

PKCS #11 Specification Version 3.1

OASIS Standard

23 July 2023

This stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/pkcs11-spec-v3.1-os.pdf> (Authoritative)

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/pkcs11-spec-v3.1-os.html>

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/pkcs11-spec-v3.1-os.docx>

Previous stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/pkcs11-spec-v3.1-cs01.pdf> (Authoritative)

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/pkcs11-spec-v3.1-cs01.html>

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/cs01/pkcs11-spec-v3.1-cs01.docx>

Latest stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.pdf> (Authoritative)

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.html>

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.docx>

Technical Committee:

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

Chairs:

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

Greg Scott ([greg.scott@cryptsoft.com](mailto:greg.scott@cryptsoft.com)), [Cryptsoft Pty Ltd](https://cryptsoft.com/)

Editors:

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

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

Additional artifacts:

This prose specification is one component of a Work Product that also includes PKCS #11 header files:

* pkcs11.h: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/include/pkcs11-v3.1/pkcs11.h>
* pkcs11f.h: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/include/pkcs11-v3.1/pkcs11f.h>
* pkcs11t.h: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/include/pkcs11-v3.1/pkcs11t.h>

Related work:

This specification replaces or supersedes:

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

This specification is related to:

* *PKCS #11 Profiles Version 3.1.* Edited by Tim Hudson. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.1/pkcs11-profiles-v3.1.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 membership of OASIS on the above date. The level of approval is also listed above. Check the "Latest stage" 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-2017-05-26/#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.

Key words:

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [[RFC2119](#RFC2119)] and [[RFC8174](#RFC8174)] when, and only when, they appear in all capitals, as shown here.

Citation format:

When referencing this document, the following citation format should be used:

[PKCS11-Spec-v3.1]

*PKCS #11 Specification Version 3.1*. Edited by Dieter Bong and Tony Cox. 23 July 2023. OASIS Standard. <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/os/pkcs11-spec-v3.1-os.html>. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.html>.

Notices:

Copyright © OASIS Open 2023. All Rights Reserved.

Distributed under the terms of the OASIS IPR Policy, [<https://www.oasis-open.org/policies-guidelines/ipr/>]. For complete copyright information please see the full Notices section in Appendix D below.

Table of Contents

[1 Introduction 18](#_Toc111203182)

[1.1 Definitions 18](#_Toc111203183)

[1.2 Symbols and abbreviations 20](#_Toc111203184)

[1.3 Normative References 22](#_Toc111203185)

[1.4 Non-Normative References 25](#_Toc111203186)

[2 Platform- and compiler-dependent directives for C or C++ 28](#_Toc111203187)

[2.1 Structure packing 28](#_Toc111203188)

[2.2 Pointer-related macros 28](#_Toc111203189)

[3 General data types 30](#_Toc111203190)

[3.1 General information 30](#_Toc111203191)

[3.2 Slot and token types 31](#_Toc111203192)

[3.3 Session types 36](#_Toc111203193)

[3.4 Object types 37](#_Toc111203194)

[3.5 Data types for mechanisms 41](#_Toc111203195)

[3.6 Function types 44](#_Toc111203196)

[3.7 Locking-related types 49](#_Toc111203197)

[4 Objects 52](#_Toc111203198)

[4.1 Creating, modifying, and copying objects 53](#_Toc111203199)

[4.1.1 Creating objects 53](#_Toc111203200)

[4.1.2 Modifying objects 54](#_Toc111203201)

[4.1.3 Copying objects 54](#_Toc111203202)

[4.2 Common attributes 54](#_Toc111203203)

[4.3 Hardware Feature Objects 55](#_Toc111203204)

[4.3.1 Definitions 55](#_Toc111203205)

[4.3.2 Overview 55](#_Toc111203206)

[4.3.3 Clock 56](#_Toc111203207)

[4.3.3.1 Definition 56](#_Toc111203208)

[4.3.3.2 Description 56](#_Toc111203209)

[4.3.4 Monotonic Counter Objects 56](#_Toc111203210)

[4.3.4.1 Definition 56](#_Toc111203211)

[4.3.4.2 Description 56](#_Toc111203212)

[4.3.5 User Interface Objects 56](#_Toc111203213)

[4.3.5.1 Definition 56](#_Toc111203214)

[4.3.5.2 Description 57](#_Toc111203215)

[4.4 Storage Objects 57](#_Toc111203216)

[4.4.1 The CKA\_UNIQUE\_ID attribute 58](#_Toc111203217)

[4.5 Data objects 59](#_Toc111203218)

[4.5.1 Definitions 59](#_Toc111203219)

[4.5.2 Overview 59](#_Toc111203220)

[4.6 Certificate objects 59](#_Toc111203221)

[4.6.1 Definitions 59](#_Toc111203222)

[4.6.2 Overview 59](#_Toc111203223)

[4.6.3 X.509 public key certificate objects 60](#_Toc111203224)

[4.6.4 WTLS public key certificate objects 62](#_Toc111203225)

[4.6.5 X.509 attribute certificate objects 63](#_Toc111203226)

[4.7 Key objects 64](#_Toc111203227)

[4.7.1 Definitions 64](#_Toc111203228)

[4.7.2 Overview 64](#_Toc111203229)

[4.8 Public key objects 65](#_Toc111203230)

[4.9 Private key objects 67](#_Toc111203231)

[4.10 Secret key objects 69](#_Toc111203232)

[4.11 Domain parameter objects 71](#_Toc111203233)

[4.11.1 Definitions 71](#_Toc111203234)

[4.11.2 Overview 71](#_Toc111203235)

[4.12 Mechanism objects 72](#_Toc111203236)

[4.12.1 Definitions 72](#_Toc111203237)

[4.12.2 Overview 72](#_Toc111203238)

[4.13 Profile objects 72](#_Toc111203239)

[4.13.1 Definitions 72](#_Toc111203240)

[4.13.2 Overview 72](#_Toc111203241)

[5 Functions 74](#_Toc111203242)

[5.1 Function return values 77](#_Toc111203243)

[5.1.1 Universal Cryptoki function return values 78](#_Toc111203244)

[5.1.2 Cryptoki function return values for functions that use a session handle 78](#_Toc111203245)

[5.1.3 Cryptoki function return values for functions that use a token 78](#_Toc111203246)

[5.1.4 Special return value for application-supplied callbacks 79](#_Toc111203247)

[5.1.5 Special return values for mutex-handling functions 79](#_Toc111203248)

[5.1.6 All other Cryptoki function return values 79](#_Toc111203249)

[5.1.7 More on relative priorities of Cryptoki errors 84](#_Toc111203250)

[5.1.8 Error code “gotchas” 85](#_Toc111203251)

[5.2 Conventions for functions returning output in a variable-length buffer 85](#_Toc111203252)

[5.3 Disclaimer concerning sample code 86](#_Toc111203253)

[5.4 General-purpose functions 86](#_Toc111203254)

[5.4.1 C\_Initialize 86](#_Toc111203255)

[5.4.2 C\_Finalize 87](#_Toc111203256)

[5.4.3 C\_GetInfo 87](#_Toc111203257)

[5.4.4 C\_GetFunctionList 88](#_Toc111203258)

[5.4.5 C\_GetInterfaceList 89](#_Toc111203259)

[5.4.6 C\_GetInterface 90](#_Toc111203260)

[5.5 Slot and token management functions 92](#_Toc111203261)

[5.5.1 C\_GetSlotList 92](#_Toc111203262)

[5.5.2 C\_GetSlotInfo 93](#_Toc111203263)

[5.5.3 C\_GetTokenInfo 94](#_Toc111203264)

[5.5.4 C\_WaitForSlotEvent 95](#_Toc111203265)

