

PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0

Committee Specification Draft 01 /  
Public Review Draft 01

29 May 2019

This version:

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.docx> (Authoritative)

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.html>

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.pdf>

Previous version:

N/A

Latest version:

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.docx> (Authoritative)

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.html>

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.pdf>

Technical Committee:

[OASIS PKCS 11 TC](https://www.oasis-open.org/committees/pkcs11/)

Chairs:

Tony Cox ([tony.cox@cryptsoft.com](mailto:tony.cox@cryptsoft.com)), [Cryptsoft Pty Ltd](https://cryptsoft.com/)

Robert Relyea ([rrelyea@redhat.com](mailto:rrelyea@redhat.com)), [Red Hat](http://www.redhat.com)

Editors:

Chris Zimman ([chris@wmpp.com](mailto:chris@wmpp.com)), Individual

Dieter Bong ([dieter.bong@utimaco.com](mailto:dieter.bong@utimaco.com)), [Utimaco IS GmbH](https://hsm.utimaco.com/)

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

* PKCS #11 header files:   
  <https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/include/pkcs11-v3.0/>

Related work:

This specification replaces or supersedes:

* *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40*. Edited by Susan Gleeson, Chris Zimman, Robert Griffin, and Tim Hudson. Latest version. <http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/pkcs11-curr-v2.40.html>.

This specification is related to:

* *PKCS #11 Cryptographic Token Interface Profiles Version 3.0.* Edited by Tim Hudson. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.0/pkcs11-profiles-v3.0.html>.
* *PKCS #11 Cryptographic Token Interface Base Specification Version 3.0.* Edited by Chris Zimman and Dieter Bong. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html>.
* *PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 3.0*. Edited by Chris Zimman and Dieter Bong. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.html>.

Abstract:

This document defines data types, functions and other basic components of the PKCS #11 Cryptoki interface.

Status:

This document was last revised or approved by the OASIS PKCS 11 TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at <https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11#technical>.

TC members should send comments on this document to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "[Send A Comment](https://www.oasis-open.org/committees/comments/index.php?wg_abbrev=pkcs11)" button on the TC's web page at <https://www.oasis-open.org/committees/pkcs11/>.

This specification is provided under the [RF on RAND Terms](https://www.oasis-open.org/policies-guidelines/ipr#RF-on-RAND-Mode) Mode of the [OASIS IPR Policy](https://www.oasis-open.org/policies-guidelines/ipr), the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (<https://www.oasis-open.org/committees/pkcs11/ipr.php>).

Note that any machine-readable content ([Computer Language Definitions](https://www.oasis-open.org/policies-guidelines/tc-process#wpComponentsCompLang)) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:

When referencing this specification the following citation format should be used:

[PKCS11-Current-v3.0]

*PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0*. Edited by Chris Zimman and Dieter Bong. 29 May 2019. OASIS Committee Specification Draft 01 / Public Review Draft 01. <https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.html>. Latest version: <https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.html>.

Notices

Copyright © OASIS Open 2019. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](https://www.oasis-open.org/policies-guidelines/ipr) may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of [OASIS](https://www.oasis-open.org/), the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see <https://www.oasis-open.org/policies-guidelines/trademark> for above guidance.

Table of Contents

[1 Introduction 16](#_Toc10560811)

[1.1 IPR Policy 16](#_Toc10560812)

[1.2 Terminology 16](#_Toc10560813)

[1.3 Definitions 16](#_Toc10560814)

[1.4 Normative References 18](#_Toc10560815)

[1.5 Non-Normative References 19](#_Toc10560816)

[2 Mechanisms 22](#_Toc10560817)

[2.1 RSA 22](#_Toc10560818)

[2.1.1 Definitions 23](#_Toc10560819)

[2.1.2 RSA public key objects 24](#_Toc10560820)

[2.1.3 RSA private key objects 25](#_Toc10560821)

[2.1.4 PKCS #1 RSA key pair generation 26](#_Toc10560822)

[2.1.5 X9.31 RSA key pair generation 27](#_Toc10560823)

[2.1.6 PKCS #1 v1.5 RSA 27](#_Toc10560824)

[2.1.7 PKCS #1 RSA OAEP mechanism parameters 28](#_Toc10560825)

[2.1.8 PKCS #1 RSA OAEP 30](#_Toc10560826)

[2.1.9 PKCS #1 RSA PSS mechanism parameters 30](#_Toc10560827)

[2.1.10 PKCS #1 RSA PSS 31](#_Toc10560828)

[2.1.11 ISO/IEC 9796 RSA 31](#_Toc10560829)

[2.1.12 X.509 (raw) RSA 32](#_Toc10560830)

[2.1.13 ANSI X9.31 RSA 33](#_Toc10560831)

[2.1.14 PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPE-MD 128 or RIPE-MD 160 34](#_Toc10560832)

[2.1.15 PKCS #1 v1.5 RSA signature with SHA-224 34](#_Toc10560833)

[2.1.16 PKCS #1 RSA PSS signature with SHA-224 34](#_Toc10560834)

[2.1.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512 34](#_Toc10560835)

[2.1.18 PKCS #1 v1.5 RSA signature with SHA3 35](#_Toc10560836)

[2.1.19 PKCS #1 RSA PSS signature with SHA3 35](#_Toc10560837)

[2.1.20 ANSI X9.31 RSA signature with SHA-1 35](#_Toc10560838)

[2.1.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA 36](#_Toc10560839)

[2.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP 36](#_Toc10560840)

[2.1.23 RSA AES KEY WRAP 37](#_Toc10560841)

[2.1.24 RSA AES KEY WRAP mechanism parameters 38](#_Toc10560842)

[2.1.25 FIPS 186-4 38](#_Toc10560843)

[2.2 DSA 38](#_Toc10560844)

[2.2.1 Definitions 39](#_Toc10560845)

[2.2.2 DSA public key objects 40](#_Toc10560846)

[2.2.3 DSA Key Restrictions 41](#_Toc10560847)

[2.2.4 DSA private key objects 41](#_Toc10560848)

[2.2.5 DSA domain parameter objects 42](#_Toc10560849)

[2.2.6 DSA key pair generation 43](#_Toc10560850)

[2.2.7 DSA domain parameter generation 43](#_Toc10560851)

[2.2.8 DSA probabilistic domain parameter generation 43](#_Toc10560852)

[2.2.9 DSA Shawe-Taylor domain parameter generation 44](#_Toc10560853)

[2.2.10 DSA base domain parameter generation 44](#_Toc10560854)

[2.2.11 DSA without hashing 44](#_Toc10560855)

[2.2.12 DSA with SHA-1 45](#_Toc10560856)

[2.2.13 FIPS 186-4 45](#_Toc10560857)

[2.2.14 DSA with SHA-224 45](#_Toc10560858)

[2.2.15 DSA with SHA-256 46](#_Toc10560859)

[2.2.16 DSA with SHA-384 46](#_Toc10560860)

[2.2.17 DSA with SHA-512 47](#_Toc10560861)

[2.2.18 DSA with SHA3-224 47](#_Toc10560862)

[2.2.19 DSA with SHA3-256 48](#_Toc10560863)

[2.2.20 DSA with SHA3-384 48](#_Toc10560864)

[2.2.21 DSA with SHA3-512 48](#_Toc10560865)

[2.3 Elliptic Curve 49](#_Toc10560866)

[2.3.1 EC Signatures 51](#_Toc10560867)

[2.3.2 Definitions 51](#_Toc10560868)

[2.3.3 ECDSA public key objects 52](#_Toc10560869)

[2.3.4 Elliptic curve private key objects 53](#_Toc10560870)

[2.3.5 Edwards Elliptic curve public key objects 54](#_Toc10560871)

[2.3.6 Edwards Elliptic curve private key objects 55](#_Toc10560872)

[2.3.7 Montgomery Elliptic curve public key objects 56](#_Toc10560873)

[2.3.8 Montgomery Elliptic curve private key objects 57](#_Toc10560874)

[2.3.9 Elliptic curve key pair generation 58](#_Toc10560875)

[2.3.10 Edwards Elliptic curve key pair generation 59](#_Toc10560876)

[2.3.11 Montgomery Elliptic curve key pair generation 59](#_Toc10560877)

[2.3.12 ECDSA without hashing 60](#_Toc10560878)

[2.3.13 ECDSA with hashing 60](#_Toc10560879)

[2.3.14 EdDSA 61](#_Toc10560880)

[2.3.15 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK\_MECHANISM\_INFO structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 8032and RFC 8410 only define curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.XEdDSA 61](#_Toc10560881)

[2.3.16 EC mechanism parameters 62](#_Toc10560882)

[2.3.17 Elliptic curve Diffie-Hellman key derivation 67](#_Toc10560883)

[2.3.18 Elliptic curve Diffie-Hellman with cofactor key derivation 67](#_Toc10560884)

[2.3.19 Elliptic curve Menezes-Qu-Vanstone key derivation 68](#_Toc10560885)

[2.3.20 ECDH AES KEY WRAP 69](#_Toc10560886)

[2.3.21 ECDH AES KEY WRAP mechanism parameters 70](#_Toc10560887)

[2.3.22 FIPS 186-4 71](#_Toc10560888)

[2.4 Diffie-Hellman 71](#_Toc10560889)

[2.4.1 Definitions 71](#_Toc10560890)

[2.4.2 Diffie-Hellman public key objects 72](#_Toc10560891)

[2.4.3 X9.42 Diffie-Hellman public key objects 72](#_Toc10560892)

[2.4.4 Diffie-Hellman private key objects 73](#_Toc10560893)

[2.4.5 X9.42 Diffie-Hellman private key objects 74](#_Toc10560894)

[2.4.6 Diffie-Hellman domain parameter objects 75](#_Toc10560895)

[2.4.7 X9.42 Diffie-Hellman domain parameters objects 76](#_Toc10560896)

[2.4.8 PKCS #3 Diffie-Hellman key pair generation 76](#_Toc10560897)

[2.4.9 PKCS #3 Diffie-Hellman domain parameter generation 77](#_Toc10560898)

[2.4.10 PKCS #3 Diffie-Hellman key derivation 77](#_Toc10560899)

[2.4.11 X9.42 Diffie-Hellman mechanism parameters 78](#_Toc10560900)

[2.4.12 X9.42 Diffie-Hellman key pair generation 81](#_Toc10560901)

[2.4.13 X9.42 Diffie-Hellman domain parameter generation 81](#_Toc10560902)

[2.4.14 X9.42 Diffie-Hellman key derivation 81](#_Toc10560903)

[2.4.15 X9.42 Diffie-Hellman hybrid key derivation 82](#_Toc10560904)

[2.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation 82](#_Toc10560905)

[2.5 Extended Triple Diffie-Hellman (x3dh) 83](#_Toc10560906)

[2.5.1 Definitions 83](#_Toc10560907)

[2.5.2 Extended Triple Diffie-Hellman key objects 83](#_Toc10560908)

[2.5.3 Initiating an Extended Triple Diffie-Hellman key exchange 84](#_Toc10560909)

[2.5.4 Responding to an Extended Triple Diffie-Hellman key exchange 84](#_Toc10560910)

[2.5.5 Extended Triple Diffie-Hellman parameters 85](#_Toc10560911)

[2.6 Double Ratchet 86](#_Toc10560912)

[2.6.1 Definitions 86](#_Toc10560913)

[2.6.2 Double Ratchet secret key objects 86](#_Toc10560914)

[2.6.3 Double Ratchet key derivation 87](#_Toc10560915)

[2.6.4 Double Ratchet Encryption mechanism 89](#_Toc10560916)

[2.6.5 Double Ratchet parameters 89](#_Toc10560917)

[2.7 Wrapping/unwrapping private keys 89](#_Toc10560918)

[2.8 Generic secret key 92](#_Toc10560919)

[2.8.1 Definitions 92](#_Toc10560920)

[2.8.2 Generic secret key objects 92](#_Toc10560921)

[2.8.3 Generic secret key generation 93](#_Toc10560922)

[2.9 HMAC mechanisms 93](#_Toc10560923)

[2.9.1 General block cipher mechanism parameters 93](#_Toc10560924)

[2.10 AES 93](#_Toc10560925)

[2.10.1 Definitions 94](#_Toc10560926)

[2.10.2 AES secret key objects 94](#_Toc10560927)

[2.10.3 AES key generation 95](#_Toc10560928)

[2.10.4 AES-ECB 95](#_Toc10560929)

[2.10.5 AES-CBC 96](#_Toc10560930)

[2.10.6 AES-CBC with PKCS padding 97](#_Toc10560931)

[2.10.7 AES-OFB 97](#_Toc10560932)

[2.10.8 AES-CFB 98](#_Toc10560933)

[2.10.9 General-length AES-MAC 98](#_Toc10560934)

[2.10.10 AES-MAC 98](#_Toc10560935)

[2.10.11 AES-XCBC-MAC 99](#_Toc10560936)

[2.10.12 AES-XCBC-MAC-96 99](#_Toc10560937)

[2.11 AES with Counter 99](#_Toc10560938)

[2.11.1 Definitions 99](#_Toc10560939)

[2.11.2 AES with Counter mechanism parameters 100](#_Toc10560940)

[2.11.3 AES with Counter Encryption / Decryption 100](#_Toc10560941)

[2.12 AES CBC with Cipher Text Stealing CTS 101](#_Toc10560942)

[2.12.1 Definitions 101](#_Toc10560943)

[2.12.2 AES CTS mechanism parameters 101](#_Toc10560944)

[2.13 Additional AES Mechanisms 101](#_Toc10560945)

[2.13.1 Definitions 101](#_Toc10560946)

[2.13.2 AES-GCM Authenticated Encryption / Decryption 102](#_Toc10560947)

[2.13.3 AES-CCM authenticated Encryption / Decryption 103](#_Toc10560948)

[2.13.4 AES-GMAC 105](#_Toc10560949)

[2.13.5 AES GCM and CCM Mechanism parameters 106](#_Toc10560950)

[2.14 AES CMAC 108](#_Toc10560951)

[2.14.1 Definitions 109](#_Toc10560952)

[2.14.2 Mechanism parameters 109](#_Toc10560953)

[2.14.3 General-length AES-CMAC 109](#_Toc10560954)

[2.14.4 AES-CMAC 109](#_Toc10560955)

[2.15 AES XTS 110](#_Toc10560956)

[2.15.1 Definitions 110](#_Toc10560957)

[2.15.2 AES-XTS secret key objects 110](#_Toc10560958)

[2.15.3 AES-XTS key generation 110](#_Toc10560959)

[2.15.4 AES-XTS 111](#_Toc10560960)

[2.16 AES Key Wrap 111](#_Toc10560961)

[2.16.1 Definitions 111](#_Toc10560962)

[2.16.2 AES Key Wrap Mechanism parameters 111](#_Toc10560963)

[2.16.3 AES Key Wrap 111](#_Toc10560964)

[2.17 Key derivation by data encryption – DES & AES 112](#_Toc10560965)

[2.17.1 Definitions 112](#_Toc10560966)

[2.17.2 Mechanism Parameters 113](#_Toc10560967)

[2.17.3 Mechanism Description 113](#_Toc10560968)

[2.18 Double and Triple-length DES 113](#_Toc10560969)

[2.18.1 Definitions 114](#_Toc10560970)

[2.18.2 DES2 secret key objects 114](#_Toc10560971)

[2.18.3 DES3 secret key objects 115](#_Toc10560972)

[2.18.4 Double-length DES key generation 115](#_Toc10560973)

[2.18.5 Triple-length DES Order of Operations 116](#_Toc10560974)

[2.18.6 Triple-length DES in CBC Mode 116](#_Toc10560975)

[2.18.7 DES and Triple length DES in OFB Mode 116](#_Toc10560976)

[2.18.8 DES and Triple length DES in CFB Mode 117](#_Toc10560977)

[2.19 Double and Triple-length DES CMAC 117](#_Toc10560978)

[2.19.1 Definitions 117](#_Toc10560979)

[2.19.2 Mechanism parameters 118](#_Toc10560980)

[2.19.3 General-length DES3-MAC 118](#_Toc10560981)

[2.19.4 DES3-CMAC 118](#_Toc10560982)

[2.20 SHA-1 118](#_Toc10560983)

[2.20.1 Definitions 119](#_Toc10560984)

[2.20.2 SHA-1 digest 119](#_Toc10560985)

[2.20.3 General-length SHA-1-HMAC 119](#_Toc10560986)

[2.20.4 SHA-1-HMAC 120](#_Toc10560987)

[2.20.5 SHA-1 key derivation 120](#_Toc10560988)

[2.20.6 SHA-1 HMAC key generation 120](#_Toc10560989)

[2.21 SHA-224 121](#_Toc10560990)

[2.21.1 Definitions 121](#_Toc10560991)

[2.21.2 SHA-224 digest 121](#_Toc10560992)

[2.21.3 General-length SHA-224-HMAC 122](#_Toc10560993)

[2.21.4 SHA-224-HMAC 122](#_Toc10560994)

[2.21.5 SHA-224 key derivation 122](#_Toc10560995)

[2.21.6 SHA-224 HMAC key generation 122](#_Toc10560996)

[2.22 SHA-256 123](#_Toc10560997)

[2.22.1 Definitions 123](#_Toc10560998)

[2.22.2 SHA-256 digest 123](#_Toc10560999)

[2.22.3 General-length SHA-256-HMAC 123](#_Toc10561000)

[2.22.4 SHA-256-HMAC 124](#_Toc10561001)

[2.22.5 SHA-256 key derivation 124](#_Toc10561002)

[2.22.6 SHA-256 HMAC key generation 124](#_Toc10561003)

[2.23 SHA-384 124](#_Toc10561004)

[2.23.1 Definitions 125](#_Toc10561005)

[2.23.2 SHA-384 digest 125](#_Toc10561006)

[2.23.3 General-length SHA-384-HMAC 125](#_Toc10561007)

[2.23.4 SHA-384-HMAC 125](#_Toc10561008)

[2.23.5 SHA-384 key derivation 126](#_Toc10561009)

[2.23.6 SHA-384 HMAC key generation 126](#_Toc10561010)

[2.24 SHA-512 126](#_Toc10561011)

[2.24.1 Definitions 126](#_Toc10561012)

[2.24.2 SHA-512 digest 126](#_Toc10561013)

[2.24.3 General-length SHA-512-HMAC 127](#_Toc10561014)

[2.24.4 SHA-512-HMAC 127](#_Toc10561015)

[2.24.5 SHA-512 key derivation 127](#_Toc10561016)

[2.24.6 SHA-512 HMAC key generation 127](#_Toc10561017)

[2.25 SHA-512/224 128](#_Toc10561018)

[2.25.1 Definitions 128](#_Toc10561019)

[2.25.2 SHA-512/224 digest 128](#_Toc10561020)

[2.25.3 General-length SHA-512/224-HMAC 128](#_Toc10561021)

[2.25.4 SHA-512/224-HMAC 129](#_Toc10561022)

[2.25.5 SHA-512/224 key derivation 129](#_Toc10561023)

[2.25.6 SHA-512/224 HMAC key generation 129](#_Toc10561024)

[2.26 SHA-512/256 129](#_Toc10561025)

[2.26.1 Definitions 130](#_Toc10561026)

[2.26.2 SHA-512/256 digest 130](#_Toc10561027)

[2.26.3 General-length SHA-512/256-HMAC 130](#_Toc10561028)

[2.26.4 SHA-512/256-HMAC 131](#_Toc10561029)

[2.26.5 SHA-512/256 key derivation 131](#_Toc10561030)

[2.26.6 SHA-512/256 HMAC key generation 131](#_Toc10561031)

[2.27 SHA-512/t 131](#_Toc10561032)

[2.27.1 Definitions 132](#_Toc10561033)

[2.27.2 SHA-512/t digest 132](#_Toc10561034)

[2.27.3 General-length SHA-512/t-HMAC 132](#_Toc10561035)

[2.27.4 SHA-512/t-HMAC 132](#_Toc10561036)

[2.27.5 SHA-512/t key derivation 133](#_Toc10561037)

[2.27.6 SHA-512/t HMAC key generation 133](#_Toc10561038)

[2.28 SHA3-224 133](#_Toc10561039)

[2.28.1 Definitions 133](#_Toc10561040)

[2.28.2 SHA3-224 digest 134](#_Toc10561041)

[2.28.3 General-length SHA3-224-HMAC 134](#_Toc10561042)

[2.28.4 SHA3-224-HMAC 134](#_Toc10561043)

[2.28.5 SHA3-224 key derivation 134](#_Toc10561044)

[2.28.6 SHA3-224 HMAC key generation 134](#_Toc10561045)

[2.29 SHA3-256 135](#_Toc10561046)

[2.29.1 Definitions 135](#_Toc10561047)

[2.29.2 SHA3-256 digest 135](#_Toc10561048)

[2.29.3 General-length SHA3-256-HMAC 135](#_Toc10561049)

[2.29.4 SHA3-256-HMAC 136](#_Toc10561050)

[2.29.5 SHA3-256 key derivation 136](#_Toc10561051)

[2.29.6 SHA3-256 HMAC key generation 136](#_Toc10561052)

[2.30 SHA3-384 136](#_Toc10561053)

[2.30.1 Definitions 137](#_Toc10561054)

[2.30.2 SHA3-384 digest 137](#_Toc10561055)

[2.30.3 General-length SHA3-384-HMAC 137](#_Toc10561056)

[2.30.4 SHA3-384-HMAC 138](#_Toc10561057)

[2.30.5 SHA3-384 key derivation 138](#_Toc10561058)

[2.30.6 SHA3-384 HMAC key generation 138](#_Toc10561059)

[2.31 SHA3-512 138](#_Toc10561060)

[2.31.1 Definitions 138](#_Toc10561061)

[2.31.2 SHA3-512 digest 139](#_Toc10561062)

[2.31.3 General-length SHA3-512-HMAC 139](#_Toc10561063)

[2.31.4 SHA3-512-HMAC 139](#_Toc10561064)

[2.31.5 SHA3-512 key derivation 139](#_Toc10561065)

[2.31.6 SHA3-512 HMAC key generation 140](#_Toc10561066)

[2.32 SHAKE 140](#_Toc10561067)

[2.32.1 Definitions 140](#_Toc10561068)

[2.32.2 SHAKE Key Derivation 140](#_Toc10561069)

[2.33 Blake2b-160 141](#_Toc10561070)

[2.33.1 Definitions 141](#_Toc10561071)

[2.33.2 BLAKE2B-160 digest 141](#_Toc10561072)

[2.33.3 General-length BLAKE2B-160-HMAC 142](#_Toc10561073)

[2.33.4 BLAKE2B-160-HMAC 142](#_Toc10561074)

[2.33.5 BLAKE2B-160 key derivation 142](#_Toc10561075)

[2.33.6 BLAKE2B-160 HMAC key generation 142](#_Toc10561076)

[2.34 BLAKE2B-256 142](#_Toc10561077)

[2.34.1 Definitions 143](#_Toc10561078)

[2.34.2 BLAKE2B-256 digest 143](#_Toc10561079)

[2.34.3 General-length BLAKE2B-256-HMAC 143](#_Toc10561080)

[2.34.4 BLAKE2B-256-HMAC 144](#_Toc10561081)

[2.34.5 BLAKE2B-256 key derivation 144](#_Toc10561082)

[2.34.6 BLAKE2B-256 HMAC key generation 144](#_Toc10561083)

[2.35 BLAKE2B-384 144](#_Toc10561084)

[2.35.1 Definitions 145](#_Toc10561085)

[2.35.2 BLAKE2B-384 digest 145](#_Toc10561086)

[2.35.3 General-length BLAKE2B-384-HMAC 145](#_Toc10561087)

[2.35.4 BLAKE2B-384-HMAC 145](#_Toc10561088)

[2.35.5 BLAKE2B-384 key derivation 146](#_Toc10561089)

[2.35.6 BLAKE2B-384 HMAC key generation 146](#_Toc10561090)

[2.36 BLAKE2B-512 146](#_Toc10561091)

[2.36.1 Definitions 146](#_Toc10561092)

[2.36.2 BLAKE2B-512 digest 146](#_Toc10561093)

[2.36.3 General-length BLAKE2B-512-HMAC 147](#_Toc10561094)

[2.36.4 BLAKE2B-512-HMAC 147](#_Toc10561095)

[2.36.5 BLAKE2B-512 key derivation 147](#_Toc10561096)

[2.36.6 BLAKE2B-512 HMAC key generation 147](#_Toc10561097)

[2.37 PKCS #5 and PKCS #5-style password-based encryption (PBE) 148](#_Toc10561098)

[2.37.1 Definitions 148](#_Toc10561099)

[2.37.2 Password-based encryption/authentication mechanism parameters 148](#_Toc10561100)

[2.37.3 PKCS #5 PBKDF2 key generation mechanism parameters 149](#_Toc10561101)

[2.37.4 PKCS #5 PBKD2 key generation 151](#_Toc10561102)

[2.38 PKCS #12 password-based encryption/authentication mechanisms 151](#_Toc10561103)

[2.38.1 SHA-1-PBE for 3-key triple-DES-CBC 152](#_Toc10561104)

[2.38.2 SHA-1-PBE for 2-key triple-DES-CBC 152](#_Toc10561105)

[2.38.3 SHA-1-PBA for SHA-1-HMAC 152](#_Toc10561106)

[2.39 SSL 153](#_Toc10561107)

[2.39.1 Definitions 153](#_Toc10561108)

[2.39.2 SSL mechanism parameters 153](#_Toc10561109)

[2.39.3 Pre-master key generation 155](#_Toc10561110)

[2.39.4 Master key derivation 156](#_Toc10561111)

[2.39.5 Master key derivation for Diffie-Hellman 156](#_Toc10561112)

[2.39.6 Key and MAC derivation 157](#_Toc10561113)

[2.39.7 MD5 MACing in SSL 3.0 158](#_Toc10561114)

[2.39.8 SHA-1 MACing in SSL 3.0 158](#_Toc10561115)

[2.40 TLS 1.2 Mechanisms 159](#_Toc10561116)

[2.40.1 Definitions 159](#_Toc10561117)

[2.40.2 TLS 1.2 mechanism parameters 159](#_Toc10561118)

[ CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS; CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR 159](#_Toc10561119)

[ CK\_TLS12\_KEY\_MAT\_PARAMS; CK\_TLS12\_KEY\_MAT\_PARAMS\_PTR 160](#_Toc10561120)

[ CK\_TLS\_KDF\_PARAMS; CK\_TLS\_KDF\_PARAMS\_PTR 161](#_Toc10561121)

[ CK\_TLS\_MAC\_PARAMS; CK\_TLS\_MAC\_PARAMS\_PTR 161](#_Toc10561122)

[2.40.3 TLS MAC 162](#_Toc10561123)

[2.40.4 Master key derivation 163](#_Toc10561124)

[2.40.5 Master key derivation for Diffie-Hellman 163](#_Toc10561125)

[2.40.6 Key and MAC derivation 164](#_Toc10561126)

[2.40.7 CKM\_TLS12\_KEY\_SAFE\_DERIVE 165](#_Toc10561127)

[2.40.8 Generic Key Derivation using the TLS PRF 165](#_Toc10561128)

[2.40.9 Generic Key Derivation using the TLS12 PRF 166](#_Toc10561129)

[2.41 WTLS 167](#_Toc10561130)

[2.41.1 Definitions 167](#_Toc10561131)

[2.41.2 WTLS mechanism parameters 167](#_Toc10561132)

[2.41.3 Pre master secret key generation for RSA key exchange suite 170](#_Toc10561133)

[2.41.4 Master secret key derivation 171](#_Toc10561134)

[2.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography 171](#_Toc10561135)

[2.41.6 WTLS PRF (pseudorandom function) 172](#_Toc10561136)

[2.41.7 Server Key and MAC derivation 172](#_Toc10561137)

[2.41.8 Client key and MAC derivation 173](#_Toc10561138)

[2.42 SP 800-108 Key Derivation 174](#_Toc10561139)

[2.42.1 Definitions 174](#_Toc10561140)

[2.42.2 Mechanism Parameters 175](#_Toc10561141)

[2.42.3 Counter Mode KDF 180](#_Toc10561142)

[2.42.4 Feedback Mode KDF 181](#_Toc10561143)

[2.42.5 Double Pipeline Mode KDF 181](#_Toc10561144)

[2.42.6 Deriving Additional Keys 182](#_Toc10561145)

[2.42.7 Key Derivation Attribute Rules 183](#_Toc10561146)

[2.42.8 Constructing PRF Input Data 183](#_Toc10561147)

[2.42.8.1 Sample Counter Mode KDF 184](#_Toc10561148)

[2.42.8.2 Sample SCP03 Counter Mode KDF 185](#_Toc10561149)

[2.42.8.3 Sample Feedback Mode KDF 186](#_Toc10561150)

[2.42.8.4 Sample Double-Pipeline Mode KDF 187](#_Toc10561151)

[2.43 Miscellaneous simple key derivation mechanisms 188](#_Toc10561152)

[2.43.1 Definitions 188](#_Toc10561153)

[2.43.2 Parameters for miscellaneous simple key derivation mechanisms 188](#_Toc10561154)

[2.43.3 Concatenation of a base key and another key 189](#_Toc10561155)

[2.43.4 Concatenation of a base key and data 190](#_Toc10561156)

[2.43.5 Concatenation of data and a base key 190](#_Toc10561157)

[2.43.6 XORing of a key and data 191](#_Toc10561158)

[2.43.7 Extraction of one key from another key 192](#_Toc10561159)

[2.44 CMS 193](#_Toc10561160)

[2.44.1 Definitions 193](#_Toc10561161)

[2.44.2 CMS Signature Mechanism Objects 193](#_Toc10561162)

[2.44.3 CMS mechanism parameters 193](#_Toc10561163)

[2.44.4 CMS signatures 195](#_Toc10561164)

[2.45 Blowfish 195](#_Toc10561165)

[2.45.1 Definitions 196](#_Toc10561166)

[2.45.2 BLOWFISH secret key objects 196](#_Toc10561167)

[2.45.3 Blowfish key generation 197](#_Toc10561168)

[2.45.4 Blowfish-CBC 197](#_Toc10561169)

[2.45.5 Blowfish-CBC with PKCS padding 197](#_Toc10561170)

[2.46 Twofish 198](#_Toc10561171)

[2.46.1 Definitions 198](#_Toc10561172)

[2.46.2 Twofish secret key objects 198](#_Toc10561173)

[2.46.3 Twofish key generation 199](#_Toc10561174)

[2.46.4 Twofish -CBC 199](#_Toc10561175)

[2.46.5 Twofish-CBC with PKCS padding 199](#_Toc10561176)

[2.47 CAMELLIA 199](#_Toc10561177)

[2.47.1 Definitions 200](#_Toc10561178)

[2.47.2 Camellia secret key objects 200](#_Toc10561179)

[2.47.3 Camellia key generation 201](#_Toc10561180)

[2.47.4 Camellia-ECB 201](#_Toc10561181)

[2.47.5 Camellia-CBC 202](#_Toc10561182)

[2.47.6 Camellia-CBC with PKCS padding 202](#_Toc10561183)

[2.47.7 CAMELLIA with Counter mechanism parameters 203](#_Toc10561184)

[2.47.8 General-length Camellia-MAC 204](#_Toc10561185)

[2.47.9 Camellia-MAC 204](#_Toc10561186)

[2.48 Key derivation by data encryption - Camellia 205](#_Toc10561187)

[2.48.1 Definitions 205](#_Toc10561188)

[2.48.2 Mechanism Parameters 205](#_Toc10561189)

[2.49 ARIA 205](#_Toc10561190)

[2.49.1 Definitions 206](#_Toc10561191)

[2.49.2 Aria secret key objects 206](#_Toc10561192)

[2.49.3 ARIA key generation 207](#_Toc10561193)

[2.49.4 ARIA-ECB 207](#_Toc10561194)

[2.49.5 ARIA-CBC 207](#_Toc10561195)

[2.49.6 ARIA-CBC with PKCS padding 208](#_Toc10561196)

[2.49.7 General-length ARIA-MAC 209](#_Toc10561197)

[2.49.8 ARIA-MAC 209](#_Toc10561198)

[2.50 Key derivation by data encryption - ARIA 209](#_Toc10561199)

[2.50.1 Definitions 210](#_Toc10561200)

[2.50.2 Mechanism Parameters 210](#_Toc10561201)

[2.51 SEED 210](#_Toc10561202)

[2.51.1 Definitions 211](#_Toc10561203)

[2.51.2 SEED secret key objects 211](#_Toc10561204)

[2.51.3 SEED key generation 212](#_Toc10561205)

[2.51.4 SEED-ECB 212](#_Toc10561206)

[2.51.5 SEED-CBC 212](#_Toc10561207)

[2.51.6 SEED-CBC with PKCS padding 212](#_Toc10561208)

[2.51.7 General-length SEED-MAC 213](#_Toc10561209)

[2.51.8 SEED-MAC 213](#_Toc10561210)

[2.52 Key derivation by data encryption - SEED 213](#_Toc10561211)

[2.52.1 Definitions 213](#_Toc10561212)

[2.52.2 Mechanism Parameters 213](#_Toc10561213)

[2.53 OTP 213](#_Toc10561214)

[2.53.1 Usage overview 213](#_Toc10561215)

[2.53.2 Case 1: Generation of OTP values 214](#_Toc10561216)

[2.53.3 Case 2: Verification of provided OTP values 215](#_Toc10561217)

[2.53.4 Case 3: Generation of OTP keys 216](#_Toc10561218)

[2.53.5 OTP objects 216](#_Toc10561219)

[2.53.5.1 Key objects 216](#_Toc10561220)

[2.53.6 OTP-related notifications 219](#_Toc10561221)

[2.53.7 OTP mechanisms 219](#_Toc10561222)

[2.53.7.1 OTP mechanism parameters 219](#_Toc10561223)

[2.53.8 RSA SecurID 223](#_Toc10561224)

[2.53.8.1 RSA SecurID secret key objects 223](#_Toc10561225)

[2.53.8.2 RSA SecurID key generation 224](#_Toc10561226)

[2.53.8.3 SecurID OTP generation and validation 225](#_Toc10561227)

[2.53.8.4 Return values 225](#_Toc10561228)

[2.53.9 OATH HOTP 225](#_Toc10561229)

[2.53.9.1 OATH HOTP secret key objects 225](#_Toc10561230)

[2.53.9.2 HOTP key generation 226](#_Toc10561231)

[2.53.9.3 HOTP OTP generation and validation 226](#_Toc10561232)

[2.53.10 ActivIdentity ACTI 226](#_Toc10561233)

[2.53.10.1 ACTI secret key objects 226](#_Toc10561234)

[2.53.10.2 ACTI key generation 227](#_Toc10561235)

[2.53.10.3 ACTI OTP generation and validation 227](#_Toc10561236)

[2.54 CT-KIP 228](#_Toc10561237)

[2.54.1 Principles of Operation 228](#_Toc10561238)

[2.54.2 Mechanisms 228](#_Toc10561239)

[2.54.3 Definitions 229](#_Toc10561240)

[2.54.4 CT-KIP Mechanism parameters 229](#_Toc10561241)

[2.54.5 CT-KIP key derivation 229](#_Toc10561242)

[2.54.6 CT-KIP key wrap and key unwrap 230](#_Toc10561243)

[2.54.7 CT-KIP signature generation 230](#_Toc10561244)

[2.55 GOST 28147-89 230](#_Toc10561245)

[2.55.1 Definitions 231](#_Toc10561246)

[2.55.2 GOST 28147-89 secret key objects 231](#_Toc10561247)

[2.55.3 GOST 28147-89 domain parameter objects 232](#_Toc10561248)

[2.55.4 GOST 28147-89 key generation 232](#_Toc10561249)

[2.55.5 GOST 28147-89-ECB 233](#_Toc10561250)

[2.55.6 GOST 28147-89 encryption mode except ECB 233](#_Toc10561251)

[2.55.7 GOST 28147-89-MAC 234](#_Toc10561252)

[2.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89 234](#_Toc10561253)

[2.56 GOST R 34.11-94 235](#_Toc10561254)

[2.56.1 Definitions 235](#_Toc10561255)

[2.56.2 GOST R 34.11-94 domain parameter objects 235](#_Toc10561256)

[2.56.3 GOST R 34.11-94 digest 236](#_Toc10561257)

[2.56.4 GOST R 34.11-94 HMAC 237](#_Toc10561258)

[2.57 GOST R 34.10-2001 237](#_Toc10561259)

[2.57.1 Definitions 238](#_Toc10561260)

[2.57.2 GOST R 34.10-2001 public key objects 238](#_Toc10561261)

[2.57.3 GOST R 34.10-2001 private key objects 239](#_Toc10561262)

[2.57.4 GOST R 34.10-2001 domain parameter objects 241](#_Toc10561263)

[2.57.5 GOST R 34.10-2001 mechanism parameters 242](#_Toc10561264)

[2.57.6 GOST R 34.10-2001 key pair generation 243](#_Toc10561265)

[2.57.7 GOST R 34.10-2001 without hashing 244](#_Toc10561266)

[2.57.8 GOST R 34.10-2001 with GOST R 34.11-94 244](#_Toc10561267)

[2.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001 245](#_Toc10561268)

[2.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys 245](#_Toc10561269)

[2.58 ChaCha20 245](#_Toc10561270)

[2.58.1 Definitions 245](#_Toc10561271)

[2.58.2 ChaCha20 secret key objects 245](#_Toc10561272)

[2.58.3 ChaCha20 mechanism parameters 246](#_Toc10561273)

[2.58.4 ChaCha20 key generation 247](#_Toc10561274)

[2.58.5 ChaCha20 mechanism 247](#_Toc10561275)

[2.59 Salsa20 248](#_Toc10561276)

[2.59.1 Definitions 248](#_Toc10561277)

[2.59.2 Salsa20 secret key objects 248](#_Toc10561278)

[2.59.3 Salsa20 mechanism parameters 249](#_Toc10561279)

[2.59.4 Salsa20 key generation 249](#_Toc10561280)

[2.59.5 Salsa20 mechanism 250](#_Toc10561281)

[2.60 Poly1305 250](#_Toc10561282)

[2.60.1 Definitions 251](#_Toc10561283)

[2.60.2 Poly1305 secret key objects 251](#_Toc10561284)

[2.60.3 Poly1305 mechanism 251](#_Toc10561285)

[2.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption 252](#_Toc10561286)

[2.61.1 Definitions 252](#_Toc10561287)

[2.61.2 Usage 252](#_Toc10561288)

[2.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters 253](#_Toc10561289)

[2.62 HKDF Mechanisms 254](#_Toc10561290)

[2.62.1 Definitions 255](#_Toc10561291)

[2.62.2 HKDF mechanism parameters 255](#_Toc10561292)

[2.62.3 HKDF derive 256](#_Toc10561293)

[2.62.4 HKDF Data 257](#_Toc10561294)

[2.62.5 HKDF Key gen 257](#_Toc10561295)

[2.63 NULL Mechanism 257](#_Toc10561296)

[2.63.1 Definitions 257](#_Toc10561297)

[2.63.2 CKM\_NULL mechanism parameters 257](#_Toc10561298)

[3 PKCS #11 Implementation Conformance 258](#_Toc10561299)

[Appendix A. Acknowledgments 259](#_Toc10561300)

[Appendix B. Manifest Constants 262](#_Toc10561301)

[B.1 Object classes 262](#_Toc10561302)

[B.2 Key types 262](#_Toc10561303)

[B.3 Key derivation functions 263](#_Toc10561304)

[B.4 Mechanisms 264](#_Toc10561305)

[B.5 Attributes 273](#_Toc10561306)

[B.6 Attribute constants 275](#_Toc10561307)

[B.7 Other constants 275](#_Toc10561308)

[B.8 Notifications 276](#_Toc10561309)

[B.9 Return values 276](#_Toc10561310)

[Appendix C. Revision History 279](#_Toc10561311)

# Introduction

This document defines mechanisms that are anticipated to be used with the current version of PKCS #11.

All text is normative unless otherwise labeled.

## IPR Policy

This specification is provided under the [RF on RAND Terms](https://www.oasis-open.org/policies-guidelines/ipr#RF-on-RAND-Mode) Mode of the [OASIS IPR Policy](https://www.oasis-open.org/policies-guidelines/ipr), the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (<https://www.oasis-open.org/committees/pkcs11/ipr.php>).

## Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119]

## Definitions

For the purposes of this standard, the following definitions apply. Please refer to the [PKCS#11-Base] for further definitions:

**AES** Advanced Encryption Standard, as defined in FIPS PUB 197.

**CAMELLIA** The Camellia encryption algorithm, as defined in RFC 3713.

**BLOWFISH** The Blowfish Encryption Algorithm of Bruce Schneier, [www.schneier.com](http://www.schneier.com/).

**CBC** Cipher-Block Chaining mode, as defined in FIPS PUB 81.

**CDMF** Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.

**CMAC** Cipher-based Message Authenticate Code as defined in [NIST sp800-38b] and [RFC 4493].

**CMS** Cryptographic Message Syntax (see RFC 2630)

**CT-KIP** Cryptographic Token Key Initialization Protocol (as defined in [CT-KIP])

**DES** Data Encryption Standard, as defined in FIPS PUB 46-3**.**

**DSA** Digital Signature Algorithm, as defined in FIPS PUB 186-2.

**EC** Elliptic Curve

**ECB** Electronic Codebook mode, as defined in FIPS PUB 81.

**ECDH** Elliptic Curve Diffie-Hellman.

**ECDSA** Elliptic Curve DSA, as in ANSI X9.62.

**ECMQV** Elliptic Curve Menezes-Qu-Vanstone

**GOST 28147-89** The encryption algorithm, as defined in Part 2 [GOST 28147-89] and [RFC 4357] [RFC 4490], and RFC [4491].

**GOST R 34.11-94** Hash algorithm, as defined in [GOST R 34.11-94] and [RFC 4357], [RFC 4490], and [RFC 4491].

**GOST R 34.10-2001** The digital signature algorithm, as defined in [GOST R 34.10-2001] and [RFC 4357], [RFC 4490], and [RFC 4491].

**IV** Initialization Vector.

**MAC** Message Authentication Code.

**MQV** Menezes-Qu-Vanstone

**OAEP** Optimal Asymmetric Encryption Padding for RSA.

**PKCS** Public-Key Cryptography Standards.

**PRF** Pseudo random function.

**PTD** Personal Trusted Device, as defined in MeT-PTD

**RSA** The RSA public-key cryptosystem.

**SHA-1** The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2.

**SHA-224** The Secure Hash Algorithm with a 224-bit message digest, as defined in RFC 3874. Also defined in FIPS PUB 180-2 with Change Notice 1.

**SHA-256** The Secure Hash Algorithm with a 256-bit message digest, as defined in FIPS PUB 180-2.

**SHA-384** The Secure Hash Algorithm with a 384-bit message digest, as defined in FIPS PUB 180-2.

**SHA-512** The Secure Hash Algorithm with a 512-bit message digest, as defined in FIPS PUB 180-2.

**SSL** The Secure Sockets Layer 3.0 protocol.

**SO** A Security Officer user.

**TLS** Transport Layer Security.

**WIM** Wireless Identification Module.

**WTLS** Wireless Transport Layer Security.

## Normative References

**[ARIA]** National Security Research Institute, Korea, “Block Cipher Algorithm ARIA”,   
URL: <http://tools.ietf.org/html/rfc5794>

**[BLOWFISH]** B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish), December 1993.  
URL: <https://www.schneier.com/paper-blowfish-fse.html>

**[CAMELLIA]** M. Matsui, J. Nakajima, S. Moriai. A Description of the Camellia Encryption Algorithm, April 2004.  
URL: <http://www.ietf.org/rfc/rfc3713.txt>

**[CDMF]** Johnson, D.B The Commercial Data Masking Facility (CDMF) data privacy algorithm, March 1994.  
URL: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5389557>

**[CHACHA]** D. Bernstein, ChaCha, a variant of Salsa20, Jan 2008.  
URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>

**[DH]** W. Diffie, M. Hellman. New Directions in Cryptography. Nov, 1976.  
URL: <http://www-ee.stanford.edu/~hellman/publications/24.pdf>

**[FIPS PUB 81]** NIST. *FIPS 81: DES Modes of Operation.* December 1980.

URL: <http://csrc.nist.gov/publications/fips/fips81/fips81.htm>

**[FIPS PUB 186-4]** NIST. FIPS 186-4: Digital Signature Standard. July 2013.  
URL: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>

**[FIPS PUB 197]** NIST. FIPS 197: Advanced Encryption Standard. November 26, 2001.  
URL: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

**[FIPS SP 800-56A]** NIST. Special Publication 800-56A Revision 2*:* *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography,* May 2013.   
URL: <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf>

**[FIPS SP 800-108]** NIST. Special Publication 800-108 (Revised): *Recommendation for Key Derivation Using Pseudorandom Functions*, October 2009.   
URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf

**[GOST]** V. Dolmatov, A. Degtyarev. GOST R. 34.11-2012: Hash Function. August 2013.   
URL: <http://tools.ietf.org/html/rfc6986>

**[MD2]** B. Kaliski. RSA Laboratories. The MD2 Message-Digest Algorithm. April, 1992.   
URL: <http://tools.ietf.org/html/rfc1319>

**[MD5]** RSA Data Security. R. Rivest. The MD5 Message-Digest Algorithm. April, 1992.   
URL: <http://tools.ietf.org/html/rfc1319>

**[OAEP]** M. Bellare, P. Rogaway. Optimal Asymmetric Encryption – How to Encrypt with RSA. Nov 19, 1995.  
URL: <http://cseweb.ucsd.edu/users/mihir/papers/oae.pdf>

[PKCS11-Base] *PKCS #11 Cryptographic Token Interface Base Specification Version 3.0.* Edited by Chris Zimman and Dieter Bong. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html>.

[PKCS11-Hist] *PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 3.0*. Edited by Chris Zimman and Dieter Bong. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.html>.

[PKCS11-Prof] *PKCS #11 Cryptographic Token Interface Profiles Version 3.0.* Edited by Tim Hudson. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.0/pkcs11-profiles-v3.0.html>.

[POLY1305] D.J. Bernstein. The Poly1305-AES message-authentication code. Jan 2005.  
URL: <https://cr.yp.to/mac/poly1305-20050329.pdf>

[RFC2119] Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, March 1997.   
URL: <http://www.ietf.org/rfc/rfc2119.txt>.

[RIPEMD] H. Dobbertin, A. Bosselaers, B. Preneel. The hash function RIPEMD-160, Feb 13, 2012.  
URL: <http://homes.esat.kuleuven.be/~bosselae/ripemd160.html>

**[SALSA]** D. Bernstein, ChaCha, a variant of Salsa20, Jan 2008.  
URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>

[SEED] KISA. SEED 128 Algorithm Specification. Sep 2003.   
URL: <http://seed.kisa.or.kr/html/egovframework/iwt/ds/ko/ref/%5B2%5D_SEED+128_Specification_english_M.pdf>

[SHA-1] NIST. FIPS 180-4: Secure Hash Standard. March 2012.   
URL: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>

[SHA-**2]** NIST. FIPS 180-4: Secure Hash Standard. March 2012.   
URL: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>

[TWOFISH] B. Schneier, J. Kelsey, D. Whiting, C. Hall, N. Ferguson. Twofish: A 128-Bit Block Cipher. June 15, 1998.   
URL: <https://www.schneier.com/paper-twofish-paper.pdf>

## Non-Normative References

**[CAP-1.2]** *Common Alerting Protocol Version 1.2*. 01 July 2010. OASIS Standard.   
URL: <http://docs.oasis-open.org/emergency/cap/v1.2/CAP-v1.2-os.html>

**[AES KEYWRAP]** National Institute of Standards and Technology, NIST Special Publication 800-38F, Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping, December 2012, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf>

**[ANSI C]** ANSI/ISO. American National Standard for Programming Languages – C. 1990.

**[ANSI X9.31]** Accredited Standards Committee X9. Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA). 1998.

**[ANSI X9.42]** Accredited Standards Committee X9. Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography. 2003.

**[ANSI X9.62]** Accredited Standards Committee X9. Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). 1998.

**[ANSI X9.63]** Accredited Standards Committee X9. Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography. 2001.   
URL: <http://webstore.ansi.org/RecordDetail.aspx?sku=X9.63-2011>

**[BRAINPOOL]** ECC Brainpool Standard Curves and Curve Generation, v1.0, 19.10.2005  
URL: http://www.ecc-brainpool.org

**[CT-KIP]** RSA Laboratories. Cryptographic Token Key Initialization Protocol. Version 1.0, December 2005.   
URL: <ftp://ftp.rsasecurity.com/pub/otps/ct-kip/ct-kip-v1-0.pdf>.

**[CC/PP]** CCPP-STRUCT-VOCAB, G. Klyne, F. Reynolds, C. , H. Ohto, J. Hjelm, M. H. Butler, L. Tran, Editors, W3C Recommendation, 15 January 2004,   
URL: <http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115/>   
Latest version available at <http://www.w3.org/TR/CCPP-struct-vocab/>

**[LEGIFRANCE]** Avis relatif aux paramètres de courbes elliptiques définis par l'Etat français (Publication of elliptic curve parameters by the French state)  
URL: https://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000024668816

**[NIST AES CTS]** National Institute of Standards and Technology, Addendum to NIST Special Publication 800-38A, “Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode”   
URL: <http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-38A.pdf>

[PKCS11-UG] *PKCS #11 Cryptographic Token Interface Usage Guide Version 2.41*. Edited by John Leiseboer and Robert Griffin. version: <http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html>.

**[RFC 2865]** Rigney et al, “Remote Authentication Dial In User Service (RADIUS)”, IETF RFC2865, June 2000.   
URL: [http://www.ietf.org/rfc/rfc2865.txt](http://ietf.org/rfc/rfc2865.txt).

**[RFC 3686]** Housley, “Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP),” IETF RFC 3686, January 2004.   
URL: [http://www.ietf.org/rfc/rfc3686.txt](http://ietf.org/rfc/rfc3686.txt).

**[RFC 3717]** Matsui, et al, ”A Description of the Camellia Encryption Algorithm,” IETF RFC 3717, April 2004.   
URL: [http://www.ietf.org/rfc/rfc3713.txt](http://ietf.org/rfc/rfc3713.txt).

**[RFC 3610]** Whiting, D., Housley, R., and N. Ferguson, “Counter with CBC-MAC (CCM)", IETF RFC 3610, September 2003.   
URL: <http://www.ietf.org/rfc/rfc3610.txt>

**[RFC 3874]** Smit et al, “A 224-bit One-way Hash Function: SHA-224,” IETF RFC 3874, June 2004.   
URL: [http://www.ietf.org/rfc/rfc3874.txt](http://ietf.org/rfc/rfc3874.txt).

**[RFC 3748]** Aboba et al, “Extensible Authentication Protocol (EAP)”, IETF RFC 3748, June 2004.   
URL: [http://www.ietf.org/rfc/rfc3748.txt](http://ietf.org/rfc/rfc3748.txt).

**[RFC 4269]** South Korean Information Security Agency (KISA) “The SEED Encryption Algorithm”, December 2005.   
URL: <ftp://ftp.rfc-editor.org/in-notes/rfc4269.txt>

**[RFC 4309]** Housley, R., “Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP),” IETF RFC 4309, December 2005.   
URL: [http://www.ietf.org/rfc/rfc4309.txt](http://ietf.org/rfc/rfc4309.txt)

**[RFC 4357]** V. Popov, I. Kurepkin, S. Leontiev “Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms”, January 2006.   
URL: <http://www.ietf.org/rfc/rfc4357.txt>

**[RFC 4490]** S. Leontiev, Ed. G. Chudov, Ed. “Using the GOST 28147-89, GOST R 34.11-94,GOST R 34.10-94, and GOST R 34.10-2001 Algorithms with Cryptographic Message Syntax (CMS)”, May 2006.   
URL: <http://www.ietf.org/rfc/rfc4490.txt>

**[RFC 4491]** S. Leontiev, Ed., D. Shefanovski, Ed., “Using the GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and CRL Profile”, May 2006.   
URL: <http://www.ietf.org/rfc/rfc4491.txt>

**[RFC 4493]** J. Song et al. *RFC 4493: The AES-CMAC Algorithm.* June 2006.   
URL: <http://www.ietf.org/rfc/rfc4493.txt>

**[RFC 5705]** Rescorla, E., “The Keying Material Exporters for Transport Layer Security (TLS)”, RFC 5705, March 2010.   
URL: http://www.ietf.org/rfc/rfc5705.txt

**[RFC 5869]** H. Krawczyk, P. Eronen, “HMAC-based Extract-and-Expand Key Derivation Function (HKDF)“, May 2010   
URL: <http://www.ietf.org/rfc/rfc5869.txt>

**[RFC 7539]** Y Nir, A. Langley. *RFC 7539: ChaCha20 and Poly1305 for IETF Protocols,* May 2015  
URL: <https://tools.ietf.org/rfc/rfc7539.txt>

**[RFC 7748]** Aboba et al, “Elliptic Curves for Security”, IETF RFC 7748, January 2016   
URL: <https://tools.ietf.org/html/rfc7748>

**[RFC 8032]** Aboba et al, “Edwards-Curve Digital Signature Algorithm (EdDSA)”, IETF RFC 8032, January 2017  
URL: https://tools.ietf.org/html/rfc8032

**[SEC 1]** Standards for Efficient Cryptography Group (SECG). *Standards for Efficient Cryptography (SEC) 1: Elliptic Curve Cryptography*. Version 1.0, September 20, 2000.

**[SEC 2]** Standards for Efficient Cryptography Group (SECG). Standards for Efficient Cryptography (SEC) 2: Recommended Elliptic Curve Domain Parameters. Version 1.0, September 20, 2000.

**[SIGNAL]** The X3DH Key Agreement Protocol, Revision 1, 2016-11-04, Moxie Marlinspike, Trevor Perrin (editor)  
URL:<https://signal.org/docs/specifications/x3dh/>

**[TLS]** [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999. http://www.ietf.org/rfc/rfc2246.txt, superseded by [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006. http://www.ietf.org/rfc/rfc4346.txt, which was superseded by [5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.   
URL: <http://www.ietf.org/rfc/rfc5246.txt>

**[TLS12]** [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.   
URL: http://www.ietf.org/rfc/rfc5246.txt

**[WIM]** WAP. Wireless Identity Module. — WAP-260-WIM-20010712-a. July 2001.   
URL: <http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-260-wim-20010712-a.pdf>

**[WPKI]** Wireless Application Protocol: Public Key Infrastructure Definition. — WAP-217-WPKI-20010424-a. April 2001.   
URL: <http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-217-wpki-20010424-a.pdf>

**[WTLS]** WAP. Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a. April 2001.   
URL: <http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/wap/wap-261-wtls-20010406-a.pdf>

**[XEDDSA]** The XEdDSA and VXEdDSA Signature Schemes - Revision 1, 2016-10-20, Trevor Perrin (editor)  
URL:<https://signal.org/docs/specifications/xeddsa/>

**[X.500]** ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1

[X.509] ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8

**[X.680]** ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. July 2002. Identical to ISO/IEC 8824-1

**[X.690]** ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical to ISO/IEC 8825-1

# Mechanisms

A mechanism specifies precisely how a certain cryptographic process is to be performed. PKCS #11 implementations MAY use one of more mechanisms defined in this document.

The following table shows which Cryptoki mechanisms are supported by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token which supports one mechanism for some operations supports any other mechanism for any other operation (or even supports that same mechanism for any other operation). For example, even if a token is able to create RSA digital signatures with the **CKM\_RSA\_PKCS** mechanism, it may or may not be the case that the same token can also perform RSA encryption with **CKM\_RSA\_PKCS**.

Each mechanism description is be preceded by a table, of the following format, mapping mechanisms to API functions.

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
|  |  |  |  |  |  |  |  |

1 SR = SignRecover, VR = VerifyRecover.

2 Single-part operations only.

3 Mechanism can only be used for wrapping, not unwrapping.

The remainder of this section will present in detail the mechanisms supported by Cryptoki and the parameters which are supplied to them.

In general, if a mechanism makes no mention of the ulMinKeyLen and ulMaxKeyLen fields of the CK\_MECHANISM\_INFO structure, then those fields have no meaning for that particular mechanism.

## RSA

*Table 1, Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_RSA\_PKCS\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_RSA\_X9\_31\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_RSA\_PKCS | ✓2 | ✓2 | ✓ |  |  | ✓ |  |
| CKM\_RSA\_PKCS\_OAEP | ✓2 |  |  |  |  | ✓ |  |
| CKM\_RSA\_PKCS\_PSS |  | ✓2 |  |  |  |  |  |
| CKM\_RSA\_9796 |  | ✓2 | ✓ |  |  |  |  |
| CKM\_RSA\_X\_509 | ✓2 | ✓2 | ✓ |  |  | ✓ |  |
| CKM\_RSA\_X9\_31 |  | ✓2 |  |  |  |  |  |
| CKM\_SHA1\_RSA\_PKCS |  | ✓ |  |  |  |  |  |
| CKM\_SHA256\_RSA\_PKCS |  | ✓ |  |  |  |  |  |
| CKM\_SHA384\_RSA\_PKCS |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_RSA\_PKCS |  | ✓ |  |  |  |  |  |
| CKM\_SHA1\_RSA\_PKCS\_PSS |  | ✓ |  |  |  |  |  |
| CKM\_SHA256\_RSA\_PKCS\_PSS |  | ✓ |  |  |  |  |  |
| CKM\_SHA384\_RSA\_PKCS\_PSS |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_RSA\_PKCS\_PSS |  | ✓ |  |  |  |  |  |
| CKM\_SHA1\_RSA\_X9\_31 |  | ✓ |  |  |  |  |  |
| CKM\_RSA\_PKCS\_TPM\_1\_1 | ✓2 |  |  |  |  | ✓ |  |
| CKM\_RSA\_PKCS\_OAEP\_TPM\_1\_1 | ✓2 |  |  |  |  | ✓ |  |
| CKM\_SHA3\_224\_RSA\_PKCS |  |  |  |  |  |  |  |
| CKM\_SHA3\_256\_RSA\_PKCS |  |  |  |  |  |  |  |
| CKM\_SHA3\_384\_RSA\_PKCS |  |  |  |  |  |  |  |
| CKM\_SHA3\_512\_RSA\_PKCS |  |  |  |  |  |  |  |
| CKM\_SHA3\_224\_RSA\_PKCS\_PSS |  |  |  |  |  |  |  |
| CKM\_SHA3\_256\_RSA\_PKCS\_PSS |  |  |  |  |  |  |  |
| CKM\_SHA3\_384\_RSA\_PKCS\_PSS |  |  |  |  |  |  |  |
| CKM\_SHA3\_512\_RSA\_PKCS\_PSS |  |  |  |  |  |  |  |

### Definitions

This section defines the RSA key type “CKK\_RSA” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of RSA key objects.

Mechanisms:

CKM\_RSA\_PKCS\_KEY\_PAIR\_GEN

CKM\_RSA\_PKCS

CKM\_RSA\_9796

CKM\_RSA\_X\_509

CKM\_MD2\_RSA\_PKCS

CKM\_MD5\_RSA\_PKCS

CKM\_SHA1\_RSA\_PKCS

CKM\_SHA224\_RSA\_PKCS

CKM\_SHA256\_RSA\_PKCS

CKM\_SHA384\_RSA\_PKCS

CKM\_SHA512\_RSA\_PKCS

CKM\_RIPEMD128\_RSA\_PKCS

CKM\_RIPEMD160\_RSA\_PKCS

CKM\_RSA\_PKCS\_OAEP

CKM\_RSA\_X9\_31\_KEY\_PAIR\_GEN

CKM\_RSA\_X9\_31

CKM\_SHA1\_RSA\_X9\_31

CKM\_RSA\_PKCS\_PSS

CKM\_SHA1\_RSA\_PKCS\_PSS

CKM\_SHA224\_RSA\_PKCS\_PSS

CKM\_SHA256\_RSA\_PKCS\_PSS

CKM\_SHA512\_RSA\_PKCS\_PSS

CKM\_SHA384\_RSA\_PKCS\_PSS

CKM\_RSA\_PKCS\_TPM\_1\_1

CKM\_RSA\_PKCS\_OAEP\_TPM\_1\_1

CKM\_RSA\_AES\_KEY\_WRAP

CKM\_SHA3\_224\_RSA\_PKCS

CKM\_SHA3\_256\_RSA\_PKCS

CKM\_SHA3\_384\_RSA\_PKCS

CKM\_SHA3\_512\_RSA\_PKCS

CKM\_SHA3\_224\_RSA\_PKCS\_PSS

CKM\_SHA3\_256\_RSA\_PKCS\_PSS

CKM\_SHA3\_384\_RSA\_PKCS\_PSS

CKM\_SHA3\_512\_RSA\_PKCS\_PSS

### RSA public key objects

RSA public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_RSA**) hold RSA public keys. The following table defines the RSA public key object attributes, in addition to the common attributes defined for this object class:

Table 2, RSA Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_MODULUS1,4 | Big integer | Modulus *n* |
| CKA\_MODULUS\_BITS2,3 | CK\_ULONG | Length in bits of modulus *n* |
| CKA\_PUBLIC\_EXPONENT1 | Big integer | Public exponent *e* |

- Refer to [PKCS11-Base] table 11 for footnotes

Depending on the token, there may be limits on the length of key components. See PKCS #1 for more information on RSA keys.

The following is a sample template for creating an RSA public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_RSA;

CK\_UTF8CHAR label[] = “An RSA public key object”;

CK\_BYTE modulus[] = {...};

CK\_BYTE exponent[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_WRAP, &true, sizeof(true)},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_MODULUS, modulus, sizeof(modulus)},

{CKA\_PUBLIC\_EXPONENT, exponent, sizeof(exponent)}

};

### RSA private key objects

RSA private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_RSA**) hold RSA private keys. The following table defines the RSA private key object attributes, in addition to the common attributes defined for this object class:

Table 3, RSA Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_MODULUS1,4,6 | Big integer | Modulus *n* |
| CKA\_PUBLIC\_EXPONENT4,6 | Big integer | Public exponent *e* |
| CKA\_PRIVATE\_EXPONENT1,4,6,7 | Big integer | Private exponent *d* |
| CKA\_PRIME\_14,6,7 | Big integer | Prime *p* |
| CKA\_PRIME\_24,6,7 | Big integer | Prime *q* |
| CKA\_EXPONENT\_14,6,7 | Big integer | Private exponent *d* modulo *p*-1 |
| CKA\_EXPONENT\_24,6,7 | Big integer | Private exponent *d* modulo *q*-1 |
| CKA\_COEFFICIENT4,6,7 | Big integer | CRT coefficient *q*-1 mod *p* |

- Refer to [PKCS11-Base] table 11 for footnotes

Depending on the token, there may be limits on the length of the key components. See PKCS #1 for more information on RSA keys.

Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above attributes, which can assist in performing rapid RSA computations. Other tokens might store only the **CKA\_MODULUS** and **CKA\_PRIVATE\_EXPONENT** values. Effective with version 2.40, tokens MUST also store CKA\_PUBLIC\_EXPONENT. This permits the retrieval of sufficient data to reconstitute the associated public key.

Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an RSA private key, it stores whichever of the fields in Table 3 it keeps track of. Later, if an application asks for the values of the key’s various attributes, Cryptoki supplies values only for attributes whose values it can obtain (*i.e.*, if Cryptoki is asked for the value of an attribute it cannot obtain, the request fails). Note that a Cryptoki implementation may or may not be able and/or willing to supply various attributes of RSA private keys which are not actually stored on the token. *E.g.*, if a particular token stores values only for the **CKA\_PRIVATE\_EXPONENT**, **CKA\_PRIME\_1**, and **CKA\_PRIME\_2** attributes, then Cryptoki is certainly *able* to report values for all the attributes above (since they can all be computed efficiently from these three values). However, a Cryptoki implementation may or may not actually do this extra computation. The only attributes from Table 3 for which a Cryptoki implementation is *required* to be able to return values are **CKA\_MODULUS** and **CKA\_PRIVATE\_EXPONENT**.

If an RSA private key object is created on a token, and more attributes from Table 3 are supplied to the object creation call than are supported by the token, the extra attributes are likely to be thrown away. If an attempt is made to create an RSA private key object on a token with insufficient attributes for that particular token, then the object creation call fails and returns CKR\_TEMPLATE\_INCOMPLETE.

Note that when generating an RSA private key, there is no **CKA\_MODULUS\_BITS** attribute specified. This is because RSA private keys are only generated as part of an RSA key *pair*, and the **CKA\_MODULUS\_BITS** attribute for the pair is specified in the template for the RSA public key.

The following is a sample template for creating an RSA private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_RSA;

CK\_UTF8CHAR label[] = “An RSA private key object”;

CK\_BYTE subject[] = {...};

CK\_BYTE id[] = {123};

CK\_BYTE modulus[] = {...};

CK\_BYTE publicExponent[] = {...};

CK\_BYTE privateExponent[] = {...};

CK\_BYTE prime1[] = {...};

CK\_BYTE prime2[] = {...};

CK\_BYTE exponent1[] = {...};

CK\_BYTE exponent2[] = {...};

CK\_BYTE coefficient[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_DECRYPT, &true, sizeof(true)},

{CKA\_SIGN, &true, sizeof(true)},

{CKA\_MODULUS, modulus, sizeof(modulus)},

{CKA\_PUBLIC\_EXPONENT, publicExponent, sizeof(publicExponent)},

{CKA\_PRIVATE\_EXPONENT, privateExponent, sizeof(privateExponent)},

{CKA\_PRIME\_1, prime1, sizeof(prime1)},

{CKA\_PRIME\_2, prime2, sizeof(prime2)},

{CKA\_EXPONENT\_1, exponent1, sizeof(exponent1)},

{CKA\_EXPONENT\_2, exponent2, sizeof(exponent2)},

{CKA\_COEFFICIENT, coefficient, sizeof(coefficient)}

};

### PKCS #1 RSA key pair generation

The PKCS #1 RSA key pair generation mechanism, denoted **CKM\_RSA\_PKCS\_KEY\_PAIR\_GEN**, is a key pair generation mechanism based on the RSA public-key cryptosystem, as defined in PKCS #1.

It does not have a parameter.

The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public exponent, as specified in the **CKA\_MODULUS\_BITS** and **CKA\_PUBLIC\_EXPONENT** attributes of the template for the public key. The **CKA\_PUBLIC\_EXPONENT** may be omitted in which case the mechanism shall supply the public exponent attribute using the default value of 0x10001 (65537). Specific implementations may use a random value or an alternative default if 0x10001 cannot be used by the token.

