	Number of bytes in the <code>plain1</code> buffer.
<code>plain2</code>	Pointer to a buffer of bytes that should be appended to the source <code>SKB_SecureData</code> object before calculating the SHA-256 hash value.  This property can be <code>NULL</code> , in which case there will be no plain data appended to the <code>SKB_SecureData</code> object.
<code>plain2_size</code>	Number of bytes in the <code>plain2</code> buffer.

### 7.10.13 SKB\_Nist800108CounterCmacAes128Parameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128` derivation algorithm is used (see §7.9.17). The purpose of this structure is to specify the necessary input parameters. For more information on this derivation algorithm, see §3.10.6.

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_Nist800108CounterCmacAes128Parameters;
```

The following table explains the properties:

Property	Description
<code>label</code>	Pointer to the label, a binary buffer that identifies the purpose for the derived key, as defined by the <i>NIST Special Publication 800-108</i> .
<code>label_size</code>	Size of the label in bytes.
<code>context</code>	Pointer to the context, a binary buffer containing the information related to the derived key, as defined by the <i>NIST Special Publication 800-108</i> .

Property	Description
<code>context_size</code>	Size of the context in bytes.
<code>output_size</code>	Size of the derivation output in bytes. It cannot exceed 4096 bytes and must be a multiple of 16.

#### 7.10.14 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 (see §7.9.17). The purpose of this structure is to specify whether the output should be encoded in little-endian or big-endian. For more information on this derivation algorithm, see §3.10.8.

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 in big-endian. Otherwise, the output will be encoded in little-endian.

#### 7.10.15 SKB\_ShaAesDerivationParameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_SHA_AES` derivation algorithm is used (see §7.9.17). The purpose of this structure is to specify the necessary input parameters. For more information on this derivation algorithm, see §3.10.10.

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 table explains the properties:

Property	Description
<code>secure_p</code>	Pointer to the <code>SKB_SecureData</code> object containing the <code>secure_p</code> value.

Property	Description
plain_1	Pointer to the plain_1 buffer.  This property may be set to <code>NULL</code> . In that case, the simplified version of the derivation algorithm will be executed (see §3.10.10).
plain_1_size	Size of the plain_1 buffer. It must be 0 if plain_1 is set to <code>NULL</code> .
plain_2	Pointer to the plain_2 buffer, which must be 16 bytes long.

### 7.10.16 SKB\_OmaDrmKdf2DerivationParameters

This structure is used by the `SKB_SecureData_Derive` method if the `SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2` derivation algorithm is used (see §7.9.17). The purpose of this structure is to specify the necessary input parameters. For more information on this derivation algorithm, see §3.10.7.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* label;
    SKB_Size        label_size;
    SKB_Size        output_size;
} SKB_OmaDrmKdf2DerivationParameters;
```

The following table explains the properties:

Property	Description
label	Pointer to the buffer containing the <code>otherInfo</code> parameter as defined in the OMA DRM specification.
label_size	Size of the <code>label</code> buffer in bytes.
output_size	Size of the derivation output in bytes.

### 7.10.17 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 (see §7.9.17). The purpose of this structure is to specify the range of bytes (first byte and the number of bytes) that should be derived from one `SKB_SecureData` object into another `SKB_SecureData` object.

The structure is declared as follows:

```
typedef struct {
    unsigned int first;
    unsigned int size;
} SKB_SliceDerivationParameters;
```

The following table explains the properties:

Property	Description
<code>first</code>	Index of the first byte of the source <code>SKB_SecureData</code> object where the derived range starts. Bytes are numbered starting with 0.  If you are using the <code>SKB_DERIVATION_ALGORITHM_BLOCK_SLICE</code> algorithm, the value must be a multiple of 16.
<code>size</code>	Number of bytes to derive starting with the byte with offset <code>first</code> .  If you are using the <code>SKB_DERIVATION_ALGORITHM_BLOCK_SLICE</code> algorithm, the value must be a multiple of 16.

### 7.10.18 SKB\_EccDomainParameters

This structure defines domain parameters for a custom ECC curve, and therefore should be employed only when the `SKB_ECC_CURVE_CUSTOM` curve type of the `SKB_EccCurve` enumeration is used (see §7.11.10). Currently, custom ECC curves are supported only for the ECDSA, ECDH, and ECC key generation algorithms. For all other cases, this structure is not used.

The structure is declared as follows:

```
typedef struct {
    SKB_Size prime_bit_length;
    SKB_Size order_bit_length;
    const unsigned int* prime;
    const unsigned int* a;
    const unsigned int* gx;
    const unsigned int* gy;
    const unsigned int* order;
} SKB_EccDomainParameters;
```

The following table explains the properties:

Property	Description
<code>prime_bit_length</code>	Bit-length of the <code>prime</code> , <code>a</code> , <code>gx</code> , and <code>gy</code> domain parameters.
<code>order_bit_length</code>	Bit-length of the <code>order</code> domain parameter.
<code>prime</code>	Pointer to the prime modulo of the field.

Property	Description
a	Pointer to the constant from the equation $y^2 = x^3 + ax + b$ .
gx	Pointer to the X coordinate of the base point.
gy	Pointer to the Y coordinate of the base point.
order	Pointer to the order of the base point.