[5.5.5 C\_GetMechanismList 96](#_Toc111203266)

[5.5.6 C\_GetMechanismInfo 97](#_Toc111203267)

[5.5.7 C\_InitToken 97](#_Toc111203268)

[5.5.8 C\_InitPIN 99](#_Toc111203269)

[5.5.9 C\_SetPIN 99](#_Toc111203270)

[5.6 Session management functions 100](#_Toc111203271)

[5.6.1 C\_OpenSession 101](#_Toc111203272)

[5.6.2 C\_CloseSession 101](#_Toc111203273)

[5.6.3 C\_CloseAllSessions 102](#_Toc111203274)

[5.6.4 C\_GetSessionInfo 103](#_Toc111203275)

[5.6.5 C\_SessionCancel 103](#_Toc111203276)

[5.6.6 C\_GetOperationState 105](#_Toc111203277)

[5.6.7 C\_SetOperationState 106](#_Toc111203278)

[5.6.8 C\_Login 108](#_Toc111203279)

[5.6.9 C\_LoginUser 109](#_Toc111203280)

[5.6.10 C\_Logout 110](#_Toc111203281)

[5.7 Object management functions 111](#_Toc111203282)

[5.7.1 C\_CreateObject 111](#_Toc111203283)

[5.7.2 C\_CopyObject 113](#_Toc111203284)

[5.7.3 C\_DestroyObject 114](#_Toc111203285)

[5.7.4 C\_GetObjectSize 115](#_Toc111203286)

[5.7.5 C\_GetAttributeValue 116](#_Toc111203287)

[5.7.6 C\_SetAttributeValue 118](#_Toc111203288)

[5.7.7 C\_FindObjectsInit 119](#_Toc111203289)

[5.7.8 C\_FindObjects 119](#_Toc111203290)

[5.7.9 C\_FindObjectsFinal 120](#_Toc111203291)

[5.8 Encryption functions 120](#_Toc111203292)

[5.8.1 C\_EncryptInit 120](#_Toc111203293)

[5.8.2 C\_Encrypt 121](#_Toc111203294)

[5.8.3 C\_EncryptUpdate 122](#_Toc111203295)

[5.8.4 C\_EncryptFinal 122](#_Toc111203296)

[5.9 Message-based encryption functions 124](#_Toc111203297)

[5.9.1 C\_MessageEncryptInit 124](#_Toc111203298)

[5.9.2 C\_EncryptMessage 125](#_Toc111203299)

[5.9.3 C\_EncryptMessageBegin 125](#_Toc111203300)

[5.9.4 C\_EncryptMessageNext 126](#_Toc111203301)

[5.9.5 C\_MessageEncryptFinal 127](#_Toc111203302)

[5.10 Decryption functions 129](#_Toc111203303)

[5.10.1 C\_DecryptInit 129](#_Toc111203304)

[5.10.2 C\_Decrypt 129](#_Toc111203305)

[5.10.3 C\_DecryptUpdate 130](#_Toc111203306)

[5.10.4 C\_DecryptFinal 131](#_Toc111203307)

[5.11 Message-based decryption functions 132](#_Toc111203308)

[5.11.1 C\_MessageDecryptInit 132](#_Toc111203309)

[5.11.2 C\_DecryptMessage 133](#_Toc111203310)

[5.11.3 C\_DecryptMessageBegin 134](#_Toc111203311)

[5.11.4 C\_DecryptMessageNext 134](#_Toc111203312)

[5.11.5 C\_MessageDecryptFinal 135](#_Toc111203313)

[5.12 Message digesting functions 135](#_Toc111203314)

[5.12.1 C\_DigestInit 136](#_Toc111203315)

[5.12.2 C\_Digest 136](#_Toc111203316)

[5.12.3 C\_DigestUpdate 137](#_Toc111203317)

[5.12.4 C\_DigestKey 137](#_Toc111203318)

[5.12.5 C\_DigestFinal 137](#_Toc111203319)

[5.13 Signing and MACing functions 139](#_Toc111203320)

[5.13.1 C\_SignInit 139](#_Toc111203321)

[5.13.2 C\_Sign 139](#_Toc111203322)

[5.13.3 C\_SignUpdate 140](#_Toc111203323)

[5.13.4 C\_SignFinal 140](#_Toc111203324)

[5.13.5 C\_SignRecoverInit 141](#_Toc111203325)

[5.13.6 C\_SignRecover 142](#_Toc111203326)

[5.14 Message-based signing and MACing functions 143](#_Toc111203327)

[5.14.1 C\_MessageSignInit 143](#_Toc111203328)

[5.14.2 C\_SignMessage 143](#_Toc111203329)

[5.14.3 C\_SignMessageBegin 144](#_Toc111203330)

[5.14.4 C\_SignMessageNext 145](#_Toc111203331)

[5.14.5 C\_MessageSignFinal 145](#_Toc111203332)

[5.15 Functions for verifying signatures and MACs 146](#_Toc111203333)

[5.15.1 C\_VerifyInit 146](#_Toc111203334)

[5.15.2 C\_Verify 146](#_Toc111203335)

[5.15.3 C\_VerifyUpdate 147](#_Toc111203336)

[5.15.4 C\_VerifyFinal 147](#_Toc111203337)

[5.15.5 C\_VerifyRecoverInit 148](#_Toc111203338)

[5.15.6 C\_VerifyRecover 149](#_Toc111203339)

[5.16 Message-based functions for verifying signatures and MACs 150](#_Toc111203340)

[5.16.1 C\_MessageVerifyInit 150](#_Toc111203341)

[5.16.2 C\_VerifyMessage 150](#_Toc111203342)

[5.16.3 C\_VerifyMessageBegin 151](#_Toc111203343)

[5.16.4 C\_VerifyMessageNext 152](#_Toc111203344)

[5.16.5 C\_MessageVerifyFinal 152](#_Toc111203345)

[5.17 Dual-function cryptographic functions 153](#_Toc111203346)

[5.17.1 C\_DigestEncryptUpdate 153](#_Toc111203347)

[5.17.2 C\_DecryptDigestUpdate 155](#_Toc111203348)

[5.17.3 C\_SignEncryptUpdate 158](#_Toc111203349)

[5.17.4 C\_DecryptVerifyUpdate 161](#_Toc111203350)

[5.18 Key management functions 163](#_Toc111203351)

[5.18.1 C\_GenerateKey 163](#_Toc111203352)

[5.18.2 C\_GenerateKeyPair 165](#_Toc111203353)

[5.18.3 C\_WrapKey 166](#_Toc111203354)

[5.18.4 C\_UnwrapKey 168](#_Toc111203355)

[5.18.5 C\_DeriveKey 169](#_Toc111203356)

[5.19 Random number generation functions 171](#_Toc111203357)

[5.19.1 C\_SeedRandom 171](#_Toc111203358)

[5.19.2 C\_GenerateRandom 172](#_Toc111203359)

[5.20 Parallel function management functions 173](#_Toc111203360)

[5.20.1 C\_GetFunctionStatus 173](#_Toc111203361)

[5.20.2 C\_CancelFunction 173](#_Toc111203362)

[5.21 Callback functions 173](#_Toc111203363)

[5.21.1 Surrender callbacks 173](#_Toc111203364)

[5.21.2 Vendor-defined callbacks 174](#_Toc111203365)

[6 Mechanisms 175](#_Toc111203366)

[6.1 RSA 175](#_Toc111203367)

[6.1.1 Definitions 175](#_Toc111203368)

[6.1.2 RSA public key objects 176](#_Toc111203369)

[6.1.3 RSA private key objects 177](#_Toc111203370)