Note: Implementations strictly compliant with version 2.11 or prior versions may generate an error if this attribute is omitted from the template. Experience has shown that many implementations of 2.11 and prior did allow the **CKA\_PUBLIC\_EXPONENT** attribute to be omitted from the template, and behaved as described above. The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_MODULUS**, and **CKA\_PUBLIC\_EXPONENT** attributes to the new public key. **CKA\_PUBLIC\_EXPONENT** will be copied from the template if supplied. **CKR\_TEMPLATE\_INCONSISTENT** shall be returned if the implementation cannot use the supplied exponent value. It contributes the **CKA\_CLASS** and **CKA\_KEY\_TYPE** attributes to the new private key; it may also contribute some of the following attributes to the new private key: **CKA\_MODULUS**, **CKA\_PUBLIC\_EXPONENT**, **CKA\_PRIVATE\_EXPONENT**, **CKA\_PRIME\_1**, **CKA\_PRIME\_2**, **CKA\_EXPONENT\_1**, **CKA\_EXPONENT\_2**, **CKA\_COEFFICIENT**. Other attributes supported by the RSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### X9.31 RSA key pair generation

The X9.31 RSA key pair generation mechanism, denoted **CKM\_RSA\_X9\_31\_KEY\_PAIR\_GEN**, is a key pair generation mechanism based on the RSA public-key cryptosystem, as defined in X9.31.

It does not have a parameter.

The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public exponent, as specified in the **CKA\_MODULUS\_BITS** and **CKA\_PUBLIC\_EXPONENT** attributes of the template for the public key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_MODULUS**, and **CKA\_PUBLIC\_EXPONENT** attributes to the new public key. It contributes the **CKA\_CLASS** and **CKA\_KEY\_TYPE** attributes to the new private key; it may also contribute some of the following attributes to the new private key: **CKA\_MODULUS**, **CKA\_PUBLIC\_EXPONENT**, **CKA\_PRIVATE\_EXPONENT**, **CKA\_PRIME\_1**, **CKA\_PRIME\_2**, **CKA\_EXPONENT\_1**, **CKA\_EXPONENT\_2**, **CKA\_COEFFICIENT**. Other attributes supported by the RSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values. Unlike the **CKM\_RSA\_PKCS\_KEY\_PAIR\_GEN** mechanism, this mechanism is guaranteed to generate *p* and *q* values, **CKA\_PRIME\_1** and **CKA\_PRIME\_2** respectively, that meet the strong primes requirement of X9.31.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### PKCS #1 v1.5 RSA

The PKCS #1 v1.5 RSA mechanism, denoted **CKM\_RSA\_PKCS**, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5. It supports single-part encryption and decryption; single-part signatures and verification with and without message recovery; key wrapping; and key unwrapping. This mechanism corresponds only to the part of PKCS #1 v1.5 that involves RSA; it does not compute a message digest or a DigestInfo encoding as specified for the md2withRSAEncryption and md5withRSAEncryption algorithms in PKCS #1 v1.5 .

This mechanism does not have a parameter.

This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the “input” to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type or any other information about the key, except the key length; the application must convey these separately. In particular, the mechanism contributes only the **CKA\_CLASS** and **CKA\_VALUE** (and **CKA\_VALUE\_LEN**, if the key has it) attributes to the recovered key during unwrapping; other attributes must be specified in the template.

Constraints on key types and the length of the data are summarized in the following table. For encryption, decryption, signatures and signature verification, the input and output data may begin at the same location in memory. In the table, *k* is the length in bytes of the RSA modulus.

Table 4, PKCS #1 v1.5 RSA: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt1 | RSA public key | ≤ *k*-11 | *k* | block type 02 |
| C\_Decrypt1 | RSA private key | *k* | ≤ *k*-11 | block type 02 |
| C\_Sign1 | RSA private key | ≤ *k*-11 | *k* | block type 01 |
| C\_SignRecover | RSA private key | ≤ *k*-11 | *k* | block type 01 |
| C\_Verify1 | RSA public key | ≤ *k*-11, *k*2 | N/A | block type 01 |
| C\_VerifyRecover | RSA public key | *k* | ≤ *k*-11 | block type 01 |
| C\_WrapKey | RSA public key | ≤ *k*-11 | *k* | block type 02 |
| C\_UnwrapKey | RSA private key | *k* | ≤ *k*-11 | block type 02 |

1 Single-part operations only.

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### PKCS #1 RSA OAEP mechanism parameters

1. CK\_RSA\_PKCS\_MGF\_TYPE; CK\_RSA\_PKCS\_MGF\_TYPE\_PTR

**CK\_RSA\_PKCS\_MGF\_TYPE**  is used to indicate the Message Generation Function (MGF) applied to a message block when formatting a message block for the PKCS #1 OAEP encryption scheme or the PKCS #1 PSS signature scheme. It is defined as follows:

typedef CK\_ULONG CK\_RSA\_PKCS\_MGF\_TYPE;

The following MGFs are defined in PKCS #1. The following table lists the defined functions.

Table 5, PKCS #1 Mask Generation Functions

|  |  |
| --- | --- |
| **Source Identifier** | **Value** |
| CKG\_MGF1\_SHA1 | 0x00000001UL |
| CKG\_MGF1\_SHA224 | 0x00000005UL |
| CKG\_MGF1\_SHA256 | 0x00000002UL |
| CKG\_MGF1\_SHA384 | 0x00000003UL |
| CKG\_MGF1\_SHA512 | 0x00000004UL |
| CKG\_MGF1\_SHA3\_224 | 0x00000006UL |
| CKG\_MGF1\_SHA3\_256 | 0x00000007UL |
| CKG\_MGF1\_SHA3\_384 | 0x00000008UL |
| CKG\_MGF1\_SHA3\_512 | 0x00000009UL |

**CK\_RSA\_PKCS\_MGF\_TYPE\_PTR** is a pointer to a **CK\_RSA\_PKCS\_ MGF\_TYPE**.

1. CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE; CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE\_PTR

**CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE**  is used to indicate the source of the encoding parameter when formatting a message block for the PKCS #1 OAEP encryption scheme. It is defined as follows:

typedef CK\_ULONG CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE;

The following encoding parameter sources are defined in PKCS #1. The following table lists the defined sources along with the corresponding data type for the *pSourceData* field in the **CK\_RSA\_PKCS\_OAEP\_PARAMS** structure defined below.

Table 6, PKCS #1 RSA OAEP: Encoding parameter sources

|  |  |  |
| --- | --- | --- |
| **Source Identifier** | **Value** | **Data Type** |
| CKZ\_DATA\_SPECIFIED | 0x00000001UL | Array of CK\_BYTE containing the value of the encoding parameter. If the parameter is empty, *pSourceData* must be NULL and *ulSourceDataLen* must be zero. |

**CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE\_PTR** is a pointer to a **CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE**.

1. CK\_RSA\_PKCS\_OAEP\_PARAMS; CK\_RSA\_PKCS\_OAEP\_PARAMS\_PTR

**CK\_RSA\_PKCS\_OAEP\_PARAMS** is a structure that provides the parameters to the **CKM\_RSA\_PKCS\_OAEP** mechanism. The structure is defined as follows:

typedef struct CK\_RSA\_PKCS\_OAEP\_PARAMS {

CK\_MECHANISM\_TYPE hashAlg;

CK\_RSA\_PKCS\_MGF\_TYPE mgf;

CK\_RSA\_PKCS\_OAEP\_SOURCE\_TYPE source;

CK\_VOID\_PTR pSourceData;

CK\_ULONG ulSourceDataLen;

} CK\_RSA\_PKCS\_OAEP\_PARAMS;

The fields of the structure have the following meanings:

hashAlg mechanism ID of the message digest algorithm used to calculate the digest of the encoding parameter

mgf mask generation function to use on the encoded block

source source of the encoding parameter

pSourceData data used as the input for the encoding parameter source

ulSourceDataLen length of the encoding parameter source input

**CK\_RSA\_PKCS\_OAEP\_PARAMS\_PTR** is a pointer to a **CK\_RSA\_PKCS\_OAEP\_PARAMS**.

Table 7, Add to following to table 7 ( PKCS #1 Mask Generation Functions) in section 2,1..7 (PKCS #1 RSA OAEP mechanism parameters)

|  |  |
| --- | --- |
| **Source Identifier** | **Value** |
| CKG\_MGF1\_SHA3\_224 | 0x00000006UL |
| CKG\_MGF1\_SHA3\_256 | 0x00000007UL |
| CKG\_MGF1\_SHA3\_384 | 0x00000008UL |
| CKG\_MGF1\_SHA3\_512 | 0x00000009UL |

### PKCS #1 RSA OAEP

The PKCS #1 RSA OAEP mechanism, denoted **CKM\_RSA\_PKCS\_OAEP**, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1. It supports single-part encryption and decryption; key wrapping; and key unwrapping.

It has a parameter, a **CK\_RSA\_PKCS\_OAEP\_PARAMS** structure.

This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the “input” to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type or any other information about the key, except the key length; the application must convey these separately. In particular, the mechanism contributes only the **CKA\_CLASS** and **CKA\_VALUE** (and **CKA\_VALUE\_LEN**, if the key has it) attributes to the recovered key during unwrapping; other attributes must be specified in the template.

Constraints on key types and the length of the data are summarized in the following table. For encryption and decryption, the input and output data may begin at the same location in memory. In the table, *k* is the length in bytes of the RSA modulus, and *hLen* is the output length of the message digest algorithm specified by the *hashAlg* field of the **CK\_RSA\_PKCS\_OAEP\_PARAMS** structure.

Table 8, PKCS #1 RSA OAEP: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Encrypt1 | RSA public key | ≤ *k*-2-2*hLen* | *k* |
| C\_Decrypt1 | RSA private key | *k* | ≤ *k*-2-2*hLen* |
| C\_WrapKey | RSA public key | ≤ *k*-2-2*hLen* | *k* |
| C\_UnwrapKey | RSA private key | *k* | ≤ *k*-2-2*hLen* |

1 Single-part operations only.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### PKCS #1 RSA PSS mechanism parameters

1. CK\_RSA\_PKCS\_PSS\_PARAMS; CK\_RSA\_PKCS\_PSS\_PARAMS\_PTR

**CK\_RSA\_PKCS\_PSS\_PARAMS** is a structure that provides the parameters to the **CKM\_RSA\_PKCS\_PSS** mechanism. The structure is defined as follows:

typedef struct CK\_RSA\_PKCS\_PSS\_PARAMS {

CK\_MECHANISM\_TYPE hashAlg;

CK\_RSA\_PKCS\_MGF\_TYPE mgf;

CK\_ULONG sLen;

} CK\_RSA\_PKCS\_PSS\_PARAMS;

The fields of the structure have the following meanings:

hashAlg hash algorithm used in the PSS encoding; if the signature mechanism does not include message hashing, then this value must be the mechanism used by the application to generate the message hash; if the signature mechanism includes hashing, then this value must match the hash algorithm indicated by the signature mechanism

mgf mask generation function to use on the encoded block

sLen length, in bytes, of the salt value used in the PSS encoding; typical values are the length of the message hash and zero

**CK\_RSA\_PKCS\_PSS\_PARAMS\_PTR** is a pointer to a **CK\_RSA\_PKCS\_PSS\_PARAMS**.

### PKCS #1 RSA PSS

The PKCS #1 RSA PSS mechanism, denoted **CKM\_RSA\_PKCS\_PSS**, is a mechanism based on the RSA public-key cryptosystem and the PSS block format defined in PKCS #1. It supports single-part signature generation and verification without message recovery. This mechanism corresponds only to the part of PKCS #1 that involves block formatting and RSA, given a hash value; it does not compute a hash value on the message to be signed.

It has a parameter, a **CK\_RSA\_PKCS\_PSS\_PARAMS** structure. The *sLen* field must be less than or equal to *k\**-2-*hLen* and *hLen* is the length of the input to the C\_Sign or C\_Verify function. *k\** is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of 8, in which case *k\** is one less than the length in bytes of the RSA modulus.

Constraints on key types and the length of the data are summarized in the following table. In the table, *k* is the length in bytes of the RSA.

Table 9, PKCS #1 RSA PSS: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | RSA private key | *hLen* | *k* |
| C\_Verify1 | RSA public key | *hLen*, *k* | N/A |

1 Single-part operations only.

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### ISO/IEC 9796 RSA

The ISO/IEC 9796 RSA mechanism, denoted **CKM\_RSA\_9796**, is a mechanism for single-part signatures and verification with and without message recovery based on the RSA public-key cryptosystem and the block formats defined in ISO/IEC 9796 and its annex A.

This mechanism processes only byte strings, whereas ISO/IEC 9796 operates on bit strings. Accordingly, the following transformations are performed:

* Data is converted between byte and bit string formats by interpreting the most-significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8).
* A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; it is converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

This mechanism does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus.

Table 10, ISO/IEC 9796 RSA: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | RSA private key | ≤ ⎣*k*/2⎦ | *k* |
| C\_SignRecover | RSA private key | ≤ ⎣*k*/2⎦ | *k* |
| C\_Verify1 | RSA public key | ≤ ⎣*k*/2⎦, *k*2 | N/A |
| C\_VerifyRecover | RSA public key | *k* | ≤ ⎣*k*/2⎦ |

1 Single-part operations only.

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### X.509 (raw) RSA

The X.509 (raw) RSA mechanism, denoted **CKM\_RSA\_X\_509**, is a multi-purpose mechanism based on the RSA public-key cryptosystem. It supports single-part encryption and decryption; single-part signatures and verification with and without message recovery; key wrapping; and key unwrapping. All these operations are based on so-called “raw” RSA, as assumed in X.509.

“Raw” RSA as defined here encrypts a byte string by converting it to an integer, most-significant byte first, applying “raw” RSA exponentiation, and converting the result to a byte string, most-significant byte first. The input string, considered as an integer, must be less than the modulus; the output string is also less than the modulus.

This mechanism does not have a parameter.

This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the “input” to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type, key length, or any other information about the key; the application must convey these separately, and supply them when unwrapping the key.

Unfortunately, X.509 does not specify how to perform padding for RSA encryption. For this mechanism, padding should be performed by prepending plaintext data with 0-valued bytes. In effect, to encrypt the sequence of plaintext bytes b1 b2 … bn (n ≤ *k*), Cryptoki forms P=2n-1b1+2n-2b2+…+bn. This number must be less than the RSA modulus. The *k*-byte ciphertext (*k* is the length in bytes of the RSA modulus) is produced by raising P to the RSA public exponent modulo the RSA modulus. Decryption of a *k*-byte ciphertext C is accomplished by raising C to the RSA private exponent modulo the RSA modulus, and returning the resulting value as a sequence of exactly *k* bytes. If the resulting plaintext is to be used to produce an unwrapped key, then however many bytes are specified in the template for the length of the key are taken *from the end* of this sequence of bytes.

Technically, the above procedures may differ very slightly from certain details of what is specified in X.509.

Executing cryptographic operations using this mechanism can result in the error returns CKR\_DATA\_INVALID (if plaintext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus) and CKR\_ENCRYPTED\_DATA\_INVALID (if ciphertext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus).

Constraints on key types and the length of input and output data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus.

Table 11, X.509 (Raw) RSA: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Encrypt1 | RSA public key | ≤ *k* | *k* |
| C\_Decrypt1 | RSA private key | *k* | *k* |
| C\_Sign1 | RSA private key | ≤ *k* | *k* |
| C\_SignRecover | RSA private key | ≤ *k* | *k* |
| C\_Verify1 | RSA public key | ≤ *k*, *k*2 | N/A |
| C\_VerifyRecover | RSA public key | *k* | *k* |
| C\_WrapKey | RSA public key | ≤ *k* | *k* |
| C\_UnwrapKey | RSA private key | *k* | ≤ *k* (specified in template) |

1 Single-part operations only.

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

This mechanism is intended for compatibility with applications that do not follow the PKCS #1 or ISO/IEC 9796 block formats.

### ANSI X9.31 RSA

The ANSI X9.31 RSA mechanism, denoted **CKM\_RSA\_X9\_31**, is a mechanism for single-part signatures and verification without message recovery based on the RSA public-key cryptosystem and the block formats defined in ANSI X9.31.

This mechanism applies the header and padding fields of the hash encapsulation. The trailer field must be applied by the application.

This mechanism processes only byte strings, whereas ANSI X9.31 operates on bit strings. Accordingly, the following transformations are performed:

* Data is converted between byte and bit string formats by interpreting the most-significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8).
* A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; it is converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

This mechanism does not have a parameter.

Constraints on key types and the length of input and output data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus. For all operations, the *k* value must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

Table 12, ANSI X9.31 RSA: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | RSA private key | ≤ *k*-2 | *k* |
| C\_Verify1 | RSA public key | ≤ *k*-2, *k*2 | N/A |

1 Single-part operations only.

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPE-MD 128 or RIPE-MD 160

The PKCS #1 v1.5 RSA signature with MD2 mechanism, denoted **CKM\_MD2\_RSA\_PKCS**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described initially in PKCS #1 v1.5 with the object identifier md2WithRSAEncryption, and as in the scheme RSASSA-PKCS1-v1\_5 in the current version of PKCS #1, where the underlying hash function is MD2.

Similarly, the PKCS #1 v1.5 RSA signature with MD5 mechanism, denoted **CKM\_MD5\_RSA\_PKCS**, performs the same operations described in PKCS #1 with the object identifier md5WithRSAEncryption. The PKCS #1 v1.5 RSA signature with SHA-1 mechanism, denoted **CKM\_SHA1\_RSA\_PKCS**, performs the same operations, except that it uses the hash function SHA-1 with object identifier sha1WithRSAEncryption.

Likewise, the PKCS #1 v1.5 RSA signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted **CKM\_SHA256\_RSA\_PKCS**, **CKM\_SHA384\_RSA\_PKCS**, and **CKM\_SHA512\_RSA\_PKCS** respectively, perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions with the object identifiers sha256WithRSAEncryption, sha384WithRSAEncryption and sha512WithRSAEncryption respectively.

The PKCS #1 v1.5 RSA signature with RIPEMD-128 or RIPEMD-160, denoted **CKM\_RIPEMD128\_RSA\_PKCS** and **CKM\_RIPEMD160\_RSA\_PKCS** respectively, perform the same operations using the RIPE-MD 128 and RIPE-MD 160 hash functions.

None of these mechanisms has a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus. For the PKCS #1 v1.5 RSA signature with MD2 and PKCS #1 v1.5 RSA signature with MD5 mechanisms, *k* must be at least 27; for the PKCS #1 v1.5 RSA signature with SHA-1 mechanism, *k* must be at least 31, and so on for other underlying hash functions, where the minimum is always 11 bytes more than the length of the hash value.

Table 13, PKCS #1 v1.5 RSA Signatures with Various Hash Functions: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Sign | RSA private key | any | *k* | block type 01 |
| C\_Verify | RSA public key | any, *k*2 | N/A | block type 01 |

2 Data length, signature length.

For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### PKCS #1 v1.5 RSA signature with SHA-224

The PKCS #1 v1.5 RSA signature with SHA-224 mechanism, denoted **CKM\_SHA224\_RSA\_PKCS,** performs similarly as the other **CKM\_SHA*X*\_RSA\_PKCS** mechanisms but uses the SHA-224 hash function.

### PKCS #1 RSA PSS signature with SHA-224

The PKCS #1 RSA PSS signature with SHA-224 mechanism, denoted **CKM\_SHA224\_RSA\_PKCS\_PSS**, performs similarly as the other **CKM\_SHA*X*\_RSA\_PSS** mechanisms but uses the SHA-224 hash function.

### PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512

The PKCS #1 RSA PSS signature with SHA-1 mechanism, denoted **CKM\_SHA1\_RSA\_PKCS\_PSS**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in PKCS #1 with the object identifier id-RSASSA-PSS, i.e., as in the scheme RSASSA-PSS in PKCS #1 where the underlying hash function is SHA-1.

The PKCS #1 RSA PSS signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted **CKM\_SHA256\_RSA\_PKCS\_PSS**, **CKM\_SHA384\_RSA\_PKCS\_PSS**, and **CKM\_SHA512\_RSA\_PKCS\_PSS** respectively, perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions.

The mechanisms have a parameter, a **CK\_RSA\_PKCS\_PSS\_PARAMS** structure. The *sLen* field must be less than or equal to *k\**-2-*hLen* where *hLen* is the length in bytes of the hash value. *k\** is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of 8, in which case *k\** is one less than the length in bytes of the RSA modulus.

Constraints on key types and the length of the data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus.

Table 14, PKCS #1 RSA PSS Signatures with Various Hash Functions: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | RSA private key | any | *k* |
| C\_Verify | RSA public key | any, *k*2 | N/A |

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### PKCS #1 v1.5 RSA signature with SHA3

The PKCS #1 v1.5 RSA signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms, denoted **CKM\_SHA3\_224\_RSA\_PKCS**, **CKM\_SHA3\_256\_RSA\_PKCS**, **CKM\_SHA3\_384\_RSA\_PKCS**, and **CKM\_SHA3\_512\_RSA\_PKCS** respectively,performs similarly as the other **CKM\_SHA*X*\_RSA\_PKCS** mechanisms but uses the corresponding SHA3 hash functions.

### PKCS #1 RSA PSS signature with SHA3

The PKCS #1 RSA PSS signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms, denoted **CKM\_SHA3\_224\_RSA\_PSS**, **CKM\_SHA3\_256\_RSA\_PSS**, **CKM\_SHA3\_384\_RSA\_PSS**, and **CKM\_SHA3\_512\_RSA\_PSS** respectively, performs similarly as the other **CKM\_SHA*X*\_RSA\_PSS** mechanisms but uses the corresponding SHA-3 hash functions.

### ANSI X9.31 RSA signature with SHA-1

The ANSI X9.31 RSA signature with SHA-1 mechanism, denoted **CKM\_SHA1\_RSA\_X9\_31**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in ANSI X9.31.

This mechanism does not have a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus. For all operations, the *k* value must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

Table 15, ANSI X9.31 RSA Signatures with SHA-1: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | RSA private key | any | *k* |
| C\_Verify | RSA public key | any, *k*2 | N/A |

2 Data length, signature length.

For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA

The TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA mechanism, denoted **CKM\_RSA\_PKCS\_TPM\_1\_1**, is a multi-use mechanism based on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5, with additional formatting rules defined in TCPA TPM Specification Version 1.1b. Additional formatting rules remained the same in TCG TPM Specification 1.2 The mechanism supports single-part encryption and decryption; key wrapping; and key unwrapping.

This mechanism does not have a parameter. It differs from the standard PKCS#1 v1.5 RSA encryption mechanism in that the plaintext is wrapped in a TCPA\_BOUND\_DATA (TPM\_BOUND\_DATA for TPM 1.2) structure before being submitted to the PKCS#1 v1.5 encryption process. On encryption, the version field of the TCPA\_BOUND\_DATA (TPM\_BOUND\_DATA for TPM 1.2) structure must contain 0x01, 0x01, 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be accepted.

This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the “input” to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type or any other information about the key, except the key length; the application must convey these separately. In particular, the mechanism contributes only the **CKA\_CLASS** and **CKA\_VALUE** (and **CKA\_VALUE\_LEN**, if the key has it) attributes to the recovered key during unwrapping; other attributes must be specified in the template.

Constraints on key types and the length of the data are summarized in the following table. For encryption and decryption, the input and output data may begin at the same location in memory. In the table, *k* is the length in bytes of the RSA modulus.

Table 16, TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Encrypt1 | RSA public key | ≤ *k*-11-5 | *k* |
| C\_Decrypt1 | RSA private key | *k* | ≤ *k*-11-5 |
| C\_WrapKey | RSA public key | ≤ *k*-11-5 | *k* |
| C\_UnwrapKey | RSA private key | *k* | ≤ *k*-11-5 |

1 Single-part operations only.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP

The TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP mechanism, denoted **CKM\_RSA\_PKCS\_OAEP\_TPM\_1\_1**, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1, with additional formatting defined in TCPA TPM Specification Version 1.1b. Additional formatting rules remained the same in TCG TPM Specification 1.2. The mechanism supports single-part encryption and decryption; key wrapping; and key unwrapping.

This mechanism does not have a parameter. It differs from the standard PKCS#1 OAEP RSA encryption mechanism in that the plaintext is wrapped in a TCPA\_BOUND\_DATA (TPM\_BOUND\_DATA for TPM 1.2) structure before being submitted to the encryption process and that all of the values of the parameters that are passed to a standard CKM\_RSA\_PKCS\_OAEP operation are fixed. On encryption, the version field of the TCPA\_BOUND\_DATA (TPM\_BOUND\_DATA for TPM 1.2) structure must contain 0x01, 0x01, 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be accepted.

This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the “input” to the encryption operation is the value of the **CKA\_VALUE** attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type or any other information about the key, except the key length; the application must convey these separately. In particular, the mechanism contributes only the **CKA\_CLASS** and **CKA\_VALUE** (and **CKA\_VALUE\_LEN**, if the key has it) attributes to the recovered key during unwrapping; other attributes must be specified in the template.

Constraints on key types and the length of the data are summarized in the following table. For encryption and decryption, the input and output data may begin at the same location in memory. In the table, *k* is the length in bytes of the RSA modulus.

Table 17, TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Encrypt1 | RSA public key | ≤ *k*-2-40-5 | *k* |
| C\_Decrypt1 | RSA private key | *k* | ≤ *k*-2-40-5 |
| C\_WrapKey | RSA public key | ≤ *k*-2-40-5 | *k* |
| C\_UnwrapKey | RSA private key | *k* | ≤ *k*-2-40-5 |

1 Single-part operations only.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of RSA modulus sizes, in bits.

### RSA AES KEY WRAP

The RSA AES key wrap mechanism, denoted **CKM\_RSA\_AES\_KEY\_WRAP**, is a mechanism based on the RSA public-key cryptosystem and the AES key wrap mechanism. It supports single-part key wrapping; and key unwrapping.

It has a parameter, a **CK\_RSA\_AES\_KEY\_WRAP\_PARAMS** structure.

The mechanism can wrap and unwrap a target asymmetric key of any length and type using an RSA key.

* A temporary AES key is used for wrapping the target key using CKM\_AES\_KEY\_WRAP\_KWP mechanism.
* The temporary AES key is wrapped with the wrapping RSA key using CKM\_RSA\_PKCS\_OAEP mechanism.

For wrapping, the mechanism -

* Generates a temporary random AES key of *ulAESKeyBits* length. This key is not accessible to the user - no handle is returned.
* Wraps the AES key with the wrapping RSA key using **CKM\_RSA\_PKCS\_OAEP** with parameters of *OAEPParams*.
* Wraps the target key with the temporary AES key using **CKM\_AES\_KEY\_WRAP\_KWP** ([AES KEYWRAP] section 6.3).
* Zeroizes the temporary AES key
* Concatenates two wrapped keys and outputs the concatenated blob. The first is the wrapped AES key, and the second is the wrapped target key.

The recommended format for an asymmetric target key being wrapped is as a PKCS8 PrivateKeyInfo

The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown

For unwrapping, the mechanism -

* Splits the input into two parts. The first is the wrapped AES key, and the second is the wrapped target key. The length of the first part is equal to the length of the unwrapping RSA key.
* Un-wraps the temporary AES key from the first part with the private RSA key using **CKM\_RSA\_PKCS\_OAEP** with parameters of *OAEPParams*.
* Un-wraps the target key from the second part with the temporary AES key using **CKM\_AES\_KEY\_WRAP\_KWP** ([AES KEYWRAP] section 6.3).
* Zeroizes the temporary AES key.
* Returns the handle to the newly unwrapped target key.

*Table 18, CKM\_RSA\_AES\_KEY\_WRAP Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_RSA\_AES\_KEY\_WRAP |  |  |  |  |  |  |  |
| 1SR = SignRecover, VR = VerifyRecover | | | | | | | |

### RSA AES KEY WRAP mechanism parameters

1. CK\_RSA\_AES\_KEY\_WRAP\_PARAMS; CK\_RSA\_AES\_KEY\_WRAP\_PARAMS\_PTR

**CK\_RSA\_AES\_KEY\_WRAP\_PARAMS** is a structure that provides the parameters to the **CKM\_RSA\_AES\_KEY\_WRAP** mechanism.  It is defined as follows:

typedef struct CK\_RSA\_AES\_KEY\_WRAP\_PARAMS {

CK\_ULONG ulAESKeyBits;

CK\_RSA\_PKCS\_OAEP\_PARAMS\_PTR pOAEPParams;

} CK\_RSA\_AES\_KEY\_WRAP\_PARAMS;

The fields of the structure have the following meanings:

ulAESKeyBits length of the temporary AES key in bits. Can be only 128, 192 or 256.

pOAEPParams pointer to the parameters of the temporary AES key wrapping. See also the description of PKCS #1 RSA OAEP mechanism parameters.

**CK\_RSA\_AES\_KEY\_WRAP\_PARAMS\_PTR** is a pointer to a **CK\_RSA\_AES\_KEY\_WRAP\_PARAMS**.

### FIPS 186-4

When CKM\_RSA\_PKCS is operated in FIPS mode, the length of the modulus SHALL only be 1024, 2048, or 3072 bits.

## DSA

*Table 19, DSA Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_DSA\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_DSA\_PARAMETER\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_DSA\_PROBABALISTIC\_PARAMETER\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_DSA\_FIPS\_G\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_DSA |  | ✓2 |  |  |  |  |  |
| CKM\_DSA\_SHA1 |  | ✓ |  |  |  |  |  |
| CKM\_DSA\_SHA224 |  | ✓ |  |  |  |  |  |
| CKM\_DSA\_SHA256 |  | ✓ |  |  |  |  |  |
| CKM\_DSA\_SHA384 |  | ✓ |  |  |  |  |  |
| CKM\_DSA\_SHA512 |  | ✓ |  |  |  |  |  |
| CKM\_DSA\_SHA3\_224 |  |  |  |  |  |  |  |
| CKM\_DSA\_SHA3\_256 |  |  |  |  |  |  |  |
| CKM\_DSA\_SHA3\_384 |  |  |  |  |  |  |  |
| CKM\_DSA\_SHA3\_512 |  |  |  |  |  |  |  |

### Definitions

This section defines the key type “CKK\_DSA” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of DSA key objects.

Mechanisms:

CKM\_DSA\_KEY\_PAIR\_GEN

CKM\_DSA

CKM\_DSA\_SHA1

CKM\_DSA\_SHA224

CKM\_DSA\_SHA256

CKM\_DSA\_SHA384

CKM\_DSA\_SHA512

CKM\_DSA\_SHA3\_224

CKM\_DSA\_SHA3\_256

CKM\_DSA\_SHA3\_384

CKM\_DSA\_SHA3\_512

CKM\_DSA\_PARAMETER\_GEN

CKM\_DSA\_PROBABLISTIC\_PARAMETER\_GEN

CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN

CKM\_DSA\_FIPS\_G\_GEN

1. CK\_DSA\_PARAMETER\_GEN\_PARAM

CK\_DSA\_PARAMETER\_GEN\_PARAM is a structure which provides and returns parameters for the NIST FIPS 186-4 parameter generating algorithms.

typedef struct CK\_DSA\_PARAMETER\_GEN\_PARAM {

CK\_MECHANISM\_TYPE hash;

CK\_BYTE\_PTR pSeed;

CK\_ULONG ulSeedLen;

CK\_ULONG ulIndex;

} CK\_DSA\_PARAMETER\_GEN\_PARAM;

The fields of the structure have the following meanings:

hash Mechanism value for the base hash used in PQG generation, Valid values are CKM\_SHA1, CKM\_SHA224, CKM\_SHA256, CKM\_SHA384, CKM\_SHA512.

pSeed Seed value used to generate PQ and G. This value is returned by CKM\_DSA\_PROBABLISTIC\_PARAMETER\_GEN, CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN, and passed into CKM\_DSA\_FIPS\_G\_GEN.

ulSeedLen Length of seed value.

ulIndex Index value for generating G. Input for CKM\_DSA\_FIPS\_G\_GEN. Ignored by CKM\_DSA\_PROBABALISTIC\_PARAMETER\_GEN and CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN.

### DSA public key objects

DSA public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_DSA**) hold DSA public keys. The following table defines the DSA public key object attributes, in addition to the common attributes defined for this object class:

Table 20, DSA Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PRIME1,3 | Big integer | Prime *p* (512 to 3072 bits, in steps of 64 bits) |
| CKA\_SUBPRIME1,3 | Big integer | Subprime *q* (160, 224 bits, or 256 bits) |
| CKA\_BASE1,3 | Big integer | Base *g* |
| CKA\_VALUE1,4 | Big integer | Public value *y* |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_PRIME**, **CKA\_SUBPRIME** and **CKA\_BASE** attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-4 for more information on DSA keys.

The following is a sample template for creating a DSA public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DSA;

CK\_UTF8CHAR label[] = “A DSA public key object”;

CK\_BYTE prime[] = {...};

CK\_BYTE subprime[] = {...};

CK\_BYTE base[] = {...};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_PRIME, prime, sizeof(prime)},

{CKA\_SUBPRIME, subprime, sizeof(subprime)},

{CKA\_BASE, base, sizeof(base)},

{CKA\_VALUE, value, sizeof(value)}

};

### DSA Key Restrictions

FIPS PUB 186-4 specifies permitted combinations of prime and sub-prime lengths. They are:

* Prime: 1024 bits, Subprime: 160
* Prime: 2048 bits, Subprime: 224
* Prime: 2048 bits, Subprime: 256
* Prime: 3072 bits, Subprime: 256

Earlier versions of FIPS 186 permitted smaller prime lengths, and those are included here for backwards compatibility. An implementation that is compliant to FIPS 186-4 does not permit the use of primes of any length less than 1024 bits.

### DSA private key objects

DSA private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_DSA**) hold DSA private keys. The following table defines the DSA private key object attributes, in addition to the common attributes defined for this object class:

Table 21, DSA Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PRIME1,4,6 | Big integer | Prime *p* (512 to 1024 bits, in steps of 64 bits) |
| CKA\_SUBPRIME1,4,6 | Big integer | Subprime *q* (160 bits, 224 bits, or 256 bits) |
| CKA\_BASE1,4,6 | Big integer | Base *g* |
| CKA\_VALUE1,4,6,7 | Big integer | Private value *x* |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_PRIME**, **CKA\_SUBPRIME** and **CKA\_BASE** attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-4 for more information on DSA keys.

Note that when generating a DSA private key, the DSA domain parameters are *not* specified in the key’s template. This is because DSA private keys are only generated as part of a DSA key *pair*, and the DSA domain parameters for the pair are specified in the template for the DSA public key.

The following is a sample template for creating a DSA private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DSA;

CK\_UTF8CHAR label[] = “A DSA private key object”;

CK\_BYTE subject[] = {...};

CK\_BYTE id[] = {123};

CK\_BYTE prime[] = {...};

CK\_BYTE subprime[] = {...};

CK\_BYTE base[] = {...};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_SIGN, &true, sizeof(true)},

{CKA\_PRIME, prime, sizeof(prime)},

{CKA\_SUBPRIME, subprime, sizeof(subprime)},

{CKA\_BASE, base, sizeof(base)},

{CKA\_VALUE, value, sizeof(value)}

};

### DSA domain parameter objects

DSA domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS,** key type **CKK\_DSA**) hold DSA domain parameters. The following table defines the DSA domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 22, DSA Domain Parameter Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PRIME1,4 | Big integer | Prime *p* (512 to 1024 bits, in steps of 64 bits) |
| CKA\_SUBPRIME1,4 | Big integer | Subprime *q* (160 bits, 224 bits, or 256 bits) |
| CKA\_BASE1,4 | Big integer | Base *g* |
| CKA\_PRIME\_BITS2,3 | CK\_ULONG | Length of the prime value. |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_PRIME**, **CKA\_SUBPRIME** and **CKA\_BASE** attribute values are collectively the “DSA domain parameters”. See FIPS PUB 186-4 for more information on DSA domain parameters.

To ensure backwards compatibility, if **CKA\_SUBPRIME\_BITS** is not specified for a call to **C\_GenerateKey**, it takes on a default based on the value of **CKA\_PRIME\_BITS** as follows:

* If **CKA\_PRIME\_BITS** is less than or equal to 1024 then CKA\_SUBPRIME\_BITS shall be 160 bits
* If **CKA\_PRIME\_BITS** equals 2048 then CKA\_SUBPRIME\_BITS shall be 224 bits
* If **CKA\_PRIME\_BITS** equals 3072 then CKA\_SUBPRIME\_BITS shall be 256 bits

The following is a sample template for creating a DSA domain parameter object:

CK\_OBJECT\_CLASS class = CKO\_DOMAIN\_PARAMETERS;

CK\_KEY\_TYPE keyType = CKK\_DSA;

CK\_UTF8CHAR label[] = “A DSA domain parameter object”;

CK\_BYTE prime[] = {...};

CK\_BYTE subprime[] = {...};

CK\_BYTE base[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_PRIME, prime, sizeof(prime)},

{CKA\_SUBPRIME, subprime, sizeof(subprime)},

{CKA\_BASE, base, sizeof(base)},

};

### DSA key pair generation

The DSA key pair generation mechanism, denoted **CKM\_DSA\_KEY\_PAIR\_GEN**, is a key pair generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.

This mechanism does not have a parameter.

The mechanism generates DSA public/private key pairs with a particular prime, subprime and base, as specified in the **CKA\_PRIME**, **CKA\_SUBPRIME**, and **CKA\_BASE** attributes of the template for the public key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_SUBPRIME**, **CKA\_BASE**, and **CKA\_VALUE** attributes to the new private key. Other attributes supported by the DSA public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of DSA prime sizes, in bits.

### DSA domain parameter generation

The DSA domain parameter generation mechanism, denoted **CKM\_DSA\_PARAMETER\_GEN**, is a domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.

This mechanism does not have a parameter.

The mechanism generates DSA domain parameters with a particular prime length in bits, as specified in the **CKA\_PRIME\_BITS** attribute of the template.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_SUBPRIME**, **CKA\_BASE** and **CKA\_PRIME\_BITS** attributes to the new object. Other attributes supported by the DSA domain parameter types may also be specified in the template, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of DSA prime sizes, in bits.

### DSA probabilistic domain parameter generation

The DSA probabilistic domain parameter generation mechanism, denoted **CKM\_DSA\_PROBABLISTIC\_PARAMETER\_GEN**, is a domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.1 Generation and Validation of Probable Primes..

This mechanism takes a **CK\_DSA\_PARAMETER\_GEN\_PARAM** which supplies the base hash and returns the seed (pSeed) and the length (ulSeedLen).

The mechanism generates DSA the prime and subprime domain parameters with a particular prime length in bits, as specified in the **CKA\_PRIME\_BITS** attribute of the template and the subprime length as specified in the **CKA\_SUBPRIME\_BITS** attribute of the template.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_SUBPRIME**, **CKA\_PRIME\_BITS, and CKA\_SUBPRIME\_BITS** attributes to the new object. **CKA\_BASE** is not set by this call. Other attributes supported by the DSA domain parameter types may also be specified in the template, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of DSA prime sizes, in bits.

### DSA Shawe-Taylor domain parameter generation

The DSA Shawe-Taylor domain parameter generation mechanism, denoted **CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN**, is a domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.2 Construction and Validation of Provable Primes p and q.

This mechanism takes a **CK\_DSA\_PARAMETER\_GEN\_PARAM** which supplies the base hash and returns the seed (pSeed) and the length (ulSeedLen).

The mechanism generates DSA the prime and subprime domain parameters with a particular prime length in bits, as specified in the CKA\_PRIME\_BITS attribute of the template and the subprime length as specified in the **CKA\_SUBPRIME\_BITS** attribute of the template.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_SUBPRIME**, **CKA\_PRIME\_BITS, and CKA\_SUBPRIME\_BITS** attributes to the new object. **CKA\_BASE** is not set by this call. Other attributes supported by the DSA domain parameter types may also be specified in the template, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of DSA prime sizes, in bits.

### DSA base domain parameter generation

The DSA base domain parameter generation mechanism, denoted **CKM\_DSA\_FIPS\_G\_GEN**, is a base parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.2 Generation of Generator G.

This mechanism takes a **CK\_DSA\_PARAMETER\_GEN\_PARAM** which supplies the base hash the seed (pSeed) and the length (ulSeedLen) and the index value.

The mechanism generates the DSA base with the domain parameter specified in the **CKA\_PRIME** and **CKA\_SUBPRIME** attributes of the template.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_BASE** attributes to the new object.Other attributes supported by the DSA domain parameter types may also be specified in the template, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of DSA prime sizes, in bits.

### DSA without hashing

The DSA without hashing mechanism, denoted **CKM\_DSA**, is a mechanism for single-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. (This mechanism corresponds only to the part of DSA that processes the 20-byte hash value; it does not compute the hash value.)

For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 23, DSA: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | DSA private key | 20, 28, 32, 48, or 64 bits | 2\*length of subprime |
| C\_Verify1 | DSA public key | (20, 28, 32, 48, or 64 bits), (2\*length of subprime)2 | N/A |

1 Single-part operations only.

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of DSA prime sizes, in bits.

### DSA with SHA-1

The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA1**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. This mechanism computes the entire DSA specification, including the hashing with SHA-1.

For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 24, DSA with SHA-1: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | DSA private key | any | 2\*subprime length |
| C\_Verify | DSA public key | any, 2\*subprime length2 | N/A |

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of DSA prime sizes, in bits.

### FIPS 186-4

When CKM\_DSA is operated in FIPS mode, only the following bit lengths of p and q, represented by L and N, SHALL be used:

L = 1024, N = 160

L = 2048, N = 224

L = 2048, N = 256

L = 3072, N = 256

### DSA with SHA-224

The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA224**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA-224.

For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 25, DSA with SHA-244: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | DSA private key | any | 2\*subprime length |
| C\_Verify | DSA public key | any, 2\*subprime length2 | N/A |

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of DSA prime sizes, in bits.

### DSA with SHA-256

The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA256**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA-256.

For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 26, DSA with SHA-256: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | DSA private key | any | 2\*subprime length |
| C\_Verify | DSA public key | any, 2\*subprime length2 | N/A |

2 Data length, signature length.

### DSA with SHA-384

The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA384**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA-384.

For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 27, DSA with SHA-384: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | DSA private key | any | 2\*subprime length |
| C\_Verify | DSA public key | any, 2\*subprime length2 | N/A |

2 Data length, signature length.

### DSA with SHA-512

The DSA with SHA-1 mechanism, denoted **CKM\_DSA\_SHA512**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA-512.

For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 28, DSA with SHA-512: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | DSA private key | any | 2\*subprime length |
| C\_Verify | DSA public key | any, 2\*subprime length2 | N/A |

2 Data length, signature length.

### DSA with SHA3-224

The DSA with SHA3-224 mechanism, denoted **CKM\_DSA\_SHA3\_224**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-224.

For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 29, DSA with SHA3-224: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | DSA private key | any | 2\*subprime length |
| C\_Verify | DSA public key | any, 2\*subprime length2 | N/A |

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of DSA prime sizes, in bits.

### DSA with SHA3-256

The DSA with SHA3-256 mechanism, denoted **CKM\_DSA\_SHA3\_256**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-256.

For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 30, DSA with SHA3-256: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | DSA private key | any | 2\*subprime length |
| C\_Verify | DSA public key | any, 2\*subprime length2 | N/A |

2 Data length, signature length.

### DSA with SHA3-384

The DSA with SHA3-384 mechanism, denoted **CKM\_DSA\_SHA3\_384**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-384.

For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 31, DSA with SHA3-384: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | DSA private key | any | 2\*subprime length |
| C\_Verify | DSA public key | any, 2\*subprime length2 | N/A |

2 Data length, signature length.

### DSA with SHA3-512

The DSA with SHA3-512 mechanism, denoted **CKM\_DSA\_SHA3-512**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SH3A-512.

For the purposes of this mechanism, a DSA signature is a string of length 2\*subprime, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 32, DSA with SHA3-512: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | DSA private key | any | 2\*subprime length |
| C\_Verify | DSA public key | any, 2\*subprime length2 | N/A |

2 Data length, signature length.

## Elliptic Curve

The Elliptic Curve (EC) cryptosystem (also related to ECDSA) in this document was originally based on the one described in the ANSI X9.62 and X9.63 standards developed by the ANSI X9F1 working group.

The EC cryptosystem developed by the ANSI X9F1 working group was created at a time when EC curves were always represented in their Weierstrass form. Since that time, new curves represented in Edwards form (RFC 8032) and Montgomery form (RFC 7748) have become more common. To support these new curves, the EC cryptosystem in this document has been extended from the original. Additional key generation mechanisms have been added as well as an additional signature generation mechanism.

*Table 33, Elliptic Curve Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_EC\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_EC\_KEY\_PAIR\_GEN\_W\_EXTRA\_BITS |  |  |  |  | ✓ |  |  |
| CKM\_EC\_EDWARDS\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_EC\_MONTGOMERY\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_ECDSA |  | ✓2 |  |  |  |  |  |
| CKM\_ECDSA\_SHA1 |  | ✓ |  |  |  |  |  |
| CKM\_ECDSA\_SHA224 |  |  |  |  |  |  |  |
| CKM\_ECDSA\_SHA256 |  |  |  |  |  |  |  |
| CKM\_ECDSA\_SHA384 |  |  |  |  |  |  |  |
| CKM\_ECDSA\_SHA512 |  |  |  |  |  |  |  |
| CKM\_ECDSA\_SHA3\_224 |  |  |  |  |  |  |  |
| CKM\_ECDSA\_SHA3\_256 |  |  |  |  |  |  |  |
| CKM\_ECDSA\_SHA3\_384 |  |  |  |  |  |  |  |
| CKM\_ECDSA\_SHA3\_512 |  |  |  |  |  |  |  |
| CKM\_EDDSA |  |  |  |  |  |  |  |
| CKM\_XEDDSA |  |  |  |  |  |  |  |
| CKM\_ECDH1\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_ECDH1\_COFACTOR\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_ECMQV\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_ECDH\_AES\_KEY\_WRAP |  |  |  |  |  |  |  |

Table 34, Mechanism Information Flags

|  |  |  |
| --- | --- | --- |
| CKF\_EC\_F\_P | 0x00100000UL | True if the mechanism can be used with EC domain parameters over *Fp* |
| CKF\_EC\_F\_2M | 0x00200000UL | True if the mechanism can be used with EC domain parameters over *F*2*m* |
| CKF\_EC\_ECPARAMETERS | 0x00400000UL | True if the mechanism can be used with EC domain parameters of the choice **ecParameters** |
| CKF\_EC\_OID | 0x00800000UL | True if the mechanism can be used with EC domain parameters of the choice **oId** |
| CKF\_EC\_UNCOMPRESS | 0x01000000UL | True if the mechanism can be used with elliptic curve point uncompressed |
| CKF\_EC\_COMPRESS | 0x02000000UL | True if the mechanism can be used with elliptic curve point compressed |
| CKF\_EC\_CURVENAME | 0x04000000UL | True of the mechanism can be used with EC domain parameters of the choice **curveName** |

Note: CKF\_EC\_NAMEDCURVE is deprecated with PKCS#11 3.00. It is replaced by CKF\_EC\_OID.

In these standards, there are two different varieties of EC defined:

1. EC using a field with an odd prime number of elements (i.e. the finite field *Fp*).
2. EC using a field of characteristic two (i.e. the finite field *F*2*m*).

An EC key in Cryptoki contains information about which variety of EC it is suited for. It is preferable that a Cryptoki library, which can perform EC mechanisms, be capable of performing operations with the two varieties of EC, however this is not required. The **CK\_MECHANISM\_INFO** structure **CKF\_EC\_F\_P** flag identifies a Cryptoki library supporting EC keys over *Fp* whereas the **CKF\_EC\_F\_2M** flag identifies a Cryptoki library supporting EC keys over *F*2*m*. A Cryptoki library that can perform EC mechanisms must set either or both of these flags for each EC mechanism.

In these specifications there are also four representation methods to define the domain parameters for an EC key. Only the **ecParameters,** the **oId** and the **curveName** choices are supported in Cryptoki. The **CK\_MECHANISM\_INFO** structure **CKF\_EC\_ECPARAMETERS** flag identifies a Cryptoki library supporting the **ecParameters** choice whereas the **CKF\_EC\_OID** flag identifies a Cryptoki library supporting the **oId** choice, and the **CKF\_EC\_CURVENAME** flag identifies a Cryptoki library supporting the **curveName** choice. A Cryptoki library that can perform EC mechanisms must set the appropriate flag(s) for each EC mechanism.

In these specifications, an EC public key (i.e. EC point *Q*) or the base point *G* when the **ecParameters** choice is used can be represented as an octet string of the uncompressed form or the compressed form. The **CK\_MECHANISM\_INFO** structure **CKF\_EC\_UNCOMPRESS** flag identifies a Cryptoki library supporting the uncompressed form whereas the **CKF\_EC\_COMPRESS** flag identifies a Cryptoki library supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or both of these flags for each EC mechanism.

Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation of domain parameters or one form may encounter difficulties achieving interoperability with other implementations.

If an attempt to create, generate, derive or unwrap an EC key of an unsupported curve is made, the attempt should fail with the error code CKR\_CURVE\_NOT\_SUPPORTED. If an attempt to create, generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain parameters is made, that attempt should fail with the error code CKR\_DOMAIN\_PARAMS\_INVALID. If an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that attempt should fail with the error code CKR\_TEMPLATE\_INCONSISTENT.

### EC Signatures

For the purposes of these mechanisms, an ECDSA signature is an octet string of even length which is at most two times *nLen* octets, where *nLen* is the length in octets of the base point order *n*. The signature octets correspond to the concatenation of the ECDSA values *r* and *s*, both represented as an octet string of equal length of at most *nLen* with the most significant byte first. If *r* and *s* have different octet length, the shorter of both must be padded with leading zero octets such that both have the same octet length. Loosely spoken, the first half of the signature is *r* and the second half is *s*. For signatures created by a token, the resulting signature is always of length 2*nLen*. For signatures passed to a token for verification, the signature may have a shorter length but must be composed as specified before.

If the length of the hash value is larger than the bit length of *n*, only the leftmost bits of the hash up to the length of *n* will be used. Any truncation is done by the token.

Note: For applications, it is recommended to encode the signature as an octet string of length two times *nLen* if possible. This ensures that the application works with PKCS#11 modules which have been implemented based on an older version of this document. Older versions required all signatures to have length two times *nLen*. It may be impossible to encode the signature with the maximum length of two times *nLen* if the application just gets the integer values of *r* and *s* (i.e. without leading zeros), but does not know the base point order *n*, because *r* and *s* can have any value between zero and the base point order *n*.

An EdDSA signature is an octet string of even length which is two times nLen octets, where nLen is calculated as EdDSA parameter b divided by 8. The signature octets correspond to the concatenation of the EdDSA values R and S as defined in [RFC 8032], both represented as an octet string of equal length of nLen bytes in little endian order.

### Definitions

This section defines the key type “CKK\_EC” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Note: CKK\_ECDSA is deprecated. It is replaced by CKK\_EC.

Mechanisms:

CKM\_EC\_KEY\_PAIR\_GEN

CKM\_EC\_EDWARDS\_KEY\_PAIR\_GEN

CKM\_EC\_MONTGOMERY\_KEY\_PAIR\_GEN

CKM\_ECDSA

CKM\_ECDSA\_SHA1

CKM\_ECDSA\_SHA224

CKM\_ECDSA\_SHA256

CKM\_ECDSA\_SHA384

CKM\_ECDSA\_SHA512

CKM\_ECDSA\_SHA3\_224

CKM\_ECDSA\_SHA3\_256

CKM\_ECDSA\_SHA3\_384

CKM\_ECDSA\_SHA3\_512

CKM\_EDDSA

CKM\_XEDDSA

CKM\_ECDH1\_DERIVE

CKM\_ECDH1\_COFACTOR\_DERIVE

CKM\_ECMQV\_DERIVE

CKM\_ECDH\_AES\_KEY\_WRAP

CKD\_NULL

CKD\_SHA1\_KDF

CKD\_SHA224\_KDF

CKD\_SHA256\_KDF

CKD\_SHA384\_KDF

CKD\_SHA512\_KDF

CKD\_SHA3\_224\_KDF

CKD\_SHA3\_256\_KDF

CKD\_SHA3\_384\_KDF

CKD\_SHA3\_512\_KDF

CKD\_SHA1\_KDF\_SP800

CKD\_SHA224\_KDF\_SP800

CKD\_SHA256\_KDF\_SP800

CKD\_SHA384\_KDF\_SP800

CKD\_SHA512\_KDF\_SP800

CKD\_SHA3\_224\_KDF\_SP800

CKD\_SHA3\_256\_KDF\_SP800

CKD\_SHA3\_384\_KDF\_SP800

CKD\_SHA3\_512\_KDF\_SP800

CKD\_BLAKE2B\_160\_KDF

CKD\_BLAKE2B\_256\_KDF

CKD\_BLAKE2B\_384\_KDF

CKD\_BLAKE2B\_512\_KDF

### ECDSA public key objects

EC (also related to ECDSA) public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_EC**) hold EC public keys. The following table defines the EC public key object attributes, in addition to the common attributes defined for this object class:

Table 35, Elliptic Curve Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,3 | Byte array | DER-encoding of an ANSI X9.62 Parameters value |
| CKA\_EC\_POINT1,4 | Byte array | DER-encoding of ANSI X9.62 ECPoint value *Q* |

- Refer to [PKCS11-Base] table 11 for footnotes

Note: CKA\_ECDSA\_PARAMS is deprecated. It is replaced by CKA\_EC\_PARAMS.

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

oId CURVES.&id({CurveNames}),

implicitlyCA NULL,

curveName PrintableString

}

This allows detailed specification of all required values using choice **ecParameters**, the use of **oId** as an object identifier substitute for a particular set of elliptic curve domain parameters, or **implicitlyCA** to indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oId** or **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.

The following is a sample template for creating an EC (ECDSA) public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “An EC public key object”;

CK\_BYTE ecParams[] = {...};

CK\_BYTE ecPoint[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_EC\_POINT, ecPoint, sizeof(ecPoint)}

};

### Elliptic curve private key objects

EC (also related to ECDSA) private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_EC**) hold EC private keys. See Section 2.3 for more information about EC. The following table defines the EC private key object attributes, in addition to the common attributes defined for this object class:

Table 36, Elliptic Curve Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,4,6 | Byte array | DER-encoding of an ANSI X9.62 Parameters value |
| CKA\_VALUE1,4,6,7 | Big integer | ANSI X9.62 private value *d* |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods with the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

oId CURVES.&id({CurveNames}),

implicitlyCA NULL,

curveName PrintableString

}

This allows detailed specification of all required values using choice **ecParameters**, the use of **oId** as an object identifier substitute for a particular set of elliptic curve domain parameters, or **implicitlyCA** to indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oId** or **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.Note that when generating an EC private key, the EC domain parameters are *not* specified in the key’s template. This is because EC private keys are only generated as part of an EC key *pair*, and the EC domain parameters for the pair are specified in the template for the EC public key.

The following is a sample template for creating an EC (ECDSA) private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “An EC private key object”;

CK\_BYTE subject[] = {...};

CK\_BYTE id[] = {123};

CK\_BYTE ecParams[] = {...};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_DERIVE, &true, sizeof(true)},

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_VALUE, value, sizeof(value)}

};

### Edwards Elliptic curve public key objects

Edwards EC public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_EC\_EDWARDS**) hold Edwards EC public keys. The following table defines the Edwards EC public key object attributes, in addition to the common attributes defined for this object class:

Table 37, Edwards Elliptic Curve Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,3 | Byte array | DER-encoding of a Parameters value as defined above |
| CKA\_EC\_POINT1,4 | Byte array | DER-encoding of the b-bit public key value in little endian order as defined in RFC 8032 |

- Refer to [PKCS #11-Base] table 11 for footnotes

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards and Montgomery Elliptic curves. The CKA\_EC\_PARAMS attribute has the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

oId CURVES.&id({CurveNames}),

implicitlyCA NULL,

curveName PrintableString

}

Edwards EC public keys only support the use of the **curveName** selection to specify a curve name as defined in [RFC 8032] and the use of the **oID** selection to specify a curve through an EdDSA algorithm as defined in [RFC 8410]. Note that keys defined by RFC 8032 and RFC 8410 are incompatible.

The following is a sample template for creating an Edwards EC public key object with Edwards25519 being specified as curveName:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “An Edwards EC public key object”;

CK\_BYTE ecParams[] = {0x13, 0x0c, 0x65, 0x64, 0x77, 0x61, 0x72, 0x64, 0x73, 0x32, 0x35, 0x35, 0x31, 0x39};

CK\_BYTE ecPoint[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_EC\_POINT, ecPoint, sizeof(ecPoint)}

};

### Edwards Elliptic curve private key objects

Edwards EC private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_EC\_EDWARDS**) hold Edwards EC private keys. See Section 2.3 for more information about EC. The following table defines the Edwards EC private key object attributes, in addition to the common attributes defined for this object class:

Table 38, Edwards Elliptic Curve Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,4,6 | Byte array | DER-encoding of a Parameters value as defined above |
| CKA\_VALUE1,4,6,7 | Big integer | b-bit private key value in little endian order as defined in RFC 8032 |

- Refer to [PKCS #11-Base] table 11 for footnotes

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards and Montgomery Elliptic curves. The CKA\_EC\_PARAMS attribute has the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

oId CURVES.&id({CurveNames}),

implicitlyCA NULL,

curveName PrintableString

}

Edwards EC private keys only support the use of the **curveName** selection to specify a curve name as defined in [RFC 8032] and the use of the **oID** selection to specify a curve through an EdDSA algorithm as defined in [RFC 8410]. Note that keys defined by RFC 8032 and RFC 8410 are incompatible.

Note that when generating an Edwards EC private key, the EC domain parameters are *not* specified in the key’s template. This is because Edwards EC private keys are only generated as part of an Edwards EC key *pair*, and the EC domain parameters for the pair are specified in the template for the Edwards EC public key.

The following is a sample template for creating an Edwards EC private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “An Edwards EC private key object”;

CK\_BYTE subject[] = {...};

CK\_BYTE id[] = {123};

CK\_BYTE ecParams[] = {...};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_DERIVE, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

### Montgomery Elliptic curve public key objects

Montgomery EC public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_EC\_MONTGOMERY**) hold Montgomery EC public keys. The following table defines the Montgomery EC public key object attributes, in addition to the common attributes defined for this object class:

Table 39, Montgomery Elliptic Curve Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,3 | Byte array | DER-encoding of a Parameters value as defined above |
| CKA\_EC\_POINT1,4 | Byte array | DER-encoding of the public key value in little endian order as defined in RFC 7748 |

- Refer to [PKCS #11-Base] table 11 for footnotes

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards and Montgomery Elliptic curves. The CKA\_EC\_PARAMS attribute has the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

oId CURVES.&id({CurveNames}),

implicitlyCA NULL,

curveName PrintableString

}

Montgomery EC public keys only support the use of the **curveName** selection to specify a curve name as defined in [RFC7748] and the use of the **oID** selection to specify a curve through an ECDH algorithm as defined in [RFC 8410]. Note that keys defined by RFC 7748 and RFC 8410 are incompatible.

The following is a sample template for creating a Montgomery EC public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “A Montgomery EC public key object”;

CK\_BYTE ecParams[] = {...};

CK\_BYTE ecPoint[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_EC\_PARAMS, ecParams, sizeof(ecParams)},

{CKA\_EC\_POINT, ecPoint, sizeof(ecPoint)}

};

### Montgomery Elliptic curve private key objects

Montgomery EC private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_EC\_MONTGOMERY**) hold Montgomery EC private keys. See Section 2.3 for more information about EC. The following table defines the Montgomery EC private key object attributes, in addition to the common attributes defined for this object class:

Table 40, Montgomery Elliptic Curve Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_EC\_PARAMS1,4,6 | Byte array | DER-encoding of a Parameters value as defined above |
| CKA\_VALUE1,4,6,7 | Big integer | Private key value in little endian order as defined in RFC 7748 |

- Refer to [PKCS #11-Base] table 11 for footnotes

The **CKA\_EC\_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards and Montgomery Elliptic curves. The CKA\_EC\_PARAMS attribute has the following syntax:

Parameters ::= CHOICE {

ecParameters ECParameters,

oId CURVES.&id({CurveNames}),

implicitlyCA NULL,

curveName PrintableString

}

Edwards EC private keys only support the use of the **curveName** selection to specify a curve name as defined in [RFC7748] and the use of the **oID** selection to specify a curve through an ECDH algorithm as defined in [RFC 8410]. Note that keys defined by RFC 7748 and RFC 8410 are incompatible.

Note that when generating a Montgomery EC private key, the EC domain parameters are *not* specified in the key’s template. This is because Montgomery EC private keys are only generated as part of a Montgomery EC key *pair*, and the EC domain parameters for the pair are specified in the template for the Montgomery EC public key.

The following is a sample template for creating a Montgomery EC private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_EC;

CK\_UTF8CHAR label[] = “A Montgomery EC private key object”;

CK\_BYTE subject[] = {...};

CK\_BYTE id[] = {123};

CK\_BYTE ecParams[] = {...};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_DERIVE, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

### Elliptic curve key pair generation

The EC (also related to ECDSA) key pair generation mechanism, denoted CKM\_EC\_KEY\_PAIR\_GEN, is a key pair generation mechanism that uses the method defined by the ANSI X9.62 and X9.63 standards.

The EC (also related to ECDSA) key pair generation mechanism, denoted CKM\_EC\_KEY\_PAIR\_GEN\_W\_EXTRA\_BITS, is a key pair generation mechanism that uses the method defined by FIPS 186-4 Appendix B.4.1.

These mechanisms do not have a parameter.

These mechanisms generate EC public/private key pairs with particular EC domain parameters, as specified in the **CKA\_EC\_PARAMS** attribute of the template for the public key. Note that this version of Cryptoki does not include a mechanism for generating these EC domain parameters.

These mechanism contribute the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_EC\_POINT** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_EC\_PARAMS** and **CKA\_VALUE** attributes to the new private key. Other attributes supported by the EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

### Edwards Elliptic curve key pair generation

The Edwards EC key pair generation mechanism, denoted **CKM\_EC\_EDWARDS\_KEY\_PAIR\_GEN**, is a key pair generation mechanism for EC keys over curves represented in Edwards form.

This mechanism does not have a parameter.

The mechanism can only generate EC public/private key pairs over the curves edwards25519 and edwards448 as defined in RFC 8032 or the curves id-Ed25519 and id-Ed448 as defined in RFC 8410. These curves can only be specified in the **CKA\_EC\_PARAMS** attribute of the template for the public key using the **curveName** or the oID methods. Attempts to generate keys over these curves using any other EC key pair generation mechanism will fail with CKR\_CURVE\_NOT\_SUPPORTED.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_EC\_POINT** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_EC\_PARAMS** and **CKA\_VALUE** attributes to the new private key. Other attributes supported by the Edwards EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### Montgomery Elliptic curve key pair generation

The Montgomery EC key pair generation mechanism, denoted **CKM\_EC\_MONTGOMERY\_KEY\_PAIR\_GEN**, is a key pair generation mechanism for EC keys over curves represented in Montgomery form.

This mechanism does not have a parameter.

The mechanism can only generate Montgomery EC public/private key pairs over the curves curve25519 and curve448 as defined in RFC 7748 or the curves id-X25519 and id-X448 as defined in RFC 8410. These curves can only be specified in the **CKA\_EC\_PARAMS** attribute of the template for the public key using the **curveName** or oId methods. Attempts to generate keys over these curves using any other EC key pair generation mechanism will fail with CKR\_CURVE\_NOT\_SUPPORTED.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_EC\_POINT** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_EC\_PARAMS** and **CKA\_VALUE** attributes to the new private key. Other attributes supported by the EC public and private key types (specifically, the flags indicating which functions the keys support) may also be specified in the templates for the keys, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 7748 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### ECDSA without hashing

Refer section 2.3.1 for signature encoding.

The ECDSA without hashing mechanism, denoted **CKM\_ECDSA**, is a mechanism for single-part signatures and verification for ECDSA. (This mechanism corresponds only to the part of ECDSA that processes the hash value, which should not be longer than 1024 bits; it does not compute the hash value.)

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 41, ECDSA without hashing: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | ECDSA private key | any3 | 2*nLen* |
| C\_Verify1 | ECDSA public key | any3, ≤2*nLen* 2 | N/A |

1 Single-part operations only.

2 Data length, signature length.

3 Input the entire raw digest. Internally, this will be truncated to the appropriate number of bits.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between 2200 and 2300 elements (inclusive), then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

### ECDSA with hashing

Refer to section 2.3.1 for signature encoding.

The ECDSA with SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanism, denoted **CKM\_ECDSA\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]** respectively, is a mechanism for single- and multiple-part signatures and verification for ECDSA. This mechanism computes the entire ECDSA specification, including the hashing with SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 42, ECDSA with hashing: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | ECDSA private key | any | 2*nLen* |
| C\_Verify | ECDSA public key | any, ≤2*nLen* 2 | N/A |

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

### EdDSA

The EdDSA mechanism, denoted **CKM\_EDDSA**, is a mechanism for single-part and multipart signatures and verification for EdDSA. This mechanism implements the five EdDSA signature schemes defined in RFC 8032 and RFC 8410.

For curves according to RFC 8032, this mechanism has an optional parameter, a **CK\_EDDSA\_PARAMS** structure. The absence or presence of the parameter as well as its content is used to identify which signature scheme is to be used. Table 32 enumerates the five signature schemes defined in RFC 8032 and all supported permutations of the mechanism parameter and its content.

Table 43, Mapping to RFC 8032 Signature Schemes

| **Signature Scheme** | **Mechanism Param** | **phFlag** | **Context Data** |
| --- | --- | --- | --- |
| Ed25519 | Not Required | N/A | N/A |
| Ed25519ctx | Required | False | Optional |
| Ed25519ph | Required | True | Optional |
| Ed448 | Required | False | Optional |
| Ed448ph | Required | True | Optional |

For curves according to RFC 8410, the mechanism is implicitly given by the curve, which is EdDSA in pure mode.

Constraints on key types and the length of data are summarized in the following table:

Table 44, EdDSA: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_EC\_EDWARDS private key | any | 2b*Len* |
| C\_Verify | CKK\_EC\_EDWARDS public key | any, ≤2b*Len* 2 | N/A |

2 Data length, signature length.

Note that for EdDSA in pure mode, Ed25519 and Ed448 the data must be processed twice. Therefore, a token might need to cache all the data, especially when used with C\_SignUpdate/C\_VerifyUpdate. If tokens are unable to do so they can return CKM\_TOKEN\_RESOURCE\_EXCEEDED.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK\_MECHANISM\_INFO structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 8032and RFC 8410 only define curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### XEdDSA

The XEdDSA mechanism, denoted **CKM\_XEDDSA**, is a mechanism for single-part signatures and verification for XEdDSA. This mechanism implements the XEdDSA signature scheme defined in **[XEDDSA]**. CKM\_XEDDSA operates on CKK\_EC\_MONTGOMERY type EC keys, which allows these keys to be used both for signing/verification and for Diffie-Hellman style key-exchanges. This double use is necessary for the Extended Triple Diffie-Hellman where the long-term identity key is used to sign short-term keys and also contributes to the DH key-exchange.