All domain parameters, except for `prime_bit_length` and `order_bit_length`, must be provided in protected form. To obtain the protected form of custom ECC domain parameters, use Custom ECC Tool as described in §5.1.

### 7.10.19 SKB\_AesWrapParameters

This structure provides a specific initialization vector to the AES algorithm when the `SKB_SecureData_Wrap` method is used (see §7.9.16). If this structure is not provided, the AES wrapping algorithm generates a random initialization vector.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* iv;
} SKB_AesWrapParameters;
```

`iv` is a pointer to the byte buffer containing the initialization vector.

### 7.10.20 SKB\_AesUnwrapParameters

A pointer to this structure can be passed to the `SKB_Engine_CreateDataFromWrapped` method (see §7.9.5) in case the CBC mode of the AES algorithm is used. This structure specifies the CBC padding type to be used. For information on available CBC padding types, see §8.2.3.

The structure is declared as follows:

```
typedef struct {
    SKB_CbcPadding padding;
} SKB_AesUnwrapParameters;
```

`padding` specifies the CBC padding type to be used. The available padding types are defined in the `SKB_CbcPadding` enumeration (see §7.11.13).

### 7.10.21 SKB\_RsaPssParameters

This structure provides additional parameters when the `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX` or `SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX` signature algorithm is used.

The structure is declared as follows:

```
typedef struct {
    const SKB_Byte* salt;
    SKB_Size salt_length;
} SKB_RsaPssParameters;
```

The following table describes the properties:

Property	Description
salt	Pointer to a byte buffer containing the salt value to be used. If this parameter is <code>NULL</code> , a random salt value with the length specified in the <code>salt_length</code> parameter will be generated.
salt_length	Length of the salt value in bytes. It must be equal or greater than 0 and must not exceed the hash function block size.

### 7.10.22 SKB\_EccParameters

This structure provides additional parameters when the ECC functions are used.

The structure is declared as follows:

```
typedef struct {
    SKB_EccCurve curve;
    SKB_EccDomainParameters* domain_parameters;
    const unsigned int* random_value;
} SKB_EccParameters;
```

The following table describes the properties:

Property	Description
curve	Specifies the ECC curve type to be used. The available curve types are defined in the <code>SKB_EccCurve</code> enumeration (see §7.11.10).
domain_parameters	Pointer to the <code>SKB_EccDomainParameters</code> structure, which provides domain parameters for a custom ECC curve (see §7.10.18). This parameter should be set only when the <code>SKB_ECC_CURVE_CUSTOM</code> 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.

Property	Description
random_value	<p>Property that allows you to provide a fixed random value to the ECDSA and ECDH algorithms.</p> <p>Typically, the value of this property should be <code>NULL</code>, in which case SKB uses an internally generated random value.</p> <p>However, you can also pass a fixed number to be used as the random value. The fixed number must be passed as an integer array containing the value in protected form. To obtain the protected form of a fixed random value, use Custom ECC Tool as described in §5.1.</p>

### 7.10.23 SKB\_PrimeDhParameters

This structure is required by the `SKB_Engine_CreateKeyAgreement` method when the classical DH algorithm (`SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH`) is selected. The structure supplies parameters necessary to execute the DH key agreement operation.

The structure is declared as follows:

```
typedef struct {
    SKB_PrimeDhLength    length;
    const SKB_Byte*     data;
    const unsigned int*  random_value;
} SKB_PrimeDhParameters;
```

The following table describes the properties:

Property	Description
length	Maximum bit-length of the DH prime P. The available values are defined in the <code>SKB_PrimeDhLength</code> enumeration (see §7.11.12).
data	Pointer to an integer array containing a combination of the prime P and generator G in protected form to be used by the DH algorithm. To obtain this protected data buffer, use Diffie-Hellman Tool as described in §5.2.
random_value	<p>Property that allows you to provide a fixed random value to the DH algorithm.</p> <p>Typically, the value of this property should be <code>NULL</code>, in which case SKB uses an internally generated random value.</p> <p>However, you can also pass a fixed number to be used as the random value. The fixed number must be passed as an integer array containing the value in protected form. To obtain the protected form of a fixed random value, use Diffie-Hellman Tool as described in §5.2.</p>

## 7.10.24 SKB\_RawBytesParameters

This structure is required by the `SKB_Engine_GenerateSecureData` method (see §7.9.8) to generate an `SKB_SecureData` object containing a protected 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` is the number of bytes to be generated.

## 7.11 Enumerations

This section describes various enumerations defined in the API.

### 7.11.1 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_RSA_PRIVATE_KEY,
    SKB_DATA_TYPE_ECC_PRIVATE_KEY
} SKB_DataType;
```

As shown, an `SKB_SecureData` object can contain raw bytes (for example, a DES or AES key), an RSA private key, or an ECC private key.

### 7.11.2 SKB\_DigestAlgorithm

This enumeration specifies the available digest algorithms, and is defined as follows:

```
typedef enum {
    SKB_DIGEST_ALGORITHM_SHA1,
    SKB_DIGEST_ALGORITHM_SHA256,
    SKB_DIGEST_ALGORITHM_SHA384,
    SKB_DIGEST_ALGORITHM_SHA512
} SKB_DigestAlgorithm;
```

### 7.11.3 SKB\_CipherAlgorithm

This enumeration specifies cryptographic algorithms that are used for encrypting and decrypting data.