[6.1.4 PKCS #1 RSA key pair generation 179](#_Toc111203371)

[6.1.5 X9.31 RSA key pair generation 179](#_Toc111203372)

[6.1.6 PKCS #1 v1.5 RSA 179](#_Toc111203373)

[6.1.7 PKCS #1 RSA OAEP mechanism parameters 180](#_Toc111203374)

[6.1.8 PKCS #1 RSA OAEP 182](#_Toc111203375)

[6.1.9 PKCS #1 RSA PSS mechanism parameters 182](#_Toc111203376)

[6.1.10 PKCS #1 RSA PSS 183](#_Toc111203377)

[6.1.11 ISO/IEC 9796 RSA 183](#_Toc111203378)

[6.1.12 X.509 (raw) RSA 184](#_Toc111203379)

[6.1.13 ANSI X9.31 RSA 185](#_Toc111203380)

[6.1.14 PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPE-MD 128 or RIPE-MD 160 186](#_Toc111203381)

[6.1.15 PKCS #1 v1.5 RSA signature with SHA-224 186](#_Toc111203382)

[6.1.16 PKCS #1 RSA PSS signature with SHA-224 186](#_Toc111203383)

[6.1.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512 186](#_Toc111203384)

[6.1.18 PKCS #1 v1.5 RSA signature with SHA3 187](#_Toc111203385)

[6.1.19 PKCS #1 RSA PSS signature with SHA3 187](#_Toc111203386)

[6.1.20 ANSI X9.31 RSA signature with SHA-1 187](#_Toc111203387)

[6.1.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA 188](#_Toc111203388)

[6.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP 188](#_Toc111203389)

[6.1.23 RSA AES KEY WRAP 189](#_Toc111203390)

[6.1.24 RSA AES KEY WRAP mechanism parameters 190](#_Toc111203391)

[6.1.25 FIPS 186-4 190](#_Toc111203392)

[6.2 DSA 190](#_Toc111203393)

[6.2.1 Definitions 191](#_Toc111203394)

[6.2.2 DSA public key objects 192](#_Toc111203395)

[6.2.3 DSA Key Restrictions 193](#_Toc111203396)

[6.2.4 DSA private key objects 193](#_Toc111203397)

[6.2.5 DSA domain parameter objects 194](#_Toc111203398)

[6.2.6 DSA key pair generation 195](#_Toc111203399)

[6.2.7 DSA domain parameter generation 195](#_Toc111203400)

[6.2.8 DSA probabilistic domain parameter generation 195](#_Toc111203401)

[6.2.9 DSA Shawe-Taylor domain parameter generation 196](#_Toc111203402)

[6.2.10 DSA base domain parameter generation 196](#_Toc111203403)

[6.2.11 DSA without hashing 196](#_Toc111203404)

[6.2.12 DSA with SHA-1 197](#_Toc111203405)

[6.2.13 FIPS 186-4 197](#_Toc111203406)

[6.2.14 DSA with SHA-224 197](#_Toc111203407)

[6.2.15 DSA with SHA-256 198](#_Toc111203408)

[6.2.16 DSA with SHA-384 198](#_Toc111203409)

[6.2.17 DSA with SHA-512 199](#_Toc111203410)

[6.2.18 DSA with SHA3-224 199](#_Toc111203411)

[6.2.19 DSA with SHA3-256 200](#_Toc111203412)

[6.2.20 DSA with SHA3-384 200](#_Toc111203413)

[6.2.21 DSA with SHA3-512 200](#_Toc111203414)

[6.3 Elliptic Curve 201](#_Toc111203415)

[6.3.1 EC Signatures 203](#_Toc111203416)

[6.3.2 Definitions 203](#_Toc111203417)

[6.3.3 Short Weierstrass Elliptic Curve public key objects 204](#_Toc111203418)

[6.3.4 Short Weierstrass Elliptic Curve private key objects 205](#_Toc111203419)

[6.3.5 Edwards Elliptic Curve public key objects 206](#_Toc111203420)

[6.3.6 Edwards Elliptic Curve private key objects 207](#_Toc111203421)

[6.3.7 Montgomery Elliptic Curve public key objects 208](#_Toc111203422)

[6.3.8 Montgomery Elliptic Curve private key objects 209](#_Toc111203423)

[6.3.9 Elliptic Curve key pair generation 210](#_Toc111203424)

[6.3.10 Edwards Elliptic Curve key pair generation 211](#_Toc111203425)

[6.3.11 Montgomery Elliptic Curve key pair generation 211](#_Toc111203426)

[6.3.12 ECDSA without hashing 212](#_Toc111203427)

[6.3.13 ECDSA with hashing 212](#_Toc111203428)

[6.3.14 EdDSA 213](#_Toc111203429)

[6.3.15 XEdDSA 213](#_Toc111203430)

[6.3.16 EC mechanism parameters 214](#_Toc111203431)

[6.3.17 Elliptic Curve Diffie-Hellman key derivation 219](#_Toc111203432)

[6.3.18 Elliptic Curve Diffie-Hellman with cofactor key derivation 219](#_Toc111203433)

[6.3.19 Elliptic Curve Menezes-Qu-Vanstone key derivation 220](#_Toc111203434)

[6.3.20 ECDH AES KEY WRAP 221](#_Toc111203435)

[6.3.21 ECDH AES KEY WRAP mechanism parameters 222](#_Toc111203436)

[6.3.22 FIPS 186-4 223](#_Toc111203437)

[6.4 Diffie-Hellman 223](#_Toc111203438)

[6.4.1 Definitions 223](#_Toc111203439)

[6.4.2 Diffie-Hellman public key objects 224](#_Toc111203440)

[6.4.3 X9.42 Diffie-Hellman public key objects 224](#_Toc111203441)

[6.4.4 Diffie-Hellman private key objects 225](#_Toc111203442)

[6.4.5 X9.42 Diffie-Hellman private key objects 226](#_Toc111203443)

[6.4.6 Diffie-Hellman domain parameter objects 227](#_Toc111203444)

[6.4.7 X9.42 Diffie-Hellman domain parameters objects 227](#_Toc111203445)

[6.4.8 PKCS #3 Diffie-Hellman key pair generation 228](#_Toc111203446)

[6.4.9 PKCS #3 Diffie-Hellman domain parameter generation 229](#_Toc111203447)

[6.4.10 PKCS #3 Diffie-Hellman key derivation 229](#_Toc111203448)

[6.4.11 X9.42 Diffie-Hellman mechanism parameters 230](#_Toc111203449)

[6.4.12 X9.42 Diffie-Hellman key pair generation 232](#_Toc111203450)

[6.4.13 X9.42 Diffie-Hellman domain parameter generation 233](#_Toc111203451)

[6.4.14 X9.42 Diffie-Hellman key derivation 233](#_Toc111203452)

[6.4.15 X9.42 Diffie-Hellman hybrid key derivation 234](#_Toc111203453)

[6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation 234](#_Toc111203454)

[6.5 Extended Triple Diffie-Hellman (x3dh) 235](#_Toc111203455)

[6.5.1 Definitions 235](#_Toc111203456)

[6.5.2 Extended Triple Diffie-Hellman key objects 235](#_Toc111203457)

[6.5.3 Initiating an Extended Triple Diffie-Hellman key exchange 235](#_Toc111203458)

[6.5.4 Responding to an Extended Triple Diffie-Hellman key exchange 236](#_Toc111203459)

[6.5.5 Extended Triple Diffie-Hellman parameters 237](#_Toc111203460)

[6.6 Double Ratchet 238](#_Toc111203461)

[6.6.1 Definitions 238](#_Toc111203462)

[6.6.2 Double Ratchet secret key objects 238](#_Toc111203463)