This mechanism has a parameter, a **CK\_XEDDSA\_PARAMS** structure.

Table 45, XEdDSA: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | CKK\_EC\_MONTGOMERY private key | any3 | 2b |
| C\_Verify1 | CKK\_EC\_MONTGOMERY public key | any3, 2b2 | N/A |

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as **[XEDDSA]** only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

### EC mechanism parameters

* **CK\_EDDSA\_PARAMS, CK\_EDDSA\_PARAMS\_PTR**

**CK\_EDDSA\_PARAMS** is a structure that provides the parameters for the **CKM\_EDDSA** signature mechanism. The structure is defined as follows:

typedef struct CK\_EDDSA\_PARAMS {

CK\_BBOOL phFlag;

CK\_ULONG ulContextDataLen;

CK\_BYTE\_PTR pContextData;

} CK\_EDDSA\_PARAMS;

The fields of the structure have the following meanings:

phFlag a Boolean value which indicates if Prehashed variant of EdDSA should used

ulContextDataLen the length in bytes of the context data where 0 <= ulContextDataLen <= 255.

pContextData context data shared between the signer and verifier

**CK\_EDDSA\_PARAMS\_PTR** is a pointer to a **CK\_EDDSA\_PARAMS**.

* **CK\_XEDDSA\_PARAMS, CK\_XEDDSA\_PARAMS\_PTR**

**CK\_XEDDSA\_PARAMS** is a structure that provides the parameters for the **CKM\_XEDDSA** signature mechanism. The structure is defined as follows:

typedef struct CK\_XEDDSA\_PARAMS {

CK\_XEDDSA\_HASH\_TYPE hash;

} CK\_XEDDSA\_PARAMS;

The fields of the structure have the following meanings:

hash a Hash mechanism to be used by the mechanism.

**CK\_XEDDSA\_PARAMS\_PTR** is a pointer to a **CK\_XEDDSA\_PARAMS**.

* **CK\_XEDDSA\_HASH\_TYPE, CK\_XEDDSA\_HASH\_TYPE\_PTR**

**CK\_XEDDSA\_HASH\_TYPE** is used to indicate the hash function used in XEDDSA. It is defined as follows:

typedef CK\_ULONG CK\_XEDDSA\_HASH\_TYPE;

The following table lists the defined functions.

Table 46, EC: Key Derivation Functions

|  |
| --- |
| **Source Identifier** |
| CKM\_BLAKE2B\_256 |
| CKM\_BLAKE2B\_512 |
| CKM\_SHA3\_256 |
| CKM\_SHA3\_512 |
| CKM\_SHA256 |
| CKM\_SHA512 |

**CK\_XEDDSA\_HASH\_TYPE\_PTR** is a pointer to a **CK\_XEDDSA\_HASH\_TYPE**.

* **CK\_EC\_KDF\_TYPE, CK\_EC\_KDF\_TYPE\_PTR**

**CK\_EC\_KDF\_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data from a shared secret. The key derivation function will be used by the EC key agreement schemes. It is defined as follows:

typedef CK\_ULONG CK\_EC\_KDF\_TYPE;

The following table lists the defined functions.

Table 47, EC: Key Derivation Functions

|  |
| --- |
| **Source Identifier** |
| CKD\_NULL |
| CKD\_SHA1\_KDF |
| CKD\_SHA224\_KDF |
| CKD\_SHA256\_KDF |
| CKD\_SHA384\_KDF |
| CKD\_SHA512\_KDF |
| CKD\_SHA3\_224\_KDF |
| CKD\_SHA3\_256\_KDF |
| CKD\_SHA3\_384\_KDF |
| CKD\_SHA3\_512\_KDF |
| CKD\_SHA1\_KDF\_SP800 |
| CKD\_SHA224\_KDF\_SP800 |
| CKD\_SHA256\_KDF\_SP800 |
| CKD\_SHA384\_KDF\_SP800 |
| CKD\_SHA512\_KDF\_SP800 |
| CKD\_SHA3\_224\_KDF\_SP800 |
| CKD\_SHA3\_256\_KDF\_SP800 |
| CKD\_SHA3\_384\_KDF\_SP800 |
| CKD\_SHA3\_512\_KDF\_SP800 |
| CKD\_BLAKE2B\_160\_KDF |
| CKD\_BLAKE2B\_256\_KDF |
| CKD\_BLAKE2B\_384\_KDF |
| CKD\_BLAKE2B\_512\_KDF |

The key derivation function **CKD\_NULL** produces a raw shared secret value without applying any key derivation function.

The key derivation functions **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF**, which arebased on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively, derive keying data from the shared secret value as defined in [ANSI X9.63].

The key derivation functions **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF\_SP800**, which arebased on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section 5.8.1.1.

The key derivation functions **CKD\_BLAKE2B\_[160|256|384|512]\_KDF**, which arebased on the Blake2b family of hashes, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section 5.8.1.1.**CK\_EC\_KDF\_TYPE\_PTR** is a pointer to a **CK\_EC\_KDF\_TYPE**.

* **CK\_ECDH1\_DERIVE\_PARAMS, CK\_ECDH1\_DERIVE\_PARAMS\_PTR**

**CK\_ECDH1\_DERIVE\_PARAMS** is a structure that provides the parameters for the **CKM\_ECDH1\_DERIVE** and **CKM\_ECDH1\_COFACTOR\_DERIVE** key derivation mechanisms, where each party contributes one key pair. The structure is defined as follows:

typedef struct CK\_ECDH1\_DERIVE\_PARAMS {

CK\_EC\_KDF\_TYPE kdf;

CK\_ULONG ulSharedDataLen;

CK\_BYTE\_PTR pSharedData;

CK\_ULONG ulPublicDataLen;

CK\_BYTE\_PTR pPublicData;

} CK\_ECDH1\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

kdf key derivation function used on the shared secret value

ulSharedDataLen the length in bytes of the shared info

pSharedData some data shared between the two parties

ulPublicDataLen the length in bytes of the other party’s EC public key

pPublicData[[1]](#footnote-1) pointer to other party’s EC public key value. A token MUST be able to accept this value encoded as a raw octet string (as per section A.5.2 of [ANSI X9.62]). A token MAY, in addition, support accepting this value as a DER-encoded ECPoint (as per section E.6 of [ANSI X9.62]) i.e. the same as a CKA\_EC\_POINT encoding. The calling application is responsible for converting the offered public key to the compressed or uncompressed forms of these encodings if the token does not support the offered form.

With the key derivation function **CKD\_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation functions **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF, CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF\_SP800**, an optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

**CK\_ECDH1\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_ECDH1\_DERIVE\_PARAMS**.

* **CK\_ECDH2\_DERIVE\_PARAMS, CK\_ECDH2\_DERIVE\_PARAMS\_PTR**

**CK\_ECDH2\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_ECMQV\_DERIVE** key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

typedef struct CK\_ECDH2\_DERIVE\_PARAMS {

CK\_EC\_KDF\_TYPE kdf;

CK\_ULONG ulSharedDataLen;

CK\_BYTE\_PTR pSharedData;

CK\_ULONG ulPublicDataLen;

CK\_BYTE\_PTR pPublicData;

CK\_ULONG ulPrivateDataLen;

CK\_OBJECT\_HANDLE hPrivateData;

CK\_ULONG ulPublicDataLen2;

CK\_BYTE\_PTR pPublicData2;

} CK\_ECDH2\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

kdf key derivation function used on the shared secret value

ulSharedDataLen the length in bytes of the shared info

pSharedData some data shared between the two parties

ulPublicDataLen the length in bytes of the other party’s first EC public key

pPublicData pointer to other party’s first EC public key value. Encoding rules are as per pPublicData of CK\_ECDH1\_DERIVE\_PARAMS

ulPrivateDataLen the length in bytes of the second EC private key

hPrivateData key handle for second EC private key value

ulPublicDataLen2 the length in bytes of the other party’s second EC public key

pPublicData2 pointer to other party’s second EC public key value. Encoding rules are as per pPublicData of CK\_ECDH1\_DERIVE\_PARAMS

With the key derivation function **CKD\_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation function **CKD\_SHA1\_KDF**, an optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

**CK\_ECDH2\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_ECDH2\_DERIVE\_PARAMS**.

* **CK\_ECMQV\_DERIVE\_PARAMS, CK\_ECMQV\_DERIVE\_PARAMS\_PTR**

**CK\_ECMQV\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_ECMQV\_DERIVE** key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

typedef struct CK\_ECMQV\_DERIVE\_PARAMS {

CK\_EC\_KDF\_TYPE kdf;

CK\_ULONG ulSharedDataLen;

CK\_BYTE\_PTR pSharedData;

CK\_ULONG ulPublicDataLen;

CK\_BYTE\_PTR pPublicData;

CK\_ULONG ulPrivateDataLen;

CK\_OBJECT\_HANDLE hPrivateData;

CK\_ULONG ulPublicDataLen2;

CK\_BYTE\_PTR pPublicData2;

CK\_OBJECT\_HANDLE publicKey;

} CK\_ECMQV\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

kdf key derivation function used on the shared secret value

ulSharedDataLen the length in bytes of the shared info

pSharedData some data shared between the two parties

ulPublicDataLen the length in bytes of the other party’s first EC public key

pPublicData pointer to other party’s first EC public key value. Encoding rules are as per pPublicData of CK\_ECDH1\_DERIVE\_PARAMS

ulPrivateDataLen the length in bytes of the second EC private key

hPrivateData key handle for second EC private key value

ulPublicDataLen2 the length in bytes of the other party’s second EC public key

pPublicData2 pointer to other party’s second EC public key value. Encoding rules are as per pPublicData of CK\_ECDH1\_DERIVE\_PARAMS

publicKey Handle to the first party’s ephemeral public key

With the key derivation function **CKD\_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be zero. With the key derivation functions **CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF, CKD\_[SHA1|SHA224|SHA384|SHA512|SHA3\_224|SHA3\_256|SHA3\_384|SHA3\_512]\_KDF\_SP800**, an optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

**CK\_ECMQV\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_ECMQV\_DERIVE\_PARAMS**.

### Elliptic curve Diffie-Hellman key derivation

The elliptic curve Diffie-Hellman (ECDH) key derivation mechanism, denoted **CKM\_ECDH1\_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes one key pair all using the same EC domain parameters.

It has a parameter, a **CK\_ECDH1\_DERIVE\_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

Constraints on key types are summarized in the following table:

Table : ECDH: Allowed Key Types

| **Function** | **Key type** |
| --- | --- |
| C\_Derive | CKK\_EC or CKK\_EC\_MONTGOMERY |

### Elliptic curve Diffie-Hellman with cofactor key derivation

The elliptic curve Diffie-Hellman (ECDH) with cofactor key derivation mechanism, denoted **CKM\_ECDH1\_COFACTOR\_DERIVE**, is a mechanism for key derivation based on the cofactor Diffie-Hellman version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes one key pair all using the same EC domain parameters. Cofactor multiplication is computationally efficient and helps to prevent security problems like small group attacks.

It has a parameter, a **CK\_ECDH1\_DERIVE\_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

Constraints on key types are summarized in the following table:

Table : ECDH with cofactor: Allowed Key Types

| **Function** | **Key type** |
| --- | --- |
| C\_Derive | CKK\_EC |

### Elliptic curve Menezes-Qu-Vanstone key derivation

The elliptic curve Menezes-Qu-Vanstone (ECMQV) key derivation mechanism, denoted **CKM\_ECMQV\_DERIVE**, is a mechanism for key derivation based the MQV version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes two key pairs all using the same EC domain parameters.

It has a parameter, a **CK\_ECMQV\_DERIVE\_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2200 and 2300 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2200 consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2300 is a 301-bit number).

Constraints on key types are summarized in the following table:

Table : ECDH MQV: Allowed Key Types

| **Function** | **Key type** |
| --- | --- |
| C\_Derive | CKK\_EC |

### ECDH AES KEY WRAP

The ECDH AES KEY WRAP mechanism, denoted **CKM\_ECDH\_AES\_KEY\_WRAP**, is a mechanism based on elliptic curve public-key crypto-system and the AES key wrap mechanism. It supports single-part key wrapping; and key unwrapping.

It has a parameter, a **CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS** structure.

The mechanism can wrap and unwrap an asymmetric target key of any length and type using an EC key.

* A temporary AES key is derived from a temporary EC key and the wrapping EC key using the **CKM\_ECDH1\_DERIVE** mechanism.
* The derived AES key is used for wrapping the target key using the **CKM\_AES\_KEY\_WRAP\_KWP** mechanism.

For wrapping, the mechanism -

* Generates a temporary random EC key (transport key) having the same parameters as the wrapping EC key (and domain parameters). Saves the transport key public key material.
* Performs ECDH operation using **CKM\_ECDH1\_DERIVE** with parameters of kdf, ulSharedDataLen and pSharedData using the private key of the transport EC key and the public key of wrapping EC key and gets the first ulAESKeyBits bits of the derived key to be the temporary AES key.
* Wraps the target key with the temporary AES key using **CKM\_AES\_KEY\_WRAP\_KWP (**[AES KEYWRAP] section 6.3).
* Zeroizes the temporary AES key and EC transport private key.
* Concatenates public key material of the transport key and output the concatenated blob. The first part is the public key material of the transport key and the second part is the wrapped target key.

The recommended format for an asymmetric target key being wrapped is as a PKCS8 PrivateKeyInfo

The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown.

For unwrapping, the mechanism -

* Splits the input into two parts. The first part is the public key material of the transport key and the second part is the wrapped target key. The length of the first part is equal to the length of the public key material of the unwrapping EC key.

*Note: since the transport key and the wrapping EC key share the same domain, the length of the public key material of the transport key is the same length of the public key material of the unwrapping EC key.*

* Performs ECDH operation using **CKM\_ECDH1\_DERIVE** with parameters of kdf, ulSharedDataLen and pSharedData using the private part of unwrapping EC key and the public part of the transport EC key and gets first ulAESKeyBits bits of the derived key to be the temporary AES key.
* Un-wraps the target key from the second part with the temporary AES key using **CKM\_AES\_KEY\_WRAP\_KWP** **(**[AES KEYWRAP] section 6.3).
* Zeroizes the temporary AES key.

*Table 51, CKM\_ECDH\_AES\_KEY\_WRAP Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_ECDH\_AES\_KEY\_WRAP |  |  |  |  |  | ✓ |  |
| 1SR = SignRecover, VR = VerifyRecover | | | | | | | |

Constraints on key types are summarized in the following table:

Table : ECDH AES Key Wrap: Allowed Key Types

| **Function** | **Key type** |
| --- | --- |
| C\_Derive | CKK\_EC or CKK\_EC\_MONTGOMERY |

### ECDH AES KEY WRAP mechanism parameters

1. CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS; CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS\_PTR

**CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS** is a structure that provides the parameters to the **CKM\_ECDH\_AES\_KEY\_WRAP** mechanism. It is defined as follows:

typedef struct CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS {

CK\_ULONG ulAESKeyBits;

CK\_EC\_KDF\_TYPE kdf;

CK\_ULONG ulSharedDataLen;

CK\_BYTE\_PTR pSharedData;

} CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS;

The fields of the structure have the following meanings:

ulAESKeyBits length of the temporary AES key in bits. Can be only 128, 192 or 256.

kdf key derivation function used on the shared secret value to generate AES key.

ulSharedDataLen the length in bytes of the shared info

pSharedData Some data shared between the two parties

**CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS\_PTR** is a pointer to a **CK\_ECDH\_AES\_KEY\_WRAP\_PARAMS**.

### FIPS 186-4

When CKM\_ECDSA is operated in FIPS mode, the curves SHALL either be NIST recommended curves (with a fixed set of domain parameters) or curves with domain parameters generated as specified by ANSI X9.64. The NIST recommended curves are:

P-192, P-224, P-256, P-384, P-521

K-163, B-163, K-233, B-233

K-283, B-283, K-409, B-409

K-571, B-571

## Diffie-Hellman

*Table 53, Diffie-Hellman Mechanisms vs. Functions*

|  | Functions | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_DH\_PKCS\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_DH\_PKCS\_PARAMETER\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_DH\_PKCS\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_X9\_42\_DH\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_X9\_42\_DH\_PKCS\_PARAMETER\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_X9\_42\_DH\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_X9\_42\_DH\_HYBRID\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_X9\_42\_MQV\_DERIVE |  |  |  |  |  |  | ✓ |

### Definitions

This section defines the key type “CKK\_DH” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of [DH] key objects.

Mechanisms:

CKM\_DH\_PKCS\_KEY\_PAIR\_GEN

CKM\_DH\_PKCS\_DERIVE

CKM\_X9\_42\_DH\_KEY\_PAIR\_GEN

CKM\_X9\_42\_DH\_DERIVE

CKM\_X9\_42\_DH\_HYBRID\_DERIVE

CKM\_X9\_42\_MQV\_DERIVE

CKM\_DH\_PKCS\_PARAMETER\_GEN

CKM\_X9\_42\_DH\_PARAMETER\_GEN

### Diffie-Hellman public key objects

Diffie-Hellman public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_DH**) hold Diffie-Hellman public keys. The following table defines the Diffie-Hellman public key object attributes, in addition to the common attributes defined for this object class:

Table 54, Diffie-Hellman Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PRIME1,3 | Big integer | Prime *p* |
| CKA\_BASE1,3 | Big integer | Base *g* |
| CKA\_VALUE1,4 | Big integer | Public value *y* |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_PRIME** and **CKA\_BASE** attribute values are collectively the “Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman keys.

The following is a sample template for creating a Diffie-Hellman public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DH;

CK\_UTF8CHAR label[] = “A Diffie-Hellman public key object”;

CK\_BYTE prime[] = {...};

CK\_BYTE base[] = {...};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_PRIME, prime, sizeof(prime)},

{CKA\_BASE, base, sizeof(base)},

{CKA\_VALUE, value, sizeof(value)}

};

### X9.42 Diffie-Hellman public key objects

X9.42 Diffie-Hellman public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_X9\_42\_DH**) hold X9.42 Diffie-Hellman public keys. The following table defines the X9.42 Diffie-Hellman public key object attributes, in addition to the common attributes defined for this object class:

Table 55, X9.42 Diffie-Hellman Public Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PRIME1,3 | Big integer | Prime *p* (≥ 1024 bits, in steps of 256 bits) |
| CKA\_BASE1,3 | Big integer | Base *g* |
| CKA\_SUBPRIME1,3 | Big integer | Subprime *q* (≥ 160 bits) |
| CKA\_VALUE1,4 | Big integer | Public value *y* |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_PRIME, CKA\_BASE** and **CKA\_SUBPRIME** attribute values are collectively the “X9.42 Diffie-Hellman domain parameters”. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman keys.

The following is a sample template for creating a X9.42 Diffie-Hellman public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_X9\_42\_DH;

CK\_UTF8CHAR label[] = “A X9.42 Diffie-Hellman public key object”;

CK\_BYTE prime[] = {...};

CK\_BYTE base[] = {...};

CK\_BYTE subprime[] = {...};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_PRIME, prime, sizeof(prime)},

{CKA\_BASE, base, sizeof(base)},

{CKA\_SUBPRIME, subprime, sizeof(subprime)},

{CKA\_VALUE, value, sizeof(value)}

};

### Diffie-Hellman private key objects

Diffie-Hellman private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_DH**) hold Diffie-Hellman private keys. The following table defines the Diffie-Hellman private key object attributes, in addition to the common attributes defined for this object class:

Table 56, Diffie-Hellman Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PRIME1,4,6 | Big integer | Prime *p* |
| CKA\_BASE1,4,6 | Big integer | Base *g* |
| CKA\_VALUE1,4,6,7 | Big integer | Private value *x* |
| CKA\_VALUE\_BITS2,6 | CK\_ULONG | Length in bits of private value *x* |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_PRIME** and **CKA\_BASE** attribute values are collectively the “Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman keys.

Note that when generating a Diffie-Hellman private key, the Diffie-Hellman parameters are *not* specified in the key’s template. This is because Diffie-Hellman private keys are only generated as part of a Diffie-Hellman key *pair*, and the Diffie-Hellman parameters for the pair are specified in the template for the Diffie-Hellman public key.

The following is a sample template for creating a Diffie-Hellman private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DH;

CK\_UTF8CHAR label[] = “A Diffie-Hellman private key object”;

CK\_BYTE subject[] = {...};

CK\_BYTE id[] = {123};

CK\_BYTE prime[] = {...};

CK\_BYTE base[] = {...};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_DERIVE, &true, sizeof(true)},

{CKA\_PRIME, prime, sizeof(prime)},

{CKA\_BASE, base, sizeof(base)},

{CKA\_VALUE, value, sizeof(value)}

};

### X9.42 Diffie-Hellman private key objects

X9.42 Diffie-Hellman private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_X9\_42\_DH**) hold X9.42 Diffie-Hellman private keys. The following table defines the X9.42 Diffie-Hellman private key object attributes, in addition to the common attributes defined for this object class:

Table 57, X9.42 Diffie-Hellman Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PRIME1,4,6 | Big integer | Prime *p* (≥ 1024 bits, in steps of 256 bits) |
| CKA\_BASE1,4,6 | Big integer | Base *g* |
| CKA\_SUBPRIME1,4,6 | Big integer | Subprime *q* (≥ 160 bits) |
| CKA\_VALUE1,4,6,7 | Big integer | Private value *x* |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_PRIME, CKA\_BASE** and **CKA\_SUBPRIME** attribute values are collectively the “X9.42 Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the key components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman keys.

Note that when generating a X9.42 Diffie-Hellman private key, the X9.42 Diffie-Hellman domain parameters are *not* specified in the key’s template. This is because X9.42 Diffie-Hellman private keys are only generated as part of a X9.42 Diffie-Hellman key *pair*, and the X9.42 Diffie-Hellman domain parameters for the pair are specified in the template for the X9.42 Diffie-Hellman public key.

The following is a sample template for creating a X9.42 Diffie-Hellman private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_X9\_42\_DH;

CK\_UTF8CHAR label[] = “A X9.42 Diffie-Hellman private key object”;

CK\_BYTE subject[] = {...};

CK\_BYTE id[] = {123};

CK\_BYTE prime[] = {...};

CK\_BYTE base[] = {...};

CK\_BYTE subprime[] = {...};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_DERIVE, &true, sizeof(true)},

{CKA\_PRIME, prime, sizeof(prime)},

{CKA\_BASE, base, sizeof(base)},

{CKA\_SUBPRIME, subprime, sizeof(subprime)},

{CKA\_VALUE, value, sizeof(value)}

};

### Diffie-Hellman domain parameter objects

Diffie-Hellman domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS,** key type **CKK\_DH**) hold Diffie-Hellman domain parameters. The following table defines the Diffie-Hellman domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 58, Diffie-Hellman Domain Parameter Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PRIME1,4 | Big integer | Prime *p* |
| CKA\_BASE1,4 | Big integer | Base *g* |
| CKA\_PRIME\_BITS2,3 | CK\_ULONG | Length of the prime value. |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_PRIME** and **CKA\_BASE** attribute values are collectively the “Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman domain parameters.

The following is a sample template for creating a Diffie-Hellman domain parameter object:

CK\_OBJECT\_CLASS class = CKO\_DOMAIN\_PARAMETERS;

CK\_KEY\_TYPE keyType = CKK\_DH;

CK\_UTF8CHAR label[] = “A Diffie-Hellman domain parameters object”;

CK\_BYTE prime[] = {...};

CK\_BYTE base[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_PRIME, prime, sizeof(prime)},

{CKA\_BASE, base, sizeof(base)},

};

### X9.42 Diffie-Hellman domain parameters objects

X9.42 Diffie-Hellman domain parameters objects (object class **CKO\_DOMAIN\_PARAMETERS,** key type **CKK\_X9\_42\_DH**) hold X9.42 Diffie-Hellman domain parameters. The following table defines the X9.42 Diffie-Hellman domain parameters object attributes, in addition to the common attributes defined for this object class:

Table 59, X9.42 Diffie-Hellman Domain Parameters Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PRIME1,4 | Big integer | Prime *p* (≥ 1024 bits, in steps of 256 bits) |
| CKA\_BASE1,4 | Big integer | Base *g* |
| CKA\_SUBPRIME1,4 | Big integer | Subprime *q* (≥ 160 bits) |
| CKA\_PRIME\_BITS2,3 | CK\_ULONG | Length of the prime value. |
| CKA\_SUBPRIME\_BITS2,3 | CK\_ULONG | Length of the subprime value. |

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA\_PRIME**, **CKA\_BASE** and **CKA\_SUBPRIME** attribute values are collectively the “X9.42 Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the domain parameters components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman domain parameters.

The following is a sample template for creating a X9.42 Diffie-Hellman domain parameters object:

CK\_OBJECT\_CLASS class = CKO\_DOMAIN\_PARAMETERS;

CK\_KEY\_TYPE keyType = CKK\_X9\_42\_DH;

CK\_UTF8CHAR label[] = “A X9.42 Diffie-Hellman domain parameters object”;

CK\_BYTE prime[] = {...};

CK\_BYTE base[] = {...};

CK\_BYTE subprime[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_PRIME, prime, sizeof(prime)},

{CKA\_BASE, base, sizeof(base)},

{CKA\_SUBPRIME, subprime, sizeof(subprime)},

};

### PKCS #3 Diffie-Hellman key pair generation

The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted **CKM\_DH\_PKCS\_KEY\_PAIR\_GEN**, is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in PKCS #3. This is what PKCS #3 calls “phase I”. It does not have a parameter.

The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as specified in the **CKA\_PRIME** and **CKA\_BASE** attributes of the template for the public key. If the **CKA\_VALUE\_BITS** attribute of the private key is specified, the mechanism limits the length in bits of the private value, as described in PKCS #3.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_BASE**, and **CKA\_VALUE** (and the **CKA\_VALUE\_BITS** attribute, if it is not already provided in the template) attributes to the new private key; other attributes required by the Diffie-Hellman public and private key types must be specified in the templates.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Diffie-Hellman prime sizes, in bits.

### PKCS #3 Diffie-Hellman domain parameter generation

The PKCS #3 Diffie-Hellman domain parameter generation mechanism, denoted **CKM\_DH\_PKCS\_PARAMETER\_GEN**, is a domain parameter generation mechanism based on Diffie-Hellman key agreement, as defined in PKCS #3.

It does not have a parameter.

The mechanism generates Diffie-Hellman domain parameters with a particular prime length in bits, as specified in the **CKA\_PRIME\_BITS** attribute of the template.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_BASE,** and **CKA\_PRIME\_BITS** attributes to the new object. Other attributes supported by the Diffie-Hellman domain parameter types may also be specified in the template, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Diffie-Hellman prime sizes, in bits.

### PKCS #3 Diffie-Hellman key derivation

The PKCS #3 Diffie-Hellman key derivation mechanism, denoted **CKM\_DH\_PKCS\_DERIVE**, is a mechanism for key derivation based on Diffie-Hellman key agreement, as defined in PKCS #3. This is what PKCS #3 calls “phase II”.

It has a parameter, which is the public value of the other party in the key agreement protocol, represented as a Cryptoki “Big integer” (*i.e.*, a sequence of bytes, most-significant byte first).

This mechanism derives a secret key from a Diffie-Hellman private key and the public value of the other party. It computes a Diffie-Hellman secret value from the public value and private key according to PKCS #3, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

This mechanism has the following rules about key sensitivity and extractability[[2]](#footnote-2):

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Diffie-Hellman prime sizes, in bits.

### X9.42 Diffie-Hellman mechanism parameters

* **CK\_X9\_42\_DH\_KDF\_TYPE, CK\_X9\_42\_DH\_KDF\_TYPE\_PTR**

**CK\_X9\_42\_DH\_KDF\_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data from a shared secret. The key derivation function will be used by the X9.42 Diffie-Hellman key agreement schemes. It is defined as follows:

typedef CK\_ULONG CK\_X9\_42\_DH\_KDF\_TYPE;

The following table lists the defined functions.

Table 60, X9.42 Diffie-Hellman Key Derivation Functions

|  |
| --- |
| **Source Identifier** |
| CKD\_NULL |
| CKD\_SHA1\_KDF\_ASN1 |
| CKD\_SHA1\_KDF\_CONCATENATE |

The key derivation function **CKD\_NULL** produces a raw shared secret value without applying any key derivation function whereas the key derivation functions **CKD\_SHA1\_KDF\_ASN1** and **CKD\_SHA1\_KDF\_CONCATENATE**, which are both based on SHA-1, derive keying data from the shared secret value as defined in the ANSI X9.42 standard.

**CK\_X9\_42\_DH\_KDF\_TYPE\_PTR** is a pointer to a **CK\_X9\_42\_DH\_KDF\_TYPE**.

1. CK\_X9\_42\_DH1\_DERIVE\_PARAMS, CK\_X9\_42\_DH1\_DERIVE\_PARAMS\_PTR

**CK\_X9\_42\_DH1\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_X9\_42\_DH\_DERIVE** key derivation mechanism, where each party contributes one key pair. The structure is defined as follows:

typedef struct CK\_X9\_42\_DH1\_DERIVE\_PARAMS {

CK\_X9\_42\_DH\_KDF\_TYPE kdf;

CK\_ULONG ulOtherInfoLen;

CK\_BYTE\_PTR pOtherInfo;

CK\_ULONG ulPublicDataLen;

CK\_BYTE\_PTR pPublicData;

} CK\_X9\_42\_DH1\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

kdf key derivation function used on the shared secret value

ulOtherInfoLen the length in bytes of the other info

pOtherInfo some data shared between the two parties

ulPublicDataLen the length in bytes of the other party’s X9.42 Diffie-Hellman public key

pPublicData pointer to other party’s X9.42 Diffie-Hellman public key value

With the key derivation function **CKD\_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero. With the key derivation function **CKD\_SHA1\_KDF\_ASN1**, *pOtherInfo* must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function **CKD\_SHA1\_KDF\_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.

**CK\_X9\_42\_DH1\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_X9\_42\_DH1\_DERIVE\_PARAMS**.

* CK\_X9\_42\_DH2\_DERIVE\_PARAMS, CK\_X9\_42\_DH2\_DERIVE\_PARAMS\_PTR

**CK\_X9\_42\_DH2\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_X9\_42\_DH\_HYBRID\_DERIVE** and **CKM\_X9\_42\_MQV\_DERIVE** key derivation mechanisms, where each party contributes two key pairs. The structure is defined as follows:

typedef struct CK\_X9\_42\_DH2\_DERIVE\_PARAMS {

CK\_X9\_42\_DH\_KDF\_TYPE kdf;

CK\_ULONG ulOtherInfoLen;

CK\_BYTE\_PTR pOtherInfo;

CK\_ULONG ulPublicDataLen;

CK\_BYTE\_PTR pPublicData;

CK\_ULONG ulPrivateDataLen;

CK\_OBJECT\_HANDLE hPrivateData;

CK\_ULONG ulPublicDataLen2;

CK\_BYTE\_PTR pPublicData2;

} CK\_X9\_42\_DH2\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

kdf key derivation function used on the shared secret value

ulOtherInfoLen the length in bytes of the other info

pOtherInfo some data shared between the two parties

ulPublicDataLen the length in bytes of the other party’s first X9.42 Diffie-Hellman public key

pPublicData pointer to other party’s first X9.42 Diffie-Hellman public key value

ulPrivateDataLen the length in bytes of the second X9.42 Diffie-Hellman private key

hPrivateData key handle for second X9.42 Diffie-Hellman private key value

ulPublicDataLen2 the length in bytes of the other party’s second X9.42 Diffie-Hellman public key

pPublicData2 pointer to other party’s second X9.42 Diffie-Hellman public key value

With the key derivation function **CKD\_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero. With the key derivation function **CKD\_SHA1\_KDF\_ASN1**, *pOtherInfo* must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function **CKD\_SHA1\_KDF\_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.

**CK\_X9\_42\_DH2\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_X9\_42\_DH2\_DERIVE\_PARAMS**.

* CK\_X9\_42\_MQV\_DERIVE\_PARAMS, CK\_X9\_42\_MQV\_DERIVE\_PARAMS\_PTR

**CK\_X9\_42\_MQV\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_X9\_42\_MQV\_DERIVE** key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

typedef struct CK\_X9\_42\_MQV\_DERIVE\_PARAMS {

CK\_X9\_42\_DH\_KDF\_TYPE kdf;

CK\_ULONG ulOtherInfoLen;

CK\_BYTE\_PTR pOtherInfo;

CK\_ULONG ulPublicDataLen;

CK\_BYTE\_PTR pPublicData;

CK\_ULONG ulPrivateDataLen;

CK\_OBJECT\_HANDLE hPrivateData;

CK\_ULONG ulPublicDataLen2;

CK\_BYTE\_PTR pPublicData2;

CK\_OBJECT\_HANDLE publicKey;

} CK\_X9\_42\_MQV\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

kdf key derivation function used on the shared secret value

ulOtherInfoLen the length in bytes of the other info

pOtherInfo some data shared between the two parties

ulPublicDataLen the length in bytes of the other party’s first X9.42 Diffie-Hellman public key

pPublicData pointer to other party’s first X9.42 Diffie-Hellman public key value

ulPrivateDataLen the length in bytes of the second X9.42 Diffie-Hellman private key

hPrivateData key handle for second X9.42 Diffie-Hellman private key value

ulPublicDataLen2 the length in bytes of the other party’s second X9.42 Diffie-Hellman public key

pPublicData2 pointer to other party’s second X9.42 Diffie-Hellman public key value

publicKey Handle to the first party’s ephemeral public key

With the key derivation function **CKD\_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero. With the key derivation function **CKD\_SHA1\_KDF\_ASN1**, *pOtherInfo* must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function **CKD\_SHA1\_KDF\_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.

**CK\_X9\_42\_MQV\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_X9\_42\_MQV\_DERIVE\_PARAMS**.

### X9.42 Diffie-Hellman key pair generation

The X9.42 Diffie-Hellman key pair generation mechanism, denoted **CKM\_X9\_42\_DH\_KEY\_PAIR\_GEN**, is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in the ANSI X9.42 standard.

It does not have a parameter.

The mechanism generates X9.42 Diffie-Hellman public/private key pairs with a particular prime, base and subprime, as specified in the **CKA\_PRIME**, **CKA\_BASE** and **CKA\_SUBPRIME** attributes of the template for the public key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_BASE**, **CKA\_SUBPRIME**, and **CKA\_VALUE** attributes to the new private key; other attributes required by the X9.42 Diffie-Hellman public and private key types must be specified in the templates.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA\_PRIME** attribute.

### X9.42 Diffie-Hellman domain parameter generation

The X9.42 Diffie-Hellman domain parameter generation mechanism, denoted **CKM\_X9\_42\_DH\_PARAMETER\_GEN**, is a domain parameters generation mechanism based on X9.42 Diffie-Hellman key agreement, as defined in the ANSI X9.42 standard.

It does not have a parameter.

The mechanism generates X9.42 Diffie-Hellman domain parameters with particular prime and subprime length in bits, as specified in the **CKA\_PRIME\_BITS** and **CKA\_SUBPRIME\_BITS** attributes of the template for the domain parameters.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_PRIME**, **CKA\_BASE, CKA\_SUBPRIME**, **CKA\_PRIME\_BITS** and **CKA\_SUBPRIME\_BITS** attributes to the new object. Other attributes supported by the X9.42 Diffie-Hellman domain parameter types may also be specified in the template for the domain parameters, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits.

### X9.42 Diffie-Hellman key derivation

The X9.42 Diffie-Hellman key derivation mechanism, denoted **CKM\_X9\_42\_DH\_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman key agreement scheme, as defined in the ANSI X9.42 standard, where each party contributes one key pair, all using the same X9.42 Diffie-Hellman domain parameters.

It has a parameter, a **CK\_X9\_42\_DH1\_DERIVE\_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template. Note that in order to validate this mechanism it may be required to use the **CKA\_VALUE** attribute as the key of a general-length MAC mechanism (e.g. **CKM\_SHA\_1\_HMAC\_GENERAL**) over some test data.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA\_PRIME** attribute.

### X9.42 Diffie-Hellman hybrid key derivation

The X9.42 Diffie-Hellman hybrid key derivation mechanism, denoted **CKM\_X9\_42\_DH\_HYBRID\_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman hybrid key agreement scheme, as defined in the ANSI X9.42 standard, where each party contributes two key pair, all using the same X9.42 Diffie-Hellman domain parameters.

It has a parameter, a **CK\_X9\_42\_DH2\_DERIVE\_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template. Note that in order to validate this mechanism it may be required to use the **CKA\_VALUE** attribute as the key of a general-length MAC mechanism (e.g. **CKM\_SHA\_1\_HMAC\_GENERAL**) over some test data.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA\_PRIME** attribute.

### X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation

The X9.42 Diffie-Hellman Menezes-Qu-Vanstone (MQV) key derivation mechanism, denoted **CKM\_X9\_42\_MQV\_DERIVE**, is a mechanism for key derivation based the MQV scheme, as defined in the ANSI X9.42 standard, where each party contributes two key pairs, all using the same X9.42 Diffie-Hellman domain parameters.

It has a parameter, a **CK\_X9\_42\_MQV\_DERIVE\_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template. Note that in order to validate this mechanism it may be required to use the **CKA\_VALUE** attribute as the key of a general-length MAC mechanism (e.g. **CKM\_SHA\_1\_HMAC\_GENERAL**) over some test data.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA\_PRIME** attribute.

## Extended Triple Diffie-Hellman (x3dh)

The Extended Triple Diffie-Hellman mechanism described here is the one described in [SIGNAL].

*Table 61, Extended Triple Diffie-Hellman Mechanisms vs. Functions*

|  | Functions | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_X3DH\_INITIATE |  |  |  |  |  |  |  |
| CKM\_X3DH\_RESPOND |  |  |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_X3DH\_INITIATE

CKM\_X3DH\_RESPOND

### Extended Triple Diffie-Hellman key objects

Extended Triple Diffie-Hellman uses Elliptic Curve keys in Montgomery representation (**CKK\_EC\_MONTGOMERY**). Three different kinds of keys are used, they differ in their lifespan:

* identity keys are long-term keys, which identify the peer,
* prekeys are short-term keys, which should be rotated often (weekly to hourly)
* onetime prekeys are keys, which should be used only once.

Any peer intending to be contacted using X3DH must publish their so-called prekey-bundle, consisting of their:

* public Identity key,
* current prekey, signed using XEDDA with their identity key
* optionally a batch of One-time public keys.

### Initiating an Extended Triple Diffie-Hellman key exchange

Initiating an Extended Triple Diffie-Hellman key exchange starts by retrieving the following required public keys (the so-called prekey-bundle) of the other peer: the Identity key, the signed public Prekey, and optionally one One-time public key.

When the necessary key material is available, the initiating party calls CKM\_X3DH\_INITIATE, also providing the following additional parameters:

* the initiators identity key
* the initiators ephemeral key (a fresh, one-time **CKK\_EC\_MONTGOMERY** type key)

**CK\_X3DH\_INITIATE\_PARAMS** is a structure that provides the parameters to the **CKM\_X3DH\_INITIATE** key exchange mechanism. The structure is defined as follows:

typedef struct CK\_X3DH\_INITIATE\_PARAMS {

CK\_X3DH\_KDF\_TYPE kdf;

CK\_OBJECT\_HANDLE pPeer\_identity;

CK\_OBJECT\_HANDLE pPeer\_prekey;

CK\_BYTE\_PTR pPrekey\_signature;

CK\_BYTE\_PTR pOnetime\_key;

CK\_OBJECT\_HANDLE pOwn\_identity;

CK\_OBJECT\_HANDLE pOwn\_ephemeral;

} CK\_X3DH\_INITIATE\_PARAMS;

Table 62, Extended Triple Diffie-Hellman Initiate Message parameters:

| **Parameter** | **Data type** | **Meaning** |
| --- | --- | --- |
| kdf | CK\_X3DH\_KDF\_TYPE | *Key derivation function* |
| pPeer\_identity | Key handle | *Peers public Identity key (from the prekey-bundle)* |
| pPeer\_prekey | Key Handle | Peers public prekey (from the prekey-bundle) |
| pPrekey\_signature | Byte array | *XEDDSA signature of PEER\_PREKEY (from prekey-bundle)* |
| pOnetime\_key | Byte array | Optional one-time public prekey of peer (from the prekey-bundle) |
| pOwn\_identity | Key Handle | Initiators Identity key |
| pOwn\_ephemeral | Key Handle | Initiators ephemeral key |

### Responding to an Extended Triple Diffie-Hellman key exchange

Responding an Extended Triple Diffie-Hellman key exchange is done by executing a CKM\_X3DH\_RESPOND mechanism. **CK\_X3DH\_RESPOND\_PARAMS** is a structure that provides the parameters to the **CKM\_X3DH\_RESPOND** key exchange mechanism. All these parameter should be supplied by the Initiator in a message to the responder. The structure is defined as follows:

typedef struct CK\_X3DH\_RESPOND\_PARAMS {

CK\_X3DH\_KDF\_TYPE kdf;

CK\_BYTE\_PTR pIdentity\_id;

CK\_BYTE\_PTR pPrekey\_id;

CK\_BYTE\_PTR pOnetime\_id;

CK\_OBJECT\_HANDLE pInitiator\_identity;

CK\_BYTE\_PTR pInitiator\_ephemeral;

} CK\_X3DH\_RESPOND\_PARAMS;

Table 63, Extended Triple Diffie-Hellman 1st Message parameters:

| **Parameter** | **Data type** | **Meaning** |
| --- | --- | --- |
| kdf | CK\_X3DH\_KDF\_TYPE | *Key derivation function* |
| pIdentity\_id | Byte array | *Peers public Identity key identifier (from the prekey-bundle)* |
| pPrekey\_id | Byte array | Peers public prekey identifier (from the prekey-bundle) |
| pOnetime\_id | Byte array | Optional one-time public prekey of peer (from the prekey-bundle) |
| pInitiator\_identity | Key handle | Initiators Identity key |
| pInitiator\_ephemeral | Byte array | Initiators ephemeral key |

Where the \*\_id fields are identifiers marking which key has been used from the prekey-bundle, these identifiers could be the keys themselves.

This mechanism has the following rules about key sensitivity and extractability[[3]](#footnote-3):

1. The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
2. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
3. Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

### Extended Triple Diffie-Hellman parameters

* CK\_X3DH\_KDF\_TYPE, CK\_X3DH\_KDF\_TYPE\_PTR

**CK\_X3DH\_KDF\_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data from a shared secret. The key derivation function will be used by the X3DH key agreement schemes. It is defined as follows:

typedef CK\_ULONG CK\_X3DH\_KDF\_TYPE;

The following table lists the defined functions.

Table 64, X3DH: Key Derivation Functions

|  |
| --- |
| **Source Identifier** |
| CKD\_NULL |
| CKD\_BLAKE2B\_256\_KDF |
| CKD\_BLAKE2B\_512\_KDF |
| CKD\_SHA3\_256\_KDF |
| CKD\_SHA256\_KDF |
| CKD\_SHA3\_512\_KDF |
| CKD\_SHA512\_KDF |

## Double Ratchet

The Double Ratchet is a key management algorithm managing the ongoing renewal and maintenance of short-lived session keys providing forward secrecy and break-in recovery for encrypt/decrypt operations. The algorithm is described in **[DoubleRatchet]**. The Signal protocol uses X3DH to exchange a shared secret in the first step, which is then used to derive a Double Ratchet secret key.

*Table 65, Double Ratchet Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_X2RATCHET\_INITALIZE |  |  |  |  |  |  | ✓ |
| CKM\_X2RATCHET\_RESPOND |  |  |  |  |  |  | ✓ |
| CKM\_X2RATCHET\_ENCRYPT | ✓ |  |  |  |  | ✓ |  |
| CKM\_X2RATCHET\_DECRYPT | ✓ |  |  |  |  | ✓ |  |

### Definitions

This section defines the key type “CKK\_X2RATCHET” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_X2RATCHET\_INITALIZE

CKM\_X2RATCHET\_RESPOND

CKM\_X2RATCHET\_ENCRYPT

CKM\_X2RATCHET\_DECRYPT

### Double Ratchet secret key objects

Double Ratchet secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_X2RATCHET) hold Double Ratchet keys. Double Ratchet secret keys can only be derived from shared secret keys using the mechanism CKM\_X2RATCHET\_INITALIZE or CKM\_X2RATCHET\_RESPOND. In the Signal protocol these are seeded with the shared secret derived from an Extended Triple Diffie-Hellman [X3DH] key-exchange. The following table defines the Double Ratchet secret key object attributes, in addition to the common attributes defined for this object class:

Table , Double Ratchet Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_X2RATCHET\_RK | Byte array | Root key |
| CKA\_X2RATCHET\_HKS | Byte array | Sender Header key |
| CKA\_X2RATCHET\_HKR | Byte array | Receiver Header key |
| CKA\_X2RATCHET\_NHKR | Byte array | Next Sender Header Key |
| CKA\_X2RATCHET\_NHKR | Byte array | Next Receiver Header Key |
| CKA\_X2RATCHET\_CKS | Byte array | Sender Chain key |
| CKA\_X2RATCHET\_CKR | Byte array | Receiver Chain key |
| CKA\_X2RATCHET\_DHS | Byte array | Sender DH secret key |
| CKA\_X2RATCHET\_DHP | Byte array | Sender DH public key |
| CKA\_X2RATCHET\_DHR | Byte array | Receiver DH public key |
| CKA\_X2RATCHET\_NS | ULONG | Message number send |
| CKA\_X2RATCHET\_NR | ULONG | Message number receive |
| CKA\_X2RATCHET\_PNS | ULONG | Previous message number send |
| CKA\_X2RATCHET\_BOBS1STMSG | BOOL | Is this bob and has he ever sent a message? |
| CKA\_X2RATCHET\_ISALICE | BOOL | Is this Alice? |
| CKA\_X2RATCHET\_BAGSIZE | ULONG | How many out-of-order keys do we store |
| CKA\_X2RATCHET\_BAG | Byte array | Out-of-order keys |

### Double Ratchet key derivation

The Double Ratchet key derivation mechanisms depend on who is the initiating party, and who the receiving, denoted **CKM\_X2RATCHET\_INITIALIZE** and **CKM\_X2RATCHET\_RESPOND**, are the key derivation mechanisms for the Double Ratchet. Usually the keys are derived from a shared secret by executing a X3DH key exchange.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Additionally the attribute flags indicating which functions the key supports are also contributed by the mechanism.

For this mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

* CK\_X2RATCHET\_INITIALIZE\_PARAMS; CK\_X2RATCHET\_INITIALIZE\_PARAMS\_PTR

**CK\_X2RATCHET\_INITIALIZE\_PARAMS** provides the parameters to the **CKM\_X2RATCHET\_INITIALIZE** mechanism. It is defined as follows:

typedef struct CK\_X2RATCHET\_INITIALIZE\_PARAMS {

CK\_BYTE\_PTR sk;

CK\_OBJECT\_HANDLE peer\_public\_prekey;

CK\_OBJECT\_HANDLE peer\_public\_identity;

CK\_OBJECT\_HANDLE own\_public\_identity;

CK\_BBOOL bEncryptedHeader;

CK\_ULONG eCurve;

CK\_MECHANISM\_TYPE aeadMechanism;

CK\_X2RATCHET\_KDF\_TYPE kdfMechanism;

} CK\_X2RATCHET\_INITIALIZE\_PARAMS;

The fields of the structure have the following meanings:

sk the shared secret with peer (derived using X3DH)

peers\_public\_prekey Peers public prekey which the Initiator used in the X3DH

peers\_public\_identity Peers public identity which the Initiator used in the X3DH

own\_public\_identity Initiators public identity as used in the X3DH

bEncryptedHeader whether the headers are encrypted

eCurve 255 for curve 25519 or 448 for curve 448

aeadMechanism a mechanism supporting AEAD encryption, e.g. CKM\_XCHACHA20

kdfMechanism a Key Derivation Mechanism, such as CKD\_BLAKE2B\_512\_KDF

* CK\_X2RATCHET\_RESPOND\_PARAMS; CK\_X2RATCHET\_RESPOND\_PARAMS\_PTR

**CK\_X2RATCHET\_RESPOND\_PARAMS** provides the parameters to the **CKM\_X2RATCHET\_RESPOND** mechanism. It is defined as follows:

typedef struct CK\_X2RATCHET\_RESPOND\_PARAMS {

CK\_BYTE\_PTR sk;

CK\_OBJECT\_HANDLE own\_prekey;

CK\_OBJECT\_HANDLE initiator\_identity;

CK\_OBJECT\_HANDLE own\_public\_identity;

CK\_BBOOL bEncryptedHeader;

CK\_ULONG eCurve;

CK\_MECHANISM\_TYPE aeadMechanism;

CK\_X2RATCHET\_KDF\_TYPE kdfMechanism;

} CK\_X2RATCHET\_RESPOND\_PARAMS;

The fields of the structure have the following meanings:

sk shared secret with the Initiator

own\_prekey Own Prekey pair that the Initiator used

initiator\_identity Initiators public identity key used

own\_public\_identity as used in the prekey bundle by the initiator in the X3DH

bEncryptedHeader whether the headers are encrypted

eCurve 255 for curve 25519 or 448 for curve 448

aeadMechanism a mechanism supporting AEAD encryption, e.g. CKM\_XCHACHA20

kdfMechanism a Key Derivation Mechanism, such as CKD\_BLAKE2B\_512\_KDF

### Double Ratchet Encryption mechanism

The Double Ratchet encryption mechanism, denoted **CKM\_X2RATCHET\_ENCRYPT** and **CKM\_X2RATCHET\_DECRYPT**, are a mechanisms for single part encryption and decryption based on the Double Ratchet and its underlying AEAD cipher.

### Double Ratchet parameters

* CK\_X2RATCHET\_KDF\_TYPE, CK\_X2RATCHET\_KDF\_TYPE\_PTR

**CK\_X2RATCHET\_KDF\_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data from a shared secret. The key derivation function will be used by the X key derivation scheme. It is defined as follows:

typedef CK\_ULONG CK\_X2RATCHET\_KDF\_TYPE;

The following table lists the defined functions.

Table 67, X2RATCHET: Key Derivation Functions

|  |
| --- |
| **Source Identifier** |
| CKD\_NULL |
| CKD\_BLAKE2B\_256\_KDF |
| CKD\_BLAKE2B\_512\_KDF |
| CKD\_SHA3\_256\_KDF |
| CKD\_SHA256\_KDF |
| CKD\_SHA3\_512\_KDF |
| CKD\_SHA512\_KDF |

## Wrapping/unwrapping private keys

Cryptoki Versions 2.01 and up allow the use of secret keys for wrapping and unwrapping RSA private keys, Diffie-Hellman private keys, X9.42 Diffie-Hellman private keys, EC (also related to ECDSA) private keys and DSA private keys. For wrapping, a private key is BER-encoded according to PKCS #8’s PrivateKeyInfo ASN.1 type. PKCS #8 requires an algorithm identifier for the type of the private key. The object identifiers for the required algorithm identifiers are as follows:

rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }

dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1 }

dhpublicnumber OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) ansi-x942(10046) number-type(2) 1 }

id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) ansi-x9-62(10045) publicKeyType(2) 1 }

id-dsa OBJECT IDENTIFIER ::= {

iso(1) member-body(2) us(840) x9-57(10040) x9cm(4) 1 }

where

pkcs-1 OBJECT IDENTIFIER ::= {

iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 1 }

pkcs-3 OBJECT IDENTIFIER ::= {

iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 3 }

These parameters for the algorithm identifiers have the following types, respectively:

NULL

DHParameter ::= SEQUENCE {

prime INTEGER, -- p

base INTEGER, -- g

privateValueLength INTEGER OPTIONAL

}

DomainParameters ::= SEQUENCE {

prime INTEGER, -- p

base INTEGER, -- g

subprime INTEGER, -- q

cofactor INTEGER OPTIONAL, -- j

validationParms ValidationParms OPTIONAL

}

ValidationParms ::= SEQUENCE {

Seed BIT STRING, -- seed

PGenCounter INTEGER -- parameter verification

}

Parameters ::= CHOICE {

ecParameters ECParameters,

namedCurve CURVES.&id({CurveNames}),

implicitlyCA NULL

}

Dss-Parms ::= SEQUENCE {

p INTEGER,

q INTEGER,

g INTEGER

}

For the X9.42 Diffie-Hellman domain parameters, the **cofactor** and the **validationParms** optional fields should not be used when wrapping or unwrapping X9.42 Diffie-Hellman private keys since their values are not stored within the token.

For the EC domain parameters, the use of **namedCurve** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.

Within the PrivateKeyInfo type:

* RSA private keys are BER-encoded according to PKCS #1’s RSAPrivateKey ASN.1 type. This type requires values to be present for *all* the attributes specific to Cryptoki’s RSA private key objects. In other words, if a Cryptoki library does not have values for an RSA private key’s **CKA\_MODULUS**, **CKA\_PUBLIC\_EXPONENT**, **CKA\_PRIVATE\_EXPONENT**, **CKA\_PRIME\_1**, **CKA\_PRIME\_2**, **CKA\_EXPONENT\_1**, **CKA\_EXPONENT2**, and **CKA\_COEFFICIENT** values, it must not create an RSAPrivateKey BER-encoding of the key, and so it must not prepare it for wrapping.
* Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.
* X9.42 Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.
* EC (also related with ECDSA) private keys are BER-encoded according to SECG SEC 1 ECPrivateKey ASN.1 type:

ECPrivateKey ::= SEQUENCE {

Version INTEGER { ecPrivkeyVer1(1) } (ecPrivkeyVer1),

privateKey OCTET STRING,

parameters [0] Parameters OPTIONAL,

publicKey [1] BIT STRING OPTIONAL

}

Since the EC domain parameters are placed in the PKCS #8’s privateKeyAlgorithm field, the optional **parameters** field in an ECPrivateKey must be omitted. A Cryptoki application must be able to unwrap an ECPrivateKey that contains the optional **publicKey** field; however, what is done with this **publicKey** field is outside the scope of Cryptoki.

* DSA private keys are represented as BER-encoded ASN.1 type INTEGER.

Once a private key has been BER-encoded as a PrivateKeyInfo type, the resulting string of bytes is encrypted with the secret key. This encryption must be done in CBC mode with PKCS padding.

Unwrapping a wrapped private key undoes the above procedure. The CBC-encrypted ciphertext is decrypted, and the PKCS padding is removed. The data thereby obtained are parsed as a PrivateKeyInfo type, and the wrapped key is produced. An error will result if the original wrapped key does not decrypt properly, or if the decrypted unpadded data does not parse properly, or its type does not match the key type specified in the template for the new key. The unwrapping mechanism contributes only those attributes specified in the PrivateKeyInfo type to the newly-unwrapped key; other attributes must be specified in the template, or will take their default values.

Earlier drafts of PKCS #11 Version 2.0 and Version 2.01 used the object identifier

DSA OBJECT IDENTIFIER ::= { algorithm 12 }

algorithm OBJECT IDENTIFIER ::= {

iso(1) identifier-organization(3) oiw(14) secsig(3) algorithm(2) }

with associated parameters

DSAParameters ::= SEQUENCE {

prime1 INTEGER, -- modulus p

prime2 INTEGER, -- modulus q

base INTEGER -- base g

}

for wrapping DSA private keys. Note that although the two structures for holding DSA domain parameters appear identical when instances of them are encoded, the two corresponding object identifiers are different.

## Generic secret key

*Table 68, Generic Secret Key Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_GENERIC\_SECRET\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_GENERIC\_SECRET” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_GENERIC\_SECRET\_KEY\_GEN

### Generic secret key objects

Generic secret key objects (object class **CKO\_SECRET\_KEY,** key type **CKK\_GENERIC\_SECRET**) hold generic secret keys. These keys do not support encryption or decryption; however, other keys can be derived from them and they can be used in HMAC operations. The following table defines the generic secret key object attributes, in addition to the common attributes defined for this object class:

These key types are used in several of the mechanisms described in this section.

Table 69, Generic Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (arbitrary length) |
| CKA\_VALUE\_LEN2,3 | CK\_ULONG | Length in bytes of key value |

- Refer to [PKCS11-Base] table 11 for footnotes

The following is a sample template for creating a generic secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_GENERIC\_SECRET;

CK\_UTF8CHAR label[] = “A generic secret key object”;

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_DERIVE, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

CKA\_CHECK\_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the SHA-1 hash of the generic secret key object’s CKA\_VALUE attribute.

### Generic secret key generation

The generic secret key generation mechanism, denoted **CKM\_GENERIC\_SECRET\_KEY\_GEN**, is used to generate generic secret keys. The generated keys take on any attributes provided in the template passed to the **C\_GenerateKey** call, and the **CKA\_VALUE\_LEN** attribute specifies the length of the key to be generated.

It does not have a parameter.

The template supplied must specify a value for the **CKA\_VALUE\_LEN** attribute. If the template specifies an object type and a class, they must have the following values:

CK\_OBJECT\_CLASS = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE = CKK\_GENERIC\_SECRET;

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of key sizes, in bits.

## HMAC mechanisms

Refer to **RFC2104** and **FIPS 198** for HMAC algorithm description. The HMAC secret key shall correspond to the PKCS11 generic secret key type or the mechanism specific key types (see mechanism definition). Such keys, for use with HMAC operations can be created using C\_CreateObject or C\_GenerateKey.

The RFC also specifies test vectors for the various hash function based HMAC mechanisms described in the respective hash mechanism descriptions. The RFC should be consulted to obtain these test vectors.

### General block cipher mechanism parameters

* CK\_MAC\_GENERAL\_PARAMS; CK\_MAC\_GENERAL\_PARAMS\_PTR

**CK\_MAC\_GENERAL\_PARAMS** provides the parameters to the general-length MACing mechanisms of the DES, DES3 (triple-DES), AES, Camellia, SEED, and ARIA ciphers.  It also provides the parameters to the general-length HMACing mechanisms (i.e.,SHA-1, SHA-256, SHA-384, SHA-512, and SHA-512/T family) and the two SSL 3.0 MACing mechanisms, (i.e., MD5 and SHA-1).  It holds the length of the MAC that these mechanisms produce.  It is defined as follows:

typedef CK\_ULONG CK\_MAC\_GENERAL\_PARAMS;

**CK\_MAC\_GENERAL\_PARAMS\_PTR** is a pointer to a **CK\_MAC\_GENERAL\_PARAMS**.

## AES

For the Advanced Encryption Standard (AES) see [FIPS PUB 197].

*Table 70, AES Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_AES\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_AES\_ECB | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_CBC | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_CBC\_PAD | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_MAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_AES\_MAC |  | ✓ |  |  |  |  |  |
| CKM\_AES\_OFB | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_CFB64 | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_CFB8 | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_CFB128 | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_CFB1 | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_XCBC\_MAC |  | ✓ |  |  |  |  |  |
| CKM\_AES\_XCBC\_MAC\_96 |  | ✓ |  |  |  |  |  |

### Definitions

This section defines the key type “CKK\_AES” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_AES\_KEY\_GEN

CKM\_AES\_ECB

CKM\_AES\_CBC

CKM\_AES\_MAC

CKM\_AES\_MAC\_GENERAL

CKM\_AES\_CBC\_PAD

CKM\_AES\_OFB

CKM\_AES\_CFB64

CKM\_AES\_CFB8

CKM\_AES\_CFB128

CKM\_AES\_CFB1

CKM\_AES\_XCBC\_MAC

CKM\_AES\_XCBC\_MAC\_96

### AES secret key objects

AES secret key objects (object class **CKO\_SECRET\_KEY,** key type **CKK\_AES**) hold AES keys. The following table defines the AES secret key object attributes, in addition to the common attributes defined for this object class:

Table 71, AES Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (16, 24, or 32 bytes) |
| CKA\_VALUE\_LEN2,3,6 | CK\_ULONG | Length in bytes of key value |

- Refer to [PKCS11-Base] table 11 for footnotes

The following is a sample template for creating an AES secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_AES;

CK\_UTF8CHAR label[] = “An AES secret key object”;

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

CKA\_CHECK\_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with the key type of the secret key object.

### AES key generation

The AES key generation mechanism, denoted **CKM\_AES\_KEY\_GEN**, is a key generation mechanism for NIST’s Advanced Encryption Standard.

It does not have a parameter.

The mechanism generates AES keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the AES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### AES-ECB

AES-ECB, denoted **CKM\_AES\_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard and electronic codebook mode.

It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 72, AES-ECB: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | AES | multiple of block size | same as input length | no final part |
| C\_Decrypt | AES | multiple of block size | same as input length | no final part |
| C\_WrapKey | AES | any | input length rounded up to multiple of block size |  |
| C\_UnwrapKey | AES | multiple of block size | determined by type of key being unwrapped or CKA\_VALUE\_LEN |  |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### AES-CBC

AES-CBC, denoted **CKM\_AES\_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST’s Advanced Encryption Standard and cipher-block chaining mode.

It has a parameter, a 16-byte initialization vector.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 73, AES-CBC: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | AES | multiple of block size | same as input length | no final part |
| C\_Decrypt | AES | multiple of block size | same as input length | no final part |
| C\_WrapKey | AES | any | input length rounded up to multiple of the block size |  |
| C\_UnwrapKey | AES | multiple of block size | determined by type of key being unwrapped or CKA\_VALUE\_LEN |  |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### AES-CBC with PKCS padding

AES-CBC with PKCS padding, denoted **CKM\_AES\_CBC\_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST’s Advanced Encryption Standard; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 16-byte initialization vector.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA\_VALUE\_LEN** attribute.

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section 2.7 for details). The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

Constraints on key types and the length of data are summarized in the following table:

Table 74, AES-CBC with PKCS Padding: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Encrypt | AES | any | input length rounded up to multiple of the block size |
| C\_Decrypt | AES | multiple of block size | between 1 and block size bytes shorter than input length |
| C\_WrapKey | AES | any | input length rounded up to multiple of the block size |
| C\_UnwrapKey | AES | multiple of block size | between 1 and block length bytes shorter than input length |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### AES-OFB

AES-OFB, denoted CKM\_AES\_OFB. It is a mechanism for single and multiple-part encryption and decryption with AES. AES-OFB mode is described in [NIST sp800-38a].

It has a parameter, an initialization vector for this mode. The initialization vector has the same length as the block size.  
  
Constraints on key types and the length of data are summarized in the following table:

Table 75, AES-OFB: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | AES | any | same as input length | no final part |
| C\_Decrypt | AES | any | same as input length | no final part |

For this mechanism the CK\_MECHANISM\_INFO structure is as specified for CBC mode.

### AES-CFB

Cipher AES has a cipher feedback mode, AES-CFB, denoted CKM\_AES\_CFB8, CKM\_AES\_CFB64, and CKM\_AES\_CFB128. It is a mechanism for single and multiple-part encryption and decryption with AES. AES-OFB mode is described [NIST sp800-38a].

It has a parameter, an initialization vector for this mode. The initialization vector has the same length as the block size.  
  
Constraints on key types and the length of data are summarized in the following table:

Table 76, AES-CFB: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | AES | any | same as input length | no final part |
| C\_Decrypt | AES | any | same as input length | no final part |

For this mechanism the CK\_MECHANISM\_INFO structure is as specified for CBC mode.

### General-length AES-MAC

General-length AES-MAC, denoted **CKM\_AES\_MAC\_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on NIST Advanced Encryption Standard as defined in FIPS PUB 197 and data authentication as defined in FIPS PUB 113.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final AES cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

Table 77, General-length AES-MAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | AES | any | 1-block size, as specified in parameters |
| C\_Verify | AES | any | 1-block size, as specified in parameters |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### AES-MAC

AES-MAC, denoted by **CKM\_AES\_MAC**, is a special case of the general-length AES-MAC mechanism. AES-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 78, AES-MAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | AES | Any | ½ block size (8 bytes) |
| C\_Verify | AES | Any | ½ block size (8 bytes) |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### AES-XCBC-MAC

AES-XCBC-MAC, denoted **CKM\_AES\_XCBC\_MAC**, is a mechanism for single and multiple part signatures and verification; based on NIST’s Advanced Encryption Standard and [RFC 3566].

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 79, AES-XCBC-MAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | AES | Any | 16 bytes |
| C\_Verify | AES | Any | 16 bytes |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### AES-XCBC-MAC-96

AES-XCBC-MAC-96, denoted **CKM\_AES\_XCBC\_MAC-96**, is a mechanism for single and multiple part signatures and verification; based on NIST’s Advanced Encryption Standard and [RFC 3566].

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 80, AES-XCBC-MAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | AES | Any | 12 bytes |
| C\_Verify | AES | Any | 12 bytes |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

## AES with Counter

*Table 81, AES with Counter Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_AES\_CTR | ✓ |  |  |  |  | ✓ |  |

### Definitions

Mechanisms:

CKM\_AES\_CTR

### AES with Counter mechanism parameters

1. CK\_AES\_CTR\_PARAMS; CK\_AES\_CTR\_PARAMS\_PTR

**CK\_AES\_CTR\_PARAMS** is a structure that provides the parameters to the **CKM\_AES\_CTR** mechanism. It is defined as follows:

typedef struct CK\_AES\_CTR\_PARAMS {

CK\_ULONG ulCounterBits;

CK\_BYTE cb[16];

} CK\_AES\_CTR\_PARAMS;

ulCounterBits specifies the number of bits in the counter block (cb) that shall be incremented. This number shall be such that 0 < *ulCounterBits* <= 128. For any values outside this range the mechanism shall return **CKR\_MECHANISM\_PARAM\_INVALID**.

It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter bits are the least significant bits of the counter block (cb). They are a big-endian value usually starting with 1. The rest of ‘cb’ is for the nonce, and maybe an optional IV.

E.g. as defined in [RFC 3686]:

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Nonce |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Initialization Vector (IV) |

| |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Block Counter |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

This construction permits each packet to consist of up to 232-1 blocks = 4,294,967,295 blocks = 68,719,476,720 octets.

**CK\_AES\_CTR\_PARAMS\_PTR** is a pointer to a **CK\_AES\_CTR\_PARAMS**.

### AES with Counter Encryption / Decryption

Generic AES counter mode is described in NIST Special Publication 800-38A and in RFC 3686. These describe encryption using a counter block which may include a nonce to guarantee uniqueness of the counter block. Since the nonce is not incremented, the mechanism parameter must specify the number of counter bits in the counter block.