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_RSA,
    SKB_CIPHER_ALGORITHM_RSA_1_5,
    SKB_CIPHER_ALGORITHM_RSA_OAEP,
    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_256_ECB,
    SKB_CIPHER_ALGORITHM_AES_256_CBC,
    SKB_CIPHER_ALGORITHM_AES_256_CTR,
    SKB_CIPHER_ALGORITHM_DES_ECB,
    SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB,
    SKB_CIPHER_ALGORITHM_NIST_AES,
    SKB_CIPHER_ALGORITHM_AES_CMLA,
    SKB_CIPHER_ALGORITHM_RSA_CMLA,
    SKB_CIPHER_ALGORITHM_XOR,
} SKB_CipherAlgorithm;
```

The following table explains the values:

Value	Description
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 <code>SKB_Engine_CreateDataFromWrapped</code> method to specify that the data to be loaded is in plain form (see §3.2).
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_RSA	1024-bit and 2048-bit RSA with no padding
SKB_CIPHER_ALGORITHM_RSA_1_5	1024-bit and 2048-bit RSA with PKCS#1 version 1.5 padding
SKB_CIPHER_ALGORITHM_RSA_OAEP	1024-bit and 2048-bit RSA with OAEP padding
SKB_CIPHER_ALGORITHM_ECC_ELGAMAL	ElGamal ECC

Value	Description
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_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_DES_ECB	DES in the ECB mode
SKB_CIPHER_ALGORITHM_TRIPLE_DES_ECB	Triple DES in the ECB mode
SKB_CIPHER_ALGORITHM_NIST_AES	AES key unwrapping algorithm defined by NIST. This cipher is supported only by the <code>SKB_Engine_CreateDataFromWrapped</code> method (see §7.9.5).
SKB_CIPHER_ALGORITHM_AES_CMLA	CMLA AES unwrapping defined by the <i>CMLA Technical Specification</i>
SKB_CIPHER_ALGORITHM_RSA_CMLA	CMLA RSA unwrapping defined by the <i>CMLA Technical Specification</i>
SKB_CIPHER_ALGORITHM_XOR	<p>Wrapping and unwrapping using XOR:</p> <ul style="list-style-type: none"> <li>▪ If the <code>SKB_SecureData_Wrap</code> function is used (see §7.9.16), the key to be wrapped is XOR-ed with the wrapping key.</li> <li>▪ If the <code>SKB_Engine_CreateDataFromWrapped</code> function is used (see §7.9.5), the wrapped buffer is XOR-ed with the unwrapping key.</li> </ul> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p> In both cases, the two XOR-ed buffers must be of equal size.</p> </div>

### 7.11.4 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_HMAC_SHA1,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA256,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA384,
    SKB_SIGNATURE_ALGORITHM_HMAC_SHA512,
    SKB_SIGNATURE_ALGORITHM_RSA,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA1,
    SKB_SIGNATURE_ALGORITHM_RSA_SHA256,
    SKB_SIGNATURE_ALGORITHM_ECDSA,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1,
    SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256,
    SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX,
} SKB_SignatureAlgorithm;
```

The following table explains the values:

Value	Description
<code>SKB_SIGNATURE_ALGORITHM_AES_128_CMAC</code>	128-bit AES-CMAC (based on OMAC1)
<code>SKB_SIGNATURE_ALGORITHM_HMAC_SHA1</code>	HMAC using SHA-1 with up to 64-byte keys
<code>SKB_SIGNATURE_ALGORITHM_HMAC_SHA256</code>	HMAC using SHA-256 with up to 64-byte keys
<code>SKB_SIGNATURE_ALGORITHM_HMAC_SHA384</code>	HMAC using SHA-384 with up to 64-byte keys
<code>SKB_SIGNATURE_ALGORITHM_HMAC_SHA512</code>	HMAC using SHA-512 with up to 64-byte keys
<code>SKB_SIGNATURE_ALGORITHM_RSA</code>	1024-bit and 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 without a hash function (can only be executed on plain input, which is a digest of some hash function)
<code>SKB_SIGNATURE_ALGORITHM_RSA_SHA1</code>	1024-bit and 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using SHA-1 as the hash function

Value	Description
<code>SKB_SIGNATURE_ALGORITHM_RSA_SHA256</code>	1024-bit and 2048-bit RSA signature algorithms standardized in version 1.5 of PKCS#1 using SHA-256 as the hash function
<code>SKB_SIGNATURE_ALGORITHM_ECDSA</code>	ECDSA with either standard or custom curves (can only be executed on plain input, which is a digest of some hash function)
<code>SKB_SIGNATURE_ALGORITHM_ECDSA_SHA1</code>	ECDSA with either standard or custom curves using SHA-1 as the hash function
<code>SKB_SIGNATURE_ALGORITHM_ECDSA_SHA256</code>	ECDSA with either standard or custom curves using SHA-256 as the hash function
<code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1</code>	1024-bit and 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using SHA-1 as the hash function.  Salt length is fixed at 20 bytes. The mask generation function is using SHA-1.
<code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1_EX</code>	Same as <code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA1</code> but allows specifying the salt value and length.
<code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256</code>	1024-bit and 2048-bit RSA signature algorithms based on the Probabilistic Signature Scheme using SHA-256 as the hash function.  Salt length is fixed at 32 bytes. The mask generation function is using SHA-256.
<code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256_EX</code>	Same as <code>SKB_SIGNATURE_ALGORITHM_RSA_PSS_SHA256</code> but allows specifying the salt value and length.