[6.6.3 Double Ratchet key derivation 239](#_Toc111203464)

[6.6.4 Double Ratchet Encryption mechanism 240](#_Toc111203465)

[6.6.5 Double Ratchet parameters 241](#_Toc111203466)

[6.7 Wrapping/unwrapping private keys 241](#_Toc111203467)

[6.8 Generic secret key 243](#_Toc111203468)

[6.8.1 Definitions 244](#_Toc111203469)

[6.8.2 Generic secret key objects 244](#_Toc111203470)

[6.8.3 Generic secret key generation 245](#_Toc111203471)

[6.9 HMAC mechanisms 245](#_Toc111203472)

[6.9.1 General block cipher mechanism parameters 245](#_Toc111203473)

[6.10 AES 245](#_Toc111203474)

[6.10.1 Definitions 246](#_Toc111203475)

[6.10.2 AES secret key objects 246](#_Toc111203476)

[6.10.3 AES key generation 247](#_Toc111203477)

[6.10.4 AES-ECB 247](#_Toc111203478)

[6.10.5 AES-CBC 248](#_Toc111203479)

[6.10.6 AES-CBC with PKCS padding 248](#_Toc111203480)

[6.10.7 AES-OFB 249](#_Toc111203481)

[6.10.8 AES-CFB 249](#_Toc111203482)

[6.10.9 General-length AES-MAC 250](#_Toc111203483)

[6.10.10 AES-MAC 250](#_Toc111203484)

[6.10.11 AES-XCBC-MAC 250](#_Toc111203485)

[6.10.12 AES-XCBC-MAC-96 251](#_Toc111203486)

[6.11 AES with Counter 251](#_Toc111203487)

[6.11.1 Definitions 251](#_Toc111203488)

[6.11.2 AES with Counter mechanism parameters 251](#_Toc111203489)

[6.11.3 AES with Counter Encryption / Decryption 252](#_Toc111203490)

[6.12 AES CBC with Cipher Text Stealing CTS 252](#_Toc111203491)

[6.12.1 Definitions 252](#_Toc111203492)

[6.12.2 AES CTS mechanism parameters 253](#_Toc111203493)

[6.13 Additional AES Mechanisms 253](#_Toc111203494)

[6.13.1 Definitions 253](#_Toc111203495)

[6.13.2 AES-GCM Authenticated Encryption / Decryption 253](#_Toc111203496)

[6.13.3 AES-CCM authenticated Encryption / Decryption 255](#_Toc111203497)

[6.13.4 AES-GMAC 257](#_Toc111203498)

[6.13.5 AES GCM and CCM Mechanism parameters 257](#_Toc111203499)

[6.14 AES CMAC 260](#_Toc111203500)

[6.14.1 Definitions 260](#_Toc111203501)

[6.14.2 Mechanism parameters 260](#_Toc111203502)

[6.14.3 General-length AES-CMAC 261](#_Toc111203503)

[6.14.4 AES-CMAC 261](#_Toc111203504)

[6.15 AES XTS 261](#_Toc111203505)

[6.15.1 Definitions 262](#_Toc111203506)

[6.15.2 AES-XTS secret key objects 262](#_Toc111203507)

[6.15.3 AES-XTS key generation 262](#_Toc111203508)

[6.15.4 AES-XTS 262](#_Toc111203509)

[6.16 AES Key Wrap 263](#_Toc111203510)

[6.16.1 Definitions 263](#_Toc111203511)

[6.16.2 AES Key Wrap Mechanism parameters 263](#_Toc111203512)

[6.16.3 AES Key Wrap 263](#_Toc111203513)

[6.17 Key derivation by data encryption – DES & AES 264](#_Toc111203514)

[6.17.1 Definitions 264](#_Toc111203515)

[6.17.2 Mechanism Parameters 265](#_Toc111203516)

[6.17.3 Mechanism Description 265](#_Toc111203517)

[6.18 Double and Triple-length DES 266](#_Toc111203518)

[6.18.1 Definitions 266](#_Toc111203519)

[6.18.2 DES2 secret key objects 266](#_Toc111203520)

[6.18.3 DES3 secret key objects 267](#_Toc111203521)

[6.18.4 Double-length DES key generation 268](#_Toc111203522)

[6.18.5 Triple-length DES Order of Operations 268](#_Toc111203523)

[6.18.6 Triple-length DES in CBC Mode 268](#_Toc111203524)

[6.18.7 DES and Triple length DES in OFB Mode 268](#_Toc111203525)

[6.18.8 DES and Triple length DES in CFB Mode 269](#_Toc111203526)

[6.19 Double and Triple-length DES CMAC 269](#_Toc111203527)

[6.19.1 Definitions 269](#_Toc111203528)

[6.19.2 Mechanism parameters 270](#_Toc111203529)

[6.19.3 General-length DES3-MAC 270](#_Toc111203530)

[6.19.4 DES3-CMAC 270](#_Toc111203531)

[6.20 SHA-1 270](#_Toc111203532)

[6.20.1 Definitions 271](#_Toc111203533)

[6.20.2 SHA-1 digest 271](#_Toc111203534)

[6.20.3 General-length SHA-1-HMAC 271](#_Toc111203535)

[6.20.4 SHA-1-HMAC 272](#_Toc111203536)

[6.20.5 SHA-1 key derivation 272](#_Toc111203537)

[6.20.6 SHA-1 HMAC key generation 272](#_Toc111203538)

[6.21 SHA-224 273](#_Toc111203539)

[6.21.1 Definitions 273](#_Toc111203540)

[6.21.2 SHA-224 digest 273](#_Toc111203541)

[6.21.3 General-length SHA-224-HMAC 273](#_Toc111203542)

[6.21.4 SHA-224-HMAC 274](#_Toc111203543)

[6.21.5 SHA-224 key derivation 274](#_Toc111203544)

[6.21.6 SHA-224 HMAC key generation 274](#_Toc111203545)

[6.22 SHA-256 274](#_Toc111203546)

[6.22.1 Definitions 275](#_Toc111203547)

[6.22.2 SHA-256 digest 275](#_Toc111203548)

[6.22.3 General-length SHA-256-HMAC 275](#_Toc111203549)

[6.22.4 SHA-256-HMAC 276](#_Toc111203550)

[6.22.5 SHA-256 key derivation 276](#_Toc111203551)

[6.22.6 SHA-256 HMAC key generation 276](#_Toc111203552)

[6.23 SHA-384 276](#_Toc111203553)

[6.23.1 Definitions 276](#_Toc111203554)

[6.23.2 SHA-384 digest 277](#_Toc111203555)

[6.23.3 General-length SHA-384-HMAC 277](#_Toc111203556)

[6.23.4 SHA-384-HMAC 277](#_Toc111203557)

[6.23.5 SHA-384 key derivation 277](#_Toc111203558)

[6.23.6 SHA-384 HMAC key generation 278](#_Toc111203559)

[6.24 SHA-512 278](#_Toc111203560)

[6.24.1 Definitions 278](#_Toc111203561)

[6.24.2 SHA-512 digest 278](#_Toc111203562)

[6.24.3 General-length SHA-512-HMAC 279](#_Toc111203563)

[6.24.4 SHA-512-HMAC 279](#_Toc111203564)

[6.24.5 SHA-512 key derivation 279](#_Toc111203565)

[6.24.6 SHA-512 HMAC key generation 279](#_Toc111203566)

[6.25 SHA-512/224 280](#_Toc111203567)

[6.25.1 Definitions 280](#_Toc111203568)

[6.25.2 SHA-512/224 digest 280](#_Toc111203569)

[6.25.3 General-length SHA-512/224-HMAC 280](#_Toc111203570)