The block counter is incremented by 1 after each block of plaintext is processed. There is no support for any other increment functions in this mechanism.

If an attempt to encrypt/decrypt is made which will cause an overflow of the counter block’s counter bits, then the mechanism shall return **CKR\_DATA\_LEN\_RANGE**. Note that the mechanism should allow the final post increment of the counter to overflow (if it implements it this way) but not allow any further processing after this point. E.g. if ulCounterBits = 2 and the counter bits start as 1 then only 3 blocks of data can be processed.

## AES CBC with Cipher Text Stealing CTS

Ref [NIST AES CTS]

This mode allows unpadded data that has length that is not a multiple of the block size to be encrypted to the same length of cipher text.

*Table 82, AES CBC with Cipher Text Stealing CTS Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_AES\_CTS | ✓ |  |  |  |  | ✓ |  |

### Definitions

Mechanisms:

CKM\_AES\_CTS

### AES CTS mechanism parameters

It has a parameter, a 16-byte initialization vector.

Table 83, AES-CTS: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | AES | Any, ≥ block size (16 bytes) | same as input length | no final part |
| C\_Decrypt | AES | any, ≥ block size (16 bytes) | same as input length | no final part |

## Additional AES Mechanisms

*Table 84, Additional AES Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_AES\_GCM | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_CCM | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_GMAC |  | ✓ |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_AES\_GCM

CKM\_AES\_CCM

CKM\_AES\_GMAC

Generator Functions:

CKG\_NO\_GENERATE

CKG\_GENERATE

CKG\_GENERATE\_COUNTER

CKG\_GENERATE\_RANDOM

### AES-GCM Authenticated Encryption / Decryption

Generic GCM mode is described in [GCM]. To set up for AES-GCM use the following process, where *K* (key) and *AAD* (additional authenticated data) are as described in [GCM]. AES-GCM uses CK\_GCM\_PARAMS for Encrypt, Decrypt and CK\_GCM\_MESSAGE\_PARAMS for MessageEncrypt and MessageDecrypt.

Encrypt:

* Set the IV length *ulIvLen* in the parameter block.
* Set the IV data *pIv* in the parameter block.
* Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD m*ay be NULL if *ulAADLen* is 0.
* Set the tag length *ulTagBits* in the parameter block.
* Call C\_EncryptInit() for **CKM\_AES\_GCM** mechanism with parameters and key *K*.
* Call C\_Encrypt(), or C\_EncryptUpdate()\*[[4]](#footnote-4) C\_EncryptFinal(), for the plaintext obtaining ciphertext and authentication tag output.

Decrypt:

* Set the IV length *ulIvLen* in the parameter block.
* Set the IV data *pIv* in the parameter block.
* Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD m*ay be NULL if ulAADLen is 0.
* Set the tag length *ulTagBits* in the parameter block.
* Call C\_DecryptInit() for **CKM\_AES\_GCM** mechanism with parameters and key *K*.
* Call C\_Decrypt(), or C\_DecryptUpdate()\*1 C\_DecryptFinal(), for the ciphertext, including the appended tag, obtaining plaintext output. Note: since **CKM\_AES\_GCM** is an AEAD cipher, no data should be returned until C\_Decrypt() or C\_DecryptFinal().

MessageEncrypt:

* Set the IV length *ulIvLen* in the parameter block.
* Set *pIv* to hold the IV data returned from C\_EncryptMessage() and C\_EncryptMessageBegin(). If *ulIvFixedBits* is not zero, then the most significant bits of *pIV* contain the fixed IV. If *ivGenerator* is set to CKG\_NO\_GENERATE, *pIv* is an input parameter with the full IV.
* Set the *ulIvFixedBits* and *ivGenerator* fields in the parameter block.
* Set the tag length *ulTagBits* in the parameter block.
* Set *pTag* to hold the tag data returned from C\_EncryptMessage() or the final C\_EncryptMessageNext().
* Call C\_MessageEncryptInit() for **CKM\_AES\_GCM** mechanism key *K*.
* Call C\_EncryptMessage(), or C\_EncryptMessageBegin() followed by C\_EncryptMessageNext()\*[[5]](#footnote-5). The mechanism parameter is passed to all three of these functions.
* Call C\_MessageEncryptFinal() to close the message decryption.

MessageDecrypt:

* Set the IV length *ulIvLen* in the parameter block.
* Set the IV data *pIv* in the parameter block.
* The *ulIvFixedBits* and *ivGenerator* fields are ignored.
* Set the tag length *ulTagBits* in the parameter block.
* Set the tag data *pTag* in the parameter block before C\_DecryptMessage() or the final C\_DecryptMessageNext().
* Call C\_MessageDecryptInit() for **CKM\_AES\_GCM** mechanism key *K*.
* Call C\_DecryptMessage(), or C\_DecryptMessageBegin followed by C\_DecryptMessageNext()\*[[6]](#footnote-6). The mechanism parameter is passed to all three of these functions.
* Call C\_MessageDecryptFinal() to close the message decryption.

In *pIv* the least significant bit of the initialization vector is the rightmost bit. *ulIvLen* is the length of the initialization vector in bytes.

On MessageEncrypt, the meaning of *ivGenerator* is as follows: CKG\_NO\_GENERATE means the IV is passed in on MessageEncrypt and no internal IV generation is done. CKG\_GENERATE means that the non-fixed portion of the IV is generated by the module internally. The generation method is not defined. CKG\_GENERATE\_COUNTER means that the non-fixed portion of the IV is generated by the module internally by use of an incrementing counter. CKG\_GENERATE\_RANDOM means that the non-fixed portion of the IV is generated by the module internally using a PRNG. In any case the entire IV, including the fixed portion, is returned in *pIV*.

Modules must implement CKG\_GENERATE. Modules may also reject *ulIvFixedBits* values which are too large. Zero is always an acceptable value for *ulIvFixedBits*.

In Encrypt and Decrypt the tag is appended to the cipher text and the least significant bit of the tag is the rightmost bit and the tag bits are the rightmost *ulTagBits* bits. In MessageEncrypt the tag is returned in the *pTag* field of CK\_GCM\_MESSAGE\_PARAMS. In MesssageDecrypt the tag is provided by the *pTag* field of CK\_GCM\_MESSAGE\_PARAMS.

The key type for *K* must be compatible with **CKM\_AES\_ECB** and the C\_EncryptInit()/C\_DecryptInit()/C\_MessageEncryptInit()/C\_MessageDecryptInit() calls shall behave, with respect to *K*, as if they were called directly with **CKM\_AES\_ECB**, *K* and NULL parameters.

### AES-CCM authenticated Encryption / Decryption

For IPsec (RFC 4309) and also for use in ZFS encryption. Generic CCM mode is described in [RFC 3610].

To set up for AES-CCM use the following process, where *K* (key), nonce and additional authenticated data are as described in [RFC 3610]. AES-CCM uses CK\_CCM\_PARAMS for Encrypt and Decrypt, and CK\_CCM\_MESSAGE\_PARAMS for MessageEncrypt and MessageDecrypt.

Encrypt:

* Set the message/data length *ulDataLen* in the parameter block.
* Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.
* Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if *ulAADLen* is 0.
* Set the MAC length *ulMACLen* in the parameter block.
* Call C\_EncryptInit() for **CKM\_AES\_CCM** mechanism with parameters and key *K*.
* Call C\_Encrypt(), C\_EncryptUpdate(), or C\_EncryptFinal(), for the plaintext obtaining the final ciphertext output and the MAC. The total length of data processed must be *ulDataLen*. The output length will be *ulDataLen + ulMACLen.*

Decrypt:

* Set the message/data length *ulDataLen* in the parameter block. This length must not include the length of the MAC that is appended to the cipher text.
* Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.
* Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD m*ay be NULL if *ulAADLen* is 0.
* Set the MAC length *ulMACLen* in the parameter block.
* Call C\_DecryptInit() for **CKM\_AES\_CCM** mechanism with parameters and key *K*.
* Call C\_Decrypt(), C\_DecryptUpdate(), or C\_DecryptFinal(), for the ciphertext, including the appended MAC, obtaining plaintext output. The total length of data processed must be *ulDataLen + ulMACLen*. Note: since **CKM\_AES\_CCM** is an AEAD cipher, no data should be returned until C\_Decrypt() or C\_DecryptFinal().

MessageEncrypt:

* Set the message/data length *ulDataLen* in the parameter block.
* Set the nonce length *ulNonceLen*.
* Set *pNonce* to hold the nonce data returned from C\_EncryptMessage() and C\_EncryptMessageBegin(). If *ulNonceFixedBits* is not zero, then the most significant bits of *pNonce* contain the fixed nonce. If *nonceGenerator* is set to CKG\_NO\_GENERATE, *pNonce* is an input parameter with the full nonce.
* Set the *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block.
* Set the MAC length *ulMACLen* in the parameter block.
* Set *pMAC* to hold the MAC data returned from C\_EncryptMessage() or the final C\_EncryptMessageNext().
* Call C\_MessageEncryptInit() for **CKM\_AES\_CCM** mechanism key *K*.
* Call C\_EncryptMessage(), or C\_EncryptMessageBegin() followed by C\_EncryptMessageNext()\*[[7]](#footnote-7).. The mechanism parameter is passed to all three functions.
* Call C\_MessageEncryptFinal() to close the message encryption.
* The MAC is returned in *pMac* of the CK\_CCM\_MESSAGE\_PARAMS structure.

MessageDecrypt:

* Set the message/data length *ulDataLen* in the parameter block.
* Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block
* The *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block are ignored.
* Set the MAC length *ulMACLen* in the parameter block.
* Set the MAC data *pMAC* in the parameter block before C\_DecryptMessage() or the final C\_DecryptMessageNext().
* Call C\_MessageDecryptInit() for **CKM\_AES\_CCM** mechanism key *K*.
* Call C\_DecryptMessage(), or C\_DecryptMessageBegin() followed by C\_DecryptMessageNext()\*[[8]](#footnote-8). The mechanism parameter is passed to all three functions.
* Call C\_MessageDecryptFinal() to close the message decryption.

In *pNonce* the least significant bit of the nonce is the rightmost bit. *ulNonceLen* is the length of the nonce in bytes.

On MessageEncrypt, the meaning of *nonceGenerator* is as follows: CKG\_NO\_GENERATE means the nonce is passed in on MessageEncrypt and no internal MAC generation is done. CKG\_GENERATE means that the non-fixed portion of the nonce is generated by the module internally. The generation method is not defined. CKG\_GENERATE\_COUNTER means that the non-fixed portion of the nonce is generated by the module internally by use of an incrementing counter. CKG\_GENERATE\_RANDOM means that the non-fixed portion of the nonce is generated by the module internally using a PRNG. In any case the entire nonce, including the fixed portion, is returned in *pNonce*.

Modules must implement CKG\_GENERATE. Modules may also reject *ulNonceFixedBits* values which are too large. Zero is always an acceptable value for *ulNonceFixedBits*.

In Encrypt and Decrypt the MAC is appended to the cipher text and the least significant byte of the MAC is the rightmost byte and the MAC bytes are the rightmost *ulMACLen* bytes. In MessageEncrypt the MAC is returned in the *pMAC* field of CK\_CCM\_MESSAGE\_PARAMS. In MesssageDecrypt the MAC is provided by the *pMAC* field of CK\_CCM\_MESSAGE\_PARAMS.

The key type for K must be compatible with **CKM\_AES\_ECB** and the C\_EncryptInit()/C\_DecryptInit()/C\_MessageEncryptInit()/C\_MessageDecryptInit() calls shall behave, with respect to K, as if they were called directly with **CKM\_AES\_ECB**, K and NULL parameters.

### AES-GMAC

AES-GMAC, denoted **CKM\_AES\_GMAC**, is a mechanism for single and multiple-part signatures and verification. It is described in NIST Special Publication 800-38D [GMAC]. GMAC is a special case of GCM that authenticates only the Additional Authenticated Data (AAD) part of the GCM mechanism parameters. When GMAC is used with C\_Sign or C\_Verify, pData points to the AAD. GMAC does not use plaintext or ciphertext.

The signature produced by GMAC, also referred to as a Tag, the tag’s length is determined by the CK\_GMAC\_PARAMS field *ulTagBits*.

The IV length is determined by the CK\_GMAC\_PARAMS field *ulIvLen*.

Constraints on key types and the length of data are summarized in the following table:

Table 85, AES-GMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_AES | < 2^64 | Depends on param’s ulTagBits |
| C\_Verify | CKK\_AES | < 2^64 | Depends on param’s ulTagBits |

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### AES GCM and CCM Mechanism parameters

1. CK\_GENERATOR\_FUNCTION

Functions to generate unique IVs and nonces.

typedef CK\_ULONG CK\_GENERATOR\_FUCNTION;

1. CK\_GCM\_PARAMS; CK\_GCM\_PARAMS\_PTR

CK\_GCM\_PARAMS is a structure that provides the parameters to the CKM\_AES\_GCM mechanism when used for Encrypt or Decrypt. It is defined as follows:

typedef struct CK\_GCM\_PARAMS {

CK\_BYTE\_PTR pIv;

CK\_ULONG ulIvLen;

CK\_BYTE\_PTR pAAD;

CK\_ULONG ulAADLen;

CK\_ULONG ulTagBits;

} CK\_GCM\_PARAMS;

The fields of the structure have the following meanings:

pIv pointer to initialization vector

ulIvLen length of initialization vector in bytes. The length of the initialization vector can be any number between 1 and (2^32) - 1. 96-bit (12 byte) IV values can be processed more efficiently, so that length is recommended for situations in which efficiency is critical.

pAAD pointer to additional authentication data. This data is authenticated but not encrypted.

ulAADLen length of pAAD in bytes. The length of the AAD can be any number between 0 and (2^32) – 1.

ulTagBits length of authentication tag (output following cipher text) in bits. Can be any value between 0 and 128.

**CK\_GCM\_PARAMS\_PTR** is a pointer to a **CK\_GCM\_PARAMS**.

1. CK\_GCM\_MESSAGE\_PARAMS; CK\_GCM\_MESSAGE\_PARAMS\_PTR

CK\_GCM\_MESSAGE\_PARAMS is a structure that provides the parameters to the CKM\_AES\_GCM mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

typedef struct CK\_GCM\_MESSAGE\_PARAMS {

CK\_BYTE\_PTR pIv;

CK\_ULONG ulIvLen;

CK\_ULONG ulIvFixedBits;

CK\_GENERATOR\_FUNCTION ivGenerator;

CK\_BYTE\_PTR pTag;

CK\_ULONG ulTagBits;

} CK\_GCM\_MESSAGE\_PARAMS;

The fields of the structure have the following meanings:

pIv pointer to initialization vector

ulIvLen length of initialization vector in bytes. The length of the initialization vector can be any number between 1 and (2^32) - 1. 96-bit (12 byte) IV values can be processed more efficiently, so that length is recommended for situations in which efficiency is critical.

ulIvFixedBits number of bits of the original IV to preserve when generating an new IV. These bits are counted from the Most significant bits (to the right).

ivGenerator Function used to generate a new IV. Each IV must be unique for a given session.

pTag location of the authentication tag which is returned on MessageEncrypt, and provided on MessageDecrypt.

ulTagBits length of authentication tag in bits. Can be any value between 0 and 128.

**CK\_GCM\_MESSAGE\_PARAMS\_PTR** is a pointer to a **CK\_GCM\_MESSAGE\_PARAMS**.

1. CK\_CCM\_PARAMS; CK\_CCM\_PARAMS\_PTR

**CK\_CCM\_PARAMS** is a structure that provides the parameters to the **CKM\_AES\_CCM** mechanism when used for Encrypt or Decrypt. It is defined as follows:

typedef struct CK\_CCM\_PARAMS {

CK\_ULONG ulDataLen; /\*plaintext or ciphertext\*/

CK\_BYTE\_PTR pNonce;

CK\_ULONG ulNonceLen;

CK\_BYTE\_PTR pAAD;

CK\_ULONG ulAADLen;

CK\_ULONG ulMACLen;

} CK\_CCM\_PARAMS;

The fields of the structure have the following meanings, where L is the size in bytes of the data length’s length (2 <= L <= 8):

ulDataLen length of the data where 0 <= ulDataLen < 2^(8L).

pNonce the nonce.

ulNonceLen length of pNonce in bytes where 7 <= ulNonceLen <= 13.

pAAD Additional authentication data. This data is authenticated but not encrypted.

ulAADLen length of pAAD in bytes where 0 <= ulAADLen <= (2^32) - 1.

ulMACLen length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16.

**CK\_CCM\_PARAMS\_PTR** is a pointer to a **CK\_CCM\_PARAMS**.

1. CK\_CCM\_MESSAGE\_PARAMS; CK\_CCM\_MESSAGE\_PARAMS\_PTR

**CK\_CCM\_MESSAGE\_PARAMS** is a structure that provides the parameters to the **CKM\_AES\_CCM** mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

typedef struct CK\_CCM\_MESSAGE\_PARAMS {

CK\_ULONG ulDataLen; /\*plaintext or ciphertext\*/

CK\_BYTE\_PTR pNonce;

CK\_ULONG ulNonceLen;

CK\_ULONG ulNonceFixedBits;

CK\_GENERATOR\_FUNCTION nonceGenerator;

CK\_BYTE\_PTR pMAC;

CK\_ULONG ulMACLen;

} CK\_CCM\_MESSAGE\_PARAMS;

The fields of the structure have the following meanings, where L is the size in bytes of the data length’s length (2 <= L <= 8):

ulDataLen length of the data where 0 <= ulDataLen < 2^(8L).

pNonce the nonce.

ulNonceLen length of pNonce in bytes where 7 <= ulNonceLen <= 13.

ulNonceFixedBits number of bits of the original nonce to preserve when generating a new nonce. These bits are counted from the Most significant bits (to the right).

nonceGenerator Function used to generate a new nonce. Each nonce must be unique for a given session.

pMAC location of the CCM MAC returned on MessageEncrypt, provided on MessageDecrypt

ulMACLen length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16.

**CK\_CCM\_MESSAGE\_PARAMS\_PTR** is a pointer to a **CK\_CCM\_MESSAGE\_PARAMS**.

## AES CMAC

Table 86, Mechanisms vs. Functions

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_AES\_CMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_AES\_CMAC |  | ✓ |  |  |  |  |  |

1 SR = SignRecover, VR = VerifyRecover.

### Definitions

Mechanisms:

CKM\_AES\_CMAC\_GENERAL

CKM\_AES\_CMAC

### Mechanism parameters

CKM\_AES\_CMAC\_GENERAL uses the existing **CK\_MAC\_GENERAL\_PARAMS** structure. CKM\_AES\_CMAC does not use a mechanism parameter.

### General-length AES-CMAC

General-length AES-CMAC, denoted **CKM\_AES\_CMAC\_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on **[**NIST SP800-38B**]** and **[**RFC 4493**]**.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final AES cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

Table 87, General-length AES-CMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_AES | any | 1-block size, as specified in parameters |
| C\_Verify | CKK\_AES | any | 1-block size, as specified in parameters |

References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller’s responsibility to follow these rules.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

### AES-CMAC

AES-CMAC, denoted **CKM\_AES\_CMAC**, is a special case of the general-length AES-CMAC mechanism. AES-MAC always produces and verifies MACs that are a full block size in length, the default output length specified by [RFC 4493].

Constraints on key types and the length of data are summarized in the following table:

Table 88, AES-CMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_AES | any | Block size (16 bytes) |
| C\_Verify | CKK\_AES | any | Block size (16 bytes) |

References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller’s responsibility to follow these rules.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of AES key sizes, in bytes.

## AES XTS

Table 89, Mechanisms vs. Functions

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_AES\_XTS | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_XTS\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_AES\_XTS” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_AES\_XTS

CKM\_AES\_CTS\_KEY\_GEN

### AES-XTS secret key objects

Table 90, AES-XTS Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (32 or 64 bytes) |
| CKA\_VALUE\_LEN2,3,6 | CK\_ULONG | Length in bytes of key value |

- Refer to [PKCS11-Base] table 11 for footnotes

### AES-XTS key generation

The double-length AES-XTS key generation mechanism, denoted **CKM\_AES\_XTS\_KEY\_GEN**, is a key generation mechanism for double-length AES-XTS keys.

The mechanism generates AES-XTS keys with a particular length in bytes as specified in the CKA\_VALUE\_LEN attributes of the template for the key.

This mechanism contributes the CKA\_CLASS, CKA\_KEY\_TYPE, and CKA\_VALUE attributes to the new key. Other attributes supported by the double-length AES-XTS key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK\_MECHANISM\_INFO structure specify the supported range of AES-XTS key sizes, in bytes.

### AES-XTS

AES-XTS (XEX-based Tweaked CodeBook mode with CipherText Stealing), denoted **CKM\_AES\_XTS**, isa mechanism for single- and multiple-part encryption and decryption. It is specified in NIST SP800-38E.

Its single parameter is a Data Unit Sequence Number 16 bytes long. Supported key lengths are 32 and 64 bytes. Keys are internally split into half-length sub-keys of 16 and 32 bytes respectively. Constraintson key types and the length of data are summarized in the following table:

Table 91, AES-XTS: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | CKK\_AES\_XTS | Any, ≥ block size (16 bytes) | Same as input length | No final part |
| C\_Decrypt | CKK\_AES\_XTS | Any, ≥ block size (16 bytes) | Same as input length | No final part |

## AES Key Wrap

*Table 92, AES Key Wrap Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_AES\_KEY\_WRAP | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_KEY\_WRAP\_PAD | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_KEY\_WRAP\_KWP | ✓ |  |  |  |  | ✓ |  |
| 1SR = SignRecover, VR = VerifyRecover | | | | | | | |

### Definitions

Mechanisms:

CKM\_AES\_KEY\_WRAP

CKM\_AES\_KEY\_WRAP\_PAD

CKM\_AES\_KEY\_WRAP\_KWP

### AES Key Wrap Mechanism parameters

The mechanisms will accept an optional mechanism parameter as the Initialization vector which, if present, must be a fixed size array of 8 bytes for CKM\_AES\_KEY\_WRAP and CKM\_AES\_KEY\_WRAP\_PAD, resp. 4 bytes for CKM\_AES\_KEY\_WRAP\_KWP; and, if NULL, will use the default initial value defined in Section 4.3 resp. 6.2 / 6.3 of [AES KEYWRAP].

The type of this parameter is CK\_BYTE\_PTR and the pointer points to the array of bytes to be used as the initial value. The length shall be either 0 and the pointer NULL; or 8 for CKM\_AES\_KEY\_WRAP / CKM\_AES\_KEY\_WRAP\_PAD, resp. 4 for CKM\_AES\_KEY\_WRAP\_KWP, and the pointer non-NULL.

### AES Key Wrap

The mechanisms support only single-part operations, single part wrapping and unwrapping, and single-part encryption and decryption.

The CKM\_AES\_KEY\_WRAP mechanism can only wrap a key resp. encrypt a block of data whose size is an exact multiple of the AES Key Wrap algorithm block size. Wrapping / encryption is done as defined in Section 6.2 of [AES KEYWRAP].

The CKM\_AES\_KEY\_WRAP\_PAD mechanism can wrap a key or encrypt a block of data of any length. It does the padding detailed in PKCS #7 of inputs (keys or data blocks), always producing wrapped output that is larger than the input key/data to be wrapped. This padding is done by the token before being passed to the AES key wrap algorithm, which then wraps / encrypts the padded block of data as defined in Section 6.2 of [AES KEYWRAP].

The CKM\_AES\_KEY\_WRAP\_KWP mechanism can wrap a key or encrypt block of data of any length. The input is padded and wrapped / encrypted as defined in Section 6.3 of [AES KEYWRAP], which produces same results as RFC 5649.

## Key derivation by data encryption – DES & AES

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C\_DeriveKey function.

*Table 93, Key derivation by data encryption Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_DES\_ECB\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_DES\_CBC\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_DES3\_ECB\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_DES3\_CBC\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_AES\_ECB\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_AES\_CBC\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |

### Definitions

Mechanisms:

CKM\_DES\_ECB\_ENCRYPT\_DATA

CKM\_DES\_CBC\_ENCRYPT\_DATA

CKM\_DES3\_ECB\_ENCRYPT\_DATA

CKM\_DES3\_CBC\_ENCRYPT\_DATA

CKM\_AES\_ECB\_ENCRYPT\_DATA

CKM\_AES\_CBC\_ENCRYPT\_DATA

typedef struct CK\_DES\_CBC\_ENCRYPT\_DATA\_PARAMS {

CK\_BYTE iv[8];

CK\_BYTE\_PTR pData;

CK\_ULONG length;

} CK\_DES\_CBC\_ENCRYPT\_DATA\_PARAMS;

typedef CK\_DES\_CBC\_ENCRYPT\_DATA\_PARAMS CK\_PTR CK\_DES\_CBC\_ENCRYPT\_DATA\_PARAMS\_PTR;

typedef struct CK\_AES\_CBC\_ENCRYPT\_DATA\_PARAMS {

CK\_BYTE iv[16];

CK\_BYTE\_PTR pData;

CK\_ULONG length;

} CK\_AES\_CBC\_ENCRYPT\_DATA\_PARAMS;

typedef CK\_AES\_CBC\_ENCRYPT\_DATA\_PARAMS CK\_PTR

CK\_AES\_CBC\_ENCRYPT\_DATA\_PARAMS\_PTR;

### Mechanism Parameters

Uses CK\_KEY\_DERIVATION\_STRING\_DATA as defined in section 2.43.2

Table 94, Mechanism Parameters

|  |  |
| --- | --- |
| CKM\_DES\_ECB\_ENCRYPT\_DATA CKM\_DES3\_ECB\_ENCRYPT\_DATA | Uses CK\_KEY\_DERIVATION\_STRING\_DATA structure. Parameter is the data to be encrypted and must be a multiple of 8 bytes long. |
| CKM\_AES\_ECB\_ENCRYPT\_DATA | Uses CK\_KEY\_DERIVATION\_STRING\_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long. |
| CKM\_DES\_CBC\_ENCRYPT\_DATA  CKM\_DES3\_CBC\_ENCRYPT\_DATA | Uses CK\_DES\_CBC\_ENCRYPT\_DATA\_PARAMS. Parameter is an 8 byte IV value followed by the data. The data value part must be a multiple of 8 bytes long. |
| CKM\_AES\_CBC\_ENCRYPT\_DATA | Uses CK\_AES\_CBC\_ENCRYPT\_DATA\_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part  must be a multiple of 16 bytes long. |

### Mechanism Description

The mechanisms will function by performing the encryption over the data provided using the base key. The resulting cipher text shall be used to create the key value of the resulting key. If not all the cipher text is used then the part discarded will be from the trailing end (least significant bytes) of the cipher text data. The derived key shall be defined by the attribute template supplied but constrained by the length of cipher text available for the key value and other normal PKCS11 derivation constraints.

Attribute template handling, attribute defaulting and key value preparation will operate as per the SHA-1 Key Derivation mechanism in section 2.20.5.

If the data is too short to make the requested key then the mechanism returns CKR\_DATA\_LEN\_RANGE.

## Double and Triple-length DES

*Table 95, Double and Triple-Length DES Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_DES2\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_DES3\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_DES3\_ECB | ✓ |  |  |  |  | ✓ |  |
| CKM\_DES3\_CBC | ✓ |  |  |  |  | ✓ |  |
| CKM\_DES3\_CBC\_PAD | ✓ |  |  |  |  | ✓ |  |
| CKM\_DES3\_MAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_DES3\_MAC |  | ✓ |  |  |  |  |  |

### Definitions

This section defines the key type “CKK\_DES2” and “CKK\_DES3” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_DES2\_KEY\_GEN

CKM\_DES3\_KEY\_GEN

CKM\_DES3\_ECB

CKM\_DES3\_CBC

CKM\_DES3\_MAC

CKM\_DES3\_MAC\_GENERAL

CKM\_DES3\_CBC\_PAD

### DES2 secret key objects

DES2 secret key objects (object class **CKO\_SECRET\_KEY,** key type **CKK\_DES2**) hold double-length DES keys. The following table defines the DES2 secret key object attributes, in addition to the common attributes defined for this object class:

Table 96, DES2 Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (always 16 bytes long) |

- Refer to [PKCS11-Base] table 11 for footnotes

DES2 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of the DES keys comprising a DES2 key must have its parity bits properly set). Attempting to create or unwrap a DES2 key with incorrect parity will return an error.

The following is a sample template for creating a double-length DES secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DES2;

CK\_UTF8CHAR label[] = “A DES2 secret key object”;

CK\_BYTE value[16] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

CKA\_CHECK\_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with the key type of the secret key object.

### DES3 secret key objects

DES3 secret key objects (object class **CKO\_SECRET\_KEY,** key type **CKK\_DES3**) hold triple-length DES keys. The following table defines the DES3 secret key object attributes, in addition to the common attributes defined for this object class:

Table 97, DES3 Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (always 24 bytes long) |

- Refer to [PKCS11-Base] table 11 for footnotes

DES3 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of the DES keys comprising a DES3 key must have its parity bits properly set). Attempting to create or unwrap a DES3 key with incorrect parity will return an error.

The following is a sample template for creating a triple-length DES secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DES3;

CK\_UTF8CHAR label[] = “A DES3 secret key object”;

CK\_BYTE value[24] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

CKA\_CHECK\_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with the key type of the secret key object.

### Double-length DES key generation

The double-length DES key generation mechanism, denoted **CKM\_DES2\_KEY\_GEN**, is a key generation mechanism for double-length DES keys. The DES keys making up a double-length DES key both have their parity bits set properly, as specified in FIPS PUB 46-3.

It does not have a parameter.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the double-length DES key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

Double-length DES keys can be used with all the same mechanisms as triple-DES keys: **CKM\_DES3\_ECB**, **CKM\_DES3\_CBC**, **CKM\_DES3\_CBC\_PAD**, **CKM\_DES3\_MAC\_GENERAL**, and **CKM\_DES3\_MAC**. Triple-DES encryption with a double-length DES key is equivalent to encryption with a triple-length DES key with K1=K3 as specified in FIPS PUB 46-3.

When double-length DES keys are generated, it is token-dependent whether or not it is possible for either of the component DES keys to be “weak” or “semi-weak” keys.

### Triple-length DES Order of Operations

Triple-length DES encryptions are carried out as specified in FIPS PUB 46-3: encrypt, decrypt, encrypt. Decryptions are carried out with the opposite three steps: decrypt, encrypt, decrypt. The mathematical representations of the encrypt and decrypt operations are as follows:

DES3-E({K1,K2,K3}, P) = E(K3, D(K2, E(K1, P)))

DES3-D({K1,K2,K3}, C) = D(K1, E(K2, D(K3, P)))

### Triple-length DES in CBC Mode

Triple-length DES operations in CBC mode, with double or triple-length keys, are performed using outer CBC as defined in X9.52. X9.52 describes this mode as TCBC. The mathematical representations of the CBC encrypt and decrypt operations are as follows:

DES3-CBC-E({K1,K2,K3}, P) = E(K3, D(K2, E(K1, P + I)))

DES3-CBC-D({K1,K2,K3}, C) = D(K1, E(K2, D(K3, P))) + I

The value *I* is either an 8-byte initialization vector or the previous block of cipher text that is added to the current input block. The addition operation is used is addition modulo-2 (XOR).

### DES and Triple length DES in OFB Mode

*Table 98, DES and Triple Length DES in OFB Mode Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_DES\_OFB64 | ✓ |  |  |  |  |  |  |
| CKM\_DES\_OFB8 | ✓ |  |  |  |  |  |  |
| CKM\_DES\_CFB64 | ✓ |  |  |  |  |  |  |
| CKM\_DES\_CFB8 | ✓ |  |  |  |  |  |  |

Cipher DES has a output feedback mode, DES-OFB, denoted **CKM\_DES\_OFB8** and **CKM\_DES\_OFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES.

It has a parameter, an initialization vector for this mode. The initialization vector has the same length as the block size.

Constraints on key types and the length of data are summarized in the following table:

Table 99, OFB: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | CKK\_DES, CKK\_DES2, CKK\_DES3 | any | same as input length | no final part |
| C\_Decrypt | CKK\_DES, CKK\_DES2, CKK\_DES3 | any | same as input length | no final part |

For this mechanism the **CK\_MECHANISM\_INFO** structure is as specified for CBC mode.

### DES and Triple length DES in CFB Mode

Cipher DES has a cipher feedback mode, DES-CFB, denoted **CKM\_DES\_CFB8** and **CKM\_DES\_CFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES.

It has a parameter, an initialization vector for this mode. The initialization vector has the same length as the block size.

Constraints on key types and the length of data are summarized in the following table:

Table 100, CFB: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | CKK\_DES, CKK\_DES2, CKK\_DES3 | any | same as input length | no final part |
| C\_Decrypt | CKK\_DES, CKK\_DES2, CKK\_DES3 | any | same as input length | no final part |

For this mechanism the **CK\_MECHANISM\_INFO** structure is as specified for CBC mode.

## Double and Triple-length DES CMAC

*Table 101, Double and Triple-length DES CMAC Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_DES3\_CMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_DES3\_CMAC |  | ✓ |  |  |  |  |  |

1 SR = SignRecover, VR = VerifyRecover.

The following additional DES3 mechanisms have been added.

### Definitions

Mechanisms:

CKM\_DES3\_CMAC\_GENERAL

CKM\_DES3\_CMAC

### Mechanism parameters

CKM\_DES3\_CMAC\_GENERAL uses the existing **CK\_MAC\_GENERAL\_PARAMS** structure. CKM\_DES3\_CMAC does not use a mechanism parameter.

### General-length DES3-MAC

General-length DES3-CMAC, denoted **CKM\_DES3\_CMAC\_GENERAL**, is a mechanism for single- and multiple-part signatures and verification with DES3 or DES2 keys, based on [NIST sp800-38b].

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final DES3 cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

Table 102, General-length DES3-CMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_DES3 CKK\_DES2 | any | 1-block size, as specified in parameters |
| C\_Verify | CKK\_DES3 CKK\_DES2 | any | 1-block size, as specified in parameters |

Reference [NIST sp800-38b] recommends that the output MAC is not truncated to less than 64 bits (which means using the entire block for DES). The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller’s responsibility to follow these rules.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used

### DES3-CMAC

DES3-CMAC, denoted **CKM\_DES3\_CMAC**, is a special case of the general-length DES3-CMAC mechanism. DES3-MAC always produces and verifies MACs that are a full block size in length, since the DES3 block length is the minimum output length recommended by [NIST sp800-38b].

Constraints on key types and the length of data are summarized in the following table:

Table 103, DES3-CMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_DES3 CKK\_DES2 | any | Block size (8 bytes) |
| C\_Verify | CKK\_DES3 CKK\_DES2 | any | Block size (8 bytes) |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

## SHA-1

*Table 104, SHA-1 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA\_1 |  |  |  | ✓ |  |  |  |
| CKM\_SHA\_1\_HMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_SHA\_1\_HMAC |  | ✓ |  |  |  |  |  |
| CKM\_SHA1\_KEY\_DERIVATION |  |  |  |  |  |  | ✓ |
| CKM\_SHA\_1\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_SHA\_1\_HMAC” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SHA\_1

CKM\_SHA\_1\_HMAC

CKM\_SHA\_1\_HMAC\_GENERAL

CKM\_SHA1\_KEY\_DERIVATION

CKM\_SHA\_1\_KEY\_GEN

### SHA-1 digest

The SHA-1 mechanism, denoted **CKM\_SHA\_1**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 160-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 105, SHA-1: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 20 |

### General-length SHA-1-HMAC

The general-length SHA-1-HMAC mechanism, denoted **CKM\_SHA\_1\_HMAC\_GENERAL**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-1 hash function. The keys it uses are generic secret keys and CKK\_SHA\_1\_HMAC.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-20 (the output size of SHA-1 is 20 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 20-byte HMAC output.

Table 106, General-length SHA-1-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret  CKK\_SHA\_1\_HMAC | any | 1-20, depending on parameters |
| C\_Verify | generic secret  CKK\_SHA\_1\_HMAC | any | 1-20, depending on parameters |

### SHA-1-HMAC

The SHA-1-HMAC mechanism, denoted **CKM\_SHA\_1\_HMAC**, is a special case of the general-length SHA-1-HMAC mechanism in Section .

It has no parameter, and always produces an output of length 20.

### SHA-1 key derivation

SHA-1 key derivation, denoted **CKM\_SHA1\_KEY\_DERIVATION**, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with SHA-1.

The value of the base key is digested once, and the result is used to make the value of derived secret key.

* If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be 20 bytes (the output size of SHA-1).
* If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
* If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn’t, an error will be returned.
* If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more than 20 bytes, such as DES3, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

### SHA-1 HMAC key generation

The SHA-1-HMAC key generation mechanism, denoted **CKM\_SHA\_1\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA-1-HMAC.

It does not have a parameter.

The mechanism generates SHA-1-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA-1-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA\_1\_HMAC** key sizes, in bytes.

## SHA-224

*Table 107, SHA-224 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA224 |  |  |  | ✓ |  |  |  |
| CKM\_SHA224\_HMAC |  | ✓ |  |  |  |  |  |
| CKM\_SHA224\_HMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_SHA224\_RSA\_PKCS |  | ✓ |  |  |  |  |  |
| CKM\_SHA224\_RSA\_PKCS\_PSS |  | ✓ |  |  |  |  |  |
| CKM\_SHA224\_KEY\_DERIVATION |  |  |  |  |  |  | ✓ |
| CKM\_SHA224\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_SHA224\_HMAC” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SHA224

CKM\_SHA224\_HMAC

CKM\_SHA224\_HMAC\_GENERAL

CKM\_SHA224\_KEY\_DERIVATION

CKM\_SHA224\_KEY\_GEN

### SHA-224 digest

The SHA-224 mechanism, denoted **CKM\_SHA224**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 224-bit message digest defined in 0.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 108, SHA-224: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 28 |

### General-length SHA-224-HMAC

The general-length SHA-224-HMAC mechanism, denoted **CKM\_SHA224\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism except that it uses the HMAC construction based on the SHA-224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and CKK\_SHA224\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-224 hash output.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-28 (the output size of SHA-224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC output.

Table 109, General-length SHA-224-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret  CKK\_SHA224\_HMAC | Any | 1-28, depending on parameters |
| C\_Verify | generic secret  CKK\_SHA224\_HMAC | Any | 1-28, depending on parameters |

### SHA-224-HMAC

The SHA-224-HMAC mechanism, denoted **CKM\_SHA224\_HMAC**, is a special case of the general-length SHA-224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

### SHA-224 key derivation

SHA-224 key derivation, denoted **CKM\_SHA224\_KEY\_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 12.21.5 except that it uses the SHA-224 hash function and the relevant length is 28 bytes.

### SHA-224 HMAC key generation

The SHA-224-HMAC key generation mechanism, denoted **CKM\_SHA224\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA224-HMAC.

It does not have a parameter.

The mechanism generates SHA224-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA224-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA224\_HMAC** key sizes, in bytes.

## SHA-256

*Table 110, SHA-256 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA256 |  |  |  | ✓ |  |  |  |
| CKM\_SHA256\_HMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_SHA256\_HMAC |  | ✓ |  |  |  |  |  |
| CKM\_SHA256\_KEY\_DERIVATION |  |  |  |  |  |  | ✓ |
| CKM\_SHA256\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_SHA256\_HMAC” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SHA256

CKM\_SHA256\_HMAC

CKM\_SHA256\_HMAC\_GENERAL

CKM\_SHA256\_KEY\_DERIVATION

CKM\_SHA256\_KEY\_GEN

### SHA-256 digest

The SHA-256 mechanism, denoted **CKM\_SHA256**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 256-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 111, SHA-256: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 32 |

### General-length SHA-256-HMAC

The general-length SHA-256-HMAC mechanism, denoted **CKM\_SHA256\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section , except that it uses the HMAC construction based on the SHA-256 hash function and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK\_SHA256\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-256 hash output.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of SHA-256 is 32 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC output.

Table 112, General-length SHA-256-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret,  CKK\_SHA256\_HMAC | Any | 1-32, depending on parameters |
| C\_Verify | generic secret,  CKK\_SHA256\_HMAC | Any | 1-32, depending on parameters |

### SHA-256-HMAC

The SHA-256-HMAC mechanism, denoted **CKM\_SHA256\_HMAC**, is a special case of the general-length SHA-256-HMAC mechanism in Section 2.22.3.

It has no parameter, and always produces an output of length 32.

### SHA-256 key derivation

SHA-256 key derivation, denoted CKM\_SHA256\_KEY\_DERIVATION, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA-256 hash function and the relevant length is 32 bytes.

### SHA-256 HMAC key generation

The SHA-256-HMAC key generation mechanism, denoted **CKM\_SHA256\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA256-HMAC.

It does not have a parameter.

The mechanism generates SHA256-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA256\_HMAC** key sizes, in bytes.

## SHA-384

*Table 113, SHA-384 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA384 |  |  |  | ✓ |  |  |  |
| CKM\_SHA384\_HMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_SHA384\_HMAC |  | ✓ |  |  |  |  |  |
| CKM\_SHA384\_KEY\_DERIVATION |  |  |  |  |  |  | ✓ |
| CKM\_SHA384\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_SHA384\_HMAC” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

CKM\_SHA384

CKM\_SHA384\_HMAC

CKM\_SHA384\_HMAC\_GENERAL

CKM\_SHA384\_KEY\_DERIVATION

CKM\_SHA384\_KEY\_GEN

### SHA-384 digest

The SHA-384 mechanism, denoted **CKM\_SHA384**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 384-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 114, SHA-384: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 48 |

### General-length SHA-384-HMAC

The general-length SHA-384-HMAC mechanism, denoted **CKM\_SHA384\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section , except that it uses the HMAC construction based on the SHA-384 hash function and length of the output should be in the range 1-48.

The keys it uses are generic secret keys and CKK\_SHA384\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 24 bytes; that is, half the size of the SHA-384 hash output.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 0-48 (the output size of SHA-384 is 48 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 48-byte HMAC output.

Table 115, General-length SHA-384-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret, CKK\_SHA384\_HMAC | Any | 1-48, depending on parameters |
| C\_Verify | generic secret,  CKK\_SHA384\_HMAC | Any | 1-48, depending on parameters |

### SHA-384-HMAC

The SHA-384-HMAC mechanism, denoted **CKM\_SHA384\_HMAC**, is a special case of the general-length SHA-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.

### SHA-384 key derivation

SHA-384 key derivation, denoted **CKM\_SHA384\_KEY\_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

### SHA-384 HMAC key generation

The SHA-384-HMAC key generation mechanism, denoted **CKM\_SHA384\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA384-HMAC.

It does not have a parameter.

The mechanism generates SHA384-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA384-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA384\_HMAC** key sizes, in bytes.

## SHA-512

*Table 116, SHA-512 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA512 |  |  |  | ✓ |  |  |  |
| CKM\_SHA512\_HMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_HMAC |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_KEY\_DERIVATION |  |  |  |  |  |  | ✓ |
| CKM\_SHA512\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_SHA512\_HMAC” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SHA512

CKM\_SHA512\_HMAC

CKM\_SHA512\_HMAC\_GENERAL

CKM\_SHA512\_KEY\_DERIVATION

CKM\_SHA512\_KEY\_GEN

### SHA-512 digest

The SHA-512 mechanism, denoted **CKM\_SHA512**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 512-bit message digest defined in FIPS PUB 180-2.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 117, SHA-512: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 64 |

### General-length SHA-512-HMAC

The general-length SHA-512-HMAC mechanism, denoted **CKM\_SHA512\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section , except that it uses the HMAC construction based on the SHA-512 hash function and length of the output should be in the range 1-64.

The keys it uses are generic secret keys and CKK\_SHA512\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 32 bytes; that is, half the size of the SHA-512 hash output.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 0-64 (the output size of SHA-512 is 64 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 64-byte HMAC output.

Table 118, General-length SHA-384-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret, CKK\_SHA512\_HMAC | Any | 1-64, depending on parameters |
| C\_Verify | generic secret,  CKK\_SHA512\_HMAC | Any | 1-64, depending on parameters |

### SHA-512-HMAC

The SHA-512-HMAC mechanism, denoted **CKM\_SHA512\_HMAC**, is a special case of the general-length SHA-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

### SHA-512 key derivation

SHA-512 key derivation, denoted **CKM\_SHA512\_KEY\_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA-512 hash function and the relevant length is 64 bytes.

### SHA-512 HMAC key generation

The SHA-512-HMAC key generation mechanism, denoted **CKM\_SHA512\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA512-HMAC.

It does not have a parameter.

The mechanism generates SHA512-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA512-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA512\_HMAC** key sizes, in bytes.

## SHA-512/224

*Table 119, SHA-512/224 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA512\_224 |  |  |  | ✓ |  |  |  |
| CKM\_SHA512\_224\_HMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_224\_HMAC |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_224\_KEY\_DERIVATION |  |  |  |  |  |  | ✓ |
| CKM\_SHA512\_224\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_SHA512\_224\_HMAC” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SHA512\_224

CKM\_SHA512\_224\_HMAC

CKM\_SHA512\_224\_HMAC\_GENERAL

CKM\_SHA512\_224\_KEY\_DERIVATION

CKM\_SHA512\_224\_KEY\_GEN

### SHA-512/224 digest

The SHA-512/224 mechanism, denoted **CKM\_SHA512\_224**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to 224 bits. **CKM\_SHA512\_224** is the same as **CKM\_SHA512\_T** with a parameter value of 224.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 120, SHA-512/224: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 28 |

### General-length SHA-512/224-HMAC

The general-length SHA-512/224-HMAC mechanism, denoted **CKM\_SHA512\_224\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section , except that it uses the HMAC construction based on the SHA-512/224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and CKK\_SHA512\_224\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-512/224 hash output.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 0-28 (the output size of SHA-512/224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC output.

Table 121, General-length SHA-384-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret, CKK\_SHA512\_224\_HMAC | Any | 1-28, depending on parameters |
| C\_Verify | generic secret,  CKK\_SHA512\_224\_HMAC | Any | 1-28, depending on parameters |

### SHA-512/224-HMAC

The SHA-512-HMAC mechanism, denoted **CKM\_SHA512\_224\_HMAC**, is a special case of the general-length SHA-512/224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

### SHA-512/224 key derivation

The SHA-512/224 key derivation, denoted **CKM\_SHA512\_224\_KEY\_DERIVATION**, is the same as the SHA-512 key derivation mechanism in section 2.25.5, except that it uses the SHA-512/224 hash function and the relevant length is 28 bytes.

### SHA-512/224 HMAC key generation

The SHA-512/224-HMAC key generation mechanism, denoted **CKM\_SHA512\_224\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA512/224-HMAC.

It does not have a parameter.

The mechanism generates SHA512/224-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA512/224-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA512\_224\_HMAC** key sizes, in bytes.

## SHA-512/256

*Table 122, SHA-512/256 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA512\_256 |  |  |  | ✓ |  |  |  |
| CKM\_SHA512\_256\_HMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_256\_HMAC |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_256\_KEY\_DERIVATION |  |  |  |  |  |  | ✓ |
| CKM\_SHA512\_256\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_SHA512\_256\_HMAC” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SHA512\_256

CKM\_SHA512\_256\_HMAC

CKM\_SHA512\_256\_HMAC\_GENERAL

CKM\_SHA512\_256\_KEY\_DERIVATION

CKM\_SHA512\_256\_KEY\_GEN

### SHA-512/256 digest

The SHA-512/256 mechanism, denoted **CKM\_SHA512\_256**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to 256 bits. **CKM\_SHA512\_256** is the same as **CKM\_SHA512\_T** with a parameter value of 256.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 123, SHA-512/256: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 32 |

### General-length SHA-512/256-HMAC

The general-length SHA-512/256-HMAC mechanism, denoted **CKM\_SHA512\_256\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section , except that it uses the HMAC construction based on the SHA-512/256 hash function and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK\_SHA512\_256\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-512/256 hash output.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of SHA-512/256 is 32 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC output.

Table 124, General-length SHA-384-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret, CKK\_SHA512\_256\_HMAC | Any | 1-32, depending on parameters |
| C\_Verify | generic secret,  CKK\_SHA512\_256\_HMAC | Any | 1-32, depending on parameters |

### SHA-512/256-HMAC

The SHA-512-HMAC mechanism, denoted **CKM\_SHA512\_256\_HMAC**, is a special case of the general-length SHA-512/256-HMAC mechanism.

It has no parameter, and always produces an output of length 32.

### SHA-512/256 key derivation

The SHA-512/256 key derivation, denoted **CKM\_SHA512\_256\_KEY\_DERIVATION**, is the same as the SHA-512 key derivation mechanism in section 2.25.5, except that it uses the SHA-512/256 hash function and the relevant length is 32 bytes.

### SHA-512/256 HMAC key generation

The SHA-512/256-HMAC key generation mechanism, denoted **CKM\_SHA512\_256\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA512/256-HMAC.

It does not have a parameter.

The mechanism generates SHA512/256-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA512/256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA512\_256\_HMAC** key sizes, in bytes.

## SHA-512/t

*Table 125, SHA-512 / t Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA512\_T |  |  |  | ✓ |  |  |  |
| CKM\_SHA512\_T\_HMAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_T\_HMAC |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_T\_KEY\_DERIVATION |  |  |  |  |  |  | ✓ |
| CKM\_SHA512\_T\_KEY\_GEN |  |  |  |  | ✓ |  |  |

### Definitions

This section defines the key type “CKK\_SHA512\_T\_HMAC” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SHA512\_T

CKM\_SHA512\_T\_HMAC

CKM\_SHA512\_T\_HMAC\_GENERAL

CKM\_SHA512\_T\_KEY\_DERIVATION

CKK\_SHA512\_T\_KEY\_GEN

### SHA-512/t digest

The SHA-512/t mechanism, denoted **CKM\_SHA512\_T**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to t bits.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the value of t in bits. The length in bytes of the desired output should be in the range of 0-⌈ t/8⌉, where 0 < t < 512, and t <> 384.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 126, SHA-512/256: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | ⌈t/8⌉, where 0 < t < 512, and t <> 384 |

### General-length SHA-512/t-HMAC

The general-length SHA-512/t-HMAC mechanism, denoted **CKM\_SHA512\_T\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section , except that it uses the HMAC construction based on the SHA-512/t hash function and length of the output should be in the range 0 – ⌈t/8⌉, where 0 < t < 512, and t <> 384.

### SHA-512/t-HMAC

The SHA-512/t-HMAC mechanism, denoted **CKM\_SHA512\_T\_HMAC**, is a special case of the general-length SHA-512/t-HMAC mechanism.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the value of t in bits. The length in bytes of the desired output should be in the range of 0-⌈t/8⌉, where 0 < t < 512, and t <> 384.

### SHA-512/t key derivation

The SHA-512/t key derivation, denoted **CKM\_SHA512\_T\_KEY\_DERIVATION**, is the same as the SHA-512 key derivation mechanism in section 2.25.5, except that it uses the SHA-512/t hash function and the relevant length is ⌈t/8⌉ bytes, where 0 < t < 512, and t <> 384.

### SHA-512/t HMAC key generation

The SHA-512/t-HMAC key generation mechanism, denoted **CKM\_SHA512\_T\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA512/t-HMAC.

It does not have a parameter.

The mechanism generates SHA512/t-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA512/t-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA512\_T\_HMAC** key sizes, in bytes.

## SHA3-224

*Table 127, SHA-224 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA3\_224 |  |  |  |  |  |  |  |
| CKM\_SHA3\_224\_HMAC |  |  |  |  |  |  |  |
| CKM\_SHA3\_224\_HMAC\_GENERAL |  |  |  |  |  |  |  |
| CKM\_SHA3\_224\_KEY\_DERIVATION |  |  |  |  |  |  |  |
| CKM\_SHA3\_224\_KEY\_GEN |  |  |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_SHA3\_224

CKM\_SHA3\_224\_HMAC

CKM\_SHA3\_224\_HMAC\_GENERAL

CKM\_SHA3\_224\_KEY\_DERIVATION

CKM\_SHA3\_224\_KEY\_GEN

CKK\_SHA3\_224\_HMAC

### SHA3-224 digest

The SHA3-224 mechanism, denoted **CKM\_SHA3\_224**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 224-bit message digest defined in FIPS Pub 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 128, SHA3-224: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 28 |

### General-length SHA3-224-HMAC

The general-length SHA3-224-HMAC mechanism, denoted **CKM\_SHA3\_224\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in section 2.20.4 except that it uses the HMAC construction based on the SHA3-224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and CKK\_SHA3\_224\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA3-224 hash output.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-28 (the output size of SHA3-224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 28-byte HMAC output.

Table 129, General-length SHA3-224-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret or CKK\_SHA3\_224\_HMAC | Any | 1-28, depending on parameters |
| C\_Verify | generic secret or CKK\_SHA3\_224\_HMAC | Any | 1-28, depending on parameters |

### SHA3-224-HMAC

The SHA3-224-HMAC mechanism, denoted **CKM\_SHA3\_224\_HMAC**, is a special case of the general-length SHA3-224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

### SHA3-224 key derivation

SHA-224 key derivation, denoted **CKM\_SHA3\_224\_KEY\_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 2.20.5 except that it uses the SHA3-224 hash function and the relevant length is 28 bytes.

### SHA3-224 HMAC key generation

The SHA3-224-HMAC key generation mechanism, denoted **CKM\_SHA3\_224\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA3-224-HMAC.

It does not have a parameter.

The mechanism generates SHA3-224-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA3-224-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA3\_224\_HMAC** key sizes, in bytes.

## SHA3-256

*Table 130, SHA3-256 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA3\_256 |  |  |  |  |  |  |  |
| CKM\_SHA3\_256\_HMAC\_GENERAL |  |  |  |  |  |  |  |
| CKM\_SHA3\_256\_HMAC |  |  |  |  |  |  |  |
| CKM\_SHA3\_256\_KEY\_DERIVATION |  |  |  |  |  |  |  |
| CKM\_SHA3\_256\_KEY\_GEN |  |  |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_SHA3\_256

CKM\_SHA3\_256\_HMAC

CKM\_SHA3\_256\_HMAC\_GENERAL

CKM\_SHA3\_256\_KEY\_DERIVATION

CKM\_SHA3\_256\_KEY\_GEN

CKK\_SHA3\_256\_HMAC

### SHA3-256 digest

The SHA3-256 mechanism, denoted **CKM\_SHA3\_256**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 256-bit message digest defined in FIPS PUB 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 131, SHA3-256: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 32 |

### General-length SHA3-256-HMAC

The general-length SHA3-256-HMAC mechanism, denoted **CKM\_SHA3\_256\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.4, except that it uses the HMAC construction based on the SHA3-256 hash function and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK\_SHA3\_256\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA3-256 hash output.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of SHA3-256 is 32 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC output.

Table 132, General-length SHA3-256-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret or CKK\_SHA3\_256\_HMAC | Any | 1-32, depending on parameters |
| C\_Verify | generic secret or  CKK\_SHA3\_256\_HMAC | Any | 1-32, depending on parameters |

### SHA3-256-HMAC

The SHA-256-HMAC mechanism, denoted **CKM\_SHA3\_256\_HMAC**, is a special case of the general-length SHA-256-HMAC mechanism in Section 2.22.3.

It has no parameter, and always produces an output of length 32.

### SHA3-256 key derivation

SHA-256 key derivation, denoted CKM\_SHA3\_256\_KEY\_DERIVATION, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA3-256 hash function and the relevant length is 32 bytes.

### SHA3-256 HMAC key generation

The SHA3-256-HMAC key generation mechanism, denoted **CKM\_SHA3\_256\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA3-256-HMAC.

It does not have a parameter.

The mechanism generates SHA3-256-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA3-256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA3\_256\_HMAC** key sizes, in bytes.

## SHA3-384

*Table 133, SHA3-384 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA3\_384 |  |  |  |  |  |  |  |
| CKM\_SHA3\_384\_HMAC\_GENERAL |  |  |  |  |  |  |  |
| CKM\_SHA3\_384\_HMAC |  |  |  |  |  |  |  |
| CKM\_SHA3\_384\_KEY\_DERIVATION |  |  |  |  |  |  |  |
| CKM\_SHA3\_384\_KEY\_GEN |  |  |  |  |  |  |  |

### Definitions

CKM\_SHA3\_384

CKM\_SHA3\_384\_HMAC

CKM\_SHA3\_384\_HMAC\_GENERAL

CKM\_SHA3\_384\_KEY\_DERIVATION

CKM\_SHA3\_384\_KEY\_GEN

CKK\_SHA3\_384\_HMAC

### SHA3-384 digest

The SHA3-384 mechanism, denoted **CKM\_SHA3\_384**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 384-bit message digest defined in FIPS PUB 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 134, SHA3-384: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 48 |

### General-length SHA3-384-HMAC

The general-length SHA3-384-HMAC mechanism, denoted **CKM\_SHA3\_384\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.4, except that it uses the HMAC construction based on the SHA-384 hash function and length of the output should be in the range 1-48.The keys it uses are generic secret keys and CKK\_SHA3\_384\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 24 bytes; that is, half the size of the SHA3-384 hash output.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-48 (the output size of SHA3-384 is 48 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

Table 135, General-length SHA3-384-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret or  CKK\_SHA3\_384\_HMAC | Any | 1-48, depending on parameters |
| C\_Verify | generic secret or  CKK\_SHA3\_384\_HMAC | Any | 1-48, depending on parameters |

### SHA3-384-HMAC

The SHA3-384-HMAC mechanism, denoted **CKM\_SHA3\_384\_HMAC**, is a special case of the general-length SHA3-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.

### SHA3-384 key derivation

SHA3-384 key derivation, denoted **CKM\_SHA3\_384\_KEY\_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

### SHA3-384 HMAC key generation

The SHA3-384-HMAC key generation mechanism, denoted **CKM\_SHA3\_384\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA3-384-HMAC.

It does not have a parameter.

The mechanism generates SHA3-384-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA3-384-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA3\_384\_HMAC** key sizes, in bytes.

## SHA3-512

*Table 136, SHA-512 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHA3\_512 |  |  |  |  |  |  |  |
| CKM\_SHA3\_512\_HMAC\_GENERAL |  |  |  |  |  |  |  |
| CKM\_SHA3\_512\_HMAC |  |  |  |  |  |  |  |
| CKM\_SHA3\_512\_KEY\_DERIVATION |  |  |  |  |  |  |  |
| CKM\_SHA3\_512\_KEY\_GEN |  |  |  |  |  |  |  |

### Definitions

CKM\_SHA3\_512

CKM\_SHA3\_512\_HMAC

CKM\_SHA3\_512\_HMAC\_GENERAL

CKM\_SHA3\_512\_KEY\_DERIVATION

CKM\_SHA3\_512\_KEY\_GEN

CKK\_SHA3\_512\_HMAC

### SHA3-512 digest

The SHA3-512 mechanism, denoted **CKM\_SHA3\_512**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 512-bit message digest defined in FIPS PUB 202.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 137, SHA3-512: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 64 |

### General-length SHA3-512-HMAC

The general-length SHA3-512-HMAC mechanism, denoted **CKM\_SHA3\_512\_HMAC\_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.4, except that it uses the HMAC construction based on the SHA3-512 hash function and length of the output should be in the range 1-64.The keys it uses are generic secret keys and CKK\_SHA3\_512\_HMAC. FIPS-198 compliant tokens may require the key length to be at least 32 bytes; that is, half the size of the SHA3-512 hash output.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-64 (the output size of SHA3-512 is 64 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

Table 138, General-length SHA3-512-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret or CKK\_SHA3\_512-HMAC | Any | 1-64, depending on parameters |
| C\_Verify | generic secret or CKK\_SHA3\_512\_HMAC | Any | 1-64, depending on parameters |

### SHA3-512-HMAC

The SHA3-512-HMAC mechanism, denoted **CKM\_SHA3\_512\_HMAC**, is a special case of the general-length SHA3-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

### SHA3-512 key derivation

SHA3-512 key derivation, denoted **CKM\_SHA3\_512\_KEY\_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA-512 hash function and the relevant length is 64 bytes.

### SHA3-512 HMAC key generation

The SHA3-512-HMAC key generation mechanism, denoted **CKM\_SHA3\_512\_KEY\_GEN**, is a key generation mechanism for NIST’s SHA3-512-HMAC.

It does not have a parameter.

The mechanism generates SHA3-512-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SHA3-512-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_SHA3\_512\_HMAC** key sizes, in bytes.

## SHAKE

*Table 139, SHA-512 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SHAKE\_128\_KEY\_DERIVATION |  |  |  |  |  |  |  |
| CKM\_SHAKE\_256\_KEY\_DERIVATION |  |  |  |  |  |  |  |

### Definitions

CKM\_SHAKE\_128\_KEY\_DERIVATION

CKM\_SHAKE\_256\_KEY\_DERIVATION

### SHAKE Key Derivation

SHAKE-128 and SHAKE-256 key derivation, denoted **CKM\_SHAKE\_128\_KEY\_DERIVATION** and **CKM\_SHAKE\_256\_KEY\_DERIVATION**, implements the SHAKE expansion function defined in FIPS 202 on the input key.

* If no length or key type is provided in the template a **CKR\_INVALID\_TEMPLATE** error is generated.
* If no key type is provided in the template, but a length is, then the key produced by this mechanism shall be a generic secret key of the specified length.
* If no length was provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism shall be of the type specified in the template. If it doesn’t, an error shall be returned.
* If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism shall be of the specified type and length.

If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key shall be set properly.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key shall as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key shall, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

## Blake2b-160

*Table 140, Blake2b-160 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_BLAKE2B\_160 |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_160\_HMAC |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_160\_HMAC\_GENERAL |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_160\_KEY\_DERIVE |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_160\_KEY\_GEN |  |  |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_BLAKE2B\_160

CKM\_BLAKE2B\_160\_HMAC

CKM\_BLAKE2B\_160\_HMAC\_GENERAL

CKM\_BLAKE2B\_160\_KEY\_DERIVE

CKM\_BLAKE2B\_160\_KEY\_GEN

CKK\_BLAKE2B\_160\_HMAC

### BLAKE2B-160 digest

The BLAKE2B-160 mechanism, denoted **CKM\_BLAKE2B\_160**, is a mechanism for message digesting, following the Blake2b Algorithm with a 160-bit message digest without a key as defined in [RFC 7693](https://tools.ietf.org/html/rfc7693).

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table , BLAKE2B-160: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 20 |

### General-length BLAKE2B-160-HMAC

The general-length BLAKE2B-160-HMAC mechanism, denoted **CKM\_BLAKE2B\_160\_HMAC\_GENERAL**, is the keyed variant of BLAKE2b-160 and length of the output should be in the range 1-20. The keys it uses are generic secret keys and CKK\_BLAKE2B\_160\_HMAC.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which holds the length in bytes of the desired output. This length should be in the range 1-20 (the output size of BLAKE2B-160 is 20 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 20-byte HMAC output.

Table 142, General-length BLAKE2B-160-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret or CKK\_BLAKE2B\_160\_HMAC | Any | 1-20, depending on parameters |
| C\_Verify | generic secret or CKK\_BLAKE2B\_160\_HMAC | Any | 1-20, depending on parameters |

### BLAKE2B-160-HMAC

The BLAKE2B-160-HMAC mechanism, denoted **CKM\_BLAKE2B\_160\_HMAC**, is a special case of the general-length BLAKE2B-160-HMAC mechanism.

It has no parameter, and always produces an output of length 20.

### BLAKE2B-160 key derivation

BLAKE2B-160 key derivation, denoted **CKM\_BLAKE2B\_160\_KEY\_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 2.20.5 except that it uses the BLAKE2B-160 hash function and the relevant length is 20 bytes.

### BLAKE2B-160 HMAC key generation

The BLAKE2B-160-HMAC key generation mechanism, denoted **CKM\_BLAKE2B\_160\_KEY\_GEN**, is a key generation mechanism for BLAKE2B-160-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-160-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-160-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_BLAKE2B\_160\_HMAC** key sizes, in bytes.

## BLAKE2B-256

*Table 143, BLAKE2B-256 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_BLAKE2B\_256 |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_256\_HMAC\_GENERAL |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_256\_HMAC |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_256\_KEY\_DERIVE |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_256\_KEY\_GEN |  |  |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_BLAKE2B\_256

CKM\_BLAKE2B\_256\_HMAC

CKM\_BLAKE2B\_256\_HMAC\_GENERAL

CKM\_BLAKE2B\_256\_KEY\_DERIVE

CKM\_BLAKE2B\_256\_KEY\_GEN

CKK\_BLAKE2B\_256\_HMAC

### BLAKE2B-256 digest

The BLAKE2B-256 mechanism, denoted **CKM\_BLAKE2B\_256**, is a mechanism for message digesting, following the Blake2b Algorithm with a 256-bit message digest without a key as defined in RFC 7693.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 144, BLAKE2B-256: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 32 |

### General-length BLAKE2B-256-HMAC

The general-length BLAKE2B-256-HMAC mechanism, denoted **CKM\_BLAKE2B\_256\_HMAC\_GENERAL**, is the keyed variant of Blake2b-256 and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK\_BLAKE2B\_256\_HMAC.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of BLAKE2B-256 is 32 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC output.

Table 145, General-length BLAKE2B-256-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret or CKK\_BLAKE2B\_256\_HMAC | Any | 1-32, depending on parameters |
| C\_Verify | generic secret or  CKK\_BLAKE2B\_256\_HMAC | Any | 1-32, depending on parameters |

### BLAKE2B-256-HMAC

The BLAKE2B-256-HMAC mechanism, denoted **CKM\_BLAKE2B\_256\_HMAC**, is a special case of the general-length BLAKE2B-256-HMAC mechanism in Section 2.22.3.

It has no parameter, and always produces an output of length 32.

### BLAKE2B-256 key derivation

BLAKE2B-256 key derivation, denoted CKM\_BLAKE2B\_256\_KEY\_DERIVE, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the BLAKE2B-256 hash function and the relevant length is 32 bytes.

### BLAKE2B-256 HMAC key generation

The BLAKE2B-256-HMAC key generation mechanism, denoted **CKM\_BLAKE2B\_256\_KEY\_GEN**, is a key generation mechanism for7 BLAKE2B-256-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-256-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_BLAKE2B\_256\_HMAC** key sizes, in bytes.