### 7.11.5 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 (see §7.9.17).

The enumeration is defined as follows:

```
typedef enum {
    SKB_DERIVATION_ALGORITHM_SLICE,
    SKB_DERIVATION_ALGORITHM_BLOCK_SLICE,
    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_REVERSE_BYTES,
    SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128,
    SKB_DERIVATION_ALGORITHM_OMA_DRM_KDF2,
    SKB_DERIVATION_ALGORITHM_RAW_BYTES_FROM_ECC_PRIVATE,
    SKB_DERIVATION_ALGORITHM_CMLA_KDF,
    SKB_DERIVATION_ALGORITHM_SHA_AES,
} SKB_DerivationAlgorithm;
```

The following list explains the values:

- `SKB_DERIVATION_ALGORITHM_SLICE`: Derives a new `SKB_SecureData` object as a substring of bytes of another `SKB_SecureData` object. For more information, see §3.10.1.
- `SKB_DERIVATION_ALGORITHM_BLOCK_SLICE`: Same as the `SKB_DERIVATION_ALGORITHM_SLICE` algorithm, but it requires the index of the first byte and the number of bytes in the substring to be multiples of 16. For more information, see §3.10.1.
- `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. For more information, see §3.10.2.
- `SKB_DERIVATION_ALGORITHM_CIPHER`: Derives a new `SKB_SecureData` object from the input `SKB_SecureData` object by encrypting or decrypting it with another key. For more information, see §3.10.3.
- `SKB_DERIVATION_ALGORITHM_SHA_1`: Obtains a hash value from the referenced `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. For more information, see §3.10.4.1.
- `SKB_DERIVATION_ALGORITHM_SHA_256`: Obtains a SHA-256 hash value from a buffer that contains a `SKB_SecureData` object, prefixed and suffixed with plain data, and stores the output as a new `SKB_SecureData` object. For more information, see §3.10.4.2.
- `SKB_DERIVATION_ALGORITHM_SHA_384`: Obtains a hash value from the referenced `SKB_SecureData` object by executing SHA-384 one time and stores the entire 48-byte output as a new `SKB_SecureData` object. For more information, see §3.10.4.3.
- `SKB_DERIVATION_ALGORITHM_REVERSE_BYTES`: Derives a new `SKB_SecureData` object where the order of bytes is reversed. You can use this derivation type to convert little-endian data buffers to big-endian and vice versa. For more information, see §3.10.5.
- `SKB_DERIVATION_ALGORITHM_NIST_800_108_COUNTER_CMAC_AES128`: Derives a new `SKB_SecureData` object according to the key derivation function specified in the *NIST Special*

*Publication 800-108*, using 128-bit AES-CMAC as the pseudorandom function in counter mode. For more information, see §3.10.6.

- `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. For more information, see §3.10.7.
- `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 with the type `SKB_DATA_TYPE_ECC_PRIVATE_KEY`. For more information, see §3.10.8.
- `SKB_DERIVATION_ALGORITHM_CMLA_KDF`: Derives a new `SKB_SecureData` object according to the key derivation function defined in the *CMLA Technical Specification*. For more information, see §3.10.9.
- `SKB_DERIVATION_ALGORITHM_SHA_AES`: Derives a new `SKB_SecureData` object using an algorithm described in 3.10.10.

### 7.11.6 SKB\_CipherDirection

This enumeration specifies the possible directions (encryption or decryption) for the `SKB_Cipher` object.

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

The enumeration is defined as follows:

```
typedef enum {
    SKB_CIPHER_DIRECTION_ENCRYPT,
    SKB_CIPHER_DIRECTION_DECRYPT
} SKB_CipherDirection;
```

### 7.11.7 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_ECC_BINARY
} SKB_DataFormat;
```

The following table explains the values:

Value	Description
<code>SKB_DATA_FORMAT_RAW</code>	Buffer of raw bytes (for example, a DES or AES key)

Value	Description
SKB_DATA_FORMAT_PKCS8	RSA private key stored according to the PKCS#8 standard
SKB_DATA_FORMAT_ECC_BINARY	ECC private key stored in the format described in §8.6

### 7.11.8 SKB\_TransformType

This enumeration specifies the available transform types used by the `SKB_Engine_CreateTransform` method to create `SKB_Transform` objects (see §7.9.9).

The enumeration is defined as follows:

```
typedef enum {
    SKB_TRANSFORM_TYPE_DIGEST,
    SKB_TRANSFORM_TYPE_SIGN,
    SKB_TRANSFORM_TYPE_VERIFY
} SKB_TransformType;
```

The following table explains the values:

Value	Description
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.

### 7.11.9 SKB\_ExportTarget

This enumeration specifies the various export types used by the `SKB_SecureData_Export` method (see §7.9.15).

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;
```

Currently, only the `SKB_EXPORT_TARGET_PERSISTENT` type is supported. With this type, the exported data can be reloaded in an engine even after a complete reboot of the system hosting the engine.