[6.25.4 SHA-512/224-HMAC 281](#_Toc111203571)

[6.25.5 SHA-512/224 key derivation 281](#_Toc111203572)

[6.25.6 SHA-512/224 HMAC key generation 281](#_Toc111203573)

[6.26 SHA-512/256 281](#_Toc111203574)

[6.26.1 Definitions 282](#_Toc111203575)

[6.26.2 SHA-512/256 digest 282](#_Toc111203576)

[6.26.3 General-length SHA-512/256-HMAC 282](#_Toc111203577)

[6.26.4 SHA-512/256-HMAC 283](#_Toc111203578)

[6.26.5 SHA-512/256 key derivation 283](#_Toc111203579)

[6.26.6 SHA-512/256 HMAC key generation 283](#_Toc111203580)

[6.27 SHA-512/t 283](#_Toc111203581)

[6.27.1 Definitions 284](#_Toc111203582)

[6.27.2 SHA-512/t digest 284](#_Toc111203583)

[6.27.3 General-length SHA-512/t-HMAC 284](#_Toc111203584)

[6.27.4 SHA-512/t-HMAC 284](#_Toc111203585)

[6.27.5 SHA-512/t key derivation 285](#_Toc111203586)

[6.27.6 SHA-512/t HMAC key generation 285](#_Toc111203587)

[6.28 SHA3-224 285](#_Toc111203588)

[6.28.1 Definitions 285](#_Toc111203589)

[6.28.2 SHA3-224 digest 286](#_Toc111203590)

[6.28.3 General-length SHA3-224-HMAC 286](#_Toc111203591)

[6.28.4 SHA3-224-HMAC 286](#_Toc111203592)

[6.28.5 SHA3-224 key derivation 286](#_Toc111203593)

[6.28.6 SHA3-224 HMAC key generation 286](#_Toc111203594)

[6.29 SHA3-256 287](#_Toc111203595)

[6.29.1 Definitions 287](#_Toc111203596)

[6.29.2 SHA3-256 digest 287](#_Toc111203597)

[6.29.3 General-length SHA3-256-HMAC 287](#_Toc111203598)

[6.29.4 SHA3-256-HMAC 288](#_Toc111203599)

[6.29.5 SHA3-256 key derivation 288](#_Toc111203600)

[6.29.6 SHA3-256 HMAC key generation 288](#_Toc111203601)

[6.30 SHA3-384 288](#_Toc111203602)

[6.30.1 Definitions 289](#_Toc111203603)

[6.30.2 SHA3-384 digest 289](#_Toc111203604)

[6.30.3 General-length SHA3-384-HMAC 289](#_Toc111203605)

[6.30.4 SHA3-384-HMAC 290](#_Toc111203606)

[6.30.5 SHA3-384 key derivation 290](#_Toc111203607)

[6.30.6 SHA3-384 HMAC key generation 290](#_Toc111203608)

[6.31 SHA3-512 290](#_Toc111203609)

[6.31.1 Definitions 290](#_Toc111203610)

[6.31.2 SHA3-512 digest 291](#_Toc111203611)

[6.31.3 General-length SHA3-512-HMAC 291](#_Toc111203612)

[6.31.4 SHA3-512-HMAC 291](#_Toc111203613)

[6.31.5 SHA3-512 key derivation 291](#_Toc111203614)

[6.31.6 SHA3-512 HMAC key generation 291](#_Toc111203615)

[6.32 SHAKE 292](#_Toc111203616)

[6.32.1 Definitions 292](#_Toc111203617)

[6.32.2 SHAKE Key Derivation 292](#_Toc111203618)

[6.33 BLAKE2B-160 293](#_Toc111203619)

[6.33.1 Definitions 293](#_Toc111203620)

[6.33.2 BLAKE2B-160 digest 293](#_Toc111203621)

[6.33.3 General-length BLAKE2B-160-HMAC 293](#_Toc111203622)

[6.33.4 BLAKE2B-160-HMAC 294](#_Toc111203623)

[6.33.5 BLAKE2B-160 key derivation 294](#_Toc111203624)

[6.33.6 BLAKE2B-160 HMAC key generation 294](#_Toc111203625)

[6.34 BLAKE2B-256 294](#_Toc111203626)

[6.34.1 Definitions 295](#_Toc111203627)

[6.34.2 BLAKE2B-256 digest 295](#_Toc111203628)

[6.34.3 General-length BLAKE2B-256-HMAC 295](#_Toc111203629)

[6.34.4 BLAKE2B-256-HMAC 296](#_Toc111203630)

[6.34.5 BLAKE2B-256 key derivation 296](#_Toc111203631)

[6.34.6 BLAKE2B-256 HMAC key generation 296](#_Toc111203632)

[6.35 BLAKE2B-384 296](#_Toc111203633)

[6.35.1 Definitions 297](#_Toc111203634)

[6.35.2 BLAKE2B-384 digest 297](#_Toc111203635)

[6.35.3 General-length BLAKE2B-384-HMAC 297](#_Toc111203636)

[6.35.4 BLAKE2B-384-HMAC 298](#_Toc111203637)

[6.35.5 BLAKE2B-384 key derivation 298](#_Toc111203638)

[6.35.6 BLAKE2B-384 HMAC key generation 298](#_Toc111203639)

[6.36 BLAKE2B-512 298](#_Toc111203640)

[6.36.1 Definitions 299](#_Toc111203641)

[6.36.2 BLAKE2B-512 digest 299](#_Toc111203642)

[6.36.3 General-length BLAKE2B-512-HMAC 299](#_Toc111203643)

[6.36.4 BLAKE2B-512-HMAC 300](#_Toc111203644)

[6.36.5 BLAKE2B-512 key derivation 300](#_Toc111203645)

[6.36.6 BLAKE2B-512 HMAC key generation 300](#_Toc111203646)

[6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE) 300](#_Toc111203647)

[6.37.1 Definitions 301](#_Toc111203648)

[6.37.2 Password-based encryption/authentication mechanism parameters 301](#_Toc111203649)

[6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters 302](#_Toc111203650)

[6.37.4 PKCS #5 PBKD2 key generation 304](#_Toc111203651)

[6.38 PKCS #12 password-based encryption/authentication mechanisms 304](#_Toc111203652)

[6.38.1 SHA-1-PBE for 3-key triple-DES-CBC 305](#_Toc111203653)

[6.38.2 SHA-1-PBE for 2-key triple-DES-CBC 305](#_Toc111203654)

[6.38.3 SHA-1-PBA for SHA-1-HMAC 305](#_Toc111203655)

[6.39 SSL 305](#_Toc111203656)

[6.39.1 Definitions 306](#_Toc111203657)

[6.39.2 SSL mechanism parameters 306](#_Toc111203658)

[6.39.3 Pre-master key generation 308](#_Toc111203659)

[6.39.4 Master key derivation 308](#_Toc111203660)

[6.39.5 Master key derivation for Diffie-Hellman 309](#_Toc111203661)

[6.39.6 Key and MAC derivation 310](#_Toc111203662)

[6.39.7 MD5 MACing in SSL 3.0 311](#_Toc111203663)

[6.39.8 SHA-1 MACing in SSL 3.0 311](#_Toc111203664)

[6.40 TLS 1.2 Mechanisms 311](#_Toc111203665)

[6.40.1 Definitions 312](#_Toc111203666)

[6.40.2 TLS 1.2 mechanism parameters 312](#_Toc111203667)

[6.40.3 TLS MAC 315](#_Toc111203668)

[6.40.4 Master key derivation 315](#_Toc111203669)

[6.40.5 Master key derivation for Diffie-Hellman 316](#_Toc111203670)

[6.40.6 Key and MAC derivation 317](#_Toc111203671)