## BLAKE2B-384

*Table 146, BLAKE2B-384 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_BLAKE2B\_384 |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_384\_HMAC\_GENERAL |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_384\_HMAC |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_384\_KEY\_DERIVE |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_384\_KEY\_GEN |  |  |  |  |  |  |  |

### Definitions

CKM\_BLAKE2B\_384

CKM\_BLAKE2B\_384\_HMAC

CKM\_BLAKE2B\_384\_HMAC\_GENERAL

CKM\_BLAKE2B\_384\_KEY\_DERIVE

CKM\_BLAKE2B\_384\_KEY\_GEN

CKK\_BLAKE2B\_384\_HMAC

### BLAKE2B-384 digest

The BLAKE2B-384 mechanism, denoted **CKM\_BLAKE2B\_384**, is a mechanism for message digesting, following the Blake2b Algorithm with a 384-bit message digest without a key as defined in RFC 7693.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 147, BLAKE2B-384: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 48 |

### General-length BLAKE2B-384-HMAC

The general-length BLAKE2B-384-HMAC mechanism, denoted **CKM\_BLAKE2B\_384\_HMAC\_GENERAL**, is the keyed variant of the Blake2b-384 hash function and length of the output should be in the range 1-48.The keys it uses are generic secret keys and CKK\_BLAKE2B\_384\_HMAC.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-48 (the output size of BLAKE2B-384 is 48 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

Table 148, General-length BLAKE2B-384-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret or  CKK\_BLAKE2B\_384\_HMAC | Any | 1-48, depending on parameters |
| C\_Verify | generic secret or  CKK\_BLAKE2B\_384\_HMAC | Any | 1-48, depending on parameters |

### BLAKE2B-384-HMAC

The BLAKE2B-384-HMAC mechanism, denoted **CKM\_BLAKE2B\_384\_HMAC**, is a special case of the general-length BLAKE2B-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.

### BLAKE2B-384 key derivation

BLAKE2B-384 key derivation, denoted **CKM\_BLAKE2B\_384\_KEY\_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 2.20.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

### BLAKE2B-384 HMAC key generation

The BLAKE2B-384-HMAC key generation mechanism, denoted **CKM\_BLAKE2B\_384\_KEY\_GEN**, is a key generation mechanism for NIST’s BLAKE2B-384-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-384-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-384-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_BLAKE2B\_384\_HMAC** key sizes, in bytes.

## BLAKE2B-512

*Table 149, SHA-512 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_BLAKE2B\_512 |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_512\_HMAC\_GENERAL |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_512\_HMAC |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_512\_KEY\_DERIVE |  |  |  |  |  |  |  |
| CKM\_BLAKE2B\_512\_KEY\_GEN |  |  |  |  |  |  |  |

### Definitions

CKM\_BLAKE2B\_512

CKM\_BLAKE2B\_512\_HMAC

CKM\_BLAKE2B\_512\_HMAC\_GENERAL

CKM\_BLAKE2B\_512\_KEY\_DERIVE

CKM\_BLAKE2B\_512\_KEY\_GEN

CKK\_BLAKE2B\_512\_HMAC

### BLAKE2B-512 digest

The BLAKE2B-512 mechanism, denoted **CKM\_BLAKE2B\_512**, is a mechanism for message digesting, following the Blake2b Algorithm with a 512-bit message digest defined in RFC 7693.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 150, BLAKE2B-512: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | any | 64 |

### General-length BLAKE2B-512-HMAC

The general-length BLAKE2B-512-HMAC mechanism, denoted **CKM\_BLAKE2B\_512\_HMAC\_GENERAL**, is the keyed variant of the BLAKE2B-512 hash function and length of the output should be in the range 1-64.The keys it uses are generic secret keys and CKK\_BLAKE2B\_512\_HMAC.

It has a parameter, a CK\_MAC\_GENERAL\_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-64 (the output size of BLAKE2B-512 is 64 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

Table 151, General-length BLAKE2B-512-HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret or CKK\_BLAKE2B\_512-HMAC | Any | 1-64, depending on parameters |
| C\_Verify | generic secret or CKK\_BLAKE2B\_512\_HMAC | Any | 1-64, depending on parameters |

### BLAKE2B-512-HMAC

The BLAKE2B-512-HMAC mechanism, denoted **CKM\_BLAKE2B\_512\_HMAC**, is a special case of the general-length BLAKE2B-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

### BLAKE2B-512 key derivation

BLAKE2B-512 key derivation, denoted **CKM\_BLAKE2B\_512\_KEY\_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section2.20.5, except that it uses the Blake2b-512 hash function and the relevant length is 64 bytes.

### BLAKE2B-512 HMAC key generation

The BLAKE2B-512-HMAC key generation mechanism, denoted **CKM\_BLAKE2B\_512\_KEY\_GEN**, is a key generation mechanism for NIST’s BLAKE2B-512-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-512-HMAC keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-512-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of **CKM\_BLAKE2B\_512\_HMAC** key sizes, in bytes.

## PKCS #5 and PKCS #5-style password-based encryption (PBE)

The mechanisms in this section are for generating keys and IVs for performing password-based encryption. The method used to generate keys and IVs is specified in PKCS #5.

*Table 152, PKCS 5 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_PBE\_SHA1\_DES3\_EDE\_CBC |  |  |  |  | ✓ |  |  |
| CKM\_PBE\_SHA1\_DES2\_EDE\_CBC |  |  |  |  | ✓ |  |  |
| CKM\_PBA\_SHA1\_WITH\_SHA1\_HMAC |  |  |  |  | ✓ |  |  |
| CKM\_PKCS5\_PBKD2 |  |  |  |  | ✓ |  |  |

### Definitions

Mechanisms:

CKM\_PBE\_SHA1\_DES3\_EDE\_CBC

CKM\_PBE\_SHA1\_DES2\_EDE\_CBC

CKM\_PKCS5\_PBKD2

CKM\_PBA\_SHA1\_WITH\_SHA1\_HMAC

### Password-based encryption/authentication mechanism parameters

1. CK\_PBE\_PARAMS; CK\_PBE\_PARAMS\_PTR

**CK\_PBE\_PARAMS** is a structure which provides all of the necessary information required by the CKM\_PBE mechanisms (see PKCS #5 and PKCS #12 for information on the PBE generation mechanisms) and the CKM\_PBA\_SHA1\_WITH\_SHA1\_HMAC mechanism. It is defined as follows:

typedef struct CK\_PBE\_PARAMS {

CK\_BYTE\_PTR pInitVector;

CK\_UTF8CHAR\_PTR pPassword;

CK\_ULONG ulPasswordLen;

CK\_BYTE\_PTR pSalt;

CK\_ULONG ulSaltLen;

CK\_ULONG ulIteration;

} CK\_PBE\_PARAMS;

The fields of the structure have the following meanings:

pInitVector pointer to the location that receives the 8-byte initialization vector (IV), if an IV is required;

pPassword points to the password to be used in the PBE key generation;

ulPasswordLen length in bytes of the password information;

pSalt points to the salt to be used in the PBE key generation;

ulSaltLen length in bytes of the salt information;

ulIteration number of iterations required for the generation.

**CK\_PBE\_PARAMS\_PTR** is a pointer to a **CK\_PBE\_PARAMS**.

### PKCS #5 PBKDF2 key generation mechanism parameters

1. CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE; CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE\_PTR

**CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE** is used to indicate the Pseudo-Random Function (PRF) used to generate key bits using PKCS #5 PBKDF2. It is defined as follows:

typedef CK\_ULONG CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE;

The following PRFs are defined in PKCS #5 v2.1. The following table lists the defined functions.

Table 153, PKCS #5 PBKDF2 Key Generation: Pseudo-random functions

|  |  |  |
| --- | --- | --- |
| **PRF Identifier** | **Value** | **Parameter Type** |
| CKP\_PKCS5\_PBKD2\_HMAC\_SHA1 | 0x00000001UL | No Parameter. *pPrfData* must be NULL and *ulPrfDataLen* must be zero. |
| CKP\_PKCS5\_PBKD2\_HMAC\_GOSTR3411 | 0x00000002UL | This PRF uses GOST R34.11-94 hash to produce secret key value. *pPrfData* should point to DER-encoded OID, indicating GOSTR34.11-94 parameters. *ulPrfDataLen* holds encoded OID length in bytes. If *pPrfData* is set to NULL\_PTR, then *id-GostR3411-94-CryptoProParamSet* parameters will be used (RFC 4357, 11.2), and *ulPrfDataLen* must be 0. |
| CKP\_PKCS5\_PBKD2\_HMAC\_SHA224 | 0x00000003UL | No Parameter. *pPrfData*must be NULL and *ulPrfDataLen*must be zero. |
| CKP\_PKCS5\_PBKD2\_HMAC\_SHA256 | 0x00000004UL | No Parameter. *pPrfData*must be NULL and *ulPrfDataLen*must be zero. |
| CKP\_PKCS5\_PBKD2\_HMAC\_SHA384 | 0x00000005UL | No Parameter. *pPrfData*must be NULL and *ulPrfDataLen*must be zero. |
| CKP\_PKCS5\_PBKD2\_HMAC\_SHA512 | 0x00000006UL | No Parameter. *pPrfData*must be NULL and *ulPrfDataLen*must be zero. |
| CKP\_PKCS5\_PBKD2\_HMAC\_SHA512\_224 | 0x00000007UL | No Parameter. *pPrfData*must be NULL and *ulPrfDataLen*must be zero. |
| CKP\_PKCS5\_PBKD2\_HMAC\_SHA512\_256 | 0x00000008UL | No Parameter. *pPrfData*must be NULL and *ulPrfDataLen*must be zero. |

**CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE\_PTR** is a pointer to a **CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE**.

1. CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE; CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE\_PTR

**CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE** is used to indicate the source of the salt value when deriving a key using PKCS #5 PBKDF2. It is defined as follows:

typedef CK\_ULONG CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE;

The following salt value sources are defined in PKCS #5 v2.1. The following table lists the defined sources along with the corresponding data type for the *pSaltSourceData* field in the **CK\_PKCS5\_PBKD2\_PARAM** structure defined below.

Table 154, PKCS #5 PBKDF2 Key Generation: Salt sources

|  |  |  |
| --- | --- | --- |
| **Source Identifier** | **Value** | **Data Type** |
| CKZ\_SALT\_SPECIFIED | 0x00000001 | Array of CK\_BYTE containing the value of the salt value. |

**CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE\_PTR** is a pointer to a **CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE**.

1. CK\_PKCS5\_PBKD2\_PARAMS; CK\_PKCS5\_PBKD2\_PARAMS\_PTR

**CK\_PKCS5\_PBKD2\_PARAMS** is a structure that provides the parameters to the **CKM\_PKCS5\_PBKD2** mechanism. The structure is defined as follows:

typedef struct CK\_PKCS5\_PBKD2\_PARAMS {

CK\_PKCS5\_PBKDF2\_SALT\_SOURCE\_TYPE saltSource;

CK\_VOID\_PTR pSaltSourceData;

CK\_ULONG ulSaltSourceDataLen;

CK\_ULONG iterations;

CK\_PKCS5\_PBKD2\_PSEUDO\_RANDOM\_FUNCTION\_TYPE prf;

CK\_VOID\_PTR pPrfData;

CK\_ULONG ulPrfDataLen;

CK\_UTF8CHAR\_PTR pPassword;

CK\_ULONG\_PTR ulPasswordLen;

} CK\_PKCS5\_PBKD2\_PARAMS;

The fields of the structure have the following meanings:

saltSource source of the salt value

pSaltSourceData data used as the input for the salt source

ulSaltSourceDataLen length of the salt source input

iterations number of iterations to perform when generating each block of random data

prf pseudo-random function used to generate the key

pPrfData data used as the input for PRF in addition to the salt value

ulPrfDataLen length of the input data for the PRF

pPassword points to the password to be used in the PBE key generation

ulPasswordLen length in bytes of the password information

**CK\_PKCS5\_PBKD2\_PARAMS\_PTR** is a pointer to a **CK\_PKCS5\_PBKD2\_PARAMS**.

### PKCS #5 PBKD2 key generation

PKCS #5 PBKDF2 key generation, denoted **CKM\_PKCS5\_PBKD2**, is a mechanism used for generating a secret key from a password and a salt value. This functionality is defined in PKCS#5 as PBKDF2.

It has a parameter, a **CK\_PKCS5\_PBKD2\_PARAMS** structure. The parameter specifies the salt value source, pseudo-random function, and iteration count used to generate the new key.

Since this mechanism can be used to generate any type of secret key, new key templates must contain the **CKA\_KEY\_TYPE** and **CKA\_VALUE\_LEN** attributes. If the key type has a fixed length the **CKA\_VALUE\_LEN** attribute may be omitted.

## PKCS #12 password-based encryption/authentication mechanisms

The mechanisms in this section are for generating keys and IVs for performing password-based encryption or authentication. The method used to generate keys and IVs is based on a method that was specified in PKCS #12.

We specify here a general method for producing various types of pseudo-random bits from a password, *p*; a string of salt bits, *s*; and an iteration count, *c*. The “type” of pseudo-random bits to be produced is identified by an identification byte, *ID*, the meaning of which will be discussed later.

Let H be a hash function built around a compression function *f:* ***Z****2u ×* ***Z****2v →* ***Z****2u* (that is, H has a chaining variable and output of length *u* bits, and the message input to the compression function of H is *v* bits). For MD2 and MD5, *u*=128 and *v*=512; for SHA-1, *u*=160 and *v*=512.

We assume here that *u* and *v* are both multiples of 8, as are the lengths in bits of the password and salt strings and the number *n* of pseudo-random bits required. In addition, *u* and *v* are of course nonzero.

1. Construct a string, *D* (the “diversifier”), by concatenating *v*/8 copies of *ID*.
2. Concatenate copies of the salt together to create a string *S* of length *v*⋅⎡*s/v*⎤ bits (the final copy of the salt may be truncated to create *S*). Note that if the salt is the empty string, then so is *S*.
3. Concatenate copies of the password together to create a string *P* of length *v*⋅⎡*p/v*⎤ bits (the final copy of the password may be truncated to create *P*). Note that if the password is the empty string, then so is *P*.
4. Set *I*=*S*||*P* to be the concatenation of *S* and *P*.
5. Set *j*=⎡*n*/*u*⎤.
6. For *i*=1, 2, …, *j*, do the following:
   1. Set *Ai*=H*c*(*D*||*I*), the *c*th hash of *D*||*I*. That is, compute the hash of *D*||*I*; compute the hash of that hash; etc.; continue in this fashion until a total of *c* hashes have been computed, each on the result of the previous hash.
   2. Concatenate copies of *Ai* to create a string *B* of length *v* bits (the final copy of *Ai* may be truncated to create *B*).
   3. Treating *I* as a concatenation *I*0, *I*1, …, *Ik*-1 of *v*-bit blocks, where *k*=⎡*s/v*⎤+⎡*p/v*⎤, modify *I* by setting *Ij*=(*Ij*+*B*+1) mod 2*v* for each *j*. To perform this addition, treat each *v*-bit block as a binary number represented most-significant bit first.
7. Concatenate *A*1, *A*2, …, *Aj* together to form a pseudo-random bit string, *A*.
8. Use the first *n* bits of *A* as the output of this entire process.

When the password-based encryption mechanisms presented in this section are used to generate a key and IV (if needed) from a password, salt, and an iteration count, the above algorithm is used. To generate a key, the identifier byte *ID* is set to the value 1; to generate an IV, the identifier byte *ID* is set to the value 2.

When the password based authentication mechanism presented in this section is used to generate a key from a password, salt, and an iteration count, the above algorithm is used. The identifier byte *ID* is set to the value 3.

### SHA-1-PBE for 3-key triple-DES-CBC

SHA-1-PBE for 3-key triple-DES-CBC, denoted **CKM\_PBE\_SHA1\_DES3\_EDE\_CBC**, is a mechanism used for generating a 3-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 3-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK\_PBE\_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

### SHA-1-PBE for 2-key triple-DES-CBC

SHA-1-PBE for 2-key triple-DES-CBC, denoted **CKM\_PBE\_SHA1\_DES2\_EDE\_CBC**, is a mechanism used for generating a 2-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 2-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK\_PBE\_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

### SHA-1-PBA for SHA-1-HMAC

SHA-1-PBA for SHA-1-HMAC, denoted **CKM\_PBA\_SHA1\_WITH\_SHA1\_HMAC**, is a mechanism used for generating a 160-bit generic secret key from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key is described above.

It has a parameter, a **CK\_PBE\_PARAMS** structure. The parameter specifies the input information for the key generation process. The parameter also has a field to hold the location of an application-supplied buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since authentication with SHA-1-HMAC does not require an IV.

The key generated by this mechanism will typically be used for computing a SHA-1 HMAC to perform password-based authentication (not *password-based encryption*). At the time of this writing, this is primarily done to ensure the integrity of a PKCS #12 PDU.

## SSL

*Table 155,SSL Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_SSL3\_MASTER\_KEY\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_SSL3\_MASTER\_KEY\_DERIVE\_DH |  |  |  |  |  |  | ✓ |
| CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_SSL3\_MD5\_MAC |  | ✓ |  |  |  |  |  |
| CKM\_SSL3\_SHA1\_MAC |  | ✓ |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN

CKM\_SSL3\_MASTER\_KEY\_DERIVE

CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE

CKM\_SSL3\_MASTER\_KEY\_DERIVE\_DH

CKM\_SSL3\_MD5\_MAC

CKM\_SSL3\_SHA1\_MAC

### SSL mechanism parameters

1. CK\_SSL3\_RANDOM\_DATA

**CK\_SSL3\_RANDOM\_DATA** is a structure which provides information about the random data of a client and a server in an SSL context. This structure is used by both the **CKM\_SSL3\_MASTER\_KEY\_DERIVE** and the **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE** mechanisms. It is defined as follows:

typedef struct CK\_SSL3\_RANDOM\_DATA {

CK\_BYTE\_PTR pClientRandom;

CK\_ULONG ulClientRandomLen;

CK\_BYTE\_PTR pServerRandom;

CK\_ULONG ulServerRandomLen;

} CK\_SSL3\_RANDOM\_DATA;

The fields of the structure have the following meanings:

pClientRandom pointer to the client’s random data

ulClientRandomLen length in bytes of the client’s random data

pServerRandom pointer to the server’s random data

ulServerRandomLen length in bytes of the server’s random data

1. CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS; CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR

**CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_SSL3\_MASTER\_KEY\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS {

CK\_SSL3\_RANDOM\_DATA RandomInfo;

CK\_VERSION\_PTR pVersion;

} CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

RandomInfo client’s and server’s random data information.

pVersion pointer to a **CK\_VERSION** structure which receives the SSL protocol version information

**CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS**.

1. CK\_SSL3\_KEY\_MAT\_OUT; CK\_SSL3\_KEY\_MAT\_OUT\_PTR

**CK\_SSL3\_KEY\_MAT\_OUT** is a structure that contains the resulting key handles and initialization vectors after performing a C\_DeriveKey function with the **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_SSL3\_KEY\_MAT\_OUT {

CK\_OBJECT\_HANDLE hClientMacSecret;

CK\_OBJECT\_HANDLE hServerMacSecret;

CK\_OBJECT\_HANDLE hClientKey;

CK\_OBJECT\_HANDLE hServerKey;

CK\_BYTE\_PTR pIVClient;

CK\_BYTE\_PTR pIVServer;

} CK\_SSL3\_KEY\_MAT\_OUT;

The fields of the structure have the following meanings:

hClientMacSecret key handle for the resulting Client MAC Secret key

hServerMacSecret key handle for the resulting Server MAC Secret key

hClientKey key handle for the resulting Client Secret key

hServerKey key handle for the resulting Server Secret key

pIVClient pointer to a location which receives the initialization vector (IV) created for the client (if any)

pIVServer pointer to a location which receives the initialization vector (IV) created for the server (if any)

**CK\_SSL3\_KEY\_MAT\_OUT\_PTR** is a pointer to a **CK\_SSL3\_KEY\_MAT\_OUT**.

1. CK\_SSL3\_KEY\_MAT\_PARAMS; CK\_SSL3\_KEY\_MAT\_PARAMS\_PTR

**CK\_SSL3\_KEY\_MAT\_PARAMS** is a structure that provides the parameters to the **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_SSL3\_KEY\_MAT\_PARAMS {

CK\_ULONG ulMacSizeInBits;

CK\_ULONG ulKeySizeInBits;

CK\_ULONG ulIVSizeInBits;

CK\_BBOOL bIsExport;

CK\_SSL3\_RANDOM\_DATA RandomInfo;

CK\_SSL3\_KEY\_MAT\_OUT\_PTR pReturnedKeyMaterial;

} CK\_SSL3\_KEY\_MAT\_PARAMS;

The fields of the structure have the following meanings:

ulMacSizeInBits the length (in bits) of the MACing keys agreed upon during the protocol handshake phase

ulKeySizeInBits the length (in bits) of the secret keys agreed upon during the protocol handshake phase

ulIVSizeInBits the length (in bits) of the IV agreed upon during the protocol handshake phase. If no IV is required, the length should be set to 0

bIsExport a Boolean value which indicates whether the keys have to be derived for an export version of the protocol

RandomInfo client’s and server’s random data information.

pReturnedKeyMaterial points to a CK\_SSL3\_KEY\_MAT\_OUT structures which receives the handles for the keys generated and the IVs

**CK\_SSL3\_KEY\_MAT\_PARAMS\_PTR** is a pointer to a **CK\_SSL3\_KEY\_MAT\_PARAMS**.

### Pre-master key generation

Pre-master key generation in SSL 3.0, denoted **CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN**, is a mechanism which generates a 48-byte generic secret key. It is used to produce the "pre\_master" key used in SSL version 3.0 for RSA-like cipher suites.

It has one parameter, a **CK\_VERSION** structure, which provides the client’s SSL version number.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C\_GenerateKey** call may indicate that the object class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure both indicate 48 bytes.

### Master key derivation

Master key derivation in SSL 3.0, denoted **CKM\_SSL3\_MASTER\_KEY\_DERIVE**, is a mechanism used to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the "master\_secret" key used in the SSL protocol from the "pre\_master" key. This mechanism returns the value of the client version, which is built into the "pre\_master" key as well as a handle to the derived "master\_secret" key.

It has a parameter, a **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section .

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template; otherwise they are assigned default values.

The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure both indicate 48 bytes.

Note that the **CK\_VERSION** structure pointed to by the **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure’s *pVersion* field will be modified by the **C\_DeriveKey** call. In particular, when the call returns, this structure will hold the SSL version associated with the supplied pre\_master key.

Note that this mechanism is only useable for cipher suites that use a 48-byte “pre\_master” secret with an embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher suites.

### Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in SSL 3.0, denoted **CKM\_SSL3\_MASTER\_KEY\_DERIVE\_DH**, is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret key. It is used to produce the "master\_secret" key used in the SSL protocol from the "pre\_master" key.

It has a parameter, a **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the passing of random data to the token. This structure is defined in Section . The *pVersion* field of the structure must be set to NULL\_PTR since the version number is not embedded in the "pre\_master" key as it is for RSA-like cipher suites.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure both indicate 48 bytes.

Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte “pre\_master” secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but excludes the RSA cipher suites.

### Key and MAC derivation

Key, MAC and IV derivation in SSL 3.0, denoted **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the "master\_secret" key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IVs created.

It has a parameter, a **CK\_SSL3\_KEY\_MAT\_PARAMS** structure, which allows for the passing of random data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a structure which receives the handles and IVs which were generated. This structure is defined in Section .

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of **CKO\_SECRET\_KEY**.

The two MACing keys ("client\_write\_MAC\_secret" and "server\_write\_MAC\_secret") are always given a type of **CKK\_GENERIC\_SECRET**. They are flagged as valid for signing, verification, and derivation operations.

The other two keys ("client\_write\_key" and "server\_write\_key") are typed according to information found in the template sent along with this mechanism during a **C\_DeriveKey** function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

IVs will be generated and returned if the *ulIVSizeInBits* field of the **CK\_SSL3\_KEY\_MAT\_PARAMS** field has a nonzero value. If they are generated, their length in bits will agree with the value in the *ulIVSizeInBits* field.

All four keys inherit the values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**, **CKA\_EXTRACTABLE**, and **CKA\_NEVER\_EXTRACTABLE** attributes from the base key. The template provided to **C\_DeriveKey** may not specify values for any of these attributes which differ from those held by the base key.

Note that the **CK\_SSL3\_KEY\_MAT\_OUT** structure pointed to by the **CK\_SSL3\_KEY\_MAT\_PARAMS** structure’s *pReturnedKeyMaterial* field will be modified by the **C\_DeriveKey** call. In particular, the four key handle fields in the **CK\_SSL3\_KEY\_MAT\_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffers pointed to by the **CK\_SSL3\_KEY\_MAT\_OUT** structure’s *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller). Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE** mechanism returns all of its key handles in the **CK\_SSL3\_KEY\_MAT\_OUT** structure pointed to by the **CK\_SSL3\_KEY\_MAT\_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C\_DeriveKey** is unnecessary, and should be a NULL\_PTR.

If a call to **C\_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the token.

### MD5 MACing in SSL 3.0

MD5 MACing in SSL3.0, denoted **CKM\_SSL3\_MD5\_MAC**, is a mechanism for single- and multiple-part signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which specifies the length in bytes of the signatures produced by this mechanism.

Constraints on key types and the length of input and output data are summarized in the following table:

Table 156, MD5 MACing in SSL 3.0: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret | any | 4-8, depending on parameters |
| C\_Verify | generic secret | any | 4-8, depending on parameters |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of generic secret key sizes, in bits.

### SHA-1 MACing in SSL 3.0

SHA-1 MACing in SSL3.0, denoted **CKM\_SSL3\_SHA1\_MAC**, is a mechanism for single- and multiple-part signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This technique is very similar to the HMAC technique.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS**, which specifies the length in bytes of the signatures produced by this mechanism.

Constraints on key types and the length of input and output data are summarized in the following table:

Table 157, SHA-1 MACing in SSL 3.0: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret | any | 4-8, depending on parameters |
| C\_Verify | generic secret | any | 4-8, depending on parameters |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of generic secret key sizes, in bits.

## TLS 1.2 Mechanisms

Details for TLS 1.2 and its key derivation and MAC mechanisms can be found in [TLS12]. TLS 1.2 mechanisms differ from TLS 1.0 and 1.1 mechanisms in that the base hash used in the underlying TLS PRF (pseudo-random function) can be negotiated. Therefore each mechanism parameter for the TLS 1.2 mechanisms contains a new value in the parameters structure to specify the hash function.

This section also specifies CKM\_TLS12\_MAC which should be used in place of **CKM\_TLS\_PRF** to calculate the verify\_data in the TLS "finished" message.

This section also specifies **CKM\_TLS\_KDF** that can be used in place of **CKM\_TLS\_PRF** to implement key material exporters.

*Table 158, TLS 1.2 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_TLS12\_MASTER\_KEY\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_TLS12\_MASTER\_KEY\_DERIVE\_DH |  |  |  |  |  |  | ✓ |
| CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_TLS12\_KEY\_SAFE\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_TLS\_KDF |  |  |  |  |  |  | ✓ |
| CKM\_TLS12\_MAC |  | ✓ |  |  |  |  |  |
| CKM\_TLS12\_KDF |  |  |  |  |  |  | ✓ |

### Definitions

Mechanisms:

CKM\_TLS12\_MASTER\_KEY\_DERIVE

CKM\_TLS12\_MASTER\_KEY\_DERIVE\_DH

CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE

CKM\_TLS12\_KEY\_SAFE\_DERIVE

CKM\_TLS\_KDF

CKM\_TLS12\_MAC

CKM\_TLS12\_KDF

### TLS 1.2 mechanism parameters

* CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS; CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR

**CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_TLS12\_MASTER\_KEY\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS {

CK\_SSL3\_RANDOM\_DATA RandomInfo;

CK\_VERSION\_PTR pVersion;

CK\_MECHANISM\_TYPE prfHashMechanism;

} CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

RandomInfo client’s and server’s random data information.

pVersion pointer to a **CK\_VERSION** structure which receives the SSL protocol version information

prfHashMechanism base hash used in the underlying TLS1.2 PRF operation used to derive the master key.

**CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_TLS12\_MASTER\_KEY\_DERIVE\_PARAMS**.

* CK\_TLS12\_KEY\_MAT\_PARAMS; CK\_TLS12\_KEY\_MAT\_PARAMS\_PTR

**CK\_TLS12\_KEY\_MAT\_PARAMS** is a structure that provides the parameters to the **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_TLS12\_KEY\_MAT\_PARAMS {

CK\_ULONG ulMacSizeInBits;

CK\_ULONG ulKeySizeInBits;

CK\_ULONG ulIVSizeInBits;

CK\_BBOOL bIsExport;

CK\_SSL3\_RANDOM\_DATA RandomInfo;

CK\_SSL3\_KEY\_MAT\_OUT\_PTR pReturnedKeyMaterial;

CK\_MECHANISM\_TYPE prfHashMechanism;

} CK\_TLS12\_KEY\_MAT\_PARAMS;

The fields of the structure have the following meanings:

ulMacSizeInBits the length (in bits) of the MACing keys agreed upon during the protocol handshake phase. If no MAC key is required, the length should be set to 0.

ulKeySizeInBits the length (in bits) of the secret keys agreed upon during the protocol handshake phase

ulIVSizeInBits the length (in bits) of the IV agreed upon during the protocol handshake phase. If no IV is required, the length should be set to 0

bIsExport must be set to CK\_FALSE because export cipher suites must not be used in TLS 1.1 and later.

RandomInfo client’s and server’s random data information.

pReturnedKeyMaterial points to a CK\_SSL3\_KEY\_MAT\_OUT structures which receives the handles for the keys generated and the IVs

prfHashMechanism base hash used in the underlying TLS1.2 PRF operation used to derive the master key.

**CK\_TLS12\_KEY\_MAT\_PARAMS\_PTR** is a pointer to a **CK\_TLS12\_KEY\_MAT\_PARAMS**.

* CK\_TLS\_KDF\_PARAMS; CK\_TLS\_KDF\_PARAMS\_PTR

**CK\_TLS\_KDF\_PARAMS** is a structure that provides the parameters to the CKM\_TLS\_KDF mechanism. It is defined as follows:

typedef struct CK\_TLS\_KDF\_PARAMS {

CK\_MECHANISM\_TYPE prfMechanism;

CK\_BYTE\_PTR pLabel;

CK\_ULONG ulLabelLength;

CK\_SSL3\_RANDOM\_DATA RandomInfo;

CK\_BYTE\_PTR pContextData;

CK\_ULONG ulContextDataLength;

} CK\_TLS\_KDF\_PARAMS;

The fields of the structure have the following meanings:

prfMechanism the hash mechanism used in the TLS1.2 PRF construct or CKM\_TLS\_PRF to use with the TLS1.0 and 1.1 PRF construct.

pLabel a pointer to the label for this key derivation

ulLabelLength length of the label in bytes

RandomInfo the random data for the key derivation

pContextData a pointer to the context data for this key derivation. NULL\_PTR if not present

ulContextDataLength length of the context data in bytes. 0 if not present.

* CK\_TLS\_MAC\_PARAMS; CK\_TLS\_MAC\_PARAMS\_PTR

**CK\_TLS\_MAC\_PARAMS** is a structure that provides the parameters to the **CKM\_TLS\_MAC** mechanism. It is defined as follows:

typedef struct CK\_TLS\_MAC\_PARAMS {

CK\_MECHANISM\_TYPE prfMechanism;

CK\_ULONG ulMacLength;

CK\_ULONG ulServerOrClient;

} CK\_TLS\_MAC\_PARAMS;

The fields of the structure have the following meanings:

prfMechanism the hash mechanism used in the TLS12 PRF construct or CKM\_TLS\_PRF to use with the TLS1.0 and 1.1 PRF construct.

ulMacLength the length of the MAC tag required or offered. Always 12 octets in TLS 1.0 and 1.1. Generally 12 octets, but may be negotiated to a longer value in TLS1.2.

ulServerOrClient 1 to use the label "server finished", 2 to use the label "client finished". All other values are invalid.

**CK\_TLS\_MAC\_PARAMS\_PTR** is a pointer to a **CK\_TLS\_MAC\_PARAMS**.

1. CK\_TLS\_PRF\_PARAMS; CK\_TLS\_PRF\_PARAMS\_PTR

**CK\_TLS\_PRF\_PARAMS** is a structure, which provides the parameters to the **CKM\_TLS\_PRF** mechanism. It is defined as follows:

typedef struct CK\_TLS\_PRF\_PARAMS {

CK\_BYTE\_PTR pSeed;

CK\_ULONG ulSeedLen;

CK\_BYTE\_PTR pLabel;

CK\_ULONG ulLabelLen;

CK\_BYTE\_PTR pOutput;

CK\_ULONG\_PTR pulOutputLen;

} CK\_TLS\_PRF\_PARAMS;

The fields of the structure have the following meanings:

pSeed pointer to the input seed

ulSeedLen length in bytes of the input seed

pLabel pointer to the identifying label

ulLabelLen length in bytes of the identifying label

pOutput pointer receiving the output of the operation

pulOutputLen pointer to the length in bytes that the output to be created shall have, has to hold the desired length as input and will receive the calculated length as output

CK\_TLS\_PRF\_PARAMS\_PTR is a pointer to a CK\_TLS\_PRF\_PARAMS.

### TLS MAC

The TLS MAC mechanism is used to generate integrity tags for the TLS "finished" message. It replaces the use of the **CKM\_TLS\_PRF** function for TLS1.0 and 1.1 and that mechanism is deprecated.

**CKM\_TLS\_MAC** takes a parameter of CK\_TLS\_MAC\_PARAMS. To use this mechanism with TLS1.0 and TLS1.1, use **CKM\_TLS\_PRF** as the value for *prfMechanism* in place of a hash mechanism. Note: Although **CKM\_TLS\_PRF** is deprecated as a mechanism for C\_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 pseudo-random function.

In TLS1.0 and 1.1 the "finished" message verify\_data (i.e. the output signature from the MAC mechanism) is always 12 bytes. In TLS1.2 the "finished" message verify\_data is a minimum of 12 bytes, defaults to 12 bytes, but may be negotiated to longer length.

Table 159, General-length TLS MAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | generic secret | any | >=12 bytes |
| C\_Verify | generic secret | any | >=12 bytes |

### Master key derivation

Master key derivation in TLS 1.0, denoted **CKM\_TLS\_MASTER\_KEY\_DERIVE**, is a mechanism used to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the "master\_secret" key used in the TLS protocol from the "pre\_master" key. This mechanism returns the value of the client version, which is built into the "pre\_master" key as well as a handle to the derived "master\_secret" key.

It has a parameter, a **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section .

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The mechanism also contributes the CKA\_ALLOWED\_MECHANISMS attribute consisting only of **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE, CKM\_TLS12\_KEY\_SAFE\_DERIVE, CKM\_TLS12\_KDF** and **CKM\_TLS12\_MAC**.

The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure both indicate 48 bytes.

Note that the **CK\_VERSION** structure pointed to by the **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure’s *pVersion* field will be modified by the **C\_DeriveKey** call. In particular, when the call returns, this structure will hold the SSL version associated with the supplied pre\_master key.

Note that this mechanism is only useable for cipher suites that use a 48-byte “pre\_master” secret with an embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher suites.

### Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in TLS 1.0, denoted **CKM\_TLS\_MASTER\_KEY\_DERIVE\_DH**, is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret key. It is used to produce the "master\_secret" key used in the TLS protocol from the "pre\_master" key.

It has a parameter, a **CK\_SSL3\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the passing of random data to the token. This structure is defined in Section . The *pVersion* field of the structure must be set to NULL\_PTR since the version number is not embedded in the "pre\_master" key as it is for RSA-like cipher suites.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The mechanism also contributes the CKA\_ALLOWED\_MECHANISMS attribute consisting only of **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE, CKM\_TLS12\_KEY\_SAFE\_DERIVE, CKM\_TLS12\_KDF** and **CKM\_TLS12\_MAC**.

The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure both indicate 48 bytes.

Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte “pre\_master” secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but excludes the RSA cipher suites.

### Key and MAC derivation

Key, MAC and IV derivation in TLS 1.0, denoted **CKM\_TLS\_KEY\_AND\_MAC\_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the "master\_secret" key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IVs created.

It has a parameter, a **CK\_SSL3\_KEY\_MAT\_PARAMS** structure, which allows for the passing of random data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a structure which receives the handles and IVs which were generated. This structure is defined in Section .

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of **CKO\_SECRET\_KEY**.

The two MACing keys ("client\_write\_MAC\_secret" and "server\_write\_MAC\_secret") (if present) are always given a type of **CKK\_GENERIC\_SECRET**. They are flagged as valid for signing and verification.

The other two keys ("client\_write\_key" and "server\_write\_key") are typed according to information found in the template sent along with this mechanism during a **C\_DeriveKey** function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

For **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE**, IVs will be generated and returned if the *ulIVSizeInBits* field of the **CK\_SSL3\_KEY\_MAT\_PARAMS** field has a nonzero value. If they are generated, their length in bits will agree with the value in the *ulIVSizeInBits* field.

Note Well: CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE produces both private (key) and public (IV) data. It is possible to "leak" private data by the simple expedient of decreasing the length of private data requested. E.g. Setting ulMacSizeInBits and ulKeySizeInBits to 0 (or other lengths less than the key size) will result in the private key data being placed in the destination designated for the IV's. Repeated calls with the same master key and same RandomInfo but with differing lengths for the private key material will result in different data being leaked.<

All four keys inherit the values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**, **CKA\_EXTRACTABLE**, and **CKA\_NEVER\_EXTRACTABLE** attributes from the base key. The template provided to **C\_DeriveKey** may not specify values for any of these attributes which differ from those held by the base key.

Note that the **CK\_SSL3\_KEY\_MAT\_OUT** structure pointed to by the **CK\_SSL3\_KEY\_MAT\_PARAMS** structure’s *pReturnedKeyMaterial* field will be modified by the **C\_DeriveKey** call. In particular, the four key handle fields in the **CK\_SSL3\_KEY\_MAT\_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffers pointed to by the **CK\_SSL3\_KEY\_MAT\_OUT** structure’s *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller). Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE** mechanism returns all of its key handles in the **CK\_SSL3\_KEY\_MAT\_OUT** structure pointed to by the **CK\_SSL3\_KEY\_MAT\_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C\_DeriveKey** is unnecessary, and should be a NULL\_PTR.

If a call to **C\_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the token.

### CKM\_TLS12\_KEY\_SAFE\_DERIVE

**CKM\_TLS12\_KEY\_SAFE\_DERIVE** is identical to **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE** except that it shall never produce IV data, and the ulIvSizeInBits field of **CK\_TLS12\_KEY\_MAT\_PARAMS** is ignored and treated as 0. All of the other conditions and behavior described for CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE, with the exception of the black box warning, apply to this mechanism.

CKM\_TLS12\_KEY\_SAFE\_DERIVE is provided as a separate mechanism to allow a client to control the export of IV material (and possible leaking of key material) through the use of the CKA\_ALLOWED\_MECHANISMS key attribute.

### Generic Key Derivation using the TLS PRF

**CKM\_TLS\_KDF** is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF function to produce additional key material for protocols that want to leverage the TLS key negotiation mechanism. **CKM\_TLS\_KDF** has a parameter of **CK\_TLS\_KDF\_PARAMS**. If the protocol using this mechanism does not use context information, the *pContextData* field shall be set to NULL\_PTR and the *ulContextDataLength* field shall be set to 0.

To use this mechanism with TLS1.0 and TLS1.1, use **CKM\_TLS\_PRF** as the value for *prfMechanism* in place of a hash mechanism. Note: Although **CKM\_TLS\_PRF** is deprecated as a mechanism for C\_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 Pseudo-random function.

This mechanism can be used to derive multiple keys (e.g. similar to **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE**) by first deriving the key stream as a **CKK\_GENERIC\_SECRET** of the necessary length and doing subsequent derives against that derived key using the **CKM\_EXTRACT\_KEY\_FROM\_KEY** mechanism to split the key stream into the actual operational keys.

The mechanism should not be used with the labels defined for use with TLS, but the token does not enforce this behavior.

This mechanism has the following rules about key sensitivity and extractability:

* If the original key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key’s **CKA\_SENSITIVE** attribute is set either from the supplied template or from the original key.
* Similarly, if the original key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key’s **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from the original key.
* The derived key’s **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the original key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
* Similarly, the derived key’s **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if the original key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

### Generic Key Derivation using the TLS12 PRF

**CKM\_TLS12\_KDF** is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF function to produce additional key material for protocols that want to leverage the TLS key negotiation mechanism. **CKM\_TLS12\_KDF** has a parameter of **CK\_TLS\_KDF\_PARAMS**. If the protocol using this mechanism does not use context information, the *pContextData* field shall be set to NULL\_PTR and the *ulContextDataLength* field shall be set to 0.

To use this mechanism with TLS1.0 and TLS1.1, use **CKM\_TLS\_PRF** as the value for *prfMechanism* in place of a hash mechanism. Note: Although **CKM\_TLS\_PRF** is deprecated as a mechanism for C\_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 Pseudo-random function.

This mechanism can be used to derive multiple keys (e.g. similar to **CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE**) by first deriving the key stream as a **CKK\_GENERIC\_SECRET** of the necessary length and doing subsequent derives against that derived key stream using the **CKM\_EXTRACT\_KEY\_FROM\_KEY** mechanism to split the key stream into the actual operational keys.

The mechanism should not be used with the labels defined for use with TLS, but the token does not enforce this behavior.

This mechanism has the following rules about key sensitivity and extractability:

* If the original key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key’s **CKA\_SENSITIVE** attribute is set either from the supplied template or from the original key.
* Similarly, if the original key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key’s **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from the original key.
* The derived key’s **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the original key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
* Similarly, the derived key’s **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if the original key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

## WTLS

Details can be found in [WTLS].

When comparing the existing TLS mechanisms with these extensions to support WTLS one could argue that there would be no need to have distinct handling of the client and server side of the handshake. However, since in WTLS the server and client use different sequence numbers, there could be instances (e.g. when WTLS is used to protect asynchronous protocols) where sequence numbers on the client and server side differ, and hence this motivates the introduced split.

*Table 160, WTLS Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_WTLS\_PRE\_MASTER\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_WTLS\_MASTER\_KEY\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_WTLS\_MASTER\_KEY\_DERIVE\_DH\_ECC |  |  |  |  |  |  | ✓ |
| CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_WTLS\_PRF |  |  |  |  |  |  | ✓ |

### Definitions

Mechanisms:

CKM\_WTLS\_PRE\_MASTER\_KEY\_GEN

CKM\_WTLS\_MASTER\_KEY\_DERIVE

CKM\_WTLS\_MASTER\_KEY\_DERIVE\_DH\_ECC

CKM\_WTLS\_PRF

CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE

CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE

### WTLS mechanism parameters

1. CK\_WTLS\_RANDOM\_DATA; CK\_WTLS\_RANDOM\_DATA\_PTR

**CK\_WTLS\_RANDOM\_DATA** is a structure, which provides information about the random data of a client and a server in a WTLS context. This structure is used by the **CKM\_WTLS\_MASTER\_KEY\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_WTLS\_RANDOM\_DATA {

CK\_BYTE\_PTR pClientRandom;

CK\_ULONG ulClientRandomLen;

CK\_BYTE\_PTR pServerRandom;

CK\_ULONG ulServerRandomLen;

} CK\_WTLS\_RANDOM\_DATA;

The fields of the structure have the following meanings:

pClientRandom pointer to the client’s random data

pClientRandomLen length in bytes of the client’s random data

pServerRaondom pointer to the server’s random data

ulServerRandomLen length in bytes of the server’s random data

**CK\_WTLS\_RANDOM\_DATA\_PTR** is a pointer to a **CK\_WTLS\_RANDOM\_DATA**.

1. CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS; CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS \_PTR

**CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS** is a structure, which provides the parameters to the **CKM\_WTLS\_MASTER\_KEY\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS {

CK\_MECHANISM\_TYPE DigestMechanism;

CK\_WTLS\_RANDOM\_DATA RandomInfo;

CK\_BYTE\_PTR pVersion;

} CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

DigestMechanism the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])

RandomInfo Client’s and server’s random data information

pVersion pointer to a **CK\_BYTE** which receives the WTLS protocol version information

**CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS**.

1. CK\_WTLS\_PRF\_PARAMS; CK\_WTLS\_PRF\_PARAMS\_PTR

**CK\_WTLS\_PRF\_PARAMS** is a structure, which provides the parameters to the **CKM\_WTLS\_PRF** mechanism. It is defined as follows:

typedef struct CK\_WTLS\_PRF\_PARAMS {

CK\_MECHANISM\_TYPE DigestMechanism;

CK\_BYTE\_PTR pSeed;

CK\_ULONG ulSeedLen;

CK\_BYTE\_PTR pLabel;

CK\_ULONG ulLabelLen;

CK\_BYTE\_PTR pOutput;

CK\_ULONG\_PTR pulOutputLen;

} CK\_WTLS\_PRF\_PARAMS;

The fields of the structure have the following meanings:

Digest Mechanism the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])

pSeed pointer to the input seed

ulSeedLen length in bytes of the input seed

pLabel pointer to the identifying label

ulLabelLen length in bytes of the identifying label

pOutput pointer receiving the output of the operation

pulOutputLen pointer to the length in bytes that the output to be created shall have, has to hold the desired length as input and will receive the calculated length as output

**CK\_WTLS\_PRF\_PARAMS\_PTR** is a pointer to a **CK\_WTLS\_PRF\_PARAMS**.

1. CK\_WTLS\_KEY\_MAT\_OUT; CK\_WTLS\_KEY\_MAT\_OUT\_PTR

**CK\_WTLS\_KEY\_MAT\_OUT** is a structure that contains the resulting key handles and initialization vectors after performing a C\_DeriveKey function with the **CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE** or with the **CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_WTLS\_KEY\_MAT\_OUT {

CK\_OBJECT\_HANDLE hMacSecret;

CK\_OBJECT\_HANDLE hKey;

CK\_BYTE\_PTR pIV;

} CK\_WTLS\_KEY\_MAT\_OUT;

The fields of the structure have the following meanings:

hMacSecret Key handle for the resulting MAC secret key

hKey Key handle for the resulting secret key

pIV Pointer to a location which receives the initialization vector (IV) created (if any)

**CK\_WTLS\_KEY\_MAT\_OUT \_PTR** is a pointer to a **CK\_WTLS\_KEY\_MAT\_OUT**.

1. CK\_WTLS\_KEY\_MAT\_PARAMS; CK\_WTLS\_KEY\_MAT\_PARAMS\_PTR

**CK\_WTLS\_KEY\_MAT\_PARAMS** is a structure that provides the parameters to the **CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE** and the **CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE** mechanisms. It is defined as follows:

typedef struct CK\_WTLS\_KEY\_MAT\_PARAMS {

CK\_MECHANISM\_TYPE DigestMechanism;

CK\_ULONG ulMacSizeInBits;

CK\_ULONG ulKeySizeInBits;

CK\_ULONG ulIVSizeInBits;

CK\_ULONG ulSequenceNumber;

CK\_BBOOL bIsExport;

CK\_WTLS\_RANDOM\_DATA RandomInfo;

CK\_WTLS\_KEY\_MAT\_OUT\_PTR pReturnedKeyMaterial;

} CK\_WTLS\_KEY\_MAT\_PARAMS;

The fields of the structure have the following meanings:

Digest Mechanism the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])

ulMaxSizeInBits the length (in bits) of the MACing key agreed upon during the protocol handshake phase

ulKeySizeInBits the length (in bits) of the secret key agreed upon during the handshake phase

ulIVSizeInBits the length (in bits) of the IV agreed upon during the handshake phase. If no IV is required, the length should be set to 0.

ulSequenceNumber the current sequence number used for records sent by the client and server respectively

bIsExport a boolean value which indicates whether the keys have to be derives for an export version of the protocol. If this value is true (i.e., the keys are exportable) then ulKeySizeInBits is the length of the key in bits before expansion. The length of the key after expansion is determined by the information found in the template sent along with this mechanism during a C\_DeriveKey function call (either the **CKA\_KEY\_TYPE** or the **CKA\_VALUE\_LEN** attribute).

RandomInfo client’s and server’s random data information

pReturnedKeyMaterial points to a **CK\_WTLS\_KEY\_MAT\_OUT** structure which receives the handles for the keys generated and the IV

**CK\_WTLS\_KEY\_MAT\_PARAMS\_PTR** is a pointer to a **CK\_WTLS\_KEY\_MAT\_PARAMS**.

### Pre master secret key generation for RSA key exchange suite

Pre master secret key generation for the RSA key exchange suite in WTLS denoted **CKM\_WTLS\_PRE\_MASTER\_KEY\_GEN**, is a mechanism, which generates a variable length secret key. It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This mechanism returns a handle to the pre master secret key.

It has one parameter, a **CK\_BYTE**, which provides the client’s WTLS version.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE** and **CKA\_VALUE** attributes to the new key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C\_GenerateKey** call may indicate that the object class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN** attribute indicates the length of the pre master secret key.

For this mechanism, the ulMinKeySize field of the **CK\_MECHANISM\_INFO** structure shall indicate 20 bytes.

### Master secret key derivation

Master secret derivation in WTLS, denoted **CKM\_WTLS\_MASTER\_KEY\_DERIVE**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client version, which is built into the pre master secret key as well as a handle to the derived master secret key.

It has a parameter, a **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for passing the mechanism type of the digest mechanism to be used as well as the passing of random data to the token as well as the returning of the protocol version number which is part of the pre master secret key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN** attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.

If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.

Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure both indicate 20 bytes.

Note that the **CK\_BYTE** pointed to by the **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS** structure’s *pVersion* field will be modified by the **C\_DeriveKey** call. In particular, when the call returns, this byte will hold the WTLS version associated with the supplied pre master secret key.

Note that this mechanism is only useable for key exchange suites that use a 20-byte pre master secret key with an embedded version number. This includes the RSA key exchange suites, but excludes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites.

### Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography

Master secret derivation for Diffie-Hellman and Elliptic Curve Cryptography in WTLS, denoted **CKM\_WTLS\_MASTER\_KEY\_DERIVE\_DH\_ECC**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns a handle to the derived master secret key.

It has a parameter, a **CK\_WTLS\_MASTER\_KEY\_DERIVE\_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used as well as random data to the token. The *pVersion* field of the structure must be set to NULL\_PTR since the version number is not embedded in the pre master secret key as it is for RSA-like key exchange suites.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key (as well as the **CKA\_VALUE\_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object class is **CKO\_SECRET\_KEY**, the key type is **CKK\_GENERIC\_SECRET**, and the **CKA\_VALUE\_LEN** attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.

If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.

Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure both indicate 20 bytes.

Note that this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

### WTLS PRF (pseudorandom function)

PRF (pseudo random function) in WTLS, denoted **CKM\_WTLS\_PRF**, is a mechanism used to produce a securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a **CK\_WTLS\_PRF\_PARAMS** structure, which allows for passing the mechanism type of the digest mechanism to be used, the passing of the input seed and its length, the passing of an identifying label and its length and the passing of the length of the output to the token and for receiving the output.

This mechanism produces securely generated pseudo-random output of the length specified in the parameter.

This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template sent along with this mechanism during a **C\_DeriveKey** function call, which means the template shall be a NULL\_PTR. For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM\_WTLS\_PRF** mechanism returns the requested number of output bytes in the **CK\_WTLS\_PRF\_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C\_DeriveKey** is unnecessary, and should be a NULL\_PTR.

If a call to **C\_DeriveKey** with this mechanism fails, then no output will be generated.

### Server Key and MAC derivation

Server key, MAC and IV derivation in WTLS, denoted **CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from the master secret key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IV created.

It has a parameter, a **CK\_WTLS\_KEY\_MAT\_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.

This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of **CKO\_SECRET\_KEY**.

The MACing key (server write MAC secret) is always given a type of **CKK\_GENERIC\_SECRET**. It is flagged as valid for signing, verification and derivation operations.

The other key (server write key) is typed according to information found in the template sent along with this mechanism during a **C\_DeriveKey** function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (server write IV) will be generated and returned if the *ulIVSizeInBits* field of the **CK\_WTLS\_KEY\_MAT\_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree with the value in the *ulIVSizeInBits* field

Both keys inherit the values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**, **CKA\_EXTRACTABLE**, and **CKA\_NEVER\_EXTRACTABLE** attributes from the base key. The template provided to **C\_DeriveKey** may not specify values for any of these attributes that differ from those held by the base key.

Note that the **CK\_WTLS\_KEY\_MAT\_OUT** structure pointed to by the **CK\_WTLS\_KEY\_MAT\_PARAMS** structure’s *pReturnedKeyMaterial* field will be modified by the **C\_DeriveKey** call. In particular, the two key handle fields in the **CK\_WTLS\_KEY\_MAT\_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the **CK\_WTLS\_KEY\_MAT\_OUT** structure’s *pIV* field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE** mechanism returns all of its key handles in the **CK\_WTLS\_KEY\_MAT\_OUT** structure pointed to by the **CK\_WTLS\_KEY\_MAT\_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C\_DeriveKey** is unnecessary, and should be a NULL\_PTR.

If a call to **C\_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

### Client key and MAC derivation

Client key, MAC and IV derivation in WTLS, denoted **CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from the master secret key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IV created.

It has a parameter, a **CK\_WTLS\_KEY\_MAT\_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.

This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of **CKO\_SECRET\_KEY**.

The MACing key (client write MAC secret) is always given a type of **CKK\_GENERIC\_SECRET**. It is flagged as valid for signing, verification and derivation operations.

The other key (client write key) is typed according to information found in the template sent along with this mechanism during a **C\_DeriveKey** function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (client write IV) will be generated and returned if the *ulIVSizeInBits* field of the **CK\_WTLS\_KEY\_MAT\_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree with the value in the *ulIVSizeInBits* field

Both keys inherit the values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**, **CKA\_EXTRACTABLE**, and **CKA\_NEVER\_EXTRACTABLE** attributes from the base key. The template provided to **C\_DeriveKey** may not specify values for any of these attributes that differ from those held by the base key.

Note that the **CK\_WTLS\_KEY\_MAT\_OUT** structure pointed to by the **CK\_WTLS\_KEY\_MAT\_PARAMS** structure’s *pReturnedKeyMaterial* field will be modified by the **C\_DeriveKey** call. In particular, the two key handle fields in the **CK\_WTLS\_KEY\_MAT\_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the **CK\_WTLS\_KEY\_MAT\_OUT** structure’s *pIV* field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C\_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE** mechanism returns all of its key handles in the **CK\_WTLS\_KEY\_MAT\_OUT** structure pointed to by the **CK\_WTLS\_KEY\_MAT\_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C\_DeriveKey** is unnecessary, and should be a NULL\_PTR.

If a call to **C\_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

## SP 800-108 Key Derivation

NIST SP800-108 defines three types of key derivation functions (KDF); a Counter Mode KDF, a Feedback Mode KDF and a Double Pipeline Mode KDF.

This section defines a unique mechanism for each type of KDF. These mechanisms can be used to derive one or more symmetric keys from a single base symmetric key.

The KDFs defined in SP800-108 are all built upon pseudo random functions (PRF). In general terms, the PRFs accepts two pieces of input; a base key and some input data. The base key is taken from the *hBaseKey* parameter to **C\_Derive**. The input data is constructed from an iteration variable (internally defined by the KDF/PRF) and the data provided in the CK\_SP800\_108\_PRF\_DATA\_PARAM array that is part of the mechanism parameter.

*Table 161, SP800-108 Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR** | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SP800\_108\_COUNTER\_KDF |  |  |  |  |  |  | ✓ |
| CKM\_SP800\_108\_FEEDBACK\_KDF |  |  |  |  |  |  | ✓ |
| CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF |  |  |  |  |  |  | ✓ |

For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the minimum and maximum supported base key size in bits. Note, these mechanisms support multiple PRF types and key types; as such the values reported by ulMinKeySize and ulMaxKeySize specify the minimum and maximum supported base key size when all PRF and keys types are considered. For example, a Cryptoki implementation may support CKK\_GENERIC\_SECRET keys that can be as small as 8-bits in length and therefore ulMinKeySize could report 8-bits. However for an AES-CMAC PRF the base key must be of type CKK\_AES and must be either 16-bytes, 24-bytes or 32-bytes in lengths and therefore the value reported by ulMinKeySize could be misleading. Depending on the PRF type selected, additional key size restrictions may apply.

### Definitions

Mechanisms:

CKM\_SP800\_108\_COUNTER\_KDF

CKM\_SP800\_108\_FEEDBACK\_KDF

CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF

Data Field Types:

CK\_SP800\_108\_ITERATION\_VARIABLE

CK\_SP800\_108\_COUNTER

CK\_SP800\_108\_DKM\_LENGTH

CK\_SP800\_108\_BYTE\_ARRAY

DKM Length Methods:

CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_KEYS

CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_SEGMENTS

### Mechanism Parameters

1. CK\_SP800\_108\_PRF\_TYPE

The **CK\_SP800\_108\_PRF\_TYPE** field of the mechanism parameter is used to specify the type of PRF that is to be used. It is defined as follows:

typedef CK\_MECHANISM\_TYPE CK\_SP800\_108\_PRF\_TYPE;

The **CK\_SP800\_108\_PRF\_TYPE** field reuses the existing mechanisms definitions. The following table lists the supported PRF types:

Table 162, SP800-108 Pseudo Random Functions

|  |
| --- |
| **Pseudo Random Function Identifiers** |
| CKM\_SHA\_1\_HMAC |
| CKM\_SHA224\_HMAC |
| CKM\_SHA256\_HMAC |
| CKM\_SHA384\_HMAC |
| CKM\_SHA512\_HMAC |
| CKM\_SHA3\_224\_HMAC |
| CKM\_SHA3\_256\_HMAC |
| CKM\_SHA3\_384\_HMAC |
| CKM\_SHA3\_512\_HMAC |
| CKM\_3DES\_CMAC |
| CKM\_AES\_CMAC |

1. CK\_PRF\_DATA\_TYPE

Each mechanism parameter contains an array of **CK\_PRF\_DATA\_PARAM** structures. The **CK\_PRF\_DATA\_PARAM** structure contains **CK\_PRF\_DATA\_TYPE** field. The **CK\_PRF\_DATA\_TYPE** field is used to identify the type of data identified by each **CK\_PRF\_DATA\_PARAM** element in the array. Depending on the type of KDF used, some data field types are mandatory, some data field types are optional and some data field types are not allowed. These requirements are defined on a per-mechanism basis in the sections below. The **CK\_PRF\_DATA\_TYPE** is defined as follows:

typedef CK\_ULONG CK\_PRF\_DATA\_TYPE;

The following table lists all of the supported data field types:

Table 163, SP800-108 PRF Data Field Types

|  |  |
| --- | --- |
| **Data Field Identifier** | **Description** |
| CK\_SP800\_108\_ITERATION\_VARIABLE | Identifies the iteration variable defined internally by the KDF. |
| CK\_SP800\_108\_COUNTER | Identifies an optional counter value represented as a binary string. Exact formatting of the counter value is defined by the CK\_SP800\_108\_COUNTER\_FORMAT structure. The value of the counter is defined by the KDF’s internal loop counter. |
| CK\_SP800\_108\_DKM\_LENGTH | Identifies the length in bits of the derived keying material (DKM) represented as a binary string. Exact formatting of the length value is defined by the CK\_SP800\_108\_DKM\_FORMAT structure. |
| CK\_SP800\_108\_BYTE\_ARRAY | Identifies a generic byte array of data. This data type can be used to provide “context”, “label”, “separator bytes” as well as any other type of encoding information required by the higher level protocol. |

1. CK\_PRF\_DATA\_PARAM

**CK\_PRF\_DATA\_PARAM** is used to define a segment of input for the PRF. Each mechanism parameter supports an array of **CK\_PRF\_DATA\_PARAM** structures. The **CK\_PRF\_DATA\_PARAM** is defined as follows:

typedef struct CK\_PRF\_DATA\_PARAM

{

CK\_PRF\_DATA\_TYPE type;

CK\_VOID\_PTR pValue;

CK\_ULONG ulValueLen;

} CK\_PRF\_DATA\_PARAM;

typedef CK\_PRF\_DATA\_PARAM CK\_PTR CK\_PRF\_DATA\_PARAM\_PTR

The fields of the **CK\_PRF\_DATA\_PARAM** structure have the following meaning:

type defines the type of data pointed to by pValue

pValue pointer to the data defined by type

ulValueLen size of the data pointed to by pValue

If the *type* field of the **CK\_PRF\_DATA\_PARAM** structure is set to CK\_SP800\_108\_ITERATION\_VARIABLE, then *pValue* must be set the appropriate value for the KDF’s iteration variable type. For the Counter Mode KDF, *pValue* must be assigned a valid CK\_SP800\_108\_COUNTER\_FORMAT\_PTR and *ulValueLen* must be set to sizeof(CK\_SP800\_108\_COUNTER\_FORMAT). For all other KDF types, *pValue must be set* to NULL\_PTR and *ulValueLen* must be set to 0.

If the *type* field of the **CK\_PRF\_DATA\_PARAM** structure is set to CK\_SP800\_108\_COUNTER, then *pValue* must be assigned a valid CK\_SP800\_108\_COUNTER\_FORMAT\_PTR and *ulValueLen* must be set to sizeof(CK\_SP800\_108\_COUNTER\_FORMAT).

If the *type* field of the **CK\_PRF\_DATA\_PARAM** structure is set to CK\_SP800\_108\_DKM\_LENGTH then *pValue* must be assigned a valid CK\_SP800\_108\_DKM\_FORMAT\_PTR and *ulValueLen* must be set to sizeof(CK\_SP800\_108\_DKM\_FORMAT).

If the *type* field of the **CK\_PRF\_DATA\_PARAM** structure is set to CK\_SP800\_108\_BYTE\_ARRAY, then *pValue* must be assigned a valid CK\_BYTE\_PTR value and *ulValueLen* must be set to a non-zero length.

1. CK\_SP800\_108\_COUNTER\_FORMAT

**CK\_SP800\_108\_COUNTER\_FORMAT** is used to define the encoding format for a counter value. The **CK\_SP800\_108\_COUNTER\_FORMAT** is defined as follows:

typedef struct CK\_SP800\_108\_COUNTER\_FORMAT

{

CK\_BBOOL bLittleEndian;

CK\_ULONG ulWidthInBits;

} CK\_SP800\_108\_COUNTER\_FORMAT;

typedef CK\_SP800\_108\_COUNTER\_FORMAT CK\_PTR CK\_SP800\_108\_COUNTER\_FORMAT\_PTR

The fields of the CK\_SP800\_108\_COUNTER\_FORMAT structure have the following meaning:

bLittleEndian defines if the counter should be represented in Big Endian or Little Endian format

ulWidthInBits defines the number of bits used to represent the counter value

1. CK\_SP800\_108\_DKM\_LENGTH\_METHOD

**CK\_SP800\_108\_DKM\_LENGTH\_METHOD** is used to define how the DKM length value is calculated. The **CK\_SP800\_108\_DKM\_LENGTH\_METHOD** type is defined as follows:

typedef CK\_ULONG CK\_SP800\_108\_DKM\_LENGTH\_METHOD;

The following table lists all of the supported DKM Length Methods:

Table 164, SP800-108 DKM Length Methods

|  |  |
| --- | --- |
| **DKM Length Method Identifier** | **Description** |
| CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_KEYS | Specifies that the DKM length should be set to the sum of the length of all keys derived by this invocation of the KDF. |
| CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_SEGMENTS | Specifies that the DKM length should be set to the sum of the length of all segments of output produced by the PRF by this invocation of the KDF. |

1. CK\_SP800\_108\_DKM\_LENGTH\_FORMAT

**CK\_SP800\_108\_DKM\_LENGTH\_FORMAT** is used to define the encoding format for the DKM length value. The **CK\_SP800\_108\_DKM\_LENGTH\_FORMAT** is defined as follows:

typedef struct CK\_SP800\_108\_DKM\_LENGTH\_FORMAT

{

CK\_SP800\_108\_DKM\_LENGTH\_METHOD dkmLengthMethod;

CK\_BBOOL bLittleEndian;

CK\_ULONG ulWidthInBits;

} CK\_SP800\_108\_DKM\_LENGTH\_FORMAT;

typedef CK\_SP800\_108\_DKM\_LENGTH\_FORMAT CK\_PTR CK\_SP800\_108\_DKM\_LENGTH\_FORMAT\_PTR

The fields of the CK\_SP800\_108\_DKM\_LENGTH\_FORMAT structure have the following meaning:

dkmLengthMethod defines the method used to calculate the DKM length value

bLittleEndian defines if the DKM length value should be represented in Big Endian or Little Endian format

ulWidthInBits defines the number of bits used to represent the DKM length value

1. CK\_DERIVED\_KEY

**CK\_DERIVED\_KEY** is used to define an additional key to be derived as well as provide a CK\_OBJECT\_HANDLE\_PTR to receive the handle for the derived keys. The **CK\_DERIVED\_KEY** is defined as follows:

typedef struct CK\_DERIVED\_KEY

{

CK\_ATTRIBUTE\_PTR pTemplate;

CK\_ULONG ulAttributeCount;

CK\_OBJECT\_HANDLE\_PTR phKey;

} CK\_DERIVED\_KEY;

typedef CK\_DERIVED\_KEY CK\_PTR CK\_DERIVED\_KEY\_PTR

The fields of the CK\_DERIVED\_KEY structure have the following meaning:

pTemplate pointer to a template that defines a key to derive

ulAttributeCount number of attributes in the template pointed to by pTemplate

phKey pointer to receive the handle for a derived key

1. CK\_SP800\_108\_KDF\_PARAMS, CK\_SP800\_108\_KDF\_PARAMS\_PTR

**CK\_SP800\_108\_KDF\_PARAMS** is a structure that provides the parameters for the **CKM\_SP800\_108\_COUNTER\_KDF** and **CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF** mechanisms.