### 7.11.10 SKB\_EccCurve

This enumeration specifies the available ECC curve types. These values must be provided when the ElGamal ECC algorithms are used.

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 table explains the values:

Value	Description
SKB_ECC_CURVE_SECP_R1_160	160-bit prime curve recommended by SECG, SECP R1
SKB_ECC_CURVE_NIST_192	192-bit prime curve recommended by NIST (same as 192-bit SECG, SECP R1)
SKB_ECC_CURVE_NIST_224	224-bit prime curve recommended by NIST (same as 224-bit SECG, SECP R1)
SKB_ECC_CURVE_NIST_256	256-bit prime curve recommended by NIST (same as 256-bit SECG, SECP R1)
SKB_ECC_CURVE_NIST_384	384-bit prime curve recommended by NIST (same as 384-bit SECG, SECP R1). Currently, this curve type is supported only for ECDSA, ECDH, and key generation, but not for decrypting and unwrapping.
SKB_ECC_CURVE_NIST_521	521-bit prime curve recommended by NIST (same as 521-bit SECG, SECP R1). 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 domain parameters. Currently, this curve type is supported only for ECDSA, ECDH, and key generation, but not for decrypting and unwrapping.



### 7.11.11 SKB\_KeyAgreementAlgorithm

This enumeration specifies the available key agreement algorithms used by the `SKB_Engine_CreateKeyAgreement` method to create `SKB_KeyAgreement` objects (see §7.9.11).

The enumeration is defined as follows:

```
typedef enum {
    SKB_KEY_AGREEMENT_ALGORITHM_ECDH,
    SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH
} SKB_KeyAgreementAlgorithm;
```

The following table explains the values:

Value	Description
<code>SKB_KEY_AGREEMENT_ALGORITHM_ECDH</code>	Elliptic curve Diffie-Hellman
<code>SKB_KEY_AGREEMENT_ALGORITHM_PRIME_DH</code>	Classical Diffie-Hellman with protected prime P and generator G

### 7.11.12 SKB\_PrimeDhLength

This enumeration specifies the available maximum bit-lengths of prime P for the classical DH key agreement algorithm. The values of this enumeration are referenced by the `length` parameter of the `SKB_PrimeDhParameters` structure (see §7.10.23).

The enumeration is defined as follows:

```
typedef enum {
    SKB_PRIME_DH_LENGTH_1024
} SKB_PrimeDhLength;
```

The value `SKB_PRIME_DH_LENGTH_1024` specifies that the maximum bit-length of prime P is 1024 bits.

### 7.11.13 SKB\_CbcPadding

This enumeration specifies the CBC mode types that can be referenced by the `SKB_AesUnwrapParameters` structure (see §7.10.20).

The enumeration is defined as follows:

```
typedef enum {
    SKB_CBC_PADDING_TYPE_NONE,
    SKB_CBC_PADDING_TYPE_XMLENC
} SKB_CbcPadding;
```

The following table explains the values:

Value	Description
SKB_CBC_PADDING_TYPE_NONE	CBC mode with no padding (see §8.2.3.1)
SKB_CBC_PADDING_TYPE_XMLENC	CBC mode with the XML encryption padding (§8.2.3.2)

#### 7.11.14 SKB\_SelectBytesDerivationVariant

This enumeration is used by the `SKB_SelectBytesDerivationParameters` structure (see §7.10.9) to specify whether odd or even bytes should be selected.

The enumeration is defined as follows:

```
typedef enum {
    SKB_SELECT_BYTES_DERIVATION_ODD_BYTES,
    SKB_SELECT_BYTES_DERIVATION_EVEN_BYTES,
} SKB_SelectBytesDerivationVariant;
```

The following table explains the values:

Value	Description
SKB_SELECT_BYTES_DERIVATION_ODD_BYTES	Odd bytes should be selected.
SKB_SELECT_BYTES_DERIVATION_EVEN_BYTES	Even bytes should be selected.

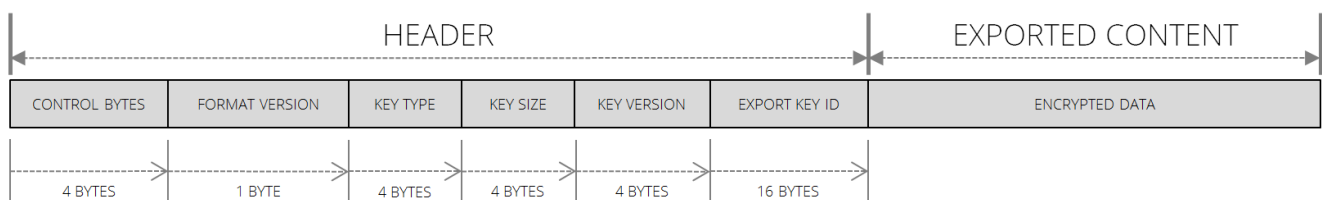
## 8 Data Formats

This is a reference chapter describing various data formats used in SKB.

### 8.1 Export Data Format

Data exported from SKB is a binary buffer that consists of a header and an encrypted content. The header can provide valuable information, especially if you are dealing with several SKB packages with different export keys, or if you are employing the one-way data upgrade deployment (see §3.7).

The following diagram shows the format of exported data.