[6.40.7 CKM\_TLS12\_KEY\_SAFE\_DERIVE 317](#_Toc111203672)

[6.40.8 Generic Key Derivation using the TLS PRF 318](#_Toc111203673)

[6.40.9 Generic Key Derivation using the TLS12 PRF 318](#_Toc111203674)

[6.41 WTLS 319](#_Toc111203675)

[6.41.1 Definitions 319](#_Toc111203676)

[6.41.2 WTLS mechanism parameters 320](#_Toc111203677)

[6.41.3 Pre master secret key generation for RSA key exchange suite 322](#_Toc111203678)

[6.41.4 Master secret key derivation 323](#_Toc111203679)

[6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography 323](#_Toc111203680)

[6.41.6 WTLS PRF (pseudorandom function) 324](#_Toc111203681)

[6.41.7 Server Key and MAC derivation 324](#_Toc111203682)

[6.41.8 Client key and MAC derivation 325](#_Toc111203683)

[6.42 SP 800-108 Key Derivation 326](#_Toc111203684)

[6.42.1 Definitions 326](#_Toc111203685)

[6.42.2 Mechanism Parameters 327](#_Toc111203686)

[6.42.3 Counter Mode KDF 332](#_Toc111203687)

[6.42.4 Feedback Mode KDF 332](#_Toc111203688)

[6.42.5 Double Pipeline Mode KDF 333](#_Toc111203689)

[6.42.6 Deriving Additional Keys 334](#_Toc111203690)

[6.42.7 Key Derivation Attribute Rules 335](#_Toc111203691)

[6.42.8 Constructing PRF Input Data 335](#_Toc111203692)

[6.42.8.1 Sample Counter Mode KDF 335](#_Toc111203693)

[6.42.8.2 Sample SCP03 Counter Mode KDF 336](#_Toc111203694)

[6.42.8.3 Sample Feedback Mode KDF 337](#_Toc111203695)

[6.42.8.4 Sample Double-Pipeline Mode KDF 339](#_Toc111203696)

[6.43 Miscellaneous simple key derivation mechanisms 340](#_Toc111203697)

[6.43.1 Definitions 340](#_Toc111203698)

[6.43.2 Parameters for miscellaneous simple key derivation mechanisms 340](#_Toc111203699)

[6.43.3 Concatenation of a base key and another key 341](#_Toc111203700)

[6.43.4 Concatenation of a base key and data 341](#_Toc111203701)

[6.43.5 Concatenation of data and a base key 342](#_Toc111203702)

[6.43.6 XORing of a key and data 343](#_Toc111203703)

[6.43.7 Extraction of one key from another key 344](#_Toc111203704)

[6.44 CMS 344](#_Toc111203705)

[6.44.1 Definitions 345](#_Toc111203706)

[6.44.2 CMS Signature Mechanism Objects 345](#_Toc111203707)

[6.44.3 CMS mechanism parameters 345](#_Toc111203708)

[6.44.4 CMS signatures 346](#_Toc111203709)

[6.45 Blowfish 347](#_Toc111203710)

[6.45.1 Definitions 348](#_Toc111203711)

[6.45.2 BLOWFISH secret key objects 348](#_Toc111203712)

[6.45.3 Blowfish key generation 349](#_Toc111203713)

[6.45.4 Blowfish-CBC 349](#_Toc111203714)

[6.45.5 Blowfish-CBC with PKCS padding 349](#_Toc111203715)

[6.46 Twofish 350](#_Toc111203716)

[6.46.1 Definitions 350](#_Toc111203717)

[6.46.2 Twofish secret key objects 350](#_Toc111203718)

[6.46.3 Twofish key generation 351](#_Toc111203719)

[6.46.4 Twofish -CBC 351](#_Toc111203720)

[6.46.5 Twofish-CBC with PKCS padding 351](#_Toc111203721)

[6.47 CAMELLIA 351](#_Toc111203722)

[6.47.1 Definitions 352](#_Toc111203723)

[6.47.2 Camellia secret key objects 352](#_Toc111203724)

[6.47.3 Camellia key generation 353](#_Toc111203725)

[6.47.4 Camellia-ECB 353](#_Toc111203726)

[6.47.5 Camellia-CBC 354](#_Toc111203727)

[6.47.6 Camellia-CBC with PKCS padding 354](#_Toc111203728)

[6.47.7 CAMELLIA with Counter mechanism parameters 355](#_Toc111203729)

[6.47.8 General-length Camellia-MAC 356](#_Toc111203730)

[6.47.9 Camellia-MAC 356](#_Toc111203731)

[6.48 Key derivation by data encryption - Camellia 357](#_Toc111203732)

[6.48.1 Definitions 357](#_Toc111203733)

[6.48.2 Mechanism Parameters 357](#_Toc111203734)

[6.49 ARIA 357](#_Toc111203735)

[6.49.1 Definitions 358](#_Toc111203736)

[6.49.2 Aria secret key objects 358](#_Toc111203737)

[6.49.3 ARIA key generation 359](#_Toc111203738)

[6.49.4 ARIA-ECB 359](#_Toc111203739)

[6.49.5 ARIA-CBC 359](#_Toc111203740)

[6.49.6 ARIA-CBC with PKCS padding 360](#_Toc111203741)

[6.49.7 General-length ARIA-MAC 361](#_Toc111203742)

[6.49.8 ARIA-MAC 361](#_Toc111203743)

[6.50 Key derivation by data encryption - ARIA 361](#_Toc111203744)

[6.50.1 Definitions 362](#_Toc111203745)

[6.50.2 Mechanism Parameters 362](#_Toc111203746)

[6.51 SEED 362](#_Toc111203747)

[6.51.1 Definitions 363](#_Toc111203748)

[6.51.2 SEED secret key objects 363](#_Toc111203749)

[6.51.3 SEED key generation 364](#_Toc111203750)

[6.51.4 SEED-ECB 364](#_Toc111203751)

[6.51.5 SEED-CBC 364](#_Toc111203752)

[6.51.6 SEED-CBC with PKCS padding 364](#_Toc111203753)

[6.51.7 General-length SEED-MAC 365](#_Toc111203754)

[6.51.8 SEED-MAC 365](#_Toc111203755)

[6.52 Key derivation by data encryption - SEED 365](#_Toc111203756)

[6.52.1 Definitions 365](#_Toc111203757)

[6.52.2 Mechanism Parameters 365](#_Toc111203758)

[6.53 OTP 366](#_Toc111203759)

[6.53.1 Usage overview 366](#_Toc111203760)

[6.53.2 Case 1: Generation of OTP values 366](#_Toc111203761)

[6.53.3 Case 2: Verification of provided OTP values 367](#_Toc111203762)

[6.53.4 Case 3: Generation of OTP keys 367](#_Toc111203763)

[6.53.5 OTP objects 368](#_Toc111203764)

[6.53.5.1 Key objects 368](#_Toc111203765)

[6.53.6 OTP-related notifications 371](#_Toc111203766)

[6.53.7 OTP mechanisms 371](#_Toc111203767)

[6.53.7.1 OTP mechanism parameters 371](#_Toc111203768)

[6.53.8 RSA SecurID 375](#_Toc111203769)

[6.53.8.1 RSA SecurID secret key objects 375](#_Toc111203770)

[6.53.8.2 RSA SecurID key generation 376](#_Toc111203771)

[6.53.8.3 SecurID OTP generation and validation 376](#_Toc111203772)

[6.53.8.4 Return values 377](#_Toc111203773)

[6.53.9 OATH HOTP 377](#_Toc111203774)

[6.53.9.1 OATH HOTP secret key objects 377](#_Toc111203775)