**typedef** **struct** CK\_SP800\_108\_KDF\_PARAMS

{

CK\_PRF\_TYPE prfType;

CK\_ULONG ulNumberOfDataParams;

CK\_PRF\_DATA\_PARAM\_PTR pDataParams;

CK\_ULONG ulAdditionalDerivedKeys;

CK\_DERIVED\_KEY pAdditionalDerivedKeys;  
} CK\_SP800\_108\_KDF\_PARAMS;

typedef CK\_SP800\_108\_KDF\_PARAMS CK\_PTR CK\_SP800\_108\_KDF\_PARAMS\_PTR;

The fields of the **CK\_SP800\_108\_KDF\_PARAMS** structure have the following meaning:

prfType type of PRF

ulNumberOfDataParams number of elements in the array pointed to by pDataParams

pDataParams an array of CK\_PRF\_DATA\_PARAM structures. The array defines input parameters that are used to construct the “data” input to the PRF.

ulAdditionalDerivedKeys number of additional keys that will be derived and the number of elements in the array pointed to by pAdditionalDerivedKeys. If pAdditionalDerivedKeys is set to NULL\_PTR, this parameter must be set to 0.

pAdditionalDerivedKeys an array of CK\_DERIVED\_KEY structures. If ulAdditionalDerivedKeys is set to 0, this parameter must be set to NULL\_PTR

1. CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS, CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS\_PTR

The **CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS** structure provides the parameters for the CKM\_SP800\_108\_FEEDBACK\_KDF mechanism. It is defined as follows:

typedef struct CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS

{

CK\_PRF\_TYPE prfType;

CK\_ULONG ulNumberOfDataParams;

CK\_PRF\_DATA\_PARAM\_PTR pDataParams;

CK\_ULONG ulIVLen;

CK\_BYTE\_PTR pIV;

CK\_ULONG ulAdditionalDerivedKeys;

CK\_DERIVED\_KEY pAdditionalDerivedKeys;  
} CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS;

typedef CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS CK\_PTR CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS\_PTR;

The fields of the **CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS** structure have the following meaning:

prfType type of PRF

ulNumberOfDataParams number of elements in the array pointed to by pDataParams

pDataParams an array of CK\_PRF\_DATA\_PARAM structures. The array defines input parameters that are used to construct the “data” input to the PRF.

ulIVLen the length in bytes of the IV. If pIV is set to NULL\_PTR, this parameter must be set to 0.

pIV an array of bytes to be used as the IV for the feedback mode KDF. This parameter is optional and can be set to NULL\_PTR. If ulIVLen is set to 0, this parameter must be set to NULL\_PTR.

ulAdditionalDerivedKeys number of additional keys that will be derived and the number of elements in the array pointed to by pAdditionalDerivedKeys. If pAdditionalDerivedKeys is set to NULL\_PTR, this parameter must be set to 0.

pAdditionalDerivedKeys an array of CK\_DERIVED\_KEYS structures. If ulAdditionalDerivedKeys is set to 0, this parameter must be set to NULL\_PTR.

### Counter Mode KDF

The SP800-108 Counter Mode KDF mechanism, denoted **CKM\_SP800\_108\_COUNTER\_KDF**, represents the KDF defined SP800-108 section 5.1. **CKM\_SP800\_108\_COUNTER\_KDF** is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a **CK\_SP800\_108\_KDF\_PARAMS** structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

Table 165, Counter Mode data field requirements

|  |  |
| --- | --- |
| **Data Field Identifier** | **Description** |
| CK\_SP800\_108\_ITERATION\_VARIABLE | This data field type is mandatory.  This data field type identifies the location of the iteration variable in the constructed PRF input data.  The iteration variable for this KDF type is a counter.  Exact formatting of the counter value is defined by the CK\_SP800\_108\_COUNTER\_FORMAT structure. |
| CK\_SP800\_108\_COUNTER | This data field type is invalid for this KDF type. |
| CK\_SP800\_108\_DKM\_LENGTH | This data field type is optional.  This data field type identifies the location of the DKM length in the constructed PRF input data.  Exact formatting of the DKM length is defined by the CK\_SP800\_108\_DKM\_LENGTH\_FORMAT structure.  If specified, only one instance of this type may be specified. |
| CK\_SP800\_108\_BYTE\_ARRAY | This data field type is optional.  This data field type identifies the location and value of a byte array of data in the constructed PRF input data.  This standard does not restrict the number of instances of this data type. |

SP800-108 limits the amount of derived keying material that can be produced by a Counter Mode KDF by limiting the internal loop counter to (2r−1), where “r” is the number of bits used to represent the counter. Therefore the maximum number of bits that can be produced is (2r−1)h, where “h” is the length in bits of the output of the selected PRF.

### Feedback Mode KDF

The SP800-108 Feedback Mode KDF mechanism, denoted **CKM\_SP800\_108\_FEEDBACK\_KDF**, represents the KDF defined SP800-108 section 5.2. **CKM\_SP800\_108\_FEEDBACK\_KDF** is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a **CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS** structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

Table 166, Feedback Mode data field requirements

|  |  |
| --- | --- |
| **Data Field Identifier** | **Description** |
| CK\_SP800\_108\_ITERATION\_VARIABLE | This data field type is mandatory.  This data field type identifies the location of the iteration variable in the constructed PRF input data.  The iteration variable is defined as K(i-1) in section 5.2 of SP800-108.  The size, format and value of this data input is defined by the internal KDF structure and PRF output.  Exact formatting of the counter value is defined by the CK\_SP800\_108\_COUNTER\_FORMAT structure. |
| CK\_ SP800\_108\_COUNTER | This data field type is optional.  This data field type identifies the location of the counter in the constructed PRF input data.  Exact formatting of the counter value is defined by the CK\_SP800\_108\_COUNTER\_FORMAT structure.  If specified, only one instance of this type may be specified. |
| CK\_ SP800\_108\_DKM\_LENGTH | This data field type is optional.  This data field type identifies the location of the DKM length in the constructed PRF input data.  Exact formatting of the DKM length is defined by the CK\_SP800\_108\_DKM\_LENGTH\_FORMAT structure.  If specified, only one instance of this type may be specified. |
| CK\_ SP800\_108\_BYTE\_ARRAY | This data field type is optional.  This data field type identifies the location and value of a byte array of data in the constructed PRF input data.  This standard does not restrict the number of instances of this data type. |

SP800-108 limits the amount of derived keying material that can be produced by a Feedback Mode KDF by limiting the internal loop counter to (232−1). Therefore the maximum number of bits that can be produced is (232−1)h, where “h” is the length in bits of the output of the selected PRF.

### Double Pipeline Mode KDF

The SP800-108 Double Pipeline Mode KDF mechanism, denoted **CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF**, represents the KDF defined SP800-108 section 5.3. **CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF** is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a CK\_SP800\_108\_KDF\_PARAMS structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

Table 167, Double Pipeline Mode data field requirements

|  |  |
| --- | --- |
| **Data Field Identifier** | **Description** |
| CK\_SP800\_108\_ITERATION\_VARIABLE | This data field type is mandatory.  This data field type identifies the location of the iteration variable in the constructed PRF input data.  The iteration variable is defined as A(i) in section 5.3 of SP800-108.  The size, format and value of this data input is defined by the internal KDF structure and PRF output.  Exact formatting of the counter value is defined by the CK\_SP800\_108\_COUNTER\_FORMAT structure. |
| CK\_ SP800\_108\_COUNTER | This data field type is optional.  This data field type identifies the location of the counter in the constructed PRF input data.  Exact formatting of the counter value is defined by the CK\_SP800\_108\_COUNTER\_FORMAT structure.  If specified, only one instance of this type may be specified. |
| CK\_ SP800\_108\_DKM\_LENGTH | This data field type is optional.  This data field type identifies the location of the DKM length in the constructed PRF input data.  Exact formatting of the DKM length is defined by the CK\_SP800\_108\_DKM\_LENGTH\_FORMAT structure.  If specified, only one instance of this type may be specified. |
| CK\_ SP800\_108\_BYTE\_ARRAY | This data field type is optional.  This data field type identifies the location and value of a byte array of data in the constructed PRF input data.  This standard does not restrict the number of instances of this data type. |

SP800-108 limits the amount of derived keying material that can be produced by a Double-Pipeline Mode KDF by limiting the internal loop counter to (232−1). Therefore the maximum number of bits that can be produced is (232−1)h, where “h” is the length in bits of the output of the selected PRF.

The Double Pipeline KDF requires an internal IV value. The IV is constructed using the same method used to construct the PRF input data; the data/values identified by the array of **CK\_PRF\_DATA\_PARAM** structures are concatenated in to a byte array that is used as the IV. As shown in SP800-108 section 5.3, the CK\_SP800\_108\_ITERATION\_VARIABLE and CK\_SP800\_108\_COUNTER data field types are not included in IV construction process. All other data field types are included in the construction process.

### Deriving Additional Keys

The KDFs defined in this section can be used to derive more than one symmetric key from the base key. The **C\_Derive** function accepts one CK\_ATTRIBUTE\_PTR to define a single derived key and one CK\_OBJECT\_HANDLE\_PTR to receive the handle for the derived key.

To derive additional keys, the mechanism parameter structure can be filled in with one or more CK\_DERIVED\_KEY structures. Each structure contains a CK\_ATTRIBUTE\_PTR to define a derived key and a CK\_OBJECT\_HANDLE\_PTR to receive the handle for the additional derived keys. The key defined by the **C\_Derive** function parameters is always derived before the keys defined by the CK\_DERIVED\_KEY array that is part of the mechanism parameter. The additional keys that are defined by the CK\_DERIVED\_KEY array are derived in the order they are defined in the array. That is to say that the derived keying material produced by the KDF is processed from left to right, and bytes are assigned first to the key defined by the **C\_Derive** function parameters, and then bytes are assigned to the keys that are defined by the CK\_DERIVED\_KEY array in the order they are defined in the array.

Each internal iteration of a KDF produces a unique segment of PRF output. Sometimes, a single iteration will produce enough keying material for the key being derived. Other times, additional internal iterations are performed to produce multiple segments which are concatenated together to produce enough keying material for the derived key(s).

When deriving multiple keys, no key can be created using part of a segment that was used for another key. All keys must be created from disjoint segments. For example, if the parameters are defined such that a 48-byte key (defined by the **C\_Derive** function parameters) and a 16-byte key (defined by the content of CK\_DERIVED\_KEY) are to be derived using **CKM\_SHA256\_HMAC** as a PRF, three internal iterations of the KDF will be performed and three segments of PRF output will be produced. The first segment and half of the second segment will be used to create the 48-byte key and the third segment will be used to create the 16-byte key.



In the above example, if the CK\_ SP800\_108\_DKM\_LENGTH data field type is specified with method CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_KEYS, then the DKM length value will be 512 bits. If the CK\_ SP800\_108\_DKM\_LENGTH data field type is specified with method CK\_SP800\_108\_DKM\_LENGTH\_SUM\_OF\_SEGMENTS, then the DKM length value will be 768 bits.

When deriving multiple keys, if any of the keys cannot be derived for any reason, none of the keys shall be derived. If the failure was caused by the content of a specific key’s template (ie the template defined by the content of *pTemplate*), the corresponding *phKey* value will be set to CK\_HANDLE\_INVALID to identify the offending template.

### Key Derivation Attribute Rules

The **CKM\_SP800\_108\_COUNTER\_KDF**, **CKM\_SP800\_108\_FEEDBACK\_KDF** and **CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF** mechanisms have the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key(s) can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

### Constructing PRF Input Data

SP800-108 defines the PRF input data for each KDF at a high level using terms like “label”, “context”, “separator”, “counter”…etc. The value, formatting and order of the input data is not strictly defined by SP800-108, instead it is described as being defined by the “encoding scheme”.

To support any encoding scheme, these mechanisms construct the PRF input data from from the array of CK\_PRF\_DATA\_PARAM structures in the mechanism parameter. All of the values defined by the CK\_PRF\_DATA\_PARAM array are concatenated in the order they are defined and passed in to the PRF as the data parameter.

#### Sample Counter Mode KDF

SP800-108 section 5.1 outlines a sample Counter Mode KDF which defines the following PRF input:

PRF (*KI,* [*i*]2 *|| Label || 0x00 || Context ||* [*L*]2)

Section 5.1 does not define the number of bits used to represent the counter (the “r” value) or the DKM length (the “L” value), so 16-bits is assumed for both cases. The following sample code shows how to define this PRF input data using an array of CK\_PRF\_DATA\_PARAM structures.

#define DIM(a) (sizeof((a))/sizeof((a)[0]))  
  
 CK\_OBJECT\_HANDLE hBaseKey;  
 CK\_OBJECT\_HANDLE hDerivedKey;  
 CK\_ATTRIBUTE derivedKeyTemplate = { … };  
  
 CK\_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};  
 CK\_ULONG ulLabelLen = sizeof(baLabel);  
 CK\_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef}};  
 CK\_ULONG ulContextLen = sizeof(baContext);  
  
 CK\_SP800\_108\_COUNTER\_FORMAT counterFormat = { 0, 16};

CK\_SP800\_108\_DKM\_FORMAT dkmFormat = {CK\_SP800\_108\_SUM\_OF\_KEYS, 0, 16};  
  
 CK\_PRF\_DATA\_PARAM dataParams[] =  
 {  
 { CK\_SP800\_108\_ITERATION\_VARIABLE,

&counterFormat, sizeof(counterFormat) },  
 { CK\_SP800\_108\_BYTE\_ARRAY, baLabel, ulLabelLen },  
 { CK\_SP800\_108\_BYTE\_ARRAY, {0x00}, 1 },   
 { CK\_SP800\_108\_BYTE\_ARRAY, baContext, ulContextLen },  
 { CK\_SP800\_108\_DKM\_LENGTH, dkmFormat, sizeof(dkmFormat) }  
 };  
  
 CK\_SP800\_108\_KDF\_PARAMS kdfParams **=**  
 {  
 CK\_PRF\_AES\_CMAC,  
 DIM(dataParams),  
 &dataParams,  
 0, */\* no addition derived keys \*/*  
 NULL */\* no addition derived keys \*/*  
 };  
  
 CK\_MECHANISM **=** mechanism  
 {  
 CKM\_FLEXIBLE\_KDF,  
 &kdfParams,  
 sizeof(kdfParams)  
 };  
  
 hBaseKey **=** GetBaseKeyHandle(**.....**);  
  
 rv = C**\_**DeriveKey(  
 hSession,  
 &mechanism,  
 hBaseKey,  
 &derivedKeyTemplate,  
 DIM(derivedKeyTemplate),  
 &hDerivedKey);

#### Sample SCP03 Counter Mode KDF

The SCP03 standard defines a variation of a counter mode KDF which defines the following PRF input:

PRF (*KI, Label || 0x00 ||* [*L*]2 *||* [*i*]2 *|| Context*)

SCP03 defines the number of bits used to represent the counter (the “r” value) and number of bits used to represent the DKM length (the “L” value) as 16-bits. The following sample code shows how to define this PRF input data using an array of CK\_PRF\_DATA\_PARAM structures.

#define DIM(a) (sizeof((a))/sizeof((a)[0]))  
  
 CK\_OBJECT\_HANDLE hBaseKey;  
 CK\_OBJECT\_HANDLE hDerivedKey;  
 CK\_ATTRIBUTE derivedKeyTemplate = { … };  
  
 CK\_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};  
 CK\_ULONG ulLabelLen = sizeof(baLabel);  
 CK\_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef}};  
 CK\_ULONG ulContextLen = sizeof(baContext);  
  
 CK\_SP800\_108\_COUNTER\_FORMAT counterFormat = { 0, 16};

CK\_SP800\_108\_DKM\_FORMAT dkmFormat = {CK\_SP800\_108\_SUM\_OF\_KEYS, 0, 16};  
  
 CK\_PRF\_DATA\_PARAM dataParams[] =  
 {  
 { CK\_SP800\_108\_BYTE\_ARRAY, baLabel, ulLabelLen },  
 { CK\_SP800\_108\_BYTE\_ARRAY, {0x00}, 1 },   
 { CK\_SP800\_108\_DKM\_LENGTH, dkmFormat, sizeof(dkmFormat) },

{ CK\_SP800\_108\_ITERATION\_VARIABLE,

&counterFormat, sizeof(counterFormat) },

{ CK\_SP800\_108\_BYTE\_ARRAY, baContext, ulContextLen }   
 };  
  
 CK\_SP800\_108\_KDF\_PARAMS kdfParams **=**  
 {  
 CK\_PRF\_AES\_CMAC,  
 DIM(dataParams),  
 &dataParams,  
 0, */\* no addition derived keys \*/*  
 NULL */\* no addition derived keys \*/*  
 };  
  
 CK\_MECHANISM **=** mechanism  
 {  
 CKM\_FLEXIBLE\_KDF,  
 &kdfParams,  
 sizeof(kdfParams)  
 };  
  
 hBaseKey **=** GetBaseKeyHandle(**.....**);  
  
 rv = C**\_**DeriveKey(  
 hSession,  
 &mechanism,  
 hBaseKey,  
 &derivedKeyTemplate,  
 DIM(derivedKeyTemplate),  
 &hDerivedKey);

#### Sample Feedback Mode KDF

SP800-108 section 5.2 outlines a sample Feedback Mode KDF which defines the following PRF input:

PRF (*KI, K*(*i-1*) {*||* [*i*]2 }*|| Label || 0x00 || Context ||* [*L*]2)

Section 5.2 does not define the number of bits used to represent the counter (the “r” value) or the DKM length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional and is included in this example. The following sample code shows how to define this PRF input data using an array of CK\_PRF\_DATA\_PARAM structures.

#define DIM(a) (sizeof((a))**/**sizeof((a)[0]))  
  
 CK\_OBJECT\_HANDLE hBaseKey;  
 CK\_OBJECT\_HANDLE hDerivedKey;  
 CK\_ATTRIBUTE derivedKeyTemplate **=** { … };  
  
 CK\_BYTE baFeedbackIV[] **=** {0x01, 0x02, 0x03 , 0x04};  
 CK\_ULONG ulFeedbackIVLen **=** sizeof(baFeedbackIV);  
 CK\_BYTE baLabel[] **=** {0xde, 0xad, 0xbe , 0xef};  
 CK\_ULONG ulLabelLen **=** sizeof(baLabel);  
 CK\_BYTE baContext[] **=** {0xfe, 0xed, 0xbe , 0xef}};  
 CK\_ULONG ulContextLen **=** sizeof(baContext);  
  
 CK\_SP800\_108\_COUNTER\_FORMAT counterFormat **=** { 0, 16};

CK\_SP800\_108\_DKM\_FORMAT dkmFormat = {CK\_SP800\_108\_SUM\_OF\_KEYS, 0, 16};  
  
 CK\_PRF\_DATA\_PARAM dataParams[] **=**  
 {  
 { CK\_SP800\_108\_ITERATION\_VARIABLE,

&counterFormat, sizeof(counterFormat) },  
 { CK\_SP800\_108\_BYTE\_ARRAY, baLabel, ulLabelLen },  
 { CK\_SP800\_108\_BYTE\_ARRAY, {0x00}, 1 },   
 { CK\_SP800\_108\_BYTE\_ARRAY, baContext, ulContextLen },  
 { CK\_SP800\_108\_DKM\_LENGTH, dkmFormat, sizeof(dkmFormat) }  
 };  
  
 CK\_SP800\_108\_FEEDBACK\_KDF\_PARAMS kdfParams **=**  
 {  
 CK\_PRF\_AES\_CMAC,  
 DIM(dataParams),  
 &dataParams,

ulFeedbackIVLen,

pFeedbackIV,  
 0, */\* no addition derived keys \*/*  
 NULL */\* no addition derived keys \*/*  
 };  
  
 CK\_MECHANISM **=** mechanism  
 {  
 CKM\_FLEXIBLE\_KDF,  
 &kdfParams,  
 sizeof(kdfParams)  
 };  
  
 hBaseKey **=** GetBaseKeyHandle(**.....**);  
  
 rv = C\_DeriveKey(  
 hSession,  
 &mechanism,  
 hBaseKey,  
 &derivedKeyTemplate,  
 DIM(derivedKeyTemplate),  
 &hDerivedKey);

#### Sample Double-Pipeline Mode KDF

SP800-108 section 5.3 outlines a sample Double-Pipeline Mode KDF which defines the two following PRF inputs:

PRF (*KI, A*(*i-*1))

PRF (*KI, K*(*i-1*) {*||* [*i*]2 }*|| Label || 0x00 || Context ||* [*L*]2)

Section 5.3 does not define the number of bits used to represent the counter (the “r” value) or the DKM length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional so it is left out in this example. The following sample code shows how to define this PRF input data using an array of CK\_PRF\_DATA\_PARAM structures.

#define DIM(a) (sizeof((a))**/**sizeof((a)[0]))  
  
 CK\_OBJECT\_HANDLE hBaseKey;  
 CK\_OBJECT\_HANDLE hDerivedKey;  
 CK\_ATTRIBUTE derivedKeyTemplate **=** { … };  
  
 CK\_BYTE baLabel[] **=** {0xde, 0xad, 0xbe , 0xef};  
 CK\_ULONG ulLabelLen **=** sizeof(baLabel);  
 CK\_BYTE baContext[] **=** {0xfe, 0xed, 0xbe , 0xef}};  
 CK\_ULONG ulContextLen **=** sizeof(baContext);

CK\_SP800\_108\_DKM\_FORMAT dkmFormat = {CK\_SP800\_108\_SUM\_OF\_KEYS, 0, 16};  
  
 CK\_PRF\_DATA\_PARAM dataParams[] **=**  
 {

{ CK\_SP800\_108\_BYTE\_ARRAY, baLabel, ulLabelLen },  
 { CK\_SP800\_108\_BYTE\_ARRAY, {0x00}, 1 },   
 { CK\_SP800\_108\_BYTE\_ARRAY, baContext, ulContextLen },  
 { CK\_SP800\_108\_DKM\_LENGTH, dkmFormat, sizeof(dkmFormat) }  
 };  
  
 CK\_SP800\_108\_KDF\_PARAMS kdfParams **=**  
 {  
 CK\_PRF\_AES\_CMAC,  
 DIM(dataParams),  
 &dataParams,  
 0, */\* no addition derived keys \*/*  
 NULL */\* no addition derived keys \*/*  
 };  
  
 CK\_MECHANISM **=** mechanism  
 {  
 CKM\_FLEXIBLE\_KDF,  
 &kdfParams,  
 sizeof(kdfParams)  
 };  
  
 hBaseKey **=** GetBaseKeyHandle(**.....**);  
  
 rv = C\_DeriveKey(  
 hSession,  
 &mechanism,  
 hBaseKey,  
 &derivedKeyTemplate,  
 DIM(derivedKeyTemplate),  
 &hDerivedKey);

## Miscellaneous simple key derivation mechanisms

*Table 168, Miscellaneous simple key derivation Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_CONCATENATE\_BASE\_AND\_KEY |  |  |  |  |  |  | ✓ |
| CKM\_CONCATENATE\_BASE\_AND\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_CONCATENATE\_DATA\_AND\_BASE |  |  |  |  |  |  | ✓ |
| CKM\_XOR\_BASE\_AND\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_EXTRACT\_KEY\_FROM\_KEY |  |  |  |  |  |  | ✓ |

### Definitions

Mechanisms:

CKM\_CONCATENATE\_BASE\_AND\_DATA

CKM\_CONCATENATE\_DATA\_AND\_BASE

CKM\_XOR\_BASE\_AND\_DATA

CKM\_EXTRACT\_KEY\_FROM\_KEY

CKM\_CONCATENATE\_BASE\_AND\_KEY

### Parameters for miscellaneous simple key derivation mechanisms

1. CK\_KEY\_DERIVATION\_STRING\_DATA; CK\_KEY\_DERIVATION\_STRING\_DATA\_PTR

CK\_KEY\_DERIVATION\_STRING\_DATA provides the parameters for the CKM\_CONCATENATE\_BASE\_AND\_DATA, CKM\_CONCATENATE\_DATA\_AND\_BASE, and CKM\_XOR\_BASE\_AND\_DATA mechanisms. It is defined as follows:

typedef struct CK\_KEY\_DERIVATION\_STRING\_DATA {

CK\_BYTE\_PTR pData;

CK\_ULONG ulLen;

} CK\_KEY\_DERIVATION\_STRING\_DATA;

The fields of the structure have the following meanings:

pData pointer to the byte string

ulLen length of the byte string

**CK\_KEY\_DERIVATION\_STRING\_DATA\_PTR** is a pointer to a **CK\_KEY\_DERIVATION\_STRING\_DATA**.

1. CK\_EXTRACT\_PARAMS; CK\_EXTRACT\_PARAMS\_PTR

**CK\_EXTRACT\_PARAMS** provides the parameter to the **CKM\_EXTRACT\_KEY\_FROM\_KEY** mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is defined as follows:

typedef CK\_ULONG CK\_EXTRACT\_PARAMS;

**CK\_EXTRACT\_PARAMS\_PTR** is a pointer to a **CK\_EXTRACT\_PARAMS**.

### Concatenation of a base key and another key

This mechanism, denoted **CKM\_CONCATENATE\_BASE\_AND\_KEY**, derives a secret key from the concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys specified are concatenated together in a buffer.

This mechanism takes a parameter, a **CK\_OBJECT\_HANDLE**. This handle produces the key value information which is appended to the end of the base key’s value information (the base key is the key whose handle is supplied as an argument to **C\_DeriveKey**).

For example, if the value of the base key is 0x01234567, and the value of the other key is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

* If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the values of the two original keys.
* If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
* If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn’t, an error will be returned.
* If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the two original keys’ values, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

* If either of the two original keys has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key’s **CKA\_SENSITIVE** attribute is set either from the supplied template or from a default value.
* Similarly, if either of the two original keys has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key’s **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
* The derived key’s **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if both of the original keys have their **CKA\_ALWAYS\_SENSITIVE** attributes set to CK\_TRUE.
* Similarly, the derived key’s **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if both of the original keys have their **CKA\_NEVER\_EXTRACTABLE** attributes set to CK\_TRUE.

### Concatenation of a base key and data

This mechanism, denoted **CKM\_CONCATENATE\_BASE\_AND\_DATA**, derives a secret key by concatenating data onto the end of a specified secret key.

This mechanism takes a parameter, a **CK\_KEY\_DERIVATION\_STRING\_DATA** structure, which specifies the length and value of the data which will be appended to the base key to derive another key.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

* If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the value of the original key and the data.
* If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
* If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn’t, an error will be returned.
* If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the original key’s value and the data, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

* If the base key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key’s **CKA\_SENSITIVE** attribute is set either from the supplied template or from a default value.
* Similarly, if the base key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key’s **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
* The derived key’s **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
* Similarly, the derived key’s **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

### Concatenation of data and a base key

This mechanism, denoted **CKM\_CONCATENATE\_DATA\_AND\_BASE**, derives a secret key by prepending data to the start of a specified secret key.

This mechanism takes a parameter, a **CK\_KEY\_DERIVATION\_STRING\_DATA** structure, which specifies the length and value of the data which will be prepended to the base key to derive another key.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x89ABCDEF01234567.

* If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the sum of the lengths of the data and the value of the original key.
* If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
* If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn’t, an error will be returned.
* If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by concatenating the data and the original key’s value, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

* If the base key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key’s **CKA\_SENSITIVE** attribute is set either from the supplied template or from a default value.
* Similarly, if the base key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key’s **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
* The derived key’s **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
* Similarly, the derived key’s **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

### XORing of a key and data

XORing key derivation, denoted **CKM\_XOR\_BASE\_AND\_DATA**, is a mechanism which provides the capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle and some data.

This mechanism takes a parameter, a **CK\_KEY\_DERIVATION\_STRING\_DATA** structure, which specifies the data with which to XOR the original key’s value.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x88888888.

* If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the minimum of the lengths of the data and the value of the original key.
* If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
* If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn’t, an error will be returned.
* If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than are available by taking the shorter of the data and the original key’s value, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

* If the base key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key’s **CKA\_SENSITIVE** attribute is set either from the supplied template or from a default value.
* Similarly, if the base key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key’s **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
* The derived key’s **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
* Similarly, the derived key’s **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

### Extraction of one key from another key

Extraction of one key from another key, denoted **CKM\_EXTRACT\_KEY\_FROM\_KEY**, is a mechanism which provides the capability of creating one secret key from the bits of another secret key.

This mechanism has a parameter, a CK\_EXTRACT\_PARAMS, which specifies which bit of the original key should be used as the first bit of the newly-derived key.

We give an example of how this mechanism works. Suppose a token has a secret key with the 4-byte value 0x329F84A9. We will derive a 2-byte secret key from this key, starting at bit position 21 (i.e., the value of the parameter to the CKM\_EXTRACT\_KEY\_FROM\_KEY mechanism is 21).

1. We write the key’s value in binary: 0011 0010 1001 1111 1000 0100 1010 1001. We regard this binary string as holding the 32 bits of the key, labeled as b0, b1, …, b31.
2. We then extract 16 consecutive bits (i.e., 2 bytes) from this binary string, starting at bit b21. We obtain the binary string 1001 0101 0010 0110.
3. The value of the new key is thus 0x9526.

Note that when constructing the value of the derived key, it is permissible to wrap around the end of the binary string representing the original key’s value.

If the original key used in this process is sensitive, then the derived key must also be sensitive for the derivation to succeed.

* If no length or key type is provided in the template, then an error will be returned.
* If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
* If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn’t, an error will be returned.
* If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than the original key has, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

* If the base key has its **CKA\_SENSITIVE** attribute set to CK\_TRUE, so does the derived key. If not, then the derived key’s **CKA\_SENSITIVE** attribute is set either from the supplied template or from a default value.
* Similarly, if the base key has its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE, so does the derived key. If not, then the derived key’s **CKA\_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
* The derived key’s **CKA\_ALWAYS\_SENSITIVE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE.
* Similarly, the derived key’s **CKA\_NEVER\_EXTRACTABLE** attribute is set to CK\_TRUE if and only if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE.

## CMS

*Table 169, CMS Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_CMS\_SIG |  | ✓ | ✓ |  |  |  |  |

### Definitions

Mechanisms:

CKM\_CMS\_SIG

### CMS Signature Mechanism Objects

These objects provide information relating to the CKM\_CMS\_SIG mechanism. CKM\_CMS\_SIG mechanism object attributes represent information about supported CMS signature attributes in the token. They are only present on tokens supporting the **CKM\_CMS\_SIG** mechanism, but must be present on those tokens.

Table 170, CMS Signature Mechanism Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_REQUIRED\_CMS\_ATTRIBUTES | Byte array | Attributes the token always will include in the set of CMS signed attributes |
| CKA\_DEFAULT\_CMS\_ATTRIBUTES | Byte array | Attributes the token will include in the set of CMS signed attributes in the absence of any attributes specified by the application |
| CKA\_SUPPORTED\_CMS\_ATTRIBUTES | Byte array | Attributes the token may include in the set of CMS signed attributes upon request by the application |

The contents of each byte array will be a DER-encoded list of CMS **Attributes** with optional accompanying values. Any attributes in the list shall be identified with its object identifier, and any values shall be DER-encoded. The list of attributes is defined in ASN.1 as:

Attributes ::= SET SIZE (1..MAX) OF Attribute

Attribute ::= SEQUENCE {

attrType OBJECT IDENTIFIER,

attrValues SET OF ANY DEFINED BY OBJECT IDENTIFIER OPTIONAL

}

The client may not set any of the attributes.

### CMS mechanism parameters

* CK\_CMS\_SIG\_PARAMS, CK\_CMS\_SIG\_PARAMS\_PTR

**CK\_CMS\_SIG\_PARAMS** is a structure that provides the parameters to the **CKM\_CMS\_SIG** mechanism. It is defined as follows:

typedef struct CK\_CMS\_SIG\_PARAMS {

CK\_OBJECT\_HANDLE certificateHandle;

CK\_MECHANISM\_PTR pSigningMechanism;

CK\_MECHANISM\_PTR pDigestMechanism;

CK\_UTF8CHAR\_PTR pContentType;

CK\_BYTE\_PTR pRequestedAttributes;

CK\_ULONG ulRequestedAttributesLen;

CK\_BYTE\_PTR pRequiredAttributes;

CK\_ULONG ulRequiredAttributesLen;

} CK\_CMS\_SIG\_PARAMS;

The fields of the structure have the following meanings:

certificateHandle Object handle for a certificate associated with the signing key. The token may use information from this certificate to identify the signer in the **SignerInfo** result value. CertificateHandle may be NULL\_PTR if the certificate is not available as a PKCS #11 object or if the calling application leaves the choice of certificate completely to the token.

pSigningMechanism Mechanism to use when signing a constructed CMS **SignedAttributes** value. E.g. **CKM\_SHA1\_RSA\_PKCS**.

pDigestMechanism Mechanism to use when digesting the data. Value shall be NULL\_PTR when the digest mechanism to use follows from the pSigningMechanism parameter.

pContentType NULL-terminated string indicating complete MIME Content-type of message to be signed; or the value NULL\_PTR if the message is a MIME object (which the token can parse to determine its MIME Content-type if required). Use the value “application/octet-stream“ if the MIME type for the message is unknown or undefined. Note that the pContentType string shall conform to the syntax specified in RFC 2045, i.e. any parameters needed for correct presentation of the content by the token (such as, for example, a non-default “charset”) must be present. The token must follow rules and procedures defined in RFC 2045 when presenting the content.

pRequestedAttributes Pointer to DER-encoded list of CMS **Attributes** the caller requests to be included in the signed attributes. Token may freely ignore this list or modify any supplied values.

ulRequestedAttributesLen Length in bytes of the value pointed to by pRequestedAttributes

pRequiredAttributes Pointer to DER-encoded list of CMS **Attributes** (with accompanying values) required to be included in the resulting signed attributes. Token must not modify any supplied values. If the token does not support one or more of the attributes, or does not accept provided values, the signature operation will fail. The token will use its own default attributes when signing if both the pRequestedAttributes and pRequiredAttributes field are set to NULL\_PTR.

ulRequiredAttributesLen Length in bytes, of the value pointed to by pRequiredAttributes.

### CMS signatures

The CMS mechanism, denoted **CKM\_CMS\_SIG**, is a multi-purpose mechanism based on the structures defined in PKCS #7 and RFC 2630. It supports single- or multiple-part signatures with and without message recovery. The mechanism is intended for use with, e.g., PTDs (see MeT-PTD) or other capable tokens. The token will construct a CMS **SignedAttributes** value and compute a signature on this value. The content of the **SignedAttributes** value is decided by the token, however the caller can suggest some attributes in the parameter *pRequestedAttributes*. The caller can also require some attributes to be present through the parameters *pRequiredAttributes*. The signature is computed in accordance with the parameter *pSigningMechanism*.

When this mechanism is used in successful calls to **C\_Sign** or **C\_SignFinal**, the *pSignature* return value will point to a DER-encoded value of type **SignerInfo**. **SignerInfo** is defined in ASN.1 as follows (for a complete definition of all fields and types, see RFC 2630):

SignerInfo ::= SEQUENCE {

version CMSVersion,

sid SignerIdentifier,

digestAlgorithm DigestAlgorithmIdentifier,

signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,

signatureAlgorithm SignatureAlgorithmIdentifier,

signature SignatureValue,

unsignedAttrs [1] IMPLICIT UnsignedAttributes OPTIONAL }

The *certificateHandle* parameter, when set, helps the token populate the **sid** field of the **SignerInfo** value. If *certificateHandle* is NULL\_PTR the choice of a suitable certificate reference in the **SignerInfo** result value is left to the token (the token could, e.g., interact with the user).

This mechanism shall not be used in calls to **C\_Verify** or **C\_VerifyFinal** (use the *pSigningMechanism* mechanism instead).

For the *pRequiredAttributes* field, the token may have to interact with the user to find out whether to accept a proposed value or not. The token should never accept any proposed attribute values without some kind of confirmation from its owner (but this could be through, e.g., configuration or policy settings and not direct interaction). If a user rejects proposed values, or the signature request as such, the value CKR\_FUNCTION\_REJECTED shall be returned.

When possible, applications should use the **CKM\_CMS\_SIG** mechanism when generating CMS-compatible signatures rather than lower-level mechanisms such as **CKM\_SHA1\_RSA\_PKCS**. This is especially true when the signatures are to be made on content that the token is able to present to a user. Exceptions may include those cases where the token does not support a particular signing attribute. Note however that the token may refuse usage of a particular signature key unless the content to be signed is known (i.e. the **CKM\_CMS\_SIG** mechanism is used).

When a token does not have presentation capabilities, the PKCS #11-aware application may avoid sending the whole message to the token by electing to use a suitable signature mechanism (e.g. **CKM\_RSA\_PKCS**) as the *pSigningMechanism* value in the **CK\_CMS\_SIG\_PARAMS** structure, and digesting the message itself before passing it to the token.

PKCS #11-aware applications making use of tokens with presentation capabilities, should attempt to provide messages to be signed by the token in a format possible for the token to present to the user. Tokens that receive multipart MIME-messages for which only certain parts are possible to present may fail the signature operation with a return value of **CKR\_DATA\_INVALID**, but may also choose to add a signing attribute indicating which parts of the message were possible to present.

## Blowfish

Blowfish, a secret-key block cipher. It is a Feistel network, iterating a simple encryption function 16 times. The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex initialization phase required before any encryption can take place, the actual encryption of data is very efficient on large microprocessors.

*Table 171, Blowfish Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_BLOWFISH\_CBC | ✓ |  |  |  |  | ✓ |  |
| CKM\_BLOWFISH\_CBC\_PAD | ✓ |  |  |  |  | ✓ |  |

### Definitions

This section defines the key type “CKK\_BLOWFISH” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_BLOWFISH\_KEY\_GEN

CKM\_BLOWFISH\_CBC

CKM\_BLOWFISH\_CBC\_PAD

### BLOWFISH secret key objects

Blowfish secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_BLOWFISH) hold Blowfish keys. The following table defines the Blowfish secret key object attributes, in addition to the common attributes defined for this object class:

Table 172, BLOWFISH Secret Key Object

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value the key can be any length up to 448 bits. Bit length restricted to a byte array. |
| CKA\_VALUE\_LEN2,3 | CK\_ULONG | Length in bytes of key value |

- Refer to [PKCS11-Base] table 11 for footnotes

The following is a sample template for creating an Blowfish secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_BLOWFISH;

CK\_UTF8CHAR label[] = “A blowfish secret key object”;

CK\_BYTE value[16] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

### Blowfish key generation

The Blowfish key generation mechanism, denoted **CKM\_BLOWFISH\_KEY\_GEN**, is a key generation mechanism Blowfish.

It does not have a parameter.

The mechanism generates Blowfish keys with a particular length, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of key sizes in bytes.

### Blowfish-CBC

Blowfish-CBC, denoted **CKM\_BLOWFISH\_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping.

It has a parameter, a 8-byte initialization vector.

This mechanism can wrap and unwrap any secret key. For wrapping, the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

*Table 173, BLOWFISH-CBC: Key and Data Length*

| **Function** | **Key type** | **Input Length** | **Output Length** |
| --- | --- | --- | --- |
| C\_Encrypt | BLOWFISH | Multiple of block size | Same as input length |
| C\_Decrypt | BLOWFISH | Multiple of block size | Same as input length |
| C\_WrapKey | BLOWFISH | Any | Input length rounded up to multiple of the block size |
| C\_UnwrapKey | BLOWFISH | Multiple of block size | Determined by type of key being unwrapped or CKA\_VALUE\_LEN |

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of BLOWFISH key sizes, in bytes.

### Blowfish-CBC with PKCS padding

Blowfish-CBC-PAD, denoted CKM\_BLOWFISH\_CBC\_PAD, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 8-byte initialization vector.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA\_VALUE\_LEN** attribute.

The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

Constraints on key types and the length of data are summarized in the following table:

*Table 174, BLOWFISH-CBC with PKCS Padding: Key and Data Length*

| **Function** | **Key type** | **Input Length** | **Output Length** |
| --- | --- | --- | --- |
| C\_Encrypt | BLOWFISH | Any | Input length rounded up to multiple of the block size |
| C\_Decrypt | BLOWFISH | Multiple of block size | Between 1 and block length block size bytes shorter than input length |
| C\_WrapKey | BLOWFISH | Any | Input length rounded up to multiple of the block size |
| C\_UnwrapKey | BLOWFISH | Multiple of block size | Between 1 and block length block size bytes shorter than input length |

## Twofish

Ref[. https://www.schneier.com/twofish.html](file:///D:\blp\data\.%20http:\www.counterpane.com\twofish-brief.html)

### Definitions

This section defines the key type “CKK\_TWOFISH” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_TWOFISH\_KEY\_GEN

CKM\_TWOFISH\_CBC

CKM\_TWOFISH\_CBC\_PAD

### Twofish secret key objects

Twofish secret key objects (object class **CKO\_SECRET\_KEY,** key type **CKK\_TWOFISH**) hold Twofish keys. The following table defines the Twofish secret key object attributes, in addition to the common attributes defined for this object class:

Table 175, Twofish Secret Key Object

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value 128-, 192-, or 256-bit key |
| CKA\_VALUE\_LEN2,3 | CK\_ULONG | Length in bytes of key value |

- Refer to [PKCS11-Base] table 11 for footnotes

The following is a sample template for creating an TWOFISH secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_TWOFISH;

CK\_UTF8CHAR label[] = “A twofish secret key object”;

CK\_BYTE value[16] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

### Twofish key generation

The Twofish key generation mechanism, denoted **CKM\_TWOFISH\_KEY\_GEN**, is a key generation mechanism Twofish.

It does not have a parameter.

The mechanism generates Blowfish keys with a particular length, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of key sizes, in bytes.

### Twofish -CBC

Twofish-CBC, denoted **CKM\_TWOFISH\_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping.

It has a parameter, a 16-byte initialization vector.

### Twofish-CBC with PKCS padding

Twofish-CBC-PAD, denoted CKM\_TWOFISH\_CBC\_PAD, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 16-byte initialization vector.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA\_VALUE\_LEN** attribute.

## CAMELLIA

Camellia is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES. Camellia is described e.g. in IETF RFC 3713.

*Table 176, Camellia Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_CAMELLIA\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_CAMELLIA\_ECB | ✓ |  |  |  |  | ✓ |  |
| CKM\_CAMELLIA\_CBC | ✓ |  |  |  |  | ✓ |  |
| CKM\_CAMELLIA\_CBC\_PAD | ✓ |  |  |  |  | ✓ |  |
| CKM\_CAMELLIA\_MAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_CAMELLIA\_MAC |  | ✓ |  |  |  |  |  |
| CKM\_CAMELLIA\_ECB\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_CAMELLIA\_CBC\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |

### Definitions

This section defines the key type “CKK\_CAMELLIA” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_CAMELLIA\_KEY\_GEN

CKM\_CAMELLIA\_ECB

CKM\_CAMELLIA\_CBC

CKM\_CAMELLIA\_MAC

CKM\_CAMELLIA\_MAC\_GENERAL

CKM\_CAMELLIA\_CBC\_PAD

### Camellia secret key objects

Camellia secret key objects (object class **CKO\_SECRET\_KEY,** key type **CKK\_CAMELLIA**) hold Camellia keys. The following table defines the Camellia secret key object attributes, in addition to the common attributes defined for this object class:

Table 177, Camellia Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (16, 24, or 32 bytes) |
| CKA\_VALUE\_LEN2,3,6 | CK\_ULONG | Length in bytes of key value |

- Refer to [PKCS11-Base] table 11 for footnotes.

The following is a sample template for creating a Camellia secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_CAMELLIA;

CK\_UTF8CHAR label[] = “A Camellia secret key object”;

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

### Camellia key generation

The Camellia key generation mechanism, denoted CKM\_CAMELLIA\_KEY\_GEN, is a key generation mechanism for Camellia.

It does not have a parameter.

The mechanism generates Camellia keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the Camellia key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Camellia key sizes, in bytes.

### Camellia-ECB

Camellia-ECB, denoted **CKM\_CAMELLIA\_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia and electronic codebook mode.

It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 178, Camellia-ECB: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | CKK\_CAMELLIA | multiple of block size | same as input length | no final part |
| C\_Decrypt | CKK\_CAMELLIA | multiple of block size | same as input length | no final part |
| C\_WrapKey | CKK\_CAMELLIA | any | input length rounded up to multiple of block size |  |
| C\_UnwrapKey | CKK\_CAMELLIA | multiple of block size | determined by type of key being unwrapped or CKA\_VALUE\_LEN |  |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Camellia key sizes, in bytes.

### Camellia-CBC

Camellia-CBC, denoted **CKM\_CAMELLIA\_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia and cipher-block chaining mode.

It has a parameter, a 16-byte initialization vector.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 179, Camellia-CBC: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | CKK\_CAMELLIA | multiple of block size | same as input length | no final part |
| C\_Decrypt | CKK\_CAMELLIA | multiple of block size | same as input length | no final part |
| C\_WrapKey | CKK\_CAMELLIA | any | input length rounded up to multiple of the block size |  |
| C\_UnwrapKey | CKK\_CAMELLIA | multiple of block size | determined by type of key being unwrapped or CKA\_VALUE\_LEN |  |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Camellia key sizes, in bytes.

### Camellia-CBC with PKCS padding

Camellia-CBC with PKCS padding, denoted **CKM\_CAMELLIA\_CBC\_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 16-byte initialization vector.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA\_VALUE\_LEN** attribute.

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section TBA for details). The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

Constraints on key types and the length of data are summarized in the following table:

Table 180, Camellia-CBC with PKCS Padding: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Encrypt | CKK\_CAMELLIA | any | input length rounded up to multiple of the block size |
| C\_Decrypt | CKK\_CAMELLIA | multiple of block size | between 1 and block size bytes shorter than input length |
| C\_WrapKey | CKK\_CAMELLIA | any | input length rounded up to multiple of the block size |
| C\_UnwrapKey | CKK\_CAMELLIA | multiple of block size | between 1 and block length bytes shorter than input length |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Camellia key sizes, in bytes.

### CAMELLIA with Counter mechanism parameters

1. CK\_CAMELLIA\_CTR\_PARAMS; CK\_CAMELLIA\_CTR\_PARAMS\_PTR

**CK\_CAMELLIA\_CTR\_PARAMS** is a structure that provides the parameters to the **CKM\_CAMELLIA\_CTR** mechanism. It is defined as follows:

typedef struct CK\_CAMELLIA\_CTR\_PARAMS {

CK\_ULONG ulCounterBits;

CK\_BYTE cb[16];

} CK\_CAMELLIA\_CTR\_PARAMS;

ulCounterBits specifies the number of bits in the counter block (cb) that shall be incremented. This number shall be such that 0 < *ulCounterBits* <= 128. For any values outside this range the mechanism shall return **CKR\_MECHANISM\_PARAM\_INVALID**.

It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter bits are the least significant bits of the counter block (cb). They are a big-endian value usually starting with 1. The rest of ‘cb’ is for the nonce, and maybe an optional IV.

E.g. as defined in [RFC 3686]:

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Nonce |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Initialization Vector (IV) |

| |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Block Counter |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

This construction permits each packet to consist of up to 232-1 blocks = 4,294,967,295 blocks = 68,719,476,720 octets.

**CK\_CAMELLIA\_CTR\_PARAMS\_PTR** is a pointer to a **CK\_CAMELLIA\_CTR\_PARAMS**.

### General-length Camellia-MAC

General-length Camellia -MAC, denoted CKM\_CAMELLIA\_MAC\_GENERAL, is a mechanism for single- and multiple-part signatures and verification, based on Camellia and data authentication as defined in.[CAMELLIA]

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final Camellia cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

Table 181, General-length Camellia-MAC: Key and Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_CAMELLIA | any | 1-block size, as specified in parameters |
| C\_Verify | CKK\_CAMELLIA | any | 1-block size, as specified in parameters |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Camellia key sizes, in bytes.

### Camellia-MAC

Camellia-MAC, denoted by **CKM\_CAMELLIA\_MAC**, is a special case of the general-length Camellia-MAC mechanism. Camellia-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 182, Camellia-MAC: Key and Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_CAMELLIA | any | ½ block size (8 bytes) |
| C\_Verify | CKK\_CAMELLIA | any | ½ block size (8 bytes) |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of Camellia key sizes, in bytes.

## Key derivation by data encryption - Camellia

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C\_DeriveKey function.

### Definitions

Mechanisms:

CKM\_CAMELLIA\_ECB\_ENCRYPT\_DATA

CKM\_CAMELLIA\_CBC\_ENCRYPT\_DATA

typedef struct CK\_CAMELLIA\_CBC\_ENCRYPT\_DATA\_PARAMS {

CK\_BYTE iv[16];

CK\_BYTE\_PTR pData;

CK\_ULONG length;

} CK\_CAMELLIA\_CBC\_ENCRYPT\_DATA\_PARAMS;

typedef CK\_CAMELLIA\_CBC\_ENCRYPT\_DATA\_PARAMS CK\_PTR CK\_CAMELLIA\_CBC\_ENCRYPT\_DATA\_PARAMS\_PTR;

### Mechanism Parameters

Uses CK\_CAMELLIA\_CBC\_ENCRYPT\_DATA\_PARAMS, and CK\_KEY\_DERIVATION\_STRING\_DATA.

Table 183, Mechanism Parameters for Camellia-based key derivation

|  |  |
| --- | --- |
| CKM\_CAMELLIA\_ECB\_ENCRYPT\_DATA | Uses CK\_KEY\_DERIVATION\_STRING\_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long. |
| CKM\_CAMELLIA\_CBC\_ENCRYPT\_DATA | Uses CK\_CAMELLIA\_CBC\_ENCRYPT\_DATA\_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long. |

## ARIA

ARIA is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES. ARIA is described in NSRI “Specification of ARIA”.

*Table 184, ARIA Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_ARIA\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_ARIA\_ECB | ✓ |  |  |  |  | ✓ |  |
| CKM\_ARIA\_CBC | ✓ |  |  |  |  | ✓ |  |
| CKM\_ARIA\_CBC\_PAD | ✓ |  |  |  |  | ✓ |  |
| CKM\_ARIA\_MAC\_GENERAL |  | ✓ |  |  |  |  |  |
| CKM\_ARIA\_MAC |  | ✓ |  |  |  |  |  |
| CKM\_ARIA\_ECB\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_ARIA\_CBC\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |

### Definitions

This section defines the key type “CKK\_ARIA” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_ARIA\_KEY\_GEN

CKM\_ARIA\_ECB

CKM\_ARIA\_CBC

CKM\_ARIA\_MAC

CKM\_ARIA\_MAC\_GENERAL

CKM\_ARIA\_CBC\_PAD

### Aria secret key objects

ARIA secret key objects (object class **CKO\_SECRET\_KEY,** key type **CKK\_ARIA**) hold ARIA keys. The following table defines the ARIA secret key object attributes, in addition to the common attributes defined for this object class:

Table 185, ARIA Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (16, 24, or 32 bytes) |
| CKA\_VALUE\_LEN2,3,6 | CK\_ULONG | Length in bytes of key value |

- Refer to [PKCS11-Base] table 11 for footnotes.

The following is a sample template for creating an ARIA secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_ARIA;

CK\_UTF8CHAR label[] = “An ARIA secret key object”;

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

### ARIA key generation

The ARIA key generation mechanism, denoted CKM\_ARIA\_KEY\_GEN, is a key generation mechanism for Aria.

It does not have a parameter.

The mechanism generates ARIA keys with a particular length in bytes, as specified in the **CKA\_VALUE\_LEN** attribute of the template for the key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the ARIA key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of ARIA key sizes, in bytes.

### ARIA-ECB

ARIA-ECB, denoted **CKM\_ARIA\_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Aria and electronic codebook mode.

It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 186, ARIA-ECB: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | CKK\_ARIA | multiple of block size | same as input length | no final part |
| C\_Decrypt | CKK\_ARIA | multiple of block size | same as input length | no final part |
| C\_WrapKey | CKK\_ARIA | any | input length rounded up to multiple of block size |  |
| C\_UnwrapKey | CKK\_ARIA | multiple of block size | determined by type of key being unwrapped or CKA\_VALUE\_LEN |  |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of ARIA key sizes, in bytes.

### ARIA-CBC

ARIA-CBC, denoted **CKM\_ARIA\_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA and cipher-block chaining mode.

It has a parameter, a 16-byte initialization vector.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

Constraints on key types and the length of data are summarized in the following table:

Table 187, ARIA-CBC: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| --- | --- | --- | --- | --- |
| C\_Encrypt | CKK\_ARIA | multiple of block size | same as input length | no final part |
| C\_Decrypt | CKK\_ARIA | multiple of block size | same as input length | no final part |
| C\_WrapKey | CKK\_ARIA | any | input length rounded up to multiple of the block size |  |
| C\_UnwrapKey | CKK\_ARIA | multiple of block size | determined by type of key being unwrapped or CKA\_VALUE\_LEN |  |

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK\_MECHANISM\_INFO structure specify the supported range of Aria key sizes, in bytes.

### ARIA-CBC with PKCS padding

ARIA-CBC with PKCS padding, denoted **CKM\_ARIA\_CBC\_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 16-byte initialization vector.

The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for the **CKA\_VALUE\_LEN** attribute.

In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA, Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section TBA for details). The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

Constraints on key types and the length of data are summarized in the following table:

Table 188, ARIA-CBC with PKCS Padding: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Encrypt | CKK\_ARIA | any | input length rounded up to multiple of the block size |
| C\_Decrypt | CKK\_ARIA | multiple of block size | between 1 and block size bytes shorter than input length |
| C\_WrapKey | CKK\_ARIA | any | input length rounded up to multiple of the block size |
| C\_UnwrapKey | CKK\_ARIA | multiple of block size | between 1 and block length bytes shorter than input length |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of ARIA key sizes, in bytes.

### General-length ARIA-MAC

General-length ARIA -MAC, denoted **CKM\_ARIA\_MAC\_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on ARIA and data authentication as defined in [FIPS 113].

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final ARIA cipher block produced in the MACing process.

Constraints on key types and the length of data are summarized in the following table:

Table 189, General-length ARIA-MAC: Key and Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_ARIA | any | 1-block size, as specified in parameters |
| C\_Verify | CKK\_ARIA | any | 1-block size, as specified in parameters |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of ARIA key sizes, in bytes.

### ARIA-MAC

ARIA-MAC, denoted by **CKM\_ARIA\_MAC**, is a special case of the general-length ARIA-MAC mechanism. ARIA-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 190, ARIA-MAC: Key and Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_ARIA | any | ½ block size (8 bytes) |
| C\_Verify | CKK\_ARIA | any | ½ block size (8 bytes) |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of ARIA key sizes, in bytes.

## Key derivation by data encryption - ARIA

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C\_DeriveKey function.

### Definitions

Mechanisms:

CKM\_ARIA\_ECB\_ENCRYPT\_DATA

CKM\_ARIA\_CBC\_ENCRYPT\_DATA

typedef struct CK\_ARIA\_CBC\_ENCRYPT\_DATA\_PARAMS {

CK\_BYTE iv[16];

CK\_BYTE\_PTR pData;

CK\_ULONG length;

} CK\_ARIA\_CBC\_ENCRYPT\_DATA\_PARAMS;

typedef CK\_ARIA\_CBC\_ENCRYPT\_DATA\_PARAMS CK\_PTR CK\_ARIA\_CBC\_ENCRYPT\_DATA\_PARAMS\_PTR;

### Mechanism Parameters

Uses CK\_ARIA\_CBC\_ENCRYPT\_DATA\_PARAMS, and CK\_KEY\_DERIVATION\_STRING\_DATA.

Table 191, Mechanism Parameters for Aria-based key derivation

|  |  |
| --- | --- |
| CKM\_ARIA\_ECB\_ENCRYPT\_DATA | Uses CK\_KEY\_DERIVATION\_STRING\_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long. |
| CKM\_ARIA\_CBC\_ENCRYPT\_DATA | Uses CK\_ARIA\_CBC\_ENCRYPT\_DATA\_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long. |

## SEED

SEED is a symmetric block cipher developed by the South Korean Information Security Agency (KISA). It has a 128-bit key size and a 128-bit block size.

Its specification has been published as Internet [RFC 4269].

RFCs have been published defining the use of SEED in

TLS <ftp://ftp.rfc-editor.org/in-notes/rfc4162.txt>

IPsec <ftp://ftp.rfc-editor.org/in-notes/rfc4196.txt>

CMS <ftp://ftp.rfc-editor.org/in-notes/rfc4010.txt>

TLS cipher suites that use SEED include:

CipherSuite TLS\_RSA\_WITH\_SEED\_CBC\_SHA = { 0x00, 0x96};

CipherSuite TLS\_DH\_DSS\_WITH\_SEED\_CBC\_SHA = { 0x00, 0x97};

CipherSuite TLS\_DH\_RSA\_WITH\_SEED\_CBC\_SHA = { 0x00, 0x98};

CipherSuite TLS\_DHE\_DSS\_WITH\_SEED\_CBC\_SHA = { 0x00, 0x99};

CipherSuite TLS\_DHE\_RSA\_WITH\_SEED\_CBC\_SHA = { 0x00, 0x9A};

CipherSuite TLS\_DH\_anon\_WITH\_SEED\_CBC\_SHA = { 0x00, 0x9B};

As with any block cipher, it can be used in the ECB, CBC, OFB and CFB modes of operation, as well as in a MAC algorithm such as HMAC.

OIDs have been published for all these uses. A list may be seen at <http://www.alvestrand.no/objectid/1.2.410.200004.1.html>

*Table 192, SEED Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SEED\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_SEED\_ECB |  |  | ✓ |  |  |  |  |
| CKM\_SEED\_CBC |  |  | ✓ |  |  |  |  |
| CKM\_SEED\_CBC\_PAD | ✓ |  |  |  |  | ✓ |  |
| CKM\_SEED\_MAC\_GENERAL |  |  | ✓ |  |  |  |  |
| CKM\_SEED\_MAC |  |  |  | ✓ |  |  |  |
| CKM\_SEED\_ECB\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |
| CKM\_SEED\_CBC\_ENCRYPT\_DATA |  |  |  |  |  |  | ✓ |

### Definitions

This section defines the key type “CKK\_SEED” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SEED\_KEY\_GEN

CKM\_SEED\_ECB

CKM\_SEED\_CBC

CKM\_SEED\_MAC

CKM\_SEED\_MAC\_GENERAL

CKM\_SEED\_CBC\_PAD

For all of these mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** are always 16.

### SEED secret key objects

SEED secret key objects (object class **CKO\_SECRET\_KEY,** key type **CKK\_SEED**) hold SEED keys. The following table defines the secret key object attributes, in addition to the common attributes defined for this object class:

Table 193, SEED Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (always 16 bytes long) |

- Refer to [PKCS11-Base] table 11 for footnotes.

The following is a sample template for creating a SEED secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_SEED;

CK\_UTF8CHAR label[] = “A SEED secret key object”;

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

### SEED key generation

The SEED key generation mechanism, denoted CKM\_SEED\_KEY\_GEN, is a key generation mechanism for SEED.

It does not have a parameter.

The mechanism generates SEED keys.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the SEED key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

### SEED-ECB

SEED-ECB, denoted **CKM\_SEED\_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED and electronic codebook mode.

It does not have a parameter.

### SEED-CBC

SEED-CBC, denoted **CKM\_SEED\_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED and cipher-block chaining mode.

It has a parameter, a 16-byte initialization vector.

### SEED-CBC with PKCS padding

SEED-CBC with PKCS padding, denoted **CKM\_SEED\_CBC\_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 16-byte initialization vector.

### General-length SEED-MAC

General-length SEED-MAC, denoted **CKM\_SEED\_MAC\_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on SEED and data authentication as defined in 0.

It has a parameter, a **CK\_MAC\_GENERAL\_PARAMS** structure, which specifies the output length desired from the mechanism.

The output bytes from this mechanism are taken from the start of the final cipher block produced in the MACing process.

### SEED-MAC

SEED-MAC, denoted by **CKM\_SEED\_MAC**, is a special case of the general-length SEED-MAC mechanism. SEED-MAC always produces and verifies MACs that are half the block size in length.

It does not have a parameter.

## Key derivation by data encryption - SEED

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C\_DeriveKey function.

### Definitions

Mechanisms:

CKM\_SEED\_ECB\_ENCRYPT\_DATA

CKM\_SEED\_CBC\_ENCRYPT\_DATA

typedef struct CK\_SEED\_CBC\_ENCRYPT\_DATA\_PARAMS CK\_CBC\_ENCRYPT\_DATA\_PARAMS;

typedef CK\_CBC\_ENCRYPT\_DATA\_PARAMS CK\_PTR CK\_CBC\_ENCRYPT\_DATA\_PARAMS\_PTR;

### Mechanism Parameters

Table 194, Mechanism Parameters for SEED-based key derivation

|  |  |
| --- | --- |
| CKM\_SEED\_ECB\_ENCRYPT\_DATA | Uses CK\_KEY\_DERIVATION\_STRING\_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long. |
| CKM\_SEED\_CBC\_ENCRYPT\_DATA | Uses CK\_CBC\_ENCRYPT\_DATA\_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long. |

## OTP

### Usage overview

OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases can be categorized according to the type of sought functionality.

### Case 1: Generation of OTP values

.

Figure : Retrieving OTP values through C\_Sign

Figure 1 shows an integration of PKCS #11 into an application that needs to authenticate users holding OTP tokens. In this particular example, a connected hardware token is used, but a software token is equally possible. The application invokes **C\_Sign** to retrieve the OTP value from the token. In the example, the application then passes the retrieved OTP value to a client API that sends it via the network to an authentication server. The client API may implement a standard authentication protocol such as RADIUS [RFC 2865] or EAP [RFC 3748], or a proprietary protocol such as that used by RSA Security's ACE/Agent® software.

### Case 2: Verification of provided OTP values



Figure : Server-side verification of OTP values

Figure 2 illustrates the server-side equivalent of the scenario depicted in Figure 1. In this case, a server application invokes **C\_Verify** with the received OTP value as the signature value to be verified.

### Case 3: Generation of OTP keys



Figure : Generation of an OTP key

Figure 3 shows an integration of PKCS #11 into an application that generates OTP keys. The application invokes **C\_GenerateKey** to generate an OTP key of a particular type on the token. The key may subsequently be used as a basis to generate OTP values.

### OTP objects

#### Key objects

OTP key objects (object class **CKO\_OTP\_KEY**) hold secret keys used by OTP tokens. The following table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys, all of which are inherited by this class:

Table 195: Common OTP key attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_OTP\_FORMAT | CK\_ULONG | Format of OTP values produced with this key:  CK\_OTP\_FORMAT\_DECIMAL = Decimal (default) (UTF8-encoded)  CK\_OTP\_FORMAT\_HEXADECIMAL = Hexadecimal (UTF8-encoded)  CK\_OTP\_FORMAT\_ALPHANUMERIC = Alphanumeric (UTF8-encoded)  CK\_OTP\_FORMAT\_BINARY = Only binary values. |
| CKA\_OTP\_LENGTH9 | CK\_ULONG | Default length of OTP values (in the CKA\_OTP\_FORMAT) produced with this key. |
| CKA\_OTP\_USER\_FRIENDLY\_MODE9 | CK\_BBOOL | Set to CK\_TRUE when the token is capable of returning OTPs suitable for human consumption. See the description of CKF\_USER\_FRIENDLY\_OTP below. |
| CKA\_OTP \_CHALLENGE\_REQUIREMENT9 | CK\_ULONG | Parameter requirements when generating or verifying OTP values with this key:  CK\_OTP\_PARAM\_MANDATORY = A challenge must be supplied.  CK\_OTP\_PARAM\_OPTIONAL = A challenge may be supplied but need not be.  CK\_OTP\_PARAM\_IGNORED = A challenge, if supplied, will be ignored. |
| CKA\_OTP\_TIME\_REQUIREMENT9 | CK\_ULONG | Parameter requirements when generating or verifying OTP values with this key:  CK\_OTP\_PARAM\_MANDATORY = A time value must be supplied.  CK\_OTP\_PARAM\_OPTIONAL = A time value may be supplied but need not be.  CK\_OTP\_PARAM\_IGNORED = A time value, if supplied, will be ignored. |
| CKA\_OTP\_COUNTER\_REQUIREMENT9 | CK\_ULONG | Parameter requirements when generating or verifying OTP values with this key:  CK\_OTP\_PARAM\_MANDATORY = A counter value must be supplied.  CK\_OTP\_PARAM\_OPTIONAL = A counter value may be supplied but need not be.  CK\_OTP\_PARAM\_IGNORED = A counter value, if supplied, will be ignored. |
| CKA\_OTP\_PIN\_REQUIREMENT9 | CK\_ULONG | Parameter requirements when generating or verifying OTP values with this key:  CK\_OTP\_PARAM\_MANDATORY = A PIN value must be supplied.  CK\_OTP\_PARAM\_OPTIONAL = A PIN value may be supplied but need not be (if not supplied, then library will be responsible for collecting it)  CK\_OTP\_PARAM\_IGNORED = A PIN value, if supplied, will be ignored. |
| CKA\_OTP\_COUNTER | Byte array | Value of the associated internal counter. Default value is empty (i.e. *ulValueLen* = 0). |
| CKA\_OTP\_TIME | RFC 2279 string | Value of the associated internal UTC time in the form YYYYMMDDhhmmss. Default value is empty (i.e. *ulValueLen*= 0). |
| CKA\_OTP\_USER\_IDENTIFIER | RFC 2279 string | Text string that identifies a user associated with the OTP key (may be used to enhance the user experience). Default value is empty (i.e. *ulValueLen* = 0). |
| CKA\_OTP\_SERVICE\_IDENTIFIER | RFC 2279 string | Text string that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. *ulValueLen* = 0). |
| CKA\_OTP\_SERVICE\_LOGO | Byte array | Logotype image that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. *ulValueLen* = 0). |
| CKA\_OTP\_SERVICE\_LOGO\_TYPE | RFC 2279 string | MIME type of the CKA\_OTP\_SERVICE\_LOGO attribute value. Default value is empty (i.e. *ulValueLen* = 0). |
| CKA\_VALUE1, 4, 6, 7 | Byte array | Value of the key. |
| CKA\_VALUE\_LEN2, 3 | CK\_ULONG | Length in bytes of key value. |

Refer to [PKCS11-Base] table 11 for footnotes.

Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-PKCS #11 application should therefore always consult the state of the CKA\_OTP\_PIN\_REQUIREMENT attribute before each call to **C\_SignInit**, as the value of this attribute may change dynamically.

For OTP tokens with multiple keys, the keys may be enumerated using **C\_FindObjects**. The **CKA\_OTP\_SERVICE\_IDENTIFIER** and/or the **CKA\_OTP\_SERVICE\_LOGO** attribute may be used to distinguish between keys. The actual choice of key for a particular operation is however application-specific and beyond the scope of this document.

For all OTP keys, the CKA\_ALLOWED\_MECHANISMS attribute should be set as required.

### OTP-related notifications

This document extends the set of defined notifications as follows:

CKN\_OTP\_CHANGED Cryptoki is informing the application that the OTP for a key on a connected token just changed. This notification is particularly useful when applications wish to display the current OTP value for time-based mechanisms.

### OTP mechanisms

The following table shows, for the OTP mechanisms defined in this document, their support by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation).

Table 196: OTP mechanisms vs. applicable functions

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SECURID\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_SECURID |  | ✓ |  |  |  |  |  |
| CKM\_HOTP\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_HOTP |  | ✓ |  |  |  |  |  |
| CKM\_ACTI\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_ACTI |  | ✓ |  |  |  |  |  |

The remainder of this section will present in detail the OTP mechanisms and the parameters that are supplied to them.