Export data format

The following table explains the components of the header:

Component	Description
Control bytes	Random bytes with specific properties that identify data exported by SKB.
Format version	Version of the export format. Currently, it is always 02.
Key type	Type of the exported key. The following values are used: <ul style="list-style-type: none"> <li>00 identifies raw bytes (for example, an AES or DES key).</li> <li>01 identifies an ECC key.</li> <li>02 identifies an RSA key.</li> </ul>
Key size	Size of the exported key.
Key version	Key version in the one-way data upgrade scheme described in §3.7.

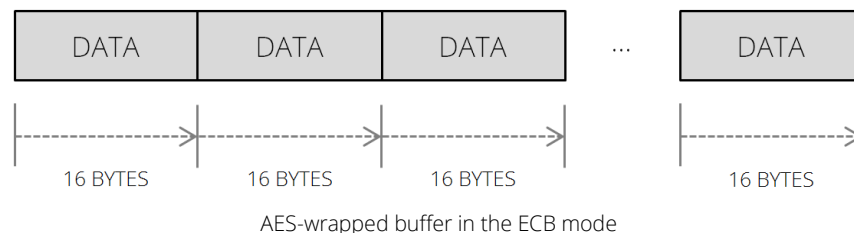
Component	Description
Export key ID	<p>Identifier of the export key that was used in exporting the data. An SKB instance that needs to import this data has to have the same export key (with the same identifier).</p> <p>You can find out the identifier of the export key of the current SKB instance using one of the following approaches:</p> <ul style="list-style-type: none"> <li>▪ Look into the <code>export.id</code> file delivered with the SKB package (see §1.5).</li> <li>▪ Call the <code>SKB_Engine_GetInfo</code> method and read the value of the <code>export_guid</code> property (see §7.9.4).</li> </ul>

## 8.2 AES-Wrapped Data Buffer

This section describes the format of an encrypted data buffer (raw bytes or RSA private key) that is either to be passed to the AES unwrapping algorithm (§3.1), or is the output of the AES wrapping algorithm (§3.3). Different modes of operation are described in separate subsections. In all modes, big-endian encoding is used.

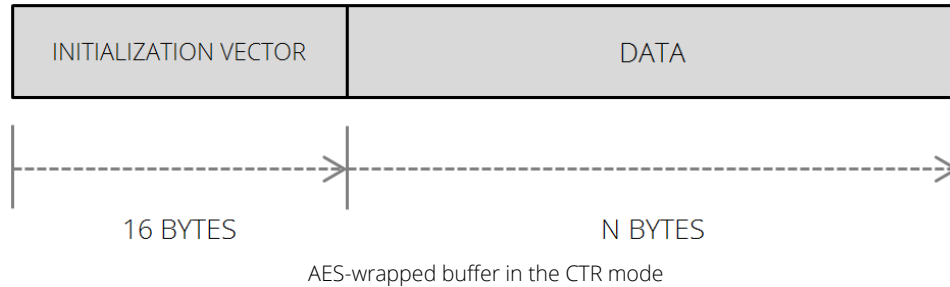
### 8.2.1 ECB Mode

In ECB mode, the size of the wrapped data buffer is an exact multiple of 16 bytes (block size for AES).



### 8.2.2 CTR Mode

In CTR mode, the wrapped data buffer begins with the initialization vector, which is 16 bytes, followed by a data buffer of N bytes. N is an arbitrary number, not necessarily a multiple of 16 (block size for AES).



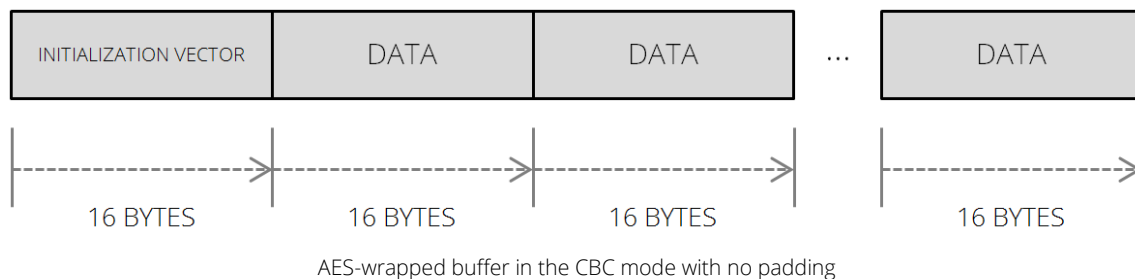
## 8.2.3 CBC Mode

In SKB, two CBC types are used — with no padding, and with XML encryption padding. In both cases, the wrapped data buffer begins with the initialization vector, which is 16 bytes, followed by a data buffer that is a multiple of 16 bytes (block size for AES).

The following subsections describe the two CBC mode types available.