[6.53.9.2 HOTP key generation 378](#_Toc111203776)

[6.53.9.3 HOTP OTP generation and validation 378](#_Toc111203777)

[6.53.10 ActivIdentity ACTI 378](#_Toc111203778)

[6.53.10.1 ACTI secret key objects 378](#_Toc111203779)

[6.53.10.2 ACTI key generation 379](#_Toc111203780)

[6.53.10.3 ACTI OTP generation and validation 379](#_Toc111203781)

[6.54 CT-KIP 380](#_Toc111203782)

[6.54.1 Principles of Operation 380](#_Toc111203783)

[6.54.2 Mechanisms 380](#_Toc111203784)

[6.54.3 Definitions 381](#_Toc111203785)

[6.54.4 CT-KIP Mechanism parameters 381](#_Toc111203786)

[6.54.5 CT-KIP key derivation 381](#_Toc111203787)

[6.54.6 CT-KIP key wrap and key unwrap 382](#_Toc111203788)

[6.54.7 CT-KIP signature generation 382](#_Toc111203789)

[6.55 GOST 28147-89 382](#_Toc111203790)

[6.55.1 Definitions 382](#_Toc111203791)

[6.55.2 GOST 28147-89 secret key objects 383](#_Toc111203792)

[6.55.3 GOST 28147-89 domain parameter objects 383](#_Toc111203793)

[6.55.4 GOST 28147-89 key generation 384](#_Toc111203794)

[6.55.5 GOST 28147-89-ECB 385](#_Toc111203795)

[6.55.6 GOST 28147-89 encryption mode except ECB 385](#_Toc111203796)

[6.55.7 GOST 28147-89-MAC 386](#_Toc111203797)

[6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89 386](#_Toc111203798)

[6.56 GOST R 34.11-94 387](#_Toc111203799)

[6.56.1 Definitions 387](#_Toc111203800)

[6.56.2 GOST R 34.11-94 domain parameter objects 387](#_Toc111203801)

[6.56.3 GOST R 34.11-94 digest 388](#_Toc111203802)

[6.56.4 GOST R 34.11-94 HMAC 389](#_Toc111203803)

[6.57 GOST R 34.10-2001 389](#_Toc111203804)

[6.57.1 Definitions 389](#_Toc111203805)

[6.57.2 GOST R 34.10-2001 public key objects 390](#_Toc111203806)

[6.57.3 GOST R 34.10-2001 private key objects 391](#_Toc111203807)

[6.57.4 GOST R 34.10-2001 domain parameter objects 393](#_Toc111203808)

[6.57.5 GOST R 34.10-2001 mechanism parameters 394](#_Toc111203809)

[6.57.6 GOST R 34.10-2001 key pair generation 395](#_Toc111203810)

[6.57.7 GOST R 34.10-2001 without hashing 395](#_Toc111203811)

[6.57.8 GOST R 34.10-2001 with GOST R 34.11-94 396](#_Toc111203812)

[6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001 396](#_Toc111203813)

[6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys 396](#_Toc111203814)

[6.58 ChaCha20 397](#_Toc111203815)

[6.58.1 Definitions 397](#_Toc111203816)

[6.58.2 ChaCha20 secret key objects 397](#_Toc111203817)

[6.58.3 ChaCha20 mechanism parameters 398](#_Toc111203818)

[6.58.4 ChaCha20 key generation 398](#_Toc111203819)

[6.58.5 ChaCha20 mechanism 398](#_Toc111203820)

[6.59 Salsa20 399](#_Toc111203821)

[6.59.1 Definitions 400](#_Toc111203822)

[6.59.2 Salsa20 secret key objects 400](#_Toc111203823)

[6.59.3 Salsa20 mechanism parameters 401](#_Toc111203824)

[6.59.4 Salsa20 key generation 401](#_Toc111203825)

[6.59.5 Salsa20 mechanism 401](#_Toc111203826)

[6.60 Poly1305 402](#_Toc111203827)

[6.60.1 Definitions 402](#_Toc111203828)

[6.60.2 Poly1305 secret key objects 402](#_Toc111203829)

[6.60.3 Poly1305 mechanism 403](#_Toc111203830)

[6.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption 403](#_Toc111203831)

[6.61.1 Definitions 404](#_Toc111203832)

[6.61.2 Usage 404](#_Toc111203833)

[6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters 406](#_Toc111203834)

[6.62 HKDF Mechanisms 407](#_Toc111203835)

[6.62.1 Definitions 407](#_Toc111203836)

[6.62.2 HKDF mechanism parameters 407](#_Toc111203837)

[6.62.3 HKDF derive 408](#_Toc111203838)

[6.62.4 HKDF Data 409](#_Toc111203839)

[6.62.5 HKDF Key gen 409](#_Toc111203840)

[6.63 NULL Mechanism 409](#_Toc111203841)

[6.63.1 Definitions 409](#_Toc111203842)

[6.63.2 CKM\_NULL mechanism parameters 409](#_Toc111203843)

[6.64 IKE Mechanisms 409](#_Toc111203844)

[6.64.1 Definitions 410](#_Toc111203845)

[6.64.2 IKE mechanism parameters 410](#_Toc111203846)

[6.64.3 IKE PRF DERIVE 412](#_Toc111203847)

[6.64.4 IKEv1 PRF DERIVE 413](#_Toc111203848)

[6.64.5 IKEv2 PRF PLUS DERIVE 413](#_Toc111203849)

[6.64.6 IKEv1 Extended Derive 414](#_Toc111203850)

[6.65 HSS 414](#_Toc111203851)

[6.65.1 Definitions 415](#_Toc111203852)

[6.65.2 HSS public key objects 415](#_Toc111203853)

[6.65.3 HSS private key objects 416](#_Toc111203854)

[6.65.4 HSS key pair generation 418](#_Toc111203855)

[6.65.5 HSS without hashing 418](#_Toc111203856)

[7 PKCS #11 Implementation Conformance 420](#_Toc111203857)

[7.1 PKCS#11 Consumer Implementation Conformance 420](#_Toc111203858)

[7.2 PKCS#11 Provider Implementation Conformance 420](#_Toc111203859)

[Appendix A. Acknowledgments 421](#_Toc111203860)

[Appendix B. Manifest constants 423](#_Toc111203861)

[Appendix C. Revision History 424](#_Toc111203862)

[Appendix D. Notices 425](#_Toc111203863)

# Introduction

This document describes the basic PKCS#11 token interface and token behavior.

The PKCS#11 standard specifies an application programming interface (API), called “Cryptoki,” for devices that hold cryptographic information and perform cryptographic functions. Cryptoki follows a simple object based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a “cryptographic token”.

This document specifies the data types and functions available to an application requiring cryptographic services using the ANSI C programming language. The supplier of a Cryptoki library implementation typically provides these data types and functions via ANSI C header files. Generic ANSI C header files for Cryptoki are available from the PKCS#11 web page. This document and up-to-date errata for Cryptoki will also be available from the same place.

Additional documents may provide a generic, language-independent Cryptoki interface and/or bindings between Cryptoki and other programming languages.

Cryptoki isolates an application from the details of the cryptographic device. The application does not have to change to interface to a different type of device or to run in a different environment; thus, the application is portable. How Cryptoki provides this isolation is beyond the scope of this document, although some conventions for the support of multiple types of device will be addressed here and possibly in a separate document.

Details of cryptographic mechanisms (algorithms) may be found in the associated PKCS#11 Mechanisms documents.