#### OTP mechanism parameters

* CK\_PARAM\_TYPE

**CK\_PARAM\_TYPE** is a value that identifies an OTP parameter type. It is defined as follows:

typedef CK\_ULONG CK\_PARAM\_TYPE;

The following **CK\_PARAM\_TYPE** types are defined:

Table 197, OTP parameter types

| **Parameter** | **Data type** | **Meaning** |
| --- | --- | --- |
| CK\_OTP\_PIN | RFC 2279 string | A UTF8 string containing a PIN for use when computing or verifying PIN-based OTP values. |
| CK\_OTP\_CHALLENGE | Byte array | Challenge to use when computing or verifying challenge-based OTP values. |
| CK\_OTP\_TIME | RFC 2279 string | UTC time value in the form YYYYMMDDhhmmss to use when computing or verifying time-based OTP values. |
| CK\_OTP\_COUNTER | Byte array | Counter value to use when computing or verifying counter-based OTP values. |
| CK\_OTP\_FLAGS | CK\_FLAGS | Bit flags indicating the characteristics of the sought OTP as defined below. |
| CK\_OTP\_OUTPUT\_LENGTH | CK\_ULONG | Desired output length (overrides any default value). A Cryptoki library will return CKR\_MECHANISM\_PARAM\_INVALID if a provided length value is not supported. |
| CK\_OTP\_FORMAT | CK\_ULONG | Returned OTP format (allowed values are the same as for CKA\_OTP\_FORMAT). This parameter is only intended for **C\_Sign** output, see paragraphs below. When not present, the returned OTP format will be the same as the value of the CKA\_OTP\_FORMAT attribute for the key in question. |
| CK\_OTP\_VALUE | Byte array | An actual OTP value. This parameter type is intended for **C\_Sign** output, see paragraphs below. |

The following table defines the possible values for the CK\_OTP\_FLAGS type:

Table 198: OTP Mechanism Flags

| **Bit flag** | **Mask** | **Meaning** |
| --- | --- | --- |
| CKF\_NEXT\_OTP | 0x00000001 | True (i.e. set) if the OTP computation shall be for the next OTP, rather than the current one (current being interpreted in the context of the algorithm, e.g. for the current counter value or current time window). A Cryptoki library shall return CKR\_MECHANISM\_PARAM\_INVALID if the CKF\_NEXT\_OTP flag is set and the OTP mechanism in question does not support the concept of “next” OTP or the library is not capable of generating the next OTP[[9]](#footnote-9). |
| CKF\_EXCLUDE\_TIME | 0x00000002 | True (i.e. set) if the OTP computation must not include a time value. Will have an effect only on mechanisms that do include a time value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR\_MECHANISM\_PARAM\_INVALID if exclusion of the value is not allowed. |
| CKF\_EXCLUDE\_COUNTER | 0x00000004 | True (i.e. set) if the OTP computation must not include a counter value. Will have an effect only on mechanisms that do include a counter value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR\_MECHANISM\_PARAM\_INVALID if exclusion of the value is not allowed. |
| CKF\_EXCLUDE\_CHALLENGE | 0x00000008 | True (i.e. set) if the OTP computation must not include a challenge. Will have an effect only on mechanisms that do include a challenge in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR\_MECHANISM\_PARAM\_INVALID if exclusion of the value is not allowed. |
| CKF\_EXCLUDE\_PIN | 0x00000010 | True (i.e. set) if the OTP computation must not include a PIN value. Will have an effect only on mechanisms that do include a PIN in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR\_MECHANISM\_PARAM\_INVALID if exclusion of the value is not allowed. |
| CKF\_USER\_FRIENDLY\_OTP | 0x00000020 | True (i.e. set) if the OTP returned shall be in a form suitable for human consumption. If this flag is set, and the call is successful, then the returned CK\_OTP\_VALUE shall be a UTF8-encoded printable string. A Cryptoki library shall return CKR\_MECHANISM\_PARAM\_INVALID if this flag is set when CKA\_OTP\_USER\_FRIENDLY\_MODE for the key in question is CK\_FALSE. |

Note: Even if CKA\_OTP\_FORMAT is not set to CK\_OTP\_FORMAT\_BINARY, then there may still be value in setting the CKF\_USER\_FRIENDLY\_OTP flag (assuming CKA\_OTP\_USER\_FRIENDLY\_MODE is CK\_TRUE, of course) if the intent is for a human to read the generated OTP value, since it may become shorter or otherwise better suited for a user. Applications that do not intend to provide a returned OTP value to a user should not set the CKF\_USER\_FRIENDLY\_OTP flag.

* CK\_OTP\_PARAM; CK\_OTP\_PARAM\_PTR

**CK\_OTP\_PARAM** is a structure that includes the type, value, and length of an OTP parameter. It is defined as follows:

typedef struct CK\_OTP\_PARAM {

CK\_PARAM\_TYPE type;

CK\_VOID\_PTR pValue;

CK\_ULONG ulValueLen;

} CK\_OTP\_PARAM;

The fields of the structure have the following meanings:

type the parameter type

pValue pointer to the value of the parameter

ulValueLen length in bytes of the value

If a parameter has no value, then *ulValueLen* = 0, and the value of *pValue* is irrelevant. Note that *pValue* is a “void” pointer, facilitating the passing of arbitrary values. Both the application and the Cryptoki library must ensure that the pointer can be safely cast to the expected type (*i.e.*, without word-alignment errors).

**CK\_OTP\_PARAM\_PTR** is a pointer to a **CK\_OTP\_PARAM**.

* CK\_OTP\_PARAMS; CK\_OTP\_PARAMS\_PTR

**CK\_OTP\_PARAMS** is a structure that is used to provide parameters for OTP mechanisms in a generic fashion. It is defined as follows:

typedef struct CK\_OTP\_PARAMS {

CK\_OTP\_PARAM\_PTR pParams;

CK\_ULONG ulCount;

} CK\_OTP\_PARAMS;

The fields of the structure have the following meanings:

pParams pointer to an array of OTP parameters

ulCount the number of parameters in the array

**CK\_OTP\_PARAMS\_PTR** is a pointer to a **CK\_OTP\_PARAMS**.

When calling C\_SignInit or C\_VerifyInit with a mechanism that takes a **CK\_OTP\_PARAMS** structure as a parameter, the **CK\_OTP\_PARAMS** structure shall be populated in accordance with the **CKA\_OTP\_*X*\_REQUIREMENT** key attributes for the identified key, where *X* is PIN, CHALLENGE, TIME, or COUNTER.

For example, if CKA\_OTP\_TIME\_REQUIREMENT = CK\_OTP\_PARAM\_MANDATORY, then the CK\_OTP\_TIME parameter shall be present. If CKA\_OTP\_TIME\_REQUIREMENT = CK\_OTP\_PARAM\_OPTIONAL, then a CK\_OTP\_TIME parameter may be present. If it is not present, then the library may collect it (during the C\_Sign call). If CKA\_OTP\_TIME\_REQUIREMENT = CK\_OTP\_PARAM\_IGNORED, then a provided CK\_OTP\_TIME parameter will always be ignored. Additionally, a provided CK\_OTP\_TIME parameter will always be ignored if CKF\_EXCLUDE\_TIME is set in a CK\_OTP\_FLAGS parameter. Similarly, if this flag is set, a library will not attempt to collect the value itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is an error (CKR\_MECHANISM\_PARAM\_INVALID) to set the CKF\_EXCLUDE\_TIME flag when the CKA\_OTP\_TIME\_REQUIREMENT attribute is CK\_OTP\_PARAM\_MANDATORY.

The above discussion holds for all CKA\_OTP\_*X*\_REQUIREMENT attributes (*i.e*., CKA\_OTP\_PIN\_REQUIREMENT, CKA\_OTP\_CHALLENGE\_REQURIEMENT, CKA\_OTP\_COUNTER\_REQUIREMENT, CKA\_OTP\_TIME\_REQUIREMENT). A library may set a particular CKA\_OTP\_*X*\_REQUIREMENT attribute to CK\_OTP\_PARAM\_OPTIONAL even if it is required by the mechanism as long as the token (or the library itself) has the capability of providing the value to the computation. One example of this is a token with an on-board clock.

In addition, applications may use the CK\_OTP\_FLAGS, the CK\_OTP\_FORMAT and the CKA\_OTP\_LENGTH parameters to set additional parameters.

* CK\_OTP\_SIGNATURE\_INFO, CK\_OTP\_SIGNATURE\_INFO\_PTR

**CK\_OTP\_SIGNATURE\_INFO** is a structure that is returned by all OTP mechanisms in successful calls to **C\_Sign** (**C\_SignFinal**). The structure informs applications of actual parameter values used in particular OTP computations in addition to the OTP value itself. It is used by all mechanisms for which the key belongs to the class CKO\_OTP\_KEY and is defined as follows:

typedef struct CK\_OTP\_SIGNATURE\_INFO {

CK\_OTP\_PARAM\_PTR pParams;

CK\_ULONG ulCount;

} CK\_OTP\_SIGNATURE\_INFO;

The fields of the structure have the following meanings:

pParams pointer to an array of OTP parameter values

ulCount the number of parameters in the array

After successful calls to **C\_Sign** or **C\_SignFinal** with an OTP mechanism, the *pSignature* parameter will be set to point to a **CK\_OTP\_SIGNATURE\_INFO** structure. One of the parameters in this structure will be the OTP value itself, identified with the **CK\_OTP\_VALUE** tag. Other parameters may be present for informational purposes, e.g. the actual time used in the OTP calculation. In order to simplify OTP validations, authentication protocols may permit authenticating parties to send some or all of these parameters in addition to OTP values themselves. Applications should therefore check for their presence in returned **CK\_OTP\_SIGNATURE\_INFO** valueswhenever such circumstances apply.

Since **C\_Sign** and **C\_SignFinal** follows the convention described in [PKCS11-Base] Section 5.2 on producing output, a call to **C\_Sign** (or **C\_SignFinal**) with *pSignature* set to NULL\_PTR will return (in the *pulSignatureLen* parameter) the required number of bytes to hold the **CK\_OTP\_SIGNATURE\_INFO** structure as well as all the data in all its **CK\_OTP\_PARAM** components. If an application allocates a memory block based on this information, it shall therefore not subsequently de-allocate components of such a received value but rather de-allocate the complete **CK\_OTP\_PARAMS** structure itself. A Cryptoki library that is called with a non-NULL *pSignature* pointer will assume that it points to a *contiguous* memory block of the size indicated by the *pulSignatureLen* parameter.

When verifying an OTP value using an OTP mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK\_OTP\_VALUE** component of a **CK\_OTP\_PARAM** structure returned by a call to **C\_Sign**. The **CK\_OTP\_PARAM** value supplied in the **C\_VerifyInit** call sets the values to use in the verification operation.

**CK\_OTP\_SIGNATURE\_INFO\_PTR** points to a **CK\_OTP\_SIGNATURE\_INFO.**

### RSA SecurID

#### RSA SecurID secret key objects

RSA SecurID secret key objects (object class **CKO\_OTP\_KEY,** key type **CKK\_SECURID**) hold RSA SecurID secret keys. The following table defines the RSA SecurID secret key object attributes, in addition to the common attributes defined for this object class:

Table 199, RSA SecurID secret key object attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_OTP\_TIME\_INTERVAL1 | CK\_ULONG | Interval between OTP values produced with this key, in seconds. Default is 60. |

Refer to [PKCS11-Base] table 11 for footnotes.

The following is a sample template for creating an RSA SecurID secret key object:

CK\_OBJECT\_CLASS class = CKO\_OTP\_KEY;

CK\_KEY\_TYPE keyType = CKK\_SECURID;

CK\_DATE endDate = {...};

CK\_UTF8CHAR label[] = “RSA SecurID secret key object”;

CK\_BYTE keyId[]= {...};

CK\_ULONG outputFormat = CK\_OTP\_FORMAT\_DECIMAL;

CK\_ULONG outputLength = 6;

CK\_ULONG needPIN = CK\_OTP\_PARAM\_MANDATORY;

CK\_ULONG timeInterval = 60;

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_END\_DATE, &endDate, sizeof(endDate)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SIGN, &true, sizeof(true)},

{CKA\_VERIFY, &true, sizeof(true)},

{CKA\_ID, keyId, sizeof(keyId)},

{CKA\_OTP\_FORMAT, &outputFormat, sizeof(outputFormat)},

{CKA\_OTP\_LENGTH, &outputLength, sizeof(outputLength)},

{CKA\_OTP\_PIN\_REQUIREMENT, &needPIN, sizeof(needPIN)},

{CKA\_OTP\_TIME\_INTERVAL, &timeInterval, sizeof(timeInterval)},

{CKA\_VALUE, value, sizeof(value)}

};

#### RSA SecurID key generation

The RSA SecurID key generation mechanism, denoted **CKM\_SECURID\_KEY\_GEN**, is a key generation mechanism for the RSA SecurID algorithm.

It does not have a parameter.

The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_VALUE\_LEN**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the RSA SecurID key type may be specified in the template for the key, or else are assigned default initial values

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of SecurID key sizes, in bytes.

#### SecurID OTP generation and validation

**CKM\_SECURID** is the mechanism for the retrieval and verification of RSA SecurID OTP values.

The mechanism takes a pointer to a **CK\_OTP\_PARAMS** structure as a parameter.

When signing or verifying using the **CKM\_SECURID** mechanism, *pData* shall be set to NULL\_PTR and *ulDataLen* shall be set to 0.

#### Return values

Support for the CKM\_SECURID mechanism extends the set of return values for C\_Verify with the following values:

* CKR\_NEW\_PIN\_MODE: The supplied OTP was not accepted and the library requests a new OTP computed using a new PIN. The new PIN is set through means out of scope for this document.
* CKR\_NEXT\_OTP: The supplied OTP was correct but indicated a larger than normal drift in the token's internal state (e.g. clock, counter). To ensure this was not due to a temporary problem, the application should provide the next one-time password to the library for verification.

### OATH HOTP

#### OATH HOTP secret key objects

HOTP secret key objects (object class **CKO\_OTP\_KEY,** key type **CKK\_HOTP**) hold generic secret keys and associated counter values.

The **CKA\_OTP\_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token’s security policy, this value may not be modified and/or may not be revealed if the object has its **CKA\_SENSITIVE** attribute set to CK\_TRUE or its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE.

For HOTP keys, the **CKA\_OTP\_COUNTER** valueshall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for a **CK\_OTP\_COUNTER** value in a **CK\_OTP\_PARAM** structure.

The following is a sample template for creating a HOTP secret key object:

CK\_OBJECT\_CLASS class = CKO\_OTP\_KEY;

CK\_KEY\_TYPE keyType = CKK\_HOTP;

CK\_UTF8CHAR label[] = “HOTP secret key object”;

CK\_BYTE keyId[]= {...};

CK\_ULONG outputFormat = CK\_OTP\_FORMAT\_DECIMAL;

CK\_ULONG outputLength = 6;

CK\_DATE endDate = {...};

CK\_BYTE counterValue[8] = {0};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_END\_DATE, &endDate, sizeof(endDate)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SIGN, &true, sizeof(true)},

{CKA\_VERIFY, &true, sizeof(true)},

{CKA\_ID, keyId, sizeof(keyId)},

{CKA\_OTP\_FORMAT, &outputFormat, sizeof(outputFormat)},

{CKA\_OTP\_LENGTH, &outputLength, sizeof(outputLength)},

{CKA\_OTP\_COUNTER, counterValue, sizeof(counterValue)},

{CKA\_VALUE, value, sizeof(value)}

};

#### HOTP key generation

The HOTP key generation mechanism, denoted **CKM\_HOTP\_KEY\_GEN**, is a key generation mechanism for the HOTP algorithm.

It does not have a parameter.

The mechanism generates HOTP keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_OTP\_COUNTER**, **CKA\_VALUE** and **CKA\_VALUE\_LEN** attributes to the new key. Other attributes supported by the HOTP key type may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of HOTP key sizes, in bytes.

#### HOTP OTP generation and validation

**CKM\_HOTP** is the mechanism for the retrieval and verification of HOTP OTP values based on the current internal counter, or a provided counter.

The mechanism takes a pointer to a **CK\_OTP\_PARAMS** structure as a parameter.

As for the **CKM\_SECURID** mechanism, when signing or verifying using the **CKM\_HOTP** mechanism, *pData* shall be set to NULL\_PTR and *ulDataLen* shall be set to 0.

For verify operations, the counter value **CK\_OTP\_COUNTER** must be provided as a **CK\_OTP\_PARAM** parameter to **C\_VerifyInit**. When verifying an OTP value using the **CKM\_HOTP** mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK\_OTP\_VALUE** component of a **CK\_OTP\_PARAM** structure in the case of an earlier call to **C\_Sign**.

### ActivIdentity ACTI

#### ACTI secret key objects

ACTI secret key objects (object class **CKO\_OTP\_KEY,** key type **CKK\_ACTI**) hold ActivIdentity ACTI secret keys.

For ACTI keys, the **CKA\_OTP\_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for the **CK\_OTP\_COUNTER** value in the **CK\_OTP\_PARAM** structure.

The **CKA\_OTP\_COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token’s security policy, this value may not be modified and/or may not be revealed if the object has its **CKA\_SENSITIVE** attribute set to CK\_TRUE or its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE.

The **CKA\_OTP\_TIME** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token’s security policy, this value may not be modified and/or may not be revealed if the object has its **CKA\_SENSITIVE** attribute set to CK\_TRUE or its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE.

The following is a sample template for creating an ACTI secret key object:

CK\_OBJECT\_CLASS class = CKO\_OTP\_KEY;

CK\_KEY\_TYPE keyType = CKK\_ACTI;

CK\_UTF8CHAR label[] = “ACTI secret key object”;

CK\_BYTE keyId[]= {...};

CK\_ULONG outputFormat = CK\_OTP\_FORMAT\_DECIMAL;

CK\_ULONG outputLength = 6;

CK\_DATE endDate = {...};

CK\_BYTE counterValue[8] = {0};

CK\_BYTE value[] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_END\_DATE, &endDate, sizeof(endDate)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SIGN, &true, sizeof(true)},

{CKA\_VERIFY, &true, sizeof(true)},

{CKA\_ID, keyId, sizeof(keyId)},

{CKA\_OTP\_FORMAT, &outputFormat,

sizeof(outputFormat)},

{CKA\_OTP\_LENGTH, &outputLength,

sizeof(outputLength)},

{CKA\_OTP\_COUNTER, counterValue,

sizeof(counterValue)},

{CKA\_VALUE, value, sizeof(value)}

};

#### ACTI key generation

The ACTI key generation mechanism, denoted **CKM\_ACTI\_KEY\_GEN**, is a key generation mechanism for the ACTI algorithm.

It does not have a parameter.

The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_VALUE** and **CKA\_VALUE\_LEN** attributes to the new key. Other attributes supported by the ACTI key type may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of ACTI key sizes, in bytes.

#### ACTI OTP generation and validation

**CKM\_ACTI** is the mechanism for the retrieval and verification of ACTI OTP values.

The mechanism takes a pointer to a **CK\_OTP\_PARAMS** structure as a parameter.

When signing or verifying using the **CKM\_ACTI** mechanism, *pData* shall be set to NULL\_PTR and *ulDataLen* shall be set to 0.

When verifying an OTP value using the **CKM\_ACTI** mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK\_OTP\_VALUE** component of a **CK\_OTP\_PARAM** structure in the case of an earlier call to **C\_Sign**.

## CT-KIP

### Principles of Operation



Figure 4: PKCS #11 and CT-KIP integration

Figure 4 shows an integration of PKCS #11 into an application that generates cryptographic keys through the use of CT-KIP. The application invokes **C\_DeriveKey** to derive a key of a particular type on the token. The key may subsequently be used as a basis to e.g., generate one-time password values. The application communicates with a CT-KIP server that participates in the key derivation and stores a copy of the key in its database. The key is transferred to the server in wrapped form, after a call to **C\_WrapKey**. The server authenticates itself to the client and the client verifies the authentication by calls to **C\_Verify**.

### Mechanisms

The following table shows, for the mechanisms defined in this document, their support by different cryptographic operations. For any particular token, of course, a particular operation may well support only a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism for some operation supports any other mechanism for any other operation (or even supports that same mechanism for any other operation).

Table 200: CT-KIP Mechanisms vs. applicable functions

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_KIP\_DERIVE |  |  |  |  |  |  | ✓ |
| CKM\_KIP\_WRAP |  |  |  |  |  | ✓ |  |
| CKM\_KIP\_MAC |  | ✓ |  |  |  |  |  |

The remainder of this section will present in detail the mechanisms and the parameters that are supplied to them.

### Definitions

Mechanisms:

CKM\_KIP\_DERIVE

CKM\_KIP\_WRAP

CKM\_KIP\_MAC

### CT-KIP Mechanism parameters

1. CK\_KIP\_PARAMS; CK\_KIP\_PARAMS\_PTR

**CK\_KIP\_PARAMS** is a structure that provides the parameters to all the CT-KIP related mechanisms: The **CKM\_KIP\_DERIVE** key derivation mechanism, the **CKM\_KIP\_WRAP** key wrap and key unwrap mechanism, and the **CKM\_KIP\_MAC** signature mechanism. The structure is defined as follows:

typedef struct CK\_KIP\_PARAMS {

CK\_MECHANISM\_PTR pMechanism;

CK\_OBJECT\_HANDLE hKey;

CK\_BYTE\_PTR pSeed;

CK\_ULONG ulSeedLen;

} CK\_KIP\_PARAMS;

The fields of the structure have the following meanings:

pMechanism pointer to the underlying cryptographic mechanism (e.g. AES, SHA-256), see further 0, Appendix D

hKey handle to a key that will contribute to the entropy of the derived key (CKM\_KIP\_DERIVE) or will be used in the MAC operation (CKM\_KIP\_MAC)

pSeed pointer to an input seed

ulSeedLen length in bytes of the input seed

**CK\_KIP\_PARAMS\_PTR** is a pointer to a **CK\_KIP\_PARAMS** structure.

### CT-KIP key derivation

The CT-KIP key derivation mechanism, denoted **CKM\_KIP\_DERIVE**, is a key derivation mechanism that is capable of generating secret keys of potentially any type, subject to token limitations.

It takes a parameter of type **CK\_KIP\_PARAMS** which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. In particular, when the *hKey* parameter is a handle to an existing key, that key will be used in the key derivation in addition to the *hBaseKey* of **C\_DeriveKey**. The *pSeed* parameter may be used to seed the key derivation operation.

The mechanism derives a secret key with a particular set of attributes as specified in the attributes of the template for the key.

The mechanism contributes the **CKA\_CLASS** and **CKA\_VALUE** attributes to the new key. Other attributes supported by the key type may be specified in the template for the key, or else will be assigned default initial values. Since the mechanism is generic, the **CKA\_KEY\_TYPE** attribute should be set in the template, if the key is to be used with a particular mechanism.

### CT-KIP key wrap and key unwrap

The CT-KIP key wrap and unwrap mechanism, denoted **CKM\_KIP\_WRAP**, is a key wrap mechanism that is capable of wrapping and unwrapping generic secret keys.

It takes a parameter of type **CK\_KIP\_PARAMS**, which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. It does not make use of the *hKey* parameter of **CK\_KIP\_PARAMS**.

### CT-KIP signature generation

The CT-KIP signature (MAC) mechanism, denoted **CKM\_KIP\_MAC**, is a mechanism used to produce a message authentication code of arbitrary length. The keys it uses are secret keys.

It takes a parameter of type **CK\_KIP\_PARAMS**, which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. The mechanism does not make use of the *pSeed* and the *ulSeedLen* parameters of **CT\_KIP\_PARAMS**.

This mechanism produces a MAC of the length specified by *pulSignatureLen* parameter in calls to **C\_Sign**.

If a call to **C\_Sign** with this mechanism fails, then no output will be generated.

## GOST 28147-89

GOST 28147-89 is a block cipher with 64-bit block size and 256-bit keys.

*Table 201, GOST 28147-89 Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Functions** | | | | | | |
| **Encrypt & Decrypt** | **Sign & Verify** | **SR & VR** | **Digest** | **Gen. Key/ Key Pair** | **Wrap & Unwrap** | **Derive** |
| CKM\_GOST28147\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_GOST28147\_ECB | ✓ |  |  |  |  | ✓ |  |
| CKM\_GOST28147 | ✓ |  |  |  |  | ✓ |  |
| CKM\_GOST28147\_MAC |  | ✓ |  |  |  |  |  |
| CKM\_GOST28147\_KEY\_WRAP |  |  |  |  |  | ✓ |  |

### Definitions

This section defines the key type “CKK\_GOST28147” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects and domain parameter objects.

Mechanisms:

CKM\_GOST28147\_KEY\_GEN

CKM\_GOST28147\_ECB

CKM\_GOST28147

CKM\_GOST28147\_MAC

CKM\_GOST28147\_KEY\_WRAP

### GOST 28147-89 secret key objects

GOST 28147‑89 secret key objects (object class **CKO\_SECRET\_KEY,** key type **CKK\_GOST28147**) hold GOST 28147‑89 keys. The following table defines the GOST 28147‑89 secret key object attributes, in addition to the common attributes defined for this object class:

*Table 202, GOST 28147-89 Secret Key Object Attributes*

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data type** | **Meaning** |
| CKA\_VALUE1,4,6,7 | Byte array | 32 bytes in little endian order |
| CKA\_GOST28147\_PARAMS1,3,5 | Byte array | DER-encoding of the object identifier indicating the data object type of GOST 28147‑89.  When key is used the domain parameter object of key type CKK\_GOST28147 must be specified with the same attribute CKA\_OBJECT\_ID |

Refer to [PKCS11-Base] Table 11 for footnotes

The following is a sample template for creating a GOST 28147‑89 secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_GOST28147;

CK\_UTF8CHAR label[] = “A GOST 28147-89 secret key object”;

CK\_BYTE value[32] = {...};

CK\_BYTE params\_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_GOST28147\_PARAMS, params\_oid, sizeof(params\_oid)},

{CKA\_VALUE, value, sizeof(value)}

};

### GOST 28147-89 domain parameter objects

GOST 28147‑89 domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS,** key type **CKK\_GOST28147**) hold GOST 28147‑89 domain parameters.

The following table defines the GOST 28147‑89 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 203, GOST 28147-89 Domain Parameter Object Attributes

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1 | Byte array | DER-encoding of the domain parameters as it was introduced in [4] section 8.1 (type *Gost28147-89-ParamSetParameters*) |
| CKA\_OBJECT\_ID1 | Byte array | DER-encoding of the object identifier indicating the domain parameters |

Refer to [PKCS11-Base] Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that **CKA\_VALUE** attribute may be inaccessible.

The following is a sample template for creating a GOST 28147‑89 domain parameter object:

CK\_OBJECT\_CLASS class = CKO\_DOMAIN\_PARAMETERS;

CK\_KEY\_TYPE keyType = CKK\_GOST28147;

CK\_UTF8CHAR label[] = “A GOST 28147-89 cryptographic parameters object”;

CK\_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};

CK\_BYTE value[] = {

0x30,0x62,0x04,0x40,0x4c,0xde,0x38,0x9c,0x29,0x89,0xef,0xb6,

0xff,0xeb,0x56,0xc5,0x5e,0xc2,0x9b,0x02,0x98,0x75,0x61,0x3b,

0x11,0x3f,0x89,0x60,0x03,0x97,0x0c,0x79,0x8a,0xa1,0xd5,0x5d,

0xe2,0x10,0xad,0x43,0x37,0x5d,0xb3,0x8e,0xb4,0x2c,0x77,0xe7,

0xcd,0x46,0xca,0xfa,0xd6,0x6a,0x20,0x1f,0x70,0xf4,0x1e,0xa4,

0xab,0x03,0xf2,0x21,0x65,0xb8,0x44,0xd8,0x02,0x01,0x00,0x02,

0x01,0x40,0x30,0x0b,0x06,0x07,0x2a,0x85,0x03,0x02,0x02,0x0e,

0x00,0x05,0x00

};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_OBJECT\_ID, oid, sizeof(oid)},

{CKA\_VALUE, value, sizeof(value)}

};

### GOST 28147-89 key generation

The GOST 28147‑89 key generation mechanism, denoted **CKM\_GOST28147\_KEY\_GEN**, is a key generation mechanism for GOST 28147‑89.

It does not have a parameter.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the GOST 28147‑89 key type may be specified for objects of object class **CKO\_SECRET\_KEY**.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** are not used.

### GOST 28147-89-ECB

GOST 28147‑89-ECB, denoted **CKM\_GOST28147\_ECB**, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147‑89 and electronic codebook mode.

It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports.

For wrapping (**C\_WrapKey**), the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size so that the resulting length is a multiple of the block size.

For unwrapping (**C\_UnwrapKey**), the mechanism decrypts the wrapped key, and truncates the result according to the **CKA\_KEY\_TYPE** attribute of the template and, if it has one, and the key type supports it, the **CKA\_VALUE\_LEN** attribute of the template. The mechanism contributes the result as the **CKA\_VALUE** attribute of the new key.

Constraints on key types and the length of data are summarized in the following table:

*Table 204, GOST 28147-89-ECB: Key and Data Length*

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Key type** | **Input length** | **Output length** |
| C\_Encrypt | CKK\_GOST28147 | Multiple of block size | Same as input length |
| C\_Decrypt | CKK\_GOST28147 | Multiple of block size | Same as input length |
| C\_WrapKey | CKK\_GOST28147 | Any | Input length rounded up to multiple of block size |
| C\_UnwrapKey | CKK\_GOST28147 | Multiple of block size | Determined by type of key being unwrapped |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

### GOST 28147-89 encryption mode except ECB

GOST 28147‑89 encryption mode except ECB, denoted **CKM\_GOST28147**, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on [GOST 28147‑89] and CFB, counter mode, and additional CBC mode defined in [RFC 4357] section 2. Encryption’s parameters are specified in object identifier of attribute **CKA\_GOST28147\_PARAMS**.

It has a parameter, which is an 8-byte initialization vector. This parameter may be omitted then a zero initialization vector is used.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports.

For wrapping (**C\_WrapKey**), the mechanism encrypts the value of the **CKA\_VALUE** attribute of the key that is wrapped.

For unwrapping (**C\_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as the **CKA\_VALUE** attribute of the new key.

Constraints on key types and the length of data are summarized in the following table:

*Table 205, GOST 28147-89 encryption modes except ECB: Key and Data Length*

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Key type** | **Input length** | **Output length** |
| C\_Encrypt | CKK\_GOST28147 | Any | For counter mode and CFB is the same as input length. For CBC is the same as input length padded on the trailing end with up to block size so that the resulting length is a multiple of the block size |
| C\_Decrypt | CKK\_GOST28147 | Any |
| C\_WrapKey | CKK\_GOST28147 | Any |
| C\_UnwrapKey | CKK\_GOST28147 | Any |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

### GOST 28147-89-MAC

GOST 28147-89-MAC, denoted **CKM\_GOST28147\_MAC**, is a mechanism for data integrity and authentication based on GOST 28147-89 and key meshing algorithms [RFC 4357] section 2.3.

MACing parameters are specified in object identifier of attribute **CKA\_GOST28147\_PARAMS**.

The output bytes from this mechanism are taken from the start of the final GOST 28147‑89 cipher block produced in the MACing process.

It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

Constraints on key types and the length of data are summarized in the following table:

*Table 206, GOST28147-89-MAC: Key and Data Length*

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Key type** | **Data length** | **Signature length** |
| C\_Sign | CKK\_GOST28147 | Any | 4 bytes |
| C\_Verify | CKK\_GOST28147 | Any | 4 bytes |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

### GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89

GOST 28147‑89 keys as a KEK (key encryption keys) for encryption GOST 28147‑89 keys, denoted by **CKM\_GOST28147\_KEY\_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on GOST 28147‑89. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147‑89.

For wrapping (**C\_WrapKey**), the mechanism first computes MAC from the value of the **CKA\_VALUE** attribute of the key that is wrapped and then encrypts in ECB mode the value of the **CKA\_VALUE** attribute of the key that is wrapped. The result is 32 bytes of the key that is wrapped and 4 bytes of MAC.

For unwrapping (**C\_UnwrapKey**), the mechanism first decrypts in ECB mode the 32 bytes of the key that was wrapped and then computes MAC from the unwrapped key. Then compared together 4 bytes MAC has computed and 4 bytes MAC of the input. If these two MACs do not match the wrapped key is disallowed. The mechanism contributes the result as the **CKA\_VALUE** attribute of the unwrapped key.

It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

Constraints on key types and the length of data are summarized in the following table:

*Table 207, GOST 28147-89 keys as KEK: Key and Data Length*

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Key type** | **Input length** | **Output length** |
| C\_WrapKey | CKK\_GOST28147 | 32 bytes | 36 bytes |
| C\_UnwrapKey | CKK\_GOST28147 | 32 bytes | 36 bytes |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

## GOST R 34.11-94

GOST R 34.11-94 is a mechanism for message digesting, following the hash algorithm with 256-bit message digest defined in [GOST R 34.11-94].

*Table 208, GOST R 34.11-94 Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Functions** | | | | | | |
| **Encrypt & Decrypt** | **Sign & Verify** | **SR & VR** | **Digest** | **Gen. Key/ Key Pair** | **Wrap & Unwrap** | **Derive** |
| CKM\_GOSTR3411 |  |  |  | ✓ |  |  |  |
| CKM\_GOSTR3411\_HMAC |  | ✓ |  |  |  |  |  |

### Definitions

This section defines the key type “CKK\_GOSTR3411” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of domain parameter objects.

Mechanisms:

CKM\_GOSTR3411

CKM\_GOSTR3411\_HMAC

### GOST R 34.11-94 domain parameter objects

GOST R 34.11-94 domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS,** key type **CKK\_GOSTR3411**) hold GOST R 34.11-94 domain parameters.

The following table defines the GOST R 34.11-94 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 209, GOST R 34.11-94 Domain Parameter Object Attributes

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1 | Byte array | DER-encoding of the domain parameters as it was introduced in [4] section 8.2 (type *GostR3411-94-ParamSetParameters*) |
| CKA\_OBJECT\_ID1 | Byte array | DER-encoding of the object identifier indicating the domain parameters |

Refer to [PKCS11-Base] Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that **CKA\_VALUE** attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.11-94 domain parameter object:

CK\_OBJECT\_CLASS class = CKO\_DOMAIN\_PARAMETERS;

CK\_KEY\_TYPE keyType = CKK\_GOSTR3411;

CK\_UTF8CHAR label[] = “A GOST R34.11-94 cryptographic parameters object”;

CK\_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};

CK\_BYTE value[] = {

0x30,0x64,0x04,0x40,0x4e,0x57,0x64,0xd1,0xab,0x8d,0xcb,0xbf,

0x94,0x1a,0x7a,0x4d,0x2c,0xd1,0x10,0x10,0xd6,0xa0,0x57,0x35,

0x8d,0x38,0xf2,0xf7,0x0f,0x49,0xd1,0x5a,0xea,0x2f,0x8d,0x94,

0x62,0xee,0x43,0x09,0xb3,0xf4,0xa6,0xa2,0x18,0xc6,0x98,0xe3,

0xc1,0x7c,0xe5,0x7e,0x70,0x6b,0x09,0x66,0xf7,0x02,0x3c,0x8b,

0x55,0x95,0xbf,0x28,0x39,0xb3,0x2e,0xcc,0x04,0x20,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00

};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_OBJECT\_ID, oid, sizeof(oid)},

{CKA\_VALUE, value, sizeof(value)}

};

### GOST R 34.11-94 digest

GOST R 34.11-94 digest, denoted **CKM\_GOSTR3411,** is a mechanism for message digesting based on GOST R 34.11-94 hash algorithm [GOST R 34.11-94].

As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357] (section 11.2) must be used.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 210, GOST R 34.11-94: Data Length

| **Function** | **Input length** | **Digest length** |
| --- | --- | --- |
| C\_Digest | Any | 32 bytes |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

### GOST R 34.11-94 HMAC

GOST R 34.11-94 HMAC mechanism, denoted **CKM\_GOSTR3411\_HMAC**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the GOST R 34.11-94 hash function [GOST R 34.11-94] and core HMAC algorithm [RFC 2104]. The keys it uses are of generic key type **CKK\_GENERIC\_SECRET** or **CKK\_GOST28147**.

To be conformed to GOST R 34.11-94 hash algorithm [GOST R 34.11-94] the block length of core HMAC algorithm is 32 bytes long (see [RFC 2104] section 2, and [RFC 4357] section 3).

As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357] (section 11.2) must be used.

Signatures (MACs) produced by this mechanism are of 32 bytes long.

Constraints on the length of input and output data are summarized in the following table:

Table 211, GOST R 34.11-94 HMAC: Key And Data Length

| **Function** | **Key type** | **Data length** | **Signature length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_GENERIC\_SECRET or CKK\_GOST28147 | Any | 32 byte |
| C\_Verify | CKK\_GENERIC\_SECRET or CKK\_GOST28147 | Any | 32 bytes |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

## GOST R 34.10-2001

GOST R 34.10-2001 is a mechanism for single- and multiple-part signatures and verification, following the digital signature algorithm defined in [GOST R 34.10-2001].

*Table 212, GOST R34.10-2001 Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Functions** | | | | | | |
| **Encrypt & Decrypt** | **Sign & Verify** | **SR & VR** | **Digest** | **Gen. Key/ Key Pair** | **Wrap & Unwrap** | **Derive** |
| CKM\_GOSTR3410\_KEY\_PAIR\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_GOSTR3410 |  | ✓1 |  |  |  |  |  |
| CKM\_GOSTR3410\_WITH\_GOST3411 |  | ✓ |  |  |  |  |  |
| CKM\_GOSTR3410\_KEY\_WRAP |  |  |  |  |  | ✓ |  |
| CKM\_GOSTR3410\_DERIVE |  |  |  |  |  |  | ✓ |

1 Single-part operations only

### Definitions

This section defines the key type “CKK\_GOSTR3410” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects and domain parameter objects.

Mechanisms:

CKM\_GOSTR3410\_KEY\_PAIR\_GEN

CKM\_GOSTR3410

CKM\_GOSTR3410\_WITH\_GOSTR3411

CKM\_GOSTR3410

CKM\_GOSTR3410\_KEY\_WRAP

CKM\_GOSTR3410\_DERIVE

### GOST R 34.10-2001 public key objects

GOST R 34.10-2001 public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_GOSTR3410**) hold GOST R 34.10-2001 public keys.

The following table defines the GOST R 34.10-2001 public key object attributes, in addition to the common attributes defined for this object class:

Table 213, GOST R 34.10-2001 Public Key Object Attributes

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4 | Byte array | 64 bytes for public key; 32 bytes for each coordinates X and Y of elliptic curve point P(X, Y) in little endian order |
| CKA\_GOSTR3410\_PARAMS1,3 | Byte array | DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001.  When key is used the domain parameter object of key type CKK\_GOSTR3410 must be specified with the same attribute CKA\_OBJECT\_ID |
| CKA\_GOSTR3411\_PARAMS1,3,8 | Byte array | DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94.  When key is used the domain parameter object of key type CKK\_GOSTR3411 must be specified with the same attribute CKA\_OBJECT\_ID |
| CKA\_GOST28147\_PARAMS8 | Byte array | DER-encoding of the object identifier indicating the data object type of GOST 28147‑89.  When key is used the domain parameter object of key type CKK\_GOST28147 must be specified with the same attribute CKA\_OBJECT\_ID. The attribute value may be omitted |

Refer to [PKCS11-Base] Table 11 for footnotes

The following is a sample template for creating an GOST R 34.10-2001 public key object:

CK\_OBJECT\_CLASS class = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_GOSTR3410;

CK\_UTF8CHAR label[] = “A GOST R34.10-2001 public key object”;

CK\_BYTE gostR3410params\_oid[] =

{0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};

CK\_BYTE gostR3411params\_oid[] =

{0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};

CK\_BYTE gost28147params\_oid[] =

{0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};

CK\_BYTE value[64] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_GOSTR3410\_PARAMS, gostR3410params\_oid, sizeof(gostR3410params\_oid)},

{CKA\_GOSTR3411\_PARAMS, gostR3411params\_oid, sizeof(gostR3411params\_oid)},

{CKA\_GOST28147\_PARAMS, gost28147params\_oid, sizeof(gost28147params\_oid)},

{CKA\_VALUE, value, sizeof(value)}

};

### GOST R 34.10-2001 private key objects

GOST R 34.10-2001 private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_GOSTR3410**) hold GOST R 34.10-2001 private keys.

The following table defines the GOST R 34.10-2001 private key object attributes, in addition to the common attributes defined for this object class:

*Table 214, GOST R 34.10-2001 Private Key Object Attributes*

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | 32 bytes for private key in little endian order |
| CKA\_GOSTR3410\_PARAMS1,4,6 | Byte array | DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001.  When key is used the domain parameter object of key type CKK\_GOSTR3410 must be specified with the same attribute CKA\_OBJECT\_ID |
| CKA\_GOSTR3411\_PARAMS1,4,6,8 | Byte array | DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94.  When key is used the domain parameter object of key type CKK\_GOSTR3411 must be specified with the same attribute CKA\_OBJECT\_ID |
| CKA\_GOST28147\_PARAMS44,6,8 | Byte array | DER-encoding of the object identifier indicating the data object type of GOST 28147‑89.  When key is used the domain parameter object of key type CKK\_GOST28147 must be specified with the same attribute CKA\_OBJECT\_ID. The attribute value may be omitted |

Refer to [PKCS11-Base] Table 11 for footnotes

Note that when generating an GOST R 34.10-2001 private key, the GOST R 34.10-2001 domain parameters are *not* specified in the key’s template. This is because GOST R 34.10-2001 private keys are only generated as part of an GOST R 34.10-2001 key *pair*, and the GOST R 34.10-2001 domain parameters for the pair are specified in the template for the GOST R 34.10-2001 public key.

The following is a sample template for creating an GOST R 34.10-2001 private key object:

CK\_OBJECT\_CLASS class = CKO\_PRIVATE\_KEY;

CK\_KEY\_TYPE keyType = CKK\_GOSTR3410;

CK\_UTF8CHAR label[] = “A GOST R34.10-2001 private key object”;

CK\_BYTE subject[] = {...};

CK\_BYTE id[] = {123};

CK\_BYTE gostR3410params\_oid[] =

{0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};

CK\_BYTE gostR3411params\_oid[] =

{0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};

CK\_BYTE gost28147params\_oid[] =

{0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};

CK\_BYTE value[32] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SUBJECT, subject, sizeof(subject)},

{CKA\_ID, id, sizeof(id)},

{CKA\_SENSITIVE, &true, sizeof(true)},

{CKA\_SIGN, &true, sizeof(true)},

{CKA\_GOSTR3410\_PARAMS, gostR3410params\_oid, sizeof(gostR3410params\_oid)},

{CKA\_GOSTR3411\_PARAMS, gostR3411params\_oid, sizeof(gostR3411params\_oid)},

{CKA\_GOST28147\_PARAMS, gost28147params\_oid, sizeof(gost28147params\_oid)},

{CKA\_VALUE, value, sizeof(value)}

};

### GOST R 34.10-2001 domain parameter objects

GOST R 34.10-2001 domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS,** key type **CKK\_GOSTR3410**) hold GOST R 34.10‑2001 domain parameters.

The following table defines the GOST R 34.10-2001 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 215, GOST R 34.10-2001 Domain Parameter Object Attributes

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1 | Byte array | DER-encoding of the domain parameters as it was introduced in [4] section 8.4 (type *GostR3410-2001-ParamSetParameters*) |
| CKA\_OBJECT\_ID1 | Byte array | DER-encoding of the object identifier indicating the domain parameters |

Refer to [PKCS11-Base] Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that **CKA\_VALUE** attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.10-2001 domain parameter object:

CK\_OBJECT\_CLASS class = CKO\_DOMAIN\_PARAMETERS;

CK\_KEY\_TYPE keyType = CKK\_GOSTR3410;

CK\_UTF8CHAR label[] = “A GOST R34.10-2001 cryptographic parameters object”;

CK\_BYTE oid[] =

{0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};

CK\_BYTE value[] = {

0x30,0x81,0x90,0x02,0x01,0x07,0x02,0x20,0x5f,0xbf,0xf4,0x98,

0xaa,0x93,0x8c,0xe7,0x39,0xb8,0xe0,0x22,0xfb,0xaf,0xef,0x40,

0x56,0x3f,0x6e,0x6a,0x34,0x72,0xfc,0x2a,0x51,0x4c,0x0c,0xe9,

0xda,0xe2,0x3b,0x7e,0x02,0x21,0x00,0x80,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x04,0x31,0x02,0x21,0x00,0x80,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x50,0xfe,

0x8a,0x18,0x92,0x97,0x61,0x54,0xc5,0x9c,0xfc,0x19,0x3a,0xcc,

0xf5,0xb3,0x02,0x01,0x02,0x02,0x20,0x08,0xe2,0xa8,0xa0,0xe6,

0x51,0x47,0xd4,0xbd,0x63,0x16,0x03,0x0e,0x16,0xd1,0x9c,0x85,

0xc9,0x7f,0x0a,0x9c,0xa2,0x67,0x12,0x2b,0x96,0xab,0xbc,0xea,

0x7e,0x8f,0xc8

};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_OBJECT\_ID, oid, sizeof(oid)},

{CKA\_VALUE, value, sizeof(value)}

};

### GOST R 34.10-2001 mechanism parameters

♦ **CK\_GOSTR3410\_KEY\_WRAP\_PARAMS**

**CK\_GOSTR3410\_KEY\_WRAP\_PARAMS** is a structure that provides the parameters to the **CKM\_GOSTR3410\_KEY\_WRAP** mechanism. It is defined as follows:

typedef struct CK\_GOSTR3410\_KEY\_WRAP\_PARAMS {

CK\_BYTE\_PTR pWrapOID;

CK\_ULONG ulWrapOIDLen;

CK\_BYTE\_PTR pUKM;

CK\_ULONG ulUKMLen;

CK\_OBJECT\_HANDLE hKey;

} CK\_GOSTR3410\_KEY\_WRAP\_PARAMS;

The fields of the structure have the following meanings:

|  |  |  |
| --- | --- | --- |
| *pWrapOID* |  | pointer to a data with DER-encoding of the object identifier indicating the data object type of GOST 28147‑89. If pointer takes NULL\_PTR value in C\_WrapKey operation then parameters are specified in object identifier of attribute CKA\_GOSTR3411\_PARAMS must be used. For C\_UnwrapKey operation the pointer is not used and must take NULL\_PTR value anytime |
| *ulWrapOIDLen* |  | length of data with DER-encoding of the object identifier indicating the data object type of GOST 28147‑89 |
| *pUKM* |  | pointer to a data with UKM. If pointer takes NULL\_PTR value in C\_WrapKey operation then random value of UKM will be used. If pointer takes non-NULL\_PTR value in C\_UnwrapKey operation then the pointer value will be compared with UKM value of wrapped key. If these two values do not match the wrapped key will be rejected |
| *ulUKMLen* |  | length of UKM data. If *pUKM*-pointer is different from NULL\_PTR then equal to 8 |
| *hKey* |  | key handle. Key handle of a sender for C\_WrapKey operation. Key handle of a receiver for C\_UnwrapKey operation. When key handle takes CK\_INVALID\_HANDLE value then an ephemeral (one time) key pair of a sender will be used |

♦ **CK\_GOSTR3410\_DERIVE\_PARAMS**

**CK\_GOSTR3410\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_GOSTR3410\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_GOSTR3410\_DERIVE\_PARAMS {

CK\_EC\_KDF\_TYPE kdf;

CK\_BYTE\_PTR pPublicData;

CK\_ULONG ulPublicDataLen;

CK\_BYTE\_PTR pUKM;

CK\_ULONG ulUKMLen;

} CK\_GOSTR3410\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

|  |  |  |
| --- | --- | --- |
| *kdf* |  | additional key diversification algorithm identifier. Possible values are CKD\_NULL and CKD\_CPDIVERSIFY\_KDF. In case of CKD\_NULL, result of the key derivation function  described in [RFC 4357], section 5.2 is used directly; In case of CKD\_CPDIVERSIFY\_KDF, the resulting key value is additionally processed with algorithm from [RFC 4357], section 6.5. |
| *pPublicData*1 |  | pointer to data with public key of a receiver |
| *ulPublicDataLen* |  | length of data with public key of a receiver (must be 64) |
| *pUKM* |  | pointer to a UKM data |
| *ulUKMLen* |  | length of UKM data in bytes (must be 8) |

1 Public key of a receiver is an octet string of 64 bytes long. The public key octets correspond to the concatenation of X and Y coordinates of a point. Any one of them is 32 bytes long and represented in little endian order.

### GOST R 34.10-2001 key pair generation

The GOST R 34.10‑2001 key pair generation mechanism, denoted **CKM\_GOSTR3410\_KEY\_PAIR\_GEN**, is a key pair generation mechanism for GOST R 34.10‑2001.

This mechanism does not have a parameter.

The mechanism generates GOST R 34.10‑2001 public/private key pairs with particular GOST R 34.10‑2001 domain parameters, as specified in the **CKA\_GOSTR3410\_PARAMS**, **CKA\_GOSTR3411\_PARAMS**, and **CKA\_GOST28147\_PARAMS** attributes of the template for the public key. Note that **CKA\_GOST28147\_PARAMS** attribute may not be present in the template.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_VALUE**, and **CKA\_GOSTR3410\_PARAMS**, **CKA\_GOSTR3411\_PARAMS**, **CKA\_GOST28147\_PARAMS** attributes to the new private key.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

### GOST R 34.10-2001 without hashing

The GOST R 34.10‑2001 without hashing mechanism, denoted **CKM\_GOSTR3410**, is a mechanism for single-part signatures and verification for GOST R 34.10‑2001. (This mechanism corresponds only to the part of GOST R 34.10‑2001 that processes the 32-bytes hash value; it does not compute the hash value.)

This mechanism does not have a parameter.

For the purposes of these mechanisms, a GOST R 34.10‑2001 signature is an octet string of 64 bytes long. The signature octets correspond to the concatenation of the GOST R 34.10‑2001 values *s* and *r’*, both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC 4490] section 3.2, and [RFC 4491] section 2.2.2.

The input for the mechanism is an octet string of 32 bytes long with digest has computed by means of GOST R 34.11‑94 hash algorithm in the context of signed or should be signed message.

Table 216, GOST R 34.10-2001 without hashing: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | CKK\_GOSTR3410 | 32 bytes | 64 bytes |
| C\_Verify1 | CKK\_GOSTR3410 | 32 bytes | 64 bytes |

1 Single-part operations only.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

### GOST R 34.10-2001 with GOST R 34.11-94

The GOST R 34.10‑2001 with GOST R 34.11‑94, denoted **CKM\_GOSTR3410\_WITH\_GOSTR3411**, is a mechanism for signatures and verification for GOST R 34.10‑2001. This mechanism computes the entire GOST R 34.10‑2001 specification, including the hashing with GOST R 34.11‑94 hash algorithm.

As a parameter this mechanism utilizes a DER-encoding of the object identifier indicating GOST R 34.11‑94 data object type. A mechanism parameter may be missed then parameters are specified in object identifier of attribute **CKA\_GOSTR3411\_PARAMS** must be used.

For the purposes of these mechanisms, a GOST R 34.10‑2001 signature is an octet string of 64 bytes long. The signature octets correspond to the concatenation of the GOST R 34.10‑2001 values *s* and *r’*, both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC 4490] section 3.2, and [RFC 4491] section 2.2.2.

The input for the mechanism is signed or should be signed message of any length. Single- and multiple-part signature operations are available.

Table 217, GOST R 34.10-2001 with GOST R 34.11-94: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_GOSTR3410 | Any | 64 bytes |
| C\_Verify | CKK\_GOSTR3410 | Any | 64 bytes |

For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK\_MECHANISM\_INFO structure are not used.

### GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001

GOST R 34.10-2001 keys as a KEK (key encryption keys) for encryption GOST 28147 keys, denoted by **CKM\_GOSTR3410\_KEY\_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on GOST R 34.10-2001. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147‑89. An encryption algorithm from [RFC 4490] (section 5.2) must be used. Encrypted key is a DER-encoded structure of ASN.1 *GostR3410-KeyTransport* type [RFC 4490] section 4.2.

It has a parameter, a **CK\_GOSTR3410\_KEY\_WRAP\_PARAMS** structure defined in section 2.57.5.

For unwrapping (**C\_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as the **CKA\_VALUE** attribute of the new key.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used.

### Common key derivation with assistance of GOST R 34.10-2001 keys

Common key derivation, denoted **CKM\_GOSTR3410\_DERIVE,** is a mechanism for key derivation with assistance of GOST R 34.10‑2001 private and public keys. The key of the mechanism must be of object class **CKO\_DOMAIN\_PARAMETERS** andkey type **CKK\_GOSTR3410**. An algorithm for key derivation from [RFC 4357] (section 5.2) must be used.

The mechanism contributes the result as the **CKA\_VALUE** attribute of the new private key. All other attributes must be specified in a template for creating private key object.

## ChaCha20

ChaCha20 is a secret-key stream cipher described in **[CHACHA].**

*Table 218, ChaCha20 Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_CHACHA20\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_CHACHA20 | ✓ |  |  |  |  | ✓ |  |

### Definitions

This section defines the key type “CKK\_CHACHA20” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_CHACHA20\_KEY\_GEN

CKM\_CHACHA20

### ChaCha20 secret key objects

ChaCha20 secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_CHACHA) hold ChaCha20 keys. The following table defines the ChaCha20 secret key object attributes, in addition to the common attributes defined for this object class:

Table 219, ChaCha20 Secret Key Object

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key length is fixed at 256 bits. Bit length restricted to a byte array. |
| CKA\_VALUE\_LEN2,3 | CK\_ULONG | Length in bytes of key value |

The following is a sample template for creating a ChaCha20 secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_CHACHA20;

CK\_UTF8CHAR label[] = “A ChaCha20 secret key object”;

CK\_BYTE value[32] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

CKA\_CHECK\_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the SHA-1 hash of the ChaCha20 secret key object’s CKA\_VALUE attribute.

### ChaCha20 mechanism parameters

1. CK\_CHACHA20\_PARAMS; CK\_CHACHA20\_PARAMS\_PTR

**CK\_CHACHA20\_PARAMS** provides the parameters to the **CKM\_CHACHA20** mechanism. It is defined as follows:

typedef struct CK\_CHACHA20\_PARAMS {

CK\_BYTE\_PTR pBlockCounter;

CK\_ULONG blockCounterBits;

CK\_BYTE\_PTR pNonce;

CK\_ULONG ulNonceBits;

} CK\_CHACHA20\_PARAMS;

The fields of the structure have the following meanings:

pBlockCounter pointer to block counter

ulblockCounterBits length of block counter in bits (can be either 32 or 64)

pNonce nonce (This should be never re-used with the same key.)

ulNonceBits length of nonce in bits (is 64 for original, 96 for IETF and 192 for xchacha20 variant)

The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption) it is necessary to address these blocks in random order, thus this counter is exposed here.

### ChaCha20 key generation

The ChaCha20 key generation mechanism, denoted **CKM\_CHACHA20\_KEY\_GEN**, is a key generation mechanism for ChaCha20.

It does not have a parameter.

The mechanism generates ChaCha20 keys of 256 bits.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of key sizes in bytes. As a practical matter, the key size for ChaCha20 is fixed at 256 bits.

### ChaCha20 mechanism

ChaCha20, denoted **CKM\_CHACHA20**, is a mechanism for single and multiple-part encryption and decryption based on the ChaCha20 stream cipher. It comes in 3 variants, which only differ in the size and handling of their nonces, affecting the safety of using random nonces and the maximum size that can be encrypted safely.

Chacha20 has a parameter, **CK\_CHACHA20\_PARAMS**, which indicates the nonce and initial block counter value.

Constraints on key types and the length of input and output data are summarized in the following table:

Table , ChaCha20: Key and Data Length

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| C\_Encrypt | ChaCha20 | Any / only up to 256 GB in case of IETF variant | Same as input length | No final part |
| C\_Decrypt | ChaCha20 | Any / only up to 256 GB in case of IETF variant | Same as input length | No final part |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of ChaCha20 key sizes, in bits.

Table , ChaCha20: Nonce and block counter lengths

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Variant** | **Nonce** | **Block counter** | **Maximum message** | **Nonce generation** |
| original | 64 bit | 64 bit | Virtually unlimited | 1st msg: nonce0=random  nth msg: noncen-1++ |
| IETF | 96 bit | 32 bit | Max ~256 GB | 1st msg: nonce0=random  nth msg: noncen-1++ |
| XChaCha20 | 192 bit | 64 bit | Virtually unlimited | Each nonce can be randomly generated. |

Nonces must not ever be reused with the same key. However due to the birthday paradox the first two variants cannot guarantee that randomly generated nonces are never repeating. Thus the recommended way to handle this is to generate the first nonce randomly, then increase this for follow-up messages. Only the last (XChaCha20) has large enough nonces so that it is virtually impossible to trigger with randomly generated nonces the birthday paradox.

## Salsa20

Salsa20 is a secret-key stream cipher described in **[SALSA].**

*Table 222, Salsa20 Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_SALSA20\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_SALSA20 | ✓ |  |  |  |  | ✓ |  |

### Definitions

This section defines the key type “CKK\_SALSA20” and “CKK\_SALSA20” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_SALSA20\_KEY\_GEN

CKM\_SALSA20

### Salsa20 secret key objects

Salsa20 secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_SALSA) hold Salsa20 keys. The following table defines the Salsa20 secret key object attributes, in addition to the common attributes defined for this object class:

Table 223, ChaCha20 Secret Key Object

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key length is fixed at 256 bits. Bit length restricted to a byte array. |
| CKA\_VALUE\_LEN2,3 | CK\_ULONG | Length in bytes of key value |

The following is a sample template for creating a Salsa20 secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_SALSA20;

CK\_UTF8CHAR label[] = “A Salsa20 secret key object”;

CK\_BYTE value[32] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_ENCRYPT, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

CKA\_CHECK\_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the SHA-1 hash of the ChaCha20 secret key object’s CKA\_VALUE attribute.

### Salsa20 mechanism parameters

1. CK\_SALSA20\_PARAMS; CK\_SALSA\_PARAMS\_PTR

**CK\_SALSA20\_PARAMS** provides the parameters to the **CKM\_SALSA20** mechanism. It is defined as follows:

typedef struct CK\_SALSA20\_PARAMS {

CK\_BYTE\_PTR pBlockCounter;

CK\_BYTE\_PTR pNonce;

CK\_ULONG ulNonceBits;

} CK\_SALSA20\_PARAMS;

The fields of the structure have the following meanings:

pBlockCounter pointer to block counter (64 bits)

pNonce nonce

ulNonceBits size of the nonce in bits (64 for classic and 192 for XSalsa20)

The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption) it is necessary to address these blocks in random order, thus this counter is exposed here.

### Salsa20 key generation

The Salsa20 key generation mechanism, denoted **CKM\_SALSA20\_KEY\_GEN**, is a key generation mechanism for Salsa20.

It does not have a parameter.

The mechanism generates Salsa20 keys of 256 bits.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, and **CKA\_VALUE** attributes to the new key. Other attributes supported by the key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of key sizes in bytes. As a practical matter, the key size for Salsa20 is fixed at 256 bits.

### Salsa20 mechanism

Salsa20, denoted **CKM\_SALSA20**, is a mechanism for single and multiple-part encryption and decryption based on the Salsa20 stream cipher. Salsa20 comes in two variants which only differ in the size and handling of their nonces, affecting the safety of using random nonces.

Salsa20 has a parameter, **CK\_SALSA20\_PARAMS**, which indicates the nonce and initial block counter value.

Constraints on key types and the length of input and output data are summarized in the following table:

Table , Salsa20: Key and Data Length

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Function** | **Key type** | **Input length** | **Output length** | **Comments** |
| C\_Encrypt | Salsa20 | Any | Same as input length | No final part |
| C\_Decrypt | Salsa20 | Any | Same as input length | No final part |

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure specify the supported range of ChaCha20 key sizes, in bits.

Table , Salsa20: Nonce sizes

|  |  |  |  |
| --- | --- | --- | --- |
| **Variant** | **Nonce** | **Maximum message** | **Nonce generation** |
| original | 64 bit | Virtually unlimited | 1st msg: nonce0=random  nth msg: noncen-1++ |
| XSalsa20 | 192 bit | Virtually unlimited | Each nonce can be randomly generated. |

Nonces must not ever be reused with the same key. However due to the birthday paradox the original variant cannot guarantee that randomly generated nonces are never repeating. Thus the recommended way to handle this is to generate the first nonce randomly, then increase this for follow-up messages. Only the XSalsa20 has large enough nonces so that it is virtually impossible to trigger with randomly generated nonces the birthday paradox.

## Poly1305

Poly1305 is a message authentication code designed by D.J Bernsterin **[POLY1305].** Poly1305 takes a 256 bit key and a message and produces a 128 bit tag that is used to verify the message.

*Table 226, Poly1305 Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_POLY1305\_KEY\_GEN |  |  |  |  | ✓ |  |  |
| CKM\_POLY1305 |  | ✓ |  |  |  |  |  |

### Definitions

This section defines the key type “CKK\_POLY1305” for type CK\_KEY\_TYPE as used in the CKA\_KEY\_TYPE attribute of key objects.

Mechanisms:

CKM\_POLY1305\_KEY\_GEN

CKM\_POLY1305\_MAC

### Poly1305 secret key objects

Poly1305 secret key objects (object class CKO\_SECRET\_KEY, key type CKK\_POLY1305) hold Poly1305 keys. The following table defines the Poly1305 secret key object attributes, in addition to the common attributes defined for this object class:

Table 227, Poly1305 Secret Key Object

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key length is fixed at 256 bits. Bit length restricted to a byte array. |
| CKA\_VALUE\_LEN2,3 | CK\_ULONG | Length in bytes of key value |

The following is a sample template for creating a Poly1305 secret key object:

CK\_OBJECT\_CLASS class = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_POLY1305;

CK\_UTF8CHAR label[] = “A Poly1305 secret key object”;

CK\_BYTE value[32] = {...};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

{CKA\_CLASS, &class, sizeof(class)},

{CKA\_KEY\_TYPE, &keyType, sizeof(keyType)},

{CKA\_TOKEN, &true, sizeof(true)},

{CKA\_LABEL, label, sizeof(label)-1},

{CKA\_SIGN, &true, sizeof(true)},

{CKA\_VALUE, value, sizeof(value)}

};

### Poly1305 mechanism

Poly1305, denoted **CKM\_POLY1305**, is a mechanism for producing an output tag based on a 256 bit key and arbitrary length input.

It has no parameters.

Signatures (MACs) produced by this mechanism will be fixed at 128 bits in size.

Table , Poly1305: Key and Data Length

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Key type** | **Data length** | **Signature Length** |
| C\_Sign | Poly1305 | Any | 128 bits |
| C\_Verify | Poly1305 | Any | 128 bits |

## Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption

The stream ciphers Salsa20 and ChaCha20 are normally used in conjunction with the Poly1305 authenticator, in such a construction they also provide Authenticated Encryption with Associated Data (AEAD). This section defines the combined mechanisms and their usage in an AEAD setting.

### Definitions

Mechanisms:

CKM\_CHACHA20\_POLY1305

CKM\_SALSA20\_POLY1305

### Usage

Generic ChaCha20, Salsa20, Poly1305 modes are described in [CHACHA], [SALSA] and [POLY1305]. To set up for ChaCha20/Poly1305 or Salsa20/Poly1305 use the following process. ChaCha20/Poly1305 and Salsa20/Poly1305 both use CK\_SALSA20\_CHACHA20\_POLY1350\_PARAM for Encrypt, Decrypt and CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAM for MessageEncrypt, and MessageDecrypt.

Encrypt:

* Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
* Set the Nonce data *pNonce* in the parameter block.
* Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD m*ay be NULL if *ulAADLen* is 0.
* Call C\_EncryptInit() for **CKM\_CHACHA20\_POLY1305** or **CKM\_SALSA20\_POLY1305** mechanism with parameters and key *K*.
* Call C\_Encrypt(), or C\_EncryptUpdate()\*[[10]](#footnote-10) C\_EncryptFinal(), for the plaintext obtaining ciphertext and authentication tag output.

Decrypt:

* Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
* Set the Nonce data *pNonce* in the parameter block.
* Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD m*ay be NULL if ulAADLen is 0.
* Call C\_DecryptInit() for **CKM\_CHACHA20\_POLY1305** or **CKM\_SALSA20\_POLY1305** mechanism with parameters and key *K*.
* Call C\_Decrypt(), or C\_DecryptUpdate()\*1 C\_DecryptFinal(), for the ciphertext, including the appended tag, obtaining plaintext output. Note: since **CKM\_CHACHA20\_POLY1305** and **CKM\_SALSA20\_POLY1305** are AEAD ciphers, no data should be returned until C\_Decrypt() or C\_DecryptFinal().

MessageEncrypt::

* Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
* Set the Nonce data *pNonce* in the parameter block.
* Set pTag to hold the tag data returned from C\_EncryptMessage() or the final C\_EncryptMessageNext().
* Call C\_MessageEncryptInit() for **CKM\_CHACHA20\_POLY1305** or **CKM\_SALSA20\_POLY1305** mechanism with key *K*.
* Call C\_EncryptMessage(), or C\_EncryptMessageBegin followed by C\_EncryptMessageNext()\*[[11]](#footnote-11). The mechanism parameter is passed to all three of these functions.
* Call C\_MessageEncryptFinal() to close the message decryption.

MessageDecrypt:

* Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
* Set the Nonce data *pNonce* in the parameter block.
* Set the tag data pTag in the parameter block before C\_DecryptMessage or the final C\_DecryptMessageNext()
* Call C\_MessageDecryptInit() for **CKM\_CHACHA20\_POLY1305** or **CKM\_SALSA20\_POLY1305** mechanism with key *K*.
* Call C\_DecryptMessage(), or C\_DecryptMessageBegin followed by C\_DecryptMessageNext()\*[[12]](#footnote-12). The mechanism parameter is passed to all three of these functions.
* Call C\_MessageDecryptFinal() to close the message decryption

*ulNonceLen* is the length of the nonce in bits.

In Encrypt and Decrypt the tag is appended to the cipher text. In MessageEncrypt the tag is returned in the pTag filed of CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS. In MesssageDecrypt the tag is provided by the pTag field of CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS. The application must provide 16 bytes of space for the tag.