### 8.2.3.1 No Padding

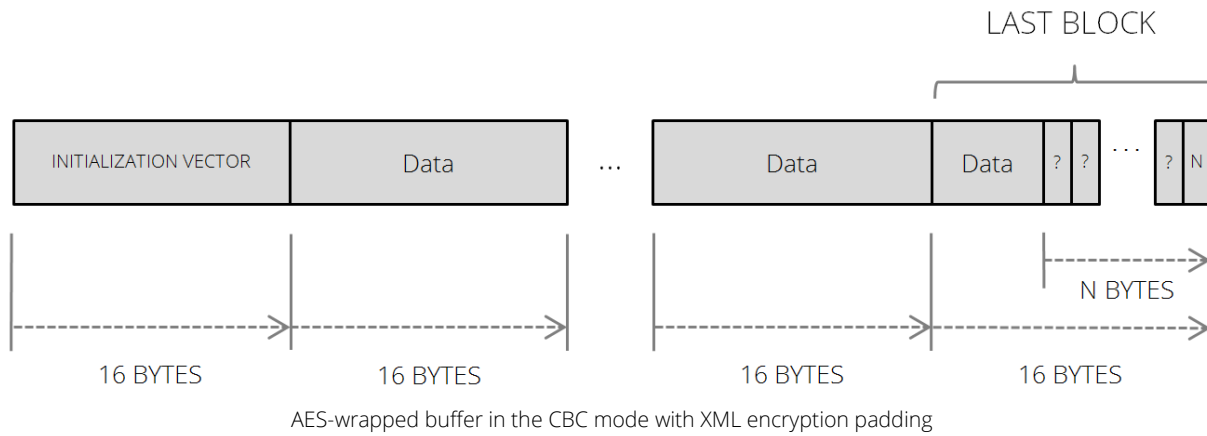
If no CBC padding is used, it is assumed that the size of the encrypted data buffer is an exact multiple of 16 bytes, and nothing is suffixed to the end of the buffer.



**⚠** This CBC type cannot be used to unwrap RSA keys. If you are unwrapping ECC keys, this CBC type can only unwrap keys of the 256-bit and 384-bit curves recommended by NIST. Other ECC curve types are not supported.

### 8.2.3.2 XML Encryption Padding

SKB supports the CBC mode with padding conventions of the standard XML encryption, which is described in <http://www.w3.org/TR/xmlenc-core/>. 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 sufficing 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.



In the preceding diagram, N is the number of bytes added to the last block. If the message size happens to be an exact multiple of 16 bytes, an additional block is added, in which the contents are arbitrary, but the last byte must contain the number 16.

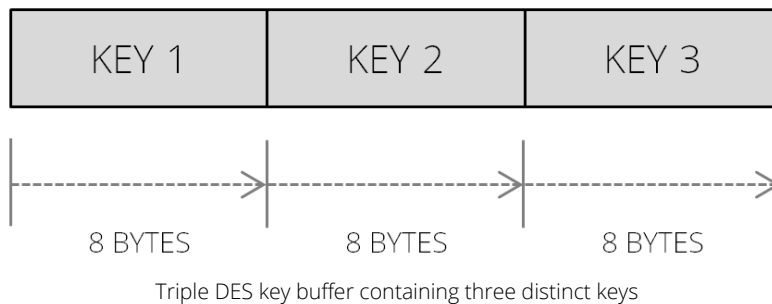
### 8.3 Key Format for the Triple DES Cipher

SKB supports two keying options for the Triple DES cipher:

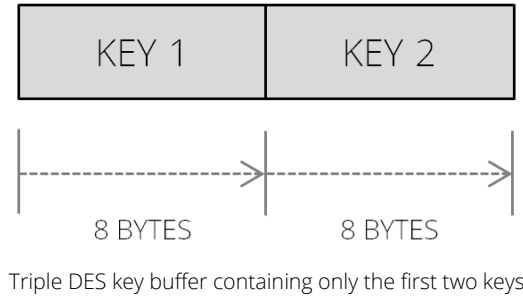
- All three keys are distinct.
- Key 1 and key 2 are distinct, and key 3 is identical to key 1.

In both cases, 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.



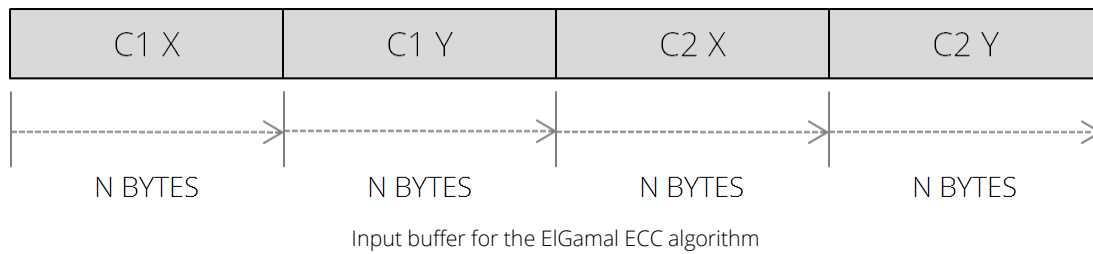
If the buffer is 128 bits long, SKB assumes the keys are provided in the following format.



In the latter case, it is assumed that key 3 is identical to key 1.

## 8.4 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 two points on an ECC curve using the following format:



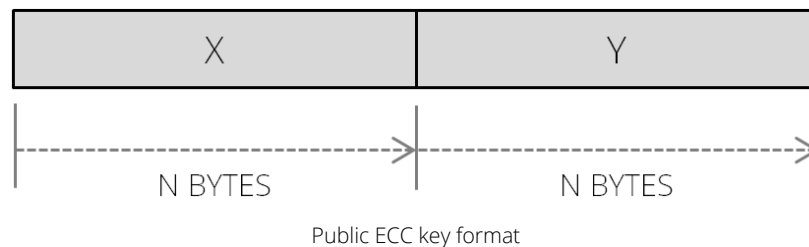
C1 X and C1 Y are the X and Y coordinates of the encrypted ciphertext 1, and C2 X and C2 Y are the X and Y coordinates of ciphertext 2 using the big-endian notation. N is the number of bytes used to store each coordinate, calculated as follows:

$$N = (L+7) / 8$$

where L is the length of the curve in bits.

## 8.5 Public ECC Key

SKB stores public ECC keys using the following format:

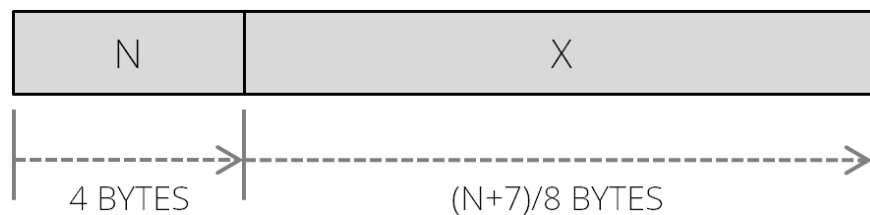


X and Y are the coordinates of the public key encoded using the big-endian notation, and N is the number of bytes used to store each coordinate. N depends on the ECC curve used as follows:

- `SKB_ECC_CURVE_SECP_R1_160`: 20 bytes
- `SKB_ECC_CURVE_NIST_192`: 24 bytes
- `SKB_ECC_CURVE_NIST_224`: 28 bytes
- `SKB_ECC_CURVE_NIST_256`: 32 bytes
- `SKB_ECC_CURVE_NIST_384`: 48 bytes
- `SKB_ECC_CURVE_NIST_521`: 66 bytes
- `SKB_ECC_CURVE_CUSTOM`: Specified bit-length of the prime domain parameter plus 7 divided by 8

## 8.6 Private ECC Key

SKB stores private ECC keys using the following format (this corresponds to the `SKB_DATA_FORMAT_ECC_BINARY` value of the `SKB_DataFormat` structure):

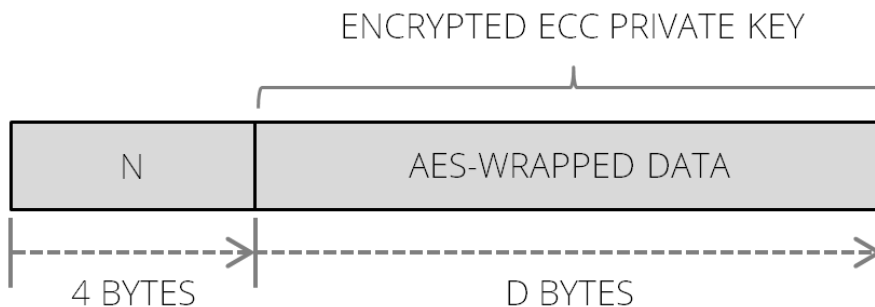


ECC private key format

N is the bit-length of the ECC curve. X is an X coordinate containing the ECC key. Both parameters are encoded in big-endian.

## 8.7 AES-Wrapped Private ECC Key

If you are unwrapping an AES-wrapped ECC private key, the input buffer to the unwrapping algorithm must have the following format (all data must be encoded in big-endian):



AES-wrapped ECC private key

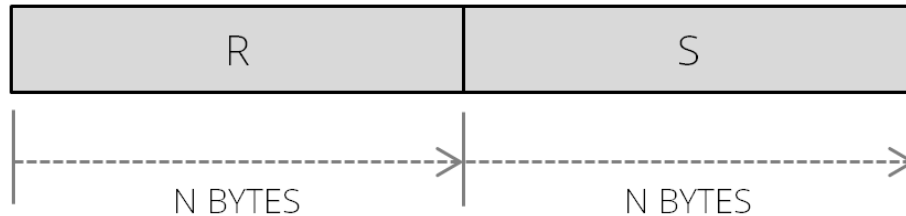
The first 4 bytes must contain the number N in plain, which is the bit-length of the ECC curve used. The rest of the buffer is AES-wrapped data, containing the wrapped ECC private key and possibly some padding bytes. The AES-wrapped portion of the buffer must be formatted as described in §8.2.



Once the AES-wrapped content is decrypted, the first  $(N+7)/8$  bytes are taken as the actual unwrapped ECC private key.

## 8.8 ECDSA Output

The output of the ECDSA algorithm is the following buffer of bytes:



ECDSA output format

R and S are the two parameters of a signature used in the ECDSA algorithm encoded using the big-endian notation, and N depends on the ECC curve used as follows:

- SKB\_ECC\_CURVE\_SECP\_R1\_160: 21 bytes
- SKB\_ECC\_CURVE\_NIST\_192: 24 bytes
- SKB\_ECC\_CURVE\_NIST\_224: 28 bytes
- SKB\_ECC\_CURVE\_NIST\_256: 32 bytes
- SKB\_ECC\_CURVE\_NIST\_384: 48 bytes
- SKB\_ECC\_CURVE\_NIST\_521: 66 bytes
- SKB\_ECC\_CURVE\_CUSTOM: specified bit-length of the order domain parameter plus 7 divided by 8
-