The key type for *K* must be compatible with **CKM\_CHACHA20** or **CKM\_SALSA20** respectively and the C\_EncryptInit/C\_DecryptInit calls shall behave, with respect to *K*, as if they were called directly with **CKM\_CHACHA20** or **CKM\_SALSA20**, *K* and NULL parameters.

Unlike the atomic Salsa20/ChaCha20 mechanism the AEAD mechanism based on them does not expose the block counter, as the AEAD construction is based on a message metaphor in which random access is not needed.

### ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters

1. CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS; CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS\_PTR

**CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS** is a structure that provides the parameters to the **CKM\_CHACHA20\_POLY1305** and **CKM\_SALSA20\_POLY1305** mechanisms. It is defined as follows:

typedef struct CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS {

CK\_BYTE\_PTR pNonce;

CK\_ULONG ulNonceLen;

CK\_BYTE\_PTR pAAD;

CK\_ULONG ulAADLen;

} CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS;

The fields of the structure have the following meanings:

pNonce nonce (This should be never re-used with the same key.)

ulNonceLen length of nonce in bits (is 64 for original, 96 for IETF (only for chacha20) and 192 for xchacha20/xsalsa20 variant)

pAAD pointer to additional authentication data. This data is authenticated but not encrypted.

ulAADLen length of pAAD in bytes.

**CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS\_PTR** is a pointer to a **CK\_SALSA20\_CHACHA20\_POLY1305\_PARAMS**.

1. CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS; CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS\_PTR

CK\_CHACHA20POLY1305\_PARAMS is a structure that provides the parameters to the CKM\_ CHACHA20\_POLY1305 mechanism. It is defined as follows:

typedef struct CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS {

CK\_BYTE\_PTR pNonce;

CK\_ULONG ulNonceLen;

CK\_BYTE\_PTR pTag;

} CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS;

The fields of the structure have the following meanings:

pNonce pointer to nonce

ulNonceLen length of nonce in bits. The length of the influences which variant of the ChaCha20 will be used (64 original, 96 IETF(only for ChaCha20), 192 XChaCha20/XSalsa20)

pTag location of the authentication tag which is returned on MessageEncrypt, and provided on MessageDecrypt.

**CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS\_PTR** is a pointer to a **CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS**.

## HKDF Mechanisms

Details for HKDF key derivation mechanisms can be found in [RFC 5869].

*Table 229, HKDF Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_HKDF\_DERIVE |  |  |  |  |  |  |  |
| CKM\_HKDF\_DATA |  |  |  |  |  |  |  |
| CKM\_HKDF\_KEY\_GEN |  |  |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_HKDF\_DERIVE

CKM\_HKDF\_DATA

CKM\_HKDF\_KEY\_GEN

Key Types:

CKK\_HKDF

### HKDF mechanism parameters

1. CK\_HKDF\_PARAMS; CK\_HKDF\_PARAMS\_PTR

**CK\_HKDF\_PARAMS** is a structure that provides the parameters to the **CKM\_HKDF\_DERIVE** and **CKM\_HKDF\_DATA** mechanisms. It is defined as follows:

typedef struct CK\_HKDF\_PARAMS {

CK\_BOOL bExtract;

CK\_BOOL bExpand;

CK\_MECHANISM\_TYPE prfHashMechanism;

CK\_ULONG ulSaltType;

CK\_BYTE\_PTR pSalt;

CK\_ULONG ulSaltLen;

CK\_HANDLE hSaltKey;

CK\_BYTE\_PTR pInfo;

CK\_ULONG ulInfoLen;

} CK\_HKDF\_PARAMS;

The fields of the structure have the following meanings:

bExtract execute the extract portion of HKDF.

bExpand execute the expand portion of HKDF.

prfHashMechanism base hash used for the HMAC in the underlying HKDF operation.

ulSaltType specifies how the salt for the extract portion of the KDF is supplied.

CKF\_HKDF\_SALT\_NULL no salt is supplied.

CKF\_HKDF\_SALT\_DATA salt is supplied as a data in pSalt with length ulSaltLen.

CKF\_HKDF\_SALT\_KEY salt is supplied as a key in hSaltKey.

pSalt pointer to the salt.

ulSaltLen length of the salt pointed to in pSalt.

hSaltKey object handle to the salt key.

pInfo info string for the expand stage.

ulInfoLen length of the info string for the expand stage.

**CK\_HKDF\_PARAMS\_PTR** is a pointer to a **CK\_HKDF\_PARAMS**.

### HKDF derive

HKDF derivation implements the HKDF as specified in RFC 5869. The two booleans bExtract and bExpand control whether the extract section of the HKDF or the expand section of the HKDF is in use.

It has a parameter, a **CK\_HKDF\_PARAMS** structure, which allows for the passing of the salt and or the expansion info. The structure contains the bools *bExtract* and *bExpand* which control whether the extract or expand portions of the HKDF is to be used. This structure is defined in Section 2.62.2.

The input key must be of type **CKK\_HKDF** or **CKK\_GENERIC\_SECRET** and the length must be the size of the underlying hash function specified in *prfHashMechanism*. The exception is a data object which has the same size as the underlying hash function, and which may be supplied as an input key. In this case bExtract should be true and non-null salt should be supplied.

Either *bExtract* or *bExpand* must be set to true. If they are both set to true, input key is first extracted then expanded. The salt is used in the extraction stage. If bExtract is set to true and no salt is given, a ‘zero’ salt (salt whose length is the same as the underlying hash and values all set to zero) is used as specified by the RFC. If bExpand is set to true, **CKA\_VALUE\_LEN** should be set to the desired key length. If it is false CKA\_VALUE\_LEN may be set to the length of the hash, but that is not necessary as the mechanism will supply this value. The salt should be ignored if *bExtract* is false. The *pInfo* should be ignored if *bExpand* is set to false.

The mechanism also contributes the **CKA\_CLASS**, and **CKA\_VALUE** attributes to the new key. Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C\_DeriveKey** call may indicate that the object class is **CKO\_SECRET\_KEY**. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

This mechanism has the following rules about key sensitivity and extractability:

* The **CKA\_SENSITIVE** and **CKA\_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK\_TRUE or CK\_FALSE. If omitted, these attributes each take on some default value.
* If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, then the derived key will as well. If the base key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_TRUE, then the derived key has its **CKA\_ALWAYS\_SENSITIVE** attribute set to the same value as its **CKA\_SENSITIVE** attribute.
* Similarly, if the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE, then the derived key will, too. If the base key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_TRUE, then the derived key has its **CKA\_NEVER\_EXTRACTABLE** attribute set to the *opposite* value from its **CKA\_EXTRACTABLE** attribute.

### HKDF Data

HKDF Data derive mechanism, denoted **CKM\_HKDF\_DATA**, is identical to HKDF Derive except the output is a **CKO\_DATA** object whose value is the result to the derive operation. Some tokens may restrict what data may be successfully derived based on the pInfo portion of the CK\_HKDF\_PARAMS. All tokens must minimally support *bExtract* set to true and *bInfo* values which start with the value “tls1.3 vi”. Future additional required combinations may be specified in the profile document and applications could then query the appropriate profile before depending on the mechanism.

### HKDF Key gen

HKDF key gen, denoted CKM\_HKDF\_KEYGEN generates a new random HKDF key. CKA\_VALUE\_LENGTH must be set in the template.

## NULL Mechanism

**CKM\_NULL** is a mechanism used to implement the trivial pass-through function.

*Table 230, CKM\_NULL Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_NULL |  |  |  |  |  |  |  |
| 1SR = SignRecover, VR = VerifyRecover | | | | | | | |

### Definitions

Mechanisms:

CKM\_NULL

### CKM\_NULL mechanism parameters

CKM\_NULL does not have a parameter.

When used for encrypting / decrypting data, the input data is copied unchanged to the output data.

When used for signing, the input data is copied to the signature. When used for signature verification, it compares the input data and the signature, and returns CKR\_OK (indicating that both are identical) or CKR\_SIGNATURE\_INVALID.

When used for digesting data, the input data is copied to the message digest.

When used for wrapping a private or secret key object, the wrapped key will be identical to the key to be wrapped. When used for unwrapping, a new object with the same value as the wrapped key will be created.

When used for deriving a key, the derived key has the same value as the base key.

# PKCS #11 Implementation Conformance

An implementation is a conforming implementation if it meets the conditions specified in one or more server profiles specified in **[PKCS11-Prof].**

If a PKCS #11 implementation claims support for a particular profile, then the implementation SHALL conform to all normative statements within the clauses specified for that profile and for any subclauses to each of those clauses.

1. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

List needs to be pasted in here

Gil Abel, Athena Smartcard Solutions, Inc.

Warren Armstrong, QuintessenceLabs

Jeff Bartell, Semper Foris Solutions LLC

Peter Bartok, Venafi, Inc.

Anthony Berglas, Cryptsoft

Joseph Brand, Semper Fortis Solutions LLC

Kelley Burgin, National Security Agency

Robert Burns, Thales e-Security

Wan-Teh Chang, Google Inc.

Hai-May Chao, Oracle

Janice Cheng, Vormetric, Inc.

Sangrae Cho, Electronics and Telecommunications Research Institute (ETRI)

Doron Cohen, SafeNet, Inc.

Fadi Cotran, Futurex

Tony Cox, Cryptsoft

Christopher Duane, EMC

Chris Dunn, SafeNet, Inc.

Valerie Fenwick, Oracle

Terry Fletcher, SafeNet, Inc.

Susan Gleeson, Oracle

Sven Gossel, Charismathics

John Green, QuintessenceLabs

Robert Griffin, EMC

Paul Grojean, Individual

Peter Gutmann, Individual

Dennis E. Hamilton, Individual

Thomas Hardjono, M.I.T.

Tim Hudson, Cryptsoft

Gershon Janssen, Individual

Seunghun Jin, Electronics and Telecommunications Research Institute (ETRI)

Wang Jingman, Feitan Technologies

Andrey Jivsov, Symantec Corp.

Mark Joseph, P6R

Stefan Kaesar, Infineon Technologies

Greg Kazmierczak, Wave Systems Corp.

Mark Knight, Thales e-Security

Darren Krahn, Google Inc.

Alex Krasnov, Infineon Technologies AG

Dina Kurktchi-Nimeh, Oracle

Mark Lambiase, SecureAuth Corporation

Lawrence Lee, GoTrust Technology Inc.

John Leiseboer, QuintessenceLabs

Sean Leon, Infineon Technologies

Geoffrey Li, Infineon Technologies

Howie Liu, Infineon Technologies

Hal Lockhart, Oracle

Robert Lockhart, Thales e-Security

Dale Moberg, Axway Software

Darren Moffat, Oracle

Valery Osheter, SafeNet, Inc.

Sean Parkinson, EMC

Rob Philpott, EMC

Mark Powers, Oracle

Ajai Puri, SafeNet, Inc.

Robert Relyea, Red Hat

Saikat Saha, Oracle

Subhash Sankuratripati, NetApp

Anthony Scarpino, Oracle

Johann Schoetz, Infineon Technologies AG

Rayees Shamsuddin, Wave Systems Corp.

Radhika Siravara, Oracle

Brian Smith, Mozilla Corporation

David Smith, Venafi, Inc.

Ryan Smith, Futurex

Jerry Smith, US Department of Defense (DoD)

Oscar So, Oracle

Graham Steel, Cryptosense

Michael Stevens, QuintessenceLabs

Michael StJohns, Individual

Jim Susoy, P6R

Sander Temme, Thales e-Security

Kiran Thota, VMware, Inc.

Walter-John Turnes, Gemini Security Solutions, Inc.

Stef Walter, Red Hat

James Wang, Vormetric

Jeff Webb, Dell

Peng Yu, Feitian Technologies

Magda Zdunkiewicz, Cryptsoft

Chris Zimman, Individual

1. Manifest Constants

The following definitions can be found in the appropriate computer language files referenced on the title page of this specification. Also, refer to **[PKCS11\_BASE]** and **[PKCS11\_HIST]** for additional definitions.

/\*

\* Copyright © Oasis Open 2019. All right reserved.

\* OASIS trademark, IPR and other policies apply.

\* <https://www.oasis-open.org/policies-guidelines/ipr>

\*/

## B.1 Object classes

#define CKO\_DATA 0x00000000

#define CKO\_CERTIFICATE 0x00000001

#define CKO\_PUBLIC\_KEY 0x00000002

#define CKO\_PRIVATE\_KEY 0x00000003

#define CKO\_SECRET\_KEY 0x00000004

#define CKO\_HW\_FEATURE 0x00000005

#define CKO\_DOMAIN\_PARAMETERS 0x00000006

#define CKO\_MECHANISM 0x00000007

#define CKO\_OTP\_KEY 0x00000008

#define CKO\_PROFILE 0x00000009

#define CKO\_VENDOR\_DEFINED 0x80000000

## B.2 Key types

#define CKK\_RSA 0x00000000

#define CKK\_DSA 0x00000001

#define CKK\_DH 0x00000002

#define CKK\_EC 0x00000003

#define CKK\_X9\_42\_DH 0x00000004

#define CKK\_KEA 0x00000005

#define CKK\_GENERIC\_SECRET 0x00000010

#define CKK\_RC2 0x00000011

#define CKK\_RC4 0x00000012

#define CKK\_DES 0x00000013

#define CKK\_DES2 0x00000014

#define CKK\_DES3 0x00000015

#define CKK\_CAST 0x00000016

#define CKK\_CAST3 0x00000017

#define CKK\_CAST128 0x00000018

#define CKK\_RC5 0x00000019

#define CKK\_IDEA 0x0000001A

#define CKK\_SKIPJACK 0x0000001B

#define CKK\_BATON 0x0000001C

#define CKK\_JUNIPER 0x0000001D

#define CKK\_CDMF 0x0000001E

#define CKK\_AES 0x0000001F

#define CKK\_BLOWFISH 0x00000020

#define CKK\_TWOFISH 0x00000021

#define CKK\_SECURID 0x00000022

#define CKK\_HOTP 0x00000023

#define CKK\_ACTI 0x00000024

#define CKK\_CAMELLIA 0x00000025

#define CKK\_ARIA 0x00000026

#define CKK\_MD5\_HMAC 0x00000027

#define CKK\_SHA\_1\_HMAC 0x00000028

#define CKK\_RIPEMD128\_HMAC 0x00000029

#define CKK\_RIPEMD160\_HMAC 0x0000002A

#define CKK\_SHA256\_HMAC 0x0000002B

#define CKK\_SHA384\_HMAC 0x0000002C

#define CKK\_SHA512\_HMAC 0x0000002D

#define CKK\_SHA224\_HMAC 0x0000002E

#define CKK\_SEED 0x0000002F

#define CKK\_GOSTR3410 0x00000030

#define CKK\_GOSTR3411 0x00000031

#define CKK\_GOST28147 0x00000032

#define CKK\_CHACHA20 0x00000033

#define CKK\_POLY1305 0x00000034

#define CKK\_AES\_XTS 0x00000035

#define CKK\_SHA3\_224\_HMAC 0x00000036

#define CKK\_SHA3\_256\_HMAC 0x00000037

#define CKK\_SHA3\_384\_HMAC 0x00000038

#define CKK\_SHA3\_512\_HMAC 0x00000039

#define CKK\_BLAKE2B\_160\_HMAC 0x0000003a

#define CKK\_BLAKE2B\_256\_HMAC 0x0000003b

#define CKK\_BLAKE2B\_384\_HMAC 0x0000003c

#define CKK\_BLAKE2B\_512\_HMAC 0x0000003d

#define CKK\_SALSA20 0x0000003e

#define CKK\_X2RATCHET 0x0000003f

#define CKK\_EC\_EDWARDS 0x00000040

#define CKK\_EC\_MONTGOMERY 0x00000041

#define CKK\_HKDF 0x00000042

#define CKK\_VENDOR\_DEFINED 0x80000000

## B.3 Key derivation functions

#define CKD\_NULL 0x00000001

#define CKD\_SHA1\_KDF 0x00000002

#define CKD\_SHA1\_KDF\_ASN1 0x00000003

#define CKD\_SHA1\_KDF\_CONCATENATE 0x00000004

#define CKD\_SHA224\_KDF 0x00000005

#define CKD\_SHA256\_KDF 0x00000006

#define CKD\_SHA384\_KDF 0x00000007

#define CKD\_SHA512\_KDF 0x00000008

#define CKD\_CPDIVERSIFY\_KDF 0x00000009

#define CKD\_SHA3\_224\_KDF 0x0000000A

#define CKD\_SHA3\_256\_KDF 0x0000000B

#define CKD\_SHA3\_384\_KDF 0x0000000C

#define CKD\_SHA3\_512\_KDF 0x0000000D

#define CKD\_SHA1\_KDF\_SP800 0x0000000E

#define CKD\_SHA224\_KDF\_SP800 0x0000000F

#define CKD\_SHA256\_KDF\_SP800 0x00000010

#define CKD\_SHA384\_KDF\_SP800 0x00000011

#define CKD\_SHA512\_KDF\_SP800 0x00000012

#define CKD\_SHA3\_224\_KDF\_SP800 0x00000013

#define CKD\_SHA3\_256\_KDF\_SP800 0x00000014

#define CKD\_SHA3\_384\_KDF\_SP800 0x00000015

#define CKD\_SHA3\_512\_KDF\_SP800 0x00000016

#define CKD\_BLAKE2B\_160\_KDF 0x00000017

#define CKD\_BLAKE2B\_256\_KDF 0x00000018

#define CKD\_BLAKE2B\_384\_KDF 0x00000019

#define CKD\_BLAKE2B\_512\_KDF 0x0000001A

## B.4 Mechanisms

#define CKM\_RSA\_PKCS\_KEY\_PAIR\_GEN 0x00000000

#define CKM\_RSA\_PKCS 0x00000001

#define CKM\_RSA\_9796 0x00000002

#define CKM\_RSA\_X\_509 0x00000003

#define CKM\_MD2\_RSA\_PKCS 0x00000004

#define CKM\_MD5\_RSA\_PKCS 0x00000005

#define CKM\_SHA1\_RSA\_PKCS 0x00000006

#define CKM\_RIPEMD128\_RSA\_PKCS 0x00000007

#define CKM\_RIPEMD160\_RSA\_PKCS 0x00000008

#define CKM\_RSA\_PKCS\_OAEP 0x00000009

#define CKM\_RSA\_X9\_31\_KEY\_PAIR\_GEN 0x0000000A

#define CKM\_RSA\_X9\_31 0x0000000B

#define CKM\_SHA1\_RSA\_X9\_31 0x0000000C

#define CKM\_RSA\_PKCS\_PSS 0x0000000D

#define CKM\_SHA1\_RSA\_PKCS\_PSS 0x0000000E

#define CKM\_DSA\_KEY\_PAIR\_GEN 0x00000010

#define CKM\_DSA 0x00000011

#define CKM\_DSA\_SHA1 0x00000012

#define CKM\_DSA\_FIPS\_G\_GEN 0x00000013

#define CKM\_DSA\_SHA224 0x00000014

#define CKM\_DSA\_SHA256 0x00000015

#define CKM\_DSA\_SHA384 0x00000016

#define CKM\_DSA\_SHA512 0x00000017

#define CKM\_DSA\_SHA3\_224 0x00000018

#define CKM\_DSA\_SHA3\_256 0x00000019

#define CKM\_DSA\_SHA3\_384 0x0000001A

#define CKM\_DSA\_SHA3\_512 0x0000001B

#define CKM\_DH\_PKCS\_KEY\_PAIR\_GEN 0x00000020

#define CKM\_DH\_PKCS\_DERIVE 0x00000021

#define CKM\_X9\_42\_DH\_KEY\_PAIR\_GEN 0x00000030

#define CKM\_X9\_42\_DH\_DERIVE 0x00000031

#define CKM\_X9\_42\_DH\_HYBRID\_DERIVE 0x00000032

#define CKM\_X9\_42\_MQV\_DERIVE 0x00000033

#define CKM\_SHA256\_RSA\_PKCS 0x00000040

#define CKM\_SHA384\_RSA\_PKCS 0x00000041

#define CKM\_SHA512\_RSA\_PKCS 0x00000042

#define CKM\_SHA256\_RSA\_PKCS\_PSS 0x00000043

#define CKM\_SHA384\_RSA\_PKCS\_PSS 0x00000044

#define CKM\_SHA512\_RSA\_PKCS\_PSS 0x00000045

#define CKM\_SHA224\_RSA\_PKCS 0x00000046

#define CKM\_SHA224\_RSA\_PKCS\_PSS 0x00000047

#define CKM\_SHA512\_224 0x00000048

#define CKM\_SHA512\_224\_HMAC 0x00000049

#define CKM\_SHA512\_224\_HMAC\_GENERAL 0x0000004A

#define CKM\_SHA512\_224\_KEY\_DERIVATION 0x0000004B

#define CKM\_SHA512\_256 0x0000004C

#define CKM\_SHA512\_256\_HMAC 0x0000004D

#define CKM\_SHA512\_256\_HMAC\_GENERAL 0x0000004E

#define CKM\_SHA512\_256\_KEY\_DERIVATION 0x0000004F

#define CKM\_SHA512\_T 0x00000050

#define CKM\_SHA512\_T\_HMAC 0x00000051

#define CKM\_SHA512\_T\_HMAC\_GENERAL 0x00000052

#define CKM\_SHA512\_T\_KEY\_DERIVATION 0x00000053

#define CKM\_SHA3\_256\_RSA\_PKCS 0x00000060

#define CKM\_SHA3\_384\_RSA\_PKCS 0x00000061

#define CKM\_SHA3\_512\_RSA\_PKCS 0x00000062

#define CKM\_SHA3\_256\_RSA\_PKCS\_PSS 0x00000063

#define CKM\_SHA3\_384\_RSA\_PKCS\_PSS 0x00000064

#define CKM\_SHA3\_512\_RSA\_PKCS\_PSS 0x00000065

#define CKM\_SHA3\_224\_RSA\_PKCS 0x00000066

#define CKM\_SHA3\_224\_RSA\_PKCS\_PSS 0x00000067

#define CKM\_RC2\_KEY\_GEN 0x00000100

#define CKM\_RC2\_ECB 0x00000101

#define CKM\_RC2\_CBC 0x00000102

#define CKM\_RC2\_MAC 0x00000103

#define CKM\_RC2\_MAC\_GENERAL 0x00000104

#define CKM\_RC2\_CBC\_PAD 0x00000105

#define CKM\_RC4\_KEY\_GEN 0x00000110

#define CKM\_RC4 0x00000111

#define CKM\_DES\_KEY\_GEN 0x00000120

#define CKM\_DES\_ECB 0x00000121

#define CKM\_DES\_CBC 0x00000122

#define CKM\_DES\_MAC 0x00000123

#define CKM\_DES\_MAC\_GENERAL 0x00000124

#define CKM\_DES\_CBC\_PAD 0x00000125

#define CKM\_DES2\_KEY\_GEN 0x00000130

#define CKM\_DES3\_KEY\_GEN 0x00000131

#define CKM\_DES3\_ECB 0x00000132

#define CKM\_DES3\_CBC 0x00000133

#define CKM\_DES3\_MAC 0x00000134

#define CKM\_DES3\_MAC\_GENERAL 0x00000135

#define CKM\_DES3\_CBC\_PAD 0x00000136

#define CKM\_DES3\_CMAC\_GENERAL 0x00000137

#define CKM\_DES3\_CMAC 0x00000138

#define CKM\_CDMF\_KEY\_GEN 0x00000140

#define CKM\_CDMF\_ECB 0x00000141

#define CKM\_CDMF\_CBC 0x00000142

#define CKM\_CDMF\_MAC 0x00000143

#define CKM\_CDMF\_MAC\_GENERAL 0x00000144

#define CKM\_CDMF\_CBC\_PAD 0x00000145

#define CKM\_DES\_OFB64 0x00000150

#define CKM\_DES\_OFB8 0x00000151

#define CKM\_DES\_CFB64 0x00000152

#define CKM\_DES\_CFB8 0x00000153

#define CKM\_MD2 0x00000200

#define CKM\_MD2\_HMAC 0x00000201

#define CKM\_MD2\_HMAC\_GENERAL 0x00000202

#define CKM\_MD5 0x00000210

#define CKM\_MD5\_HMAC 0x00000211

#define CKM\_MD5\_HMAC\_GENERAL 0x00000212

#define CKM\_SHA\_1 0x00000220

#define CKM\_SHA\_1\_HMAC 0x00000221

#define CKM\_SHA\_1\_HMAC\_GENERAL 0x00000222

#define CKM\_RIPEMD128 0x00000230

#define CKM\_RIPEMD128\_HMAC 0x00000231

#define CKM\_RIPEMD128\_HMAC\_GENERAL 0x00000232

#define CKM\_RIPEMD160 0x00000240

#define CKM\_RIPEMD160\_HMAC 0x00000241

#define CKM\_RIPEMD160\_HMAC\_GENERAL 0x00000242

#define CKM\_SHA256 0x00000250

#define CKM\_SHA256\_HMAC 0x00000251

#define CKM\_SHA256\_HMAC\_GENERAL 0x00000252

#define CKM\_SHA224 0x00000255

#define CKM\_SHA224\_HMAC 0x00000256

#define CKM\_SHA224\_HMAC\_GENERAL 0x00000257

#define CKM\_SHA384 0x00000260

#define CKM\_SHA384\_HMAC 0x00000261

#define CKM\_SHA384\_HMAC\_GENERAL 0x00000262

#define CKM\_SHA512 0x00000270

#define CKM\_SHA512\_HMAC 0x00000271

#define CKM\_SHA512\_HMAC\_GENERAL 0x00000272

#define CKM\_SECURID\_KEY\_GEN 0x00000280

#define CKM\_SECURID 0x00000282

#define CKM\_HOTP\_KEY\_GEN 0x00000290

#define CKM\_HOTP 0x00000291

#define CKM\_ACTI 0x000002A0

#define CKM\_ACTI\_KEY\_GEN 0x000002A1

#define CKM\_SHA3\_256 0x000002B0

#define CKM\_SHA3\_256\_HMAC 0x000002B1

#define CKM\_SHA3\_256\_HMAC\_GENERAL 0x000002B2

#define CKM\_SHA3\_256\_KEY\_GEN 0x000002B3

#define CKM\_SHA3\_224 0x000002B5

#define CKM\_SHA3\_224\_HMAC 0x000002B6

#define CKM\_SHA3\_224\_HMAC\_GENERAL 0x000002B7

#define CKM\_SHA3\_224\_KEY\_GEN 0x000002B8

#define CKM\_SHA3\_384 0x000002C0

#define CKM\_SHA3\_384\_HMAC 0x000002C1

#define CKM\_SHA3\_384\_HMAC\_GENERAL 0x000002C2

#define CKM\_SHA3\_384\_KEY\_GEN 0x000002C3

#define CKM\_SHA3\_512 0x000002D0

#define CKM\_SHA3\_512\_HMAC 0x000002D1

#define CKM\_SHA3\_512\_HMAC\_GENERAL 0x000002D2

#define CKM\_SHA3\_512\_KEY\_GEN 0x000002D3

#define CKM\_CAST\_KEY\_GEN 0x00000300

#define CKM\_CAST\_ECB 0x00000301

#define CKM\_CAST\_CBC 0x00000302

#define CKM\_CAST\_MAC 0x00000303

#define CKM\_CAST\_MAC\_GENERAL 0x00000304

#define CKM\_CAST\_CBC\_PAD 0x00000305

#define CKM\_CAST3\_KEY\_GEN 0x00000310

#define CKM\_CAST3\_ECB 0x00000311

#define CKM\_CAST3\_CBC 0x00000312

#define CKM\_CAST3\_MAC 0x00000313

#define CKM\_CAST3\_MAC\_GENERAL 0x00000314

#define CKM\_CAST3\_CBC\_PAD 0x00000315

#define CKM\_CAST128\_KEY\_GEN 0x00000320

#define CKM\_CAST128\_ECB 0x00000321

#define CKM\_CAST128\_CBC 0x00000322

#define CKM\_CAST128\_MAC 0x00000323

#define CKM\_CAST128\_MAC\_GENERAL 0x00000324

#define CKM\_CAST128\_CBC\_PAD 0x00000325

#define CKM\_RC5\_KEY\_GEN 0x00000330

#define CKM\_RC5\_ECB 0x00000331

#define CKM\_RC5\_CBC 0x00000332

#define CKM\_RC5\_MAC 0x00000333

#define CKM\_RC5\_MAC\_GENERAL 0x00000334

#define CKM\_RC5\_CBC\_PAD 0x00000335

#define CKM\_IDEA\_KEY\_GEN 0x00000340

#define CKM\_IDEA\_ECB 0x00000341

#define CKM\_IDEA\_CBC 0x00000342

#define CKM\_IDEA\_MAC 0x00000343

#define CKM\_IDEA\_MAC\_GENERAL 0x00000344

#define CKM\_IDEA\_CBC\_PAD 0x00000345

#define CKM\_GENERIC\_SECRET\_KEY\_GEN 0x00000350

#define CKM\_CONCATENATE\_BASE\_AND\_KEY 0x00000360

#define CKM\_CONCATENATE\_BASE\_AND\_DATA 0x00000362

#define CKM\_CONCATENATE\_DATA\_AND\_BASE 0x00000363

#define CKM\_XOR\_BASE\_AND\_DATA 0x00000364

#define CKM\_EXTRACT\_KEY\_FROM\_KEY 0x00000365

#define CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN 0x00000370

#define CKM\_SSL3\_MASTER\_KEY\_DERIVE 0x00000371

#define CKM\_SSL3\_KEY\_AND\_MAC\_DERIVE 0x00000372

#define CKM\_SSL3\_MASTER\_KEY\_DERIVE\_DH 0x00000373

#define CKM\_TLS\_PRE\_MASTER\_KEY\_GEN 0x00000374

#define CKM\_TLS\_MASTER\_KEY\_DERIVE 0x00000375

#define CKM\_TLS\_KEY\_AND\_MAC\_DERIVE 0x00000376

#define CKM\_TLS\_MASTER\_KEY\_DERIVE\_DH 0x00000377

#define CKM\_TLS\_PRF 0x00000378

#define CKM\_SSL3\_MD5\_MAC 0x00000380

#define CKM\_SSL3\_SHA1\_MAC 0x00000381

#define CKM\_MD5\_KEY\_DERIVATION 0x00000390

#define CKM\_MD2\_KEY\_DERIVATION 0x00000391

#define CKM\_SHA1\_KEY\_DERIVATION 0x00000392

#define CKM\_SHA256\_KEY\_DERIVATION 0x00000393

#define CKM\_SHA384\_KEY\_DERIVATION 0x00000394

#define CKM\_SHA512\_KEY\_DERIVATION 0x00000395

#define CKM\_SHA224\_KEY\_DERIVATION 0x00000396

#define CKM\_SHA3\_256\_KEY\_DERIVE 0x00000397

#define CKM\_SHA3\_224\_KEY\_DERIVE 0x00000398

#define CKM\_SHA3\_384\_KEY\_DERIVE 0x00000399

#define CKM\_SHA3\_512\_KEY\_DERIVE 0x0000039A

#define CKM\_SHAKE\_128\_KEY\_DERIVE 0x0000039B

#define CKM\_SHAKE\_256\_KEY\_DERIVE 0x0000039C

#define CKM\_PBE\_MD2\_DES\_CBC 0x000003A0

#define CKM\_PBE\_MD5\_DES\_CBC 0x000003A1

#define CKM\_PBE\_MD5\_CAST\_CBC 0x000003A2

#define CKM\_PBE\_MD5\_CAST3\_CBC 0x000003A3

#define CKM\_PBE\_MD5\_CAST5\_CBC 0x000003A4

#define CKM\_PBE\_MD5\_CAST128\_CBC 0x000003A4

#define CKM\_PBE\_SHA1\_CAST128\_CBC 0x000003A5

#define CKM\_PBE\_SHA1\_RC4\_128 0x000003A6

#define CKM\_PBE\_SHA1\_RC4\_40 0x000003A7

#define CKM\_PBE\_SHA1\_DES3\_EDE\_CBC 0x000003A8

#define CKM\_PBE\_SHA1\_DES2\_EDE\_CBC 0x000003A9

#define CKM\_PBE\_SHA1\_RC2\_128\_CBC 0x000003AA

#define CKM\_PBE\_SHA1\_RC2\_40\_CBC 0x000003AB

#define CKM\_SP800\_108\_COUNTER\_KDF 0x000003AC

#define CKM\_SP800\_108\_FEEDBACK\_KDF 0x000003AD

#define CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF 0x000003AE

#define CKM\_PKCS5\_PBKD2 0x000003B0

#define CKM\_PBA\_SHA1\_WITH\_SHA1\_HMAC 0x000003C0

#define CKM\_WTLS\_PRE\_MASTER\_KEY\_GEN 0x000003D0

#define CKM\_WTLS\_MASTER\_KEY\_DERIVE 0x000003D1

#define CKM\_WTLS\_MASTER\_KEY\_DERIVE\_DH\_ECC 0x000003D2

#define CKM\_WTLS\_PRF 0x000003D3

#define CKM\_WTLS\_SERVER\_KEY\_AND\_MAC\_DERIVE 0x000003D4

#define CKM\_WTLS\_CLIENT\_KEY\_AND\_MAC\_DERIVE 0x000003D5

#define CKM\_TLS12\_MAC 0x000003D8

#define CKM\_TLS12\_KDF 0x000003D9

#define CKM\_TLS12\_MASTER\_KEY\_DERIVE 0x000003E0

#define CKM\_TLS12\_KEY\_AND\_MAC\_DERIVE 0x000003E1

#define CKM\_TLS12\_MASTER\_KEY\_DERIVE\_DH 0x000003E2

#define CKM\_TLS12\_KEY\_SAFE\_DERIVE 0x000003E3

#define CKM\_TLS\_MAC 0x000003E4

#define CKM\_TLS\_KDF 0x000003E5

#define CKM\_KEY\_WRAP\_LYNKS 0x00000400

#define CKM\_KEY\_WRAP\_SET\_OAEP 0x00000401

#define CKM\_CMS\_SIG 0x00000500

#define CKM\_KIP\_DERIVE 0x00000510

#define CKM\_KIP\_WRAP 0x00000511

#define CKM\_KIP\_MAC 0x00000512

#define CKM\_CAMELLIA\_KEY\_GEN 0x00000550

#define CKM\_CAMELLIA\_ECB 0x00000551

#define CKM\_CAMELLIA\_CBC 0x00000552

#define CKM\_CAMELLIA\_MAC 0x00000553

#define CKM\_CAMELLIA\_MAC\_GENERAL 0x00000554

#define CKM\_CAMELLIA\_CBC\_PAD 0x00000555

#define CKM\_CAMELLIA\_ECB\_ENCRYPT\_DATA 0x00000556

#define CKM\_CAMELLIA\_CBC\_ENCRYPT\_DATA 0x00000557

#define CKM\_CAMELLIA\_CTR 0x00000558

#define CKM\_ARIA\_KEY\_GEN 0x00000560

#define CKM\_ARIA\_ECB 0x00000561

#define CKM\_ARIA\_CBC 0x00000562

#define CKM\_ARIA\_MAC 0x00000563

#define CKM\_ARIA\_MAC\_GENERAL 0x00000564

#define CKM\_ARIA\_CBC\_PAD 0x00000565

#define CKM\_ARIA\_ECB\_ENCRYPT\_DATA 0x00000566

#define CKM\_ARIA\_CBC\_ENCRYPT\_DATA 0x00000567

#define CKM\_SEED\_KEY\_GEN 0x00000650

#define CKM\_SEED\_ECB 0x00000651

#define CKM\_SEED\_CBC 0x00000652

#define CKM\_SEED\_MAC 0x00000653

#define CKM\_SEED\_MAC\_GENERAL 0x00000654

#define CKM\_SEED\_CBC\_PAD 0x00000655

#define CKM\_SEED\_ECB\_ENCRYPT\_DATA 0x00000656

#define CKM\_SEED\_CBC\_ENCRYPT\_DATA 0x00000657

#define CKM\_SKIPJACK\_KEY\_GEN 0x00001000

#define CKM\_SKIPJACK\_ECB64 0x00001001

#define CKM\_SKIPJACK\_CBC64 0x00001002

#define CKM\_SKIPJACK\_OFB64 0x00001003

#define CKM\_SKIPJACK\_CFB64 0x00001004

#define CKM\_SKIPJACK\_CFB32 0x00001005

#define CKM\_SKIPJACK\_CFB16 0x00001006

#define CKM\_SKIPJACK\_CFB8 0x00001007

#define CKM\_SKIPJACK\_WRAP 0x00001008

#define CKM\_SKIPJACK\_PRIVATE\_WRAP 0x00001009

#define CKM\_SKIPJACK\_RELAYX 0x0000100A

#define CKM\_KEA\_KEY\_PAIR\_GEN 0x00001010

#define CKM\_KEA\_KEY\_DERIVE 0x00001011

#define CKM\_KEA\_DERIVE 0x00001012

#define CKM\_FORTEZZA\_TIMESTAMP 0x00001020

#define CKM\_BATON\_KEY\_GEN 0x00001030

#define CKM\_BATON\_ECB128 0x00001031

#define CKM\_BATON\_ECB96 0x00001032

#define CKM\_BATON\_CBC128 0x00001033

#define CKM\_BATON\_COUNTER 0x00001034

#define CKM\_BATON\_SHUFFLE 0x00001035

#define CKM\_BATON\_WRAP 0x00001036

#define CKM\_EC\_KEY\_PAIR\_GEN 0x00001040

#define CKM\_ECDSA 0x00001041

#define CKM\_ECDSA\_SHA1 0x00001042

#define CKM\_ECDSA\_SHA224 0x00001043

#define CKM\_ECDSA\_SHA256 0x00001044

#define CKM\_ECDSA\_SHA384 0x00001045

#define CKM\_ECDSA\_SHA512 0x00001046

#define CKM\_ECDSA\_SHA3\_224 0x00001047

#define CKM\_ECDSA\_SHA3\_256 0x00001048

#define CKM\_ECDSA\_SHA3\_384 0x00001049

#define CKM\_ECDSA\_SHA3\_512 0x0000104A

#define CKM\_ECDH1\_DERIVE 0x00001050

#define CKM\_ECDH1\_COFACTOR\_DERIVE 0x00001051

#define CKM\_ECMQV\_DERIVE 0x00001052

#define CKM\_ECDH\_AES\_KEY\_WRAP 0x00001053

#define CKM\_RSA\_AES\_KEY\_WRAP 0x00001054

#define CKM\_EC\_EDWARDS\_KEY\_PAIR\_GEN 0x00001055

#define CKM\_EC\_MONTGOMERY\_KEY\_PAIR\_GEN 0x00001056

#define CKM\_EDDSA 0x00001057

#define CKM\_JUNIPER\_KEY\_GEN 0x00001060

#define CKM\_JUNIPER\_ECB128 0x00001061

#define CKM\_JUNIPER\_CBC128 0x00001062

#define CKM\_JUNIPER\_COUNTER 0x00001063

#define CKM\_JUNIPER\_SHUFFLE 0x00001064

#define CKM\_JUNIPER\_WRAP 0x00001065

#define CKM\_FASTHASH 0x00001070

#define CKM\_AES\_XTS 0x00001071

#define CKM\_AEX\_XTS\_KEY\_GEN 0x00001072

#define CKM\_AES\_KEY\_GEN 0x00001080

#define CKM\_AES\_ECB 0x00001081

#define CKM\_AES\_CBC 0x00001082

#define CKM\_AES\_MAC 0x00001083

#define CKM\_AES\_MAC\_GENERAL 0x00001084

#define CKM\_AES\_CBC\_PAD 0x00001085

#define CKM\_AES\_CTR 0x00001086

#define CKM\_AES\_GCM 0x00001087

#define CKM\_AES\_CCM 0x00001088

#define CKM\_AES\_CMAC\_GENERAL 0x00001089

#define CKM\_AES\_CMAC 0x0000108A

#define CKM\_AES\_CTS 0x0000108B

#define CKM\_AES\_XCBC\_MAC 0x0000108C

#define CKM\_AES\_XCBC\_MAC\_96 0x0000108D

#define CKM\_AES\_GMAC 0x0000108E

#define CKM\_BLOWFISH\_KEY\_GEN 0x00001090

#define CKM\_BLOWFISH\_CBC 0x00001091

#define CKM\_TWOFISH\_KEY\_GEN 0x00001092

#define CKM\_TWOFISH\_CBC 0x00001093

#define CKM\_BLOWFISH\_CBC\_PAD 0x00001094

#define CKM\_TWOFISH\_CBC\_PAD 0x00001095

#define CKM\_DES\_ECB\_ENCRYPT\_DATA 0x00001100

#define CKM\_DES\_CBC\_ENCRYPT\_DATA 0x00001101

#define CKM\_DES3\_ECB\_ENCRYPT\_DATA 0x00001102

#define CKM\_DES3\_CBC\_ENCRYPT\_DATA 0x00001103

#define CKM\_AES\_ECB\_ENCRYPT\_DATA 0x00001104

#define CKM\_AES\_CBC\_ENCRYPT\_DATA 0x00001105

#define CKM\_GOSTR3410\_KEY\_PAIR\_GEN 0x00001200

#define CKM\_GOSTR3410 0x00001201

#define CKM\_GOSTR3410\_WITH\_GOSTR3411 0x00001202

#define CKM\_GOSTR3410\_KEY\_WRAP 0x00001203

#define CKM\_GOSTR3410\_DERIVE 0x00001204

#define CKM\_GOSTR3411 0x00001210

#define CKM\_GOSTR3411\_HMAC 0x00001211

#define CKM\_GOST28147\_KEY\_GEN 0x00001220

#define CKM\_GOST28147\_ECB 0x00001221

#define CKM\_GOST28147 0x00001222

#define CKM\_GOST28147\_MAC 0x00001223

#define CKM\_GOST28147\_KEY\_WRAP 0x00001224

#define CKM\_CHACHA20\_KEY\_GEN 0x00001225

#define CKM\_CHACHA20 0x00001226

#define CKM\_POLY1305\_KEY\_GEN 0x00001227

#define CKM\_POLY1305 0x00001228

#define CKM\_DSA\_PARAMETER\_GEN 0x00002000

#define CKM\_DH\_PKCS\_PARAMETER\_GEN 0x00002001

#define CKM\_X9\_42\_DH\_PKCS\_PARAMETER\_GEN 0x00002002

#define CKM\_DSA\_PROBABLISTIC\_PARAMETER\_GEN 0x00002003

#define CKM\_DSA\_SHAWE\_TAYLOR\_PARAMETER\_GEN 0x00002004

#define CKM\_AES\_OFB 0x00002104

#define CKM\_AES\_CFB64 0x00002105

#define CKM\_AES\_CFB8 0x00002106

#define CKM\_AES\_CFB128 0x00002107

#define CKM\_AES\_CFB1 0x00002108

#define CKM\_AES\_KEY\_WRAP 0x00002109

#define CKM\_AES\_KEY\_WRAP\_PAD 0x0000210A

#define CKM\_AES\_KEY\_WRAP\_KWP 0x0000210B

#define CKM\_RSA\_PKCS\_TPM\_1\_1 0x00004001

#define CKM\_RSA\_PKCS\_OAEP\_TPM\_1\_1 0x00004002

#define CKM\_SHA\_1\_KEY\_GEN 0x00004003

#define CKM\_SHA224\_KEY\_GEN 0x00004004

#define CKM\_SHA256\_KEY\_GEN 0x00004005

#define CKM\_SHA384\_KEY\_GEN 0x00004006

#define CKM\_SHA512\_KEY\_GEN 0x00004007

#define CKM\_SHA512\_224\_KEY\_GEN 0x00004008

#define CKM\_SHA512\_256\_KEY\_GEN 0x00004009

#define CKM\_SHA512\_T\_KEY\_GEN 0x0000400A

#define CKM\_NULL 0x0000400B

#define CKM\_BLAKE2B\_160 0x0000400C

#define CKM\_BLAKE2B\_160\_HMAC 0x0000400D

#define CKM\_BLAKE2B\_160\_HMAC\_GENERAL 0x0000400E

#define CKM\_BLAKE2B\_160\_KEY\_DERIVE 0x0000400F

#define CKM\_BLAKE2B\_160\_KEY\_GEN 0x00004010

#define CKM\_BLAKE2B\_256 0x00004011

#define CKM\_BLAKE2B\_256\_HMAC 0x00004012

#define CKM\_BLAKE2B\_256\_HMAC\_GENERAL 0x00004013

#define CKM\_BLAKE2B\_256\_KEY\_DERIVE 0x00004014

#define CKM\_BLAKE2B\_256\_KEY\_GEN 0x00004015

#define CKM\_BLAKE2B\_384 0x00004016

#define CKM\_BLAKE2B\_384\_HMAC 0x00004017

#define CKM\_BLAKE2B\_384\_HMAC\_GENERAL 0x00004018

#define CKM\_BLAKE2B\_384\_KEY\_DERIVE 0x00004019

#define CKM\_BLAKE2B\_384\_KEY\_GEN 0x0000401A

#define CKM\_BLAKE2B\_512 0x0000401B

#define CKM\_BLAKE2B\_512\_HMAC 0x0000401C

#define CKM\_BLAKE2B\_512\_HMAC\_GENERAL 0x0000401D

#define CKM\_BLAKE2B\_512\_KEY\_DERIVE 0x0000401E

#define CKM\_BLAKE2B\_512\_KEY\_GEN 0x0000401F

#define CKM\_SALSA20 0x00004020

#define CKM\_CHACHA20\_POLY1305 0x00004021

#define CKM\_SALSA20\_POLY1305 0x00004022

#define CKM\_X3DH\_INITIALIZE 0x00004023

#define CKM\_X3DH\_RESPOND 0x00004024

#define CKM\_X2RATCHET\_INITIALIZE 0x00004025

#define CKM\_X2RATCHET\_RESPOND 0x00004026

#define CKM\_X2RATCHET\_ENCRYPT 0x00004027

#define CKM\_X2RATCHET\_DECRYPT 0x00004028

#define CKM\_XEDDSA 0x00004029

#define CKM\_HKDF\_DERIVE 0x0000402A

#define CKM\_HKDF\_DATA 0x0000402B

#define CKM\_HKDF\_KEY\_GEN 0x0000402C

#define CKM\_VENDOR\_DEFINED 0x80000000

## B.5 Attributes

#define CKA\_CLASS 0x00000000

#define CKA\_TOKEN 0x00000001

#define CKA\_PRIVATE 0x00000002

#define CKA\_LABEL 0x00000003

#define CKA\_UNIQUE\_ID 0x00000004

#define CKA\_APPLICATION 0x00000010

#define CKA\_VALUE 0x00000011

#define CKA\_OBJECT\_ID 0x00000012

#define CKA\_CERTIFICATE\_TYPE 0x00000080

#define CKA\_ISSUER 0x00000081

#define CKA\_SERIAL\_NUMBER 0x00000082

#define CKA\_AC\_ISSUER 0x00000083

#define CKA\_OWNER 0x00000084

#define CKA\_ATTR\_TYPES 0x00000085

#define CKA\_TRUSTED 0x00000086

#define CKA\_CERTIFICATE\_CATEGORY 0x00000087

#define CKA\_JAVA\_MIDP\_SECURITY\_DOMAIN 0x00000088

#define CKA\_URL 0x00000089

#define CKA\_HASH\_OF\_SUBJECT\_PUBLIC\_KEY 0x0000008A

#define CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY 0x0000008B

#define CKA\_NAME\_HASH\_ALGORITHM 0x0000008C

#define CKA\_CHECK\_VALUE 0x00000090

#define CKA\_KEY\_TYPE 0x00000100

#define CKA\_SUBJECT 0x00000101

#define CKA\_ID 0x00000102

#define CKA\_SENSITIVE 0x00000103

#define CKA\_ENCRYPT 0x00000104

#define CKA\_DECRYPT 0x00000105

#define CKA\_WRAP 0x00000106

#define CKA\_UNWRAP 0x00000107

#define CKA\_SIGN 0x00000108

#define CKA\_SIGN\_RECOVER 0x00000109

#define CKA\_VERIFY 0x0000010A

#define CKA\_VERIFY\_RECOVER 0x0000010B

#define CKA\_DERIVE 0x0000010C

#define CKA\_START\_DATE 0x00000110

#define CKA\_END\_DATE 0x00000111

#define CKA\_MODULUS 0x00000120

#define CKA\_MODULUS\_BITS 0x00000121

#define CKA\_PUBLIC\_EXPONENT 0x00000122

#define CKA\_PRIVATE\_EXPONENT 0x00000123

#define CKA\_PRIME\_1 0x00000124

#define CKA\_PRIME\_2 0x00000125

#define CKA\_EXPONENT\_1 0x00000126

#define CKA\_EXPONENT\_2 0x00000127

#define CKA\_COEFFICIENT 0x00000128

#define CKA\_PUBLIC\_KEY\_INFO 0x00000129

#define CKA\_PRIME 0x00000130

#define CKA\_SUBPRIME 0x00000131

#define CKA\_BASE 0x00000132

#define CKA\_PRIME\_BITS 0x00000133

#define CKA\_SUBPRIME\_BITS 0x00000134

#define CKA\_SUB\_PRIME\_BITS CKA\_SUBPRIME\_BITS

#define CKA\_VALUE\_BITS 0x00000160

#define CKA\_VALUE\_LEN 0x00000161

#define CKA\_EXTRACTABLE 0x00000162

#define CKA\_LOCAL 0x00000163

#define CKA\_NEVER\_EXTRACTABLE 0x00000164

#define CKA\_ALWAYS\_SENSITIVE 0x00000165

#define CKA\_KEY\_GEN\_MECHANISM 0x00000166

#define CKA\_MODIFIABLE 0x00000170

#define CKA\_COPYABLE 0x00000171

#define CKA\_DESTROYABLE 0x00000172

#define CKA\_EC\_PARAMS 0x00000180

#define CKA\_EC\_POINT 0x00000181

#define CKA\_WRAP\_WITH\_TRUSTED 0x00000210

#define CKA\_WRAP\_TEMPLATE (CKF\_ARRAY\_ATTRIBUTE|0x00000211)

#define CKA\_UNWRAP\_TEMPLATE (CKF\_ARRAY\_ATTRIBUTE|0x00000212)

#define CKA\_DERIVE\_TEMPLATE (CKF\_ARRAY\_ATTRIBUTE|0x00000213)

#define CKA\_OTP\_FORMAT 0x00000220

#define CKA\_OTP\_LENGTH 0x00000221

#define CKA\_OTP\_TIME\_INTERVAL 0x00000222

#define CKA\_OTP\_USER\_FRIENDLY\_MODE 0x00000223

#define CKA\_OTP\_CHALLENGE\_REQUIREMENT 0x00000224

#define CKA\_OTP\_TIME\_REQUIREMENT 0x00000225

#define CKA\_OTP\_COUNTER\_REQUIREMENT 0x00000226

#define CKA\_OTP\_PIN\_REQUIREMENT 0x00000227

#define CKA\_OTP\_USER\_IDENTIFIER 0x0000022A

#define CKA\_OTP\_SERVICE\_IDENTIFIER 0x0000022B

#define CKA\_OTP\_SERVICE\_LOGO 0x0000022C

#define CKA\_OTP\_SERVICE\_LOGO\_TYPE 0x0000022D

#define CKA\_OTP\_COUNTER 0x0000022E

#define CKA\_OTP\_TIME 0x0000022F

#define CKA\_GOSTR3410\_PARAMS 0x00000250

#define CKA\_GOSTR3411\_PARAMS 0x00000251

#define CKA\_GOST28147\_PARAMS 0x00000252

#define CKA\_HW\_FEATURE\_TYPE 0x00000300

#define CKA\_RESET\_ON\_INIT 0x00000301

#define CKA\_HAS\_RESET 0x00000302

#define CKA\_PIXEL\_X 0x00000400

#define CKA\_PIXEL\_Y 0x00000401

#define CKA\_RESOLUTION 0x00000402

#define CKA\_CHAR\_ROWS 0x00000403

#define CKA\_CHAR\_COLUMNS 0x00000404

#define CKA\_COLOR 0x00000405

#define CKA\_BITS\_PER\_PIXEL 0x00000406

#define CKA\_CHAR\_SETS 0x00000480

#define CKA\_ENCODING\_METHODS 0x00000481

#define CKA\_MIME\_TYPES 0x00000482

#define CKA\_MECHANISM\_TYPE 0x00000500

#define CKA\_REQUIRED\_CMS\_ATTRIBUTES 0x00000501

#define CKA\_DEFAULT\_CMS\_ATTRIBUTES 0x00000502

#define CKA\_SUPPORTED\_CMS\_ATTRIBUTES 0x00000503

#define CKA\_ALLOWED\_MECHANISMS (CKF\_ARRAY\_ATTRIBUTE|0x00000600)

#define CKA\_VENDOR\_DEFINED 0x80000000

## B.6 Attribute constants

#define CK\_OTP\_FORMAT\_DECIMAL 0x00000000

#define CK\_OTP\_FORMAT\_HEXADECIMAL 0x00000001

#define CK\_OTP\_FORMAT\_ALPHANUMERIC 0x00000002

#define CK\_OTP\_FORMAT\_BINARY 0x00000003

#define CK\_OTP\_PARAM\_IGNORED 0x00000000

#define CK\_OTP\_PARAM\_OPTIONAL 0x00000001

#define CK\_OTP\_PARAM\_MANDATORY 0x00000002

#define CK\_OTP\_VALUE 0x00000000

#define CK\_OTP\_PIN 0x00000001

#define CK\_OTP\_CHALLENGE 0x00000002

#define CK\_OTP\_TIME 0x00000003

#define CK\_OTP\_COUNTER 0x00000004

#define CK\_OTP\_FLAGS 0x00000005

#define CK\_OTP\_OUTPUT\_LENGTH 0x00000006

#define CK\_OTP\_FORMAT 0x00000007

## B.7 Other constants

#define CKF\_NEXT\_OTP 0x00000001

#define CKF\_EXCLUDE\_TIME 0x00000002

#define CKF\_EXCLUDE\_COUNTER 0x00000004

#define CKF\_EXCLUDE\_CHALLENGE 0x00000008

#define CKF\_EXCLUDE\_PIN 0x00000010

#define CKF\_USER\_FRIENDLY\_OTP 0x00000020

#define CKF\_HKDF\_SALT\_NULL 0x00000001

#define CKF\_HKDF\_SALT\_DATA 0x00000002

#define CKF\_HKDF\_SALT\_KEY 0x00000004

## B.8 Notifications

#define CKN\_OTP\_CHANGED 0x00000001

## B.9 Return values

#define CKR\_OK 0x00000000

#define CKR\_CANCEL 0x00000001

#define CKR\_HOST\_MEMORY 0x00000002

#define CKR\_SLOT\_ID\_INVALID 0x00000003

#define CKR\_GENERAL\_ERROR 0x00000005

#define CKR\_FUNCTION\_FAILED 0x00000006

#define CKR\_ARGUMENTS\_BAD 0x00000007

#define CKR\_NO\_EVENT 0x00000008

#define CKR\_NEED\_TO\_CREATE\_THREADS 0x00000009

#define CKR\_CANT\_LOCK 0x0000000A

#define CKR\_ATTRIBUTE\_READ\_ONLY 0x00000010

#define CKR\_ATTRIBUTE\_SENSITIVE 0x00000011

#define CKR\_ATTRIBUTE\_TYPE\_INVALID 0x00000012

#define CKR\_ATTRIBUTE\_VALUE\_INVALID 0x00000013

#define CKR\_ACTION\_PROHIBITED 0x0000001B

#define CKR\_DATA\_INVALID 0x00000020

#define CKR\_DATA\_LEN\_RANGE 0x00000021

#define CKR\_DEVICE\_ERROR 0x00000030

#define CKR\_DEVICE\_MEMORY 0x00000031

#define CKR\_DEVICE\_REMOVED 0x00000032

#define CKR\_ENCRYPTED\_DATA\_INVALID 0x00000040

#define CKR\_ENCRYPTED\_DATA\_LEN\_RANGE 0x00000041

#define CKR\_AEAD\_DECRYPT\_FAILED 0x00000042

#define CKR\_FUNCTION\_CANCELED 0x00000050

#define CKR\_FUNCTION\_NOT\_PARALLEL 0x00000051

#define CKR\_FUNCTION\_NOT\_SUPPORTED 0x00000054

#define CKR\_KEY\_HANDLE\_INVALID 0x00000060

#define CKR\_KEY\_SIZE\_RANGE 0x00000062

#define CKR\_KEY\_TYPE\_INCONSISTENT 0x00000063

#define CKR\_KEY\_NOT\_NEEDED 0x00000064

#define CKR\_KEY\_CHANGED 0x00000065

#define CKR\_KEY\_NEEDED 0x00000066

#define CKR\_KEY\_INDIGESTIBLE 0x00000067

#define CKR\_KEY\_FUNCTION\_NOT\_PERMITTED 0x00000068

#define CKR\_KEY\_NOT\_WRAPPABLE 0x00000069

#define CKR\_KEY\_UNEXTRACTABLE 0x0000006A

#define CKR\_MECHANISM\_INVALID 0x00000070

#define CKR\_MECHANISM\_PARAM\_INVALID 0x00000071

#define CKR\_OBJECT\_HANDLE\_INVALID 0x00000082

#define CKR\_OPERATION\_ACTIVE 0x00000090

#define CKR\_OPERATION\_NOT\_INITIALIZED 0x00000091

#define CKR\_PIN\_INCORRECT 0x000000A0

#define CKR\_PIN\_INVALID 0x000000A1

#define CKR\_PIN\_LEN\_RANGE 0x000000A2

#define CKR\_PIN\_EXPIRED 0x000000A3

#define CKR\_PIN\_LOCKED 0x000000A4

#define CKR\_SESSION\_CLOSED 0x000000B0

#define CKR\_SESSION\_COUNT 0x000000B1

#define CKR\_SESSION\_HANDLE\_INVALID 0x000000B3

#define CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED 0x000000B4

#define CKR\_SESSION\_READ\_ONLY 0x000000B5

#define CKR\_SESSION\_EXISTS 0x000000B6

#define CKR\_SESSION\_READ\_ONLY\_EXISTS 0x000000B7

#define CKR\_SESSION\_READ\_WRITE\_SO\_EXISTS 0x000000B8

#define CKR\_SIGNATURE\_INVALID 0x000000C0

#define CKR\_SIGNATURE\_LEN\_RANGE 0x000000C1

#define CKR\_TEMPLATE\_INCOMPLETE 0x000000D0

#define CKR\_TEMPLATE\_INCONSISTENT 0x000000D1

#define CKR\_TOKEN\_NOT\_PRESENT 0x000000E0

#define CKR\_TOKEN\_NOT\_RECOGNIZED 0x000000E1

#define CKR\_TOKEN\_WRITE\_PROTECTED 0x000000E2

#define CKR\_UNWRAPPING\_KEY\_HANDLE\_INVALID 0x000000F0

#define CKR\_UNWRAPPING\_KEY\_SIZE\_RANGE 0x000000F1

#define CKR\_UNWRAPPING\_KEY\_TYPE\_INCONSISTENT 0x000000F2

#define CKR\_USER\_ALREADY\_LOGGED\_IN 0x00000100

#define CKR\_USER\_NOT\_LOGGED\_IN 0x00000101

#define CKR\_USER\_PIN\_NOT\_INITIALIZED 0x00000102

#define CKR\_USER\_TYPE\_INVALID 0x00000103

#define CKR\_USER\_ANOTHER\_ALREADY\_LOGGED\_IN 0x00000104

#define CKR\_USER\_TOO\_MANY\_TYPES 0x00000105

#define CKR\_WRAPPED\_KEY\_INVALID 0x00000110

#define CKR\_WRAPPED\_KEY\_LEN\_RANGE 0x00000112

#define CKR\_WRAPPING\_KEY\_HANDLE\_INVALID 0x00000113

#define CKR\_WRAPPING\_KEY\_SIZE\_RANGE 0x00000114

#define CKR\_WRAPPING\_KEY\_TYPE\_INCONSISTENT 0x00000115

#define CKR\_RANDOM\_SEED\_NOT\_SUPPORTED 0x00000120

#define CKR\_RANDOM\_NO\_RNG 0x00000121

#define CKR\_DOMAIN\_PARAMS\_INVALID 0x00000130

#define CKR\_CURVE\_NOT\_SUPPORTED 0x00000140

#define CKR\_BUFFER\_TOO\_SMALL 0x00000150

#define CKR\_SAVED\_STATE\_INVALID 0x00000160

#define CKR\_INFORMATION\_SENSITIVE 0x00000170

#define CKR\_STATE\_UNSAVEABLE 0x00000180

#define CKR\_CRYPTOKI\_NOT\_INITIALIZED 0x00000190

#define CKR\_CRYPTOKI\_ALREADY\_INITIALIZED 0x00000191

#define CKR\_MUTEX\_BAD 0x000001A0

#define CKR\_MUTEX\_NOT\_LOCKED 0x000001A1

#define CKR\_NEW\_PIN\_MODE 0x000001B0

#define CKR\_NEXT\_OTP 0x000001B1

#define CKR\_EXCEEDED\_MAX\_ITERATIONS 0x000001B5

#define CKR\_FIPS\_SELF\_TEST\_FAILED 0x000001B6

#define CKR\_LIBRARY\_LOAD\_FAILED 0x000001B7

#define CKR\_PIN\_TOO\_WEAK 0x000001B8

#define CKR\_PUBLIC\_KEY\_INVALID 0x000001B9

#define CKR\_FUNCTION\_REJECTED 0x00000200

#define CKR\_VENDOR\_DEFINED 0x80000000

1. Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Editor** | **Changes Made** |
| Wd02 | 18-Jun-2017 | Chris Zimman | Initial version incorporating recent changes since 2.41 |
| WD07 | 16-Oct-2018 | Dieter Bong | See pkcs11-curr-v3.0-updates-from-wd06-to-wd07.docx |
| WD08 | 22-Oct-2018 | Dieter Bong | Added references [SALSA] to section 1.3  Split *GOST Mechanisms vs. Functions* table for GOST into separate tables for the respective GOST nnn algorithms.  Changed format of Salsa20 to Heading2, making it section 2.59  Removed section B.1 OTP definitions (OTP definitions have become pat of the standard header files and are now covered in B.6 Attribute constants)  Copyright updated from “2013” to “2018” |
| WD09 | 26-Mar-2019 | Daniel Minder | Added CKF\_EC\_CURVENAME to table 34  Changed CK\_GCM\_AEAD\_PARAMS to CK\_GCM\_MESSAGE\_PARAMS  Reworked section 2.13 (additional AES mechanisms) during F2F  Removed Derive for CKM\_AES\_GMAC in Table 80 since this is not defined in section 2.13.4  Removed solved comments of Chris  typos and formatting  Edwards curves and RFC 8410:  - Added reference to RFC 8410 at several places in 2.3.5 - 2.3.14  - Clarified that Edwards/Montgomery curves specified with curveName are incompatible with curves specified with oId (since RFC 8410 is designed like this)  - Added explanation for CKM\_TOKEN\_RESOURCE\_EXCEEDED error in 2.3.14  - Changed sample template for edwards public key objects in 2.3.5 since the parameter spec was in ecPoint instead of ecParams  - Added "allowed key types" table in 2.3.17 - 2.3.20 (they were lost when copying from the proposal), but corrected it for ECDH with cofactor since this is not possible according to RFC8032.  Corrected some formatting issues raised by Darren |
| WD10 | 29 Apr 2019 | Dieter Bong | * Updated section Related work * Added Dieter Bong as Editor * Put year 2019 in Copyright * Added section 2.62 HKDF Mechanisms; HKDF constants in Appendix B; RFC5869 in section 1.4 Non-Normative References * Section 2.40: added CKM\_NULL, removed CKM\_TLS10\_MAC\_\* * Section 2.52.7.1 reference to base specification corrrected * Replaced reference to [PKCS11-Base] table 10 by [PKCS11-Base] table 11 throughout whole document * Removed all occurences of CKK\_ECDSA and CKA\_ECDSA\_PARAMS and added notices that they are deprecated * Removed #define’s for CKA\_SECONDARY\_AUTH, CKA\_AUTH\_PIN\_FLAGS and CKA\_ALWAYS\_AUTHENTICATE * Removed #define’s for CAST5 mechanisms |
| WD10 Rev. 2 | 7 May 2019 | Dieter Bong | * Moved CKM\_NULL to own section 2.63 * Removed 2 remaining occurences of CKA\_ECDSA\_PARAMS |
| WD11 | May 28, 2019 | Tony Cox | * Final cleanup of front introductory texts and links prior to CSPRD |

1. The encoding in V2.20 was not specified and resulted in different implementations choosing different encodings. Applications relying only on a V2.20 encoding (e.g. the DER variant) other than the one specified now (raw) may not work with all V2.30 compliant tokens. [↑](#footnote-ref-1)
2. Note that the rules regarding the CKA\_SENSITIVE, CKA\_EXTRACTABLE, CKA\_ALWAYS\_SENSITIVE, and CKA\_NEVER\_EXTRACTABLE attributes have changed in version 2.11 to match the policy used by other key derivation mechanisms such as CKM\_SSL3\_MASTER\_KEY\_DERIVE. [↑](#footnote-ref-2)
3. Note that the rules regarding the CKA\_SENSITIVE, CKA\_EXTRACTABLE, CKA\_ALWAYS\_SENSITIVE, and CKA\_NEVER\_EXTRACTABLE attributes have changed in version 2.11 to match the policy used by other key derivation mechanisms such as CKM\_SSL3\_MASTER\_KEY\_DERIVE. [↑](#footnote-ref-3)
4. “\*” indicates 0 or more calls may be made as required [↑](#footnote-ref-4)
5. “\*” indicates 0 or more calls may be made as required [↑](#footnote-ref-5)
6. “\*” indicates 0 or more calls may be made as required [↑](#footnote-ref-6)
7. “\*” indicates 0 or more calls may be made as required [↑](#footnote-ref-7)
8. “\*” indicates 0 or more calls may be made as required [↑](#footnote-ref-8)
9. Applications that may need to retrieve the next OTP should be prepared to handle this situation. For example, an application could store the OTP value returned by C\_Sign so that, if a next OTP is required, it can compare it to the OTP value returned by subsequent calls to C\_Sign should it turn out that the library does not support the CKF\_NEXT\_OTP flag. [↑](#footnote-ref-9)
10. “\*” indicates 0 or more calls may be made as required [↑](#footnote-ref-10)
11. “\*” indicates 0 or more calls may be made as required [↑](#footnote-ref-11)
12. “\*” indicates 0 or more calls may be made as required [↑](#footnote-ref-12)