## Definitions

For the purposes of this standard, the following definitions apply:

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

**API** Application programming interface.

**Application** Any computer program that calls the Cryptoki interface.

**ASN.1** Abstract Syntax Notation One, as defined in X.680.

**Attribute** A characteristic of an object.

**BER** Basic Encoding Rules, as defined in X.690.

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

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

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

**Certificate** A signed message binding a subject name and a public key, or a subject name and a set of attributes.

**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 5652)

**Cryptographic Device** A device storing cryptographic information and possibly performing cryptographic functions. May be implemented as a smart card, smart disk, PCMCIA card, or with some other technology, including software-only.

**Cryptoki** The Cryptographic Token Interface defined in this standard.

**Cryptoki library** A library that implements the functions specified in this standard.

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

**DER** Distinguished Encoding Rules, as defined in X.690.

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

**DSA** Digital Signature Algorithm, as defined in FIPS PUB 186-4.

**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.

**Mechanism** A process for implementing a cryptographic operation.

**MQV** Menezes-Qu-Vanstone

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

**Object** An item that is stored on a token. May be data, a certificate, or a key.

**PIN** Personal Identification Number.

**PKCS** Public-Key Cryptography Standards.

**PRF** Pseudo random function.

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

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

**Reader** The means by which information is exchanged with a device.

**Session** A logical connection between an application and a token.

**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.

**Slot** A logical reader that potentially contains a token.

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

**Subject Name** The X.500 distinguished name of the entity to which a key is assigned.

**SO** A Security Officer user.

**TLS** Transport Layer Security.

**Token** The logical view of a cryptographic device defined by Cryptoki.

**User** The person using an application that interfaces to Cryptoki.

**UTF-8** Universal Character Set (UCS) transformation format (UTF) that represents ISO 10646 and UNICODE strings with a variable number of octets.

**WTLS** Wireless Transport Layer Security.

## Symbols and abbreviations

The following symbols are used in this standard:

Table , Symbols

|  |  |
| --- | --- |
| **Symbol** | **Definition** |
| N/A | Not applicable |
| R/O | Read-only |
| R/W | Read/write |

The following prefixes are used in this standard:

Table , Prefixes

| **Prefix** | **Description** |
| --- | --- |
| C\_ | Function |
| CK\_ | Data type or general constant |
| CKA\_ | Attribute |
| CKC\_ | Certificate type |
| CKD\_ | Key derivation function |
| CKF\_ | Bit flag |
| CKG\_ | Mask generation function |
| CKH\_ | Hardware feature type |
| CKK\_ | Key type |
| CKM\_ | Mechanism type |
| CKN\_ | Notification |
| CKO\_ | Object class |
| CKP\_ | Pseudo-random function |
| CKS\_ | Session state |
| CKR\_ | Return value |
| CKU\_ | User type |
| CKZ\_ | Salt/Encoding parameter source |
| h | a handle |
| ul | a CK\_ULONG |
| p | a pointer |
| pb | a pointer to a CK\_BYTE |
| ph | a pointer to a handle |
| pul | a pointer to a CK\_ULONG |

Cryptoki is based on ANSI C types, and defines the following data types:

/\* an unsigned 8-bit value \*/

typedef unsigned char CK\_BYTE;

/\* an unsigned 8-bit character \*/

typedef CK\_BYTE CK\_CHAR;

/\* an 8-bit UTF-8 character \*/

typedef CK\_BYTE CK\_UTF8CHAR;

/\* a BYTE-sized Boolean flag \*/

typedef CK\_BYTE CK\_BBOOL;

/\* an unsigned value, at least 32 bits long \*/

typedef unsigned long int CK\_ULONG;

/\* a signed value, the same size as a CK\_ULONG \*/

typedef long int CK\_LONG;

/\* at least 32 bits; each bit is a Boolean flag \*/

typedef CK\_ULONG CK\_FLAGS;

Cryptoki also uses pointers to some of these data types, as well as to the type void, which are implementation-dependent. These pointer types are:

CK\_BYTE\_PTR /\* Pointer to a CK\_BYTE \*/

CK\_CHAR\_PTR /\* Pointer to a CK\_CHAR \*/

CK\_UTF8CHAR\_PTR /\* Pointer to a CK\_UTF8CHAR \*/

CK\_ULONG\_PTR /\* Pointer to a CK\_ULONG \*/

CK\_VOID\_PTR /\* Pointer to a void \*/

Cryptoki also defines a pointer to a CK\_VOID\_PTR, which is implementation-dependent:

CK\_VOID\_PTR\_PTR /\* Pointer to a CK\_VOID\_PTR \*/

In addition, Cryptoki defines a C-style NULL pointer, which is distinct from any valid pointer:

NULL\_PTR /\* A NULL pointer \*/

It follows that many of the data and pointer types will vary somewhat from one environment to another (*e.g.*, a CK\_ULONG will sometimes be 32 bits, and sometimes perhaps 64 bits). However, these details should not affect an application, assuming it is compiled with Cryptoki header files consistent with the Cryptoki library to which the application is linked.

All numbers and values expressed in this document are decimal, unless they are preceded by “0x”, in which case they are hexadecimal values.

The **CK\_CHAR** data type holds characters from the following table, taken from ANSI C:

Table , Character Set

| **Category** | **Characters** |
| --- | --- |
| Letters | A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z |
| Numbers | 0 1 2 3 4 5 6 7 8 9 |
| Graphic characters | ! “ # % & ‘ ( ) \* + , - . / : ; < = > ? [ \ ] ^ \_ { | } ~ |
| Blank character | ‘ ‘ |

The **CK\_UTF8CHAR** data type holds UTF-8 encoded Unicode characters as specified in RFC2279. UTF-8 allows internationalization while maintaining backward compatibility with the Local String definition of PKCS #11 version 2.01.

In Cryptoki, the **CK\_BBOOL** data type is a Boolean type that can be true or false. A zero value means false, and a nonzero value means true. Similarly, an individual bit flag, **CKF\_**..., can also be set (true) or unset (false). For convenience, Cryptoki defines the following macros for use with values of type **CK\_BBOOL**:

#define CK\_FALSE 0

#define CK\_TRUE 1

For backwards compatibility, header files for this version of Cryptoki also define TRUE and FALSE as (CK\_DISABLE\_TRUE\_FALSE may be set by the application vendor):

#ifndef CK\_DISABLE\_TRUE\_FALSE

#ifndef FALSE

#define FALSE CK\_FALSE

#endif

#ifndef TRUE

#define TRUE CK\_TRUE

#endif

#endif

## Normative References

**[ARIA]** National Security Research Institute, Korea, “Block Cipher Algorithm ARIA”,URL: <https://www.ietf.org/rfc/rfc5794.txt>

**[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”, January 2008.  
URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>

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

**[FIPS PUB 46-3]** NIST. “FIPS 46-3: Data Encryption Standard”, October 1999.   
URL: <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.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 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: <https://datatracker.ietf.org/doc/html/rfc6986>

**[MD2]** B. Kaliski. RSA Laboratories. “The MD2 Message-Digest Algorithm”, April 1992.   
URL: <https://www.ietf.org/rfc/rfc1319.txt>

**[MD5]** RSA Data Security. R. Rivest. “The MD5 Message-Digest Algorithm”, April 1992.   
URL: <https://www.ietf.org/rfc/rfc1321.txt>

**[NIST 802-208]** NIST “Special Publication 800-208: Recommendation for Stateful Hash-Based Signature Schemes”, October 2020.   
URL: <https://csrc.nist.gov/publications/detail/sp/800-208/final>

**[OAEP]** M. Bellare, P. Rogaway. “Optimal Asymmetric Encryption – How to Encrypt with RSA”, November 1995.

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

**[PKCS11-Prof]** *PKCS #11 Profiles Version 3.1*. Edited by Tim Hudson. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.1/pkcs11-profiles-v3.1.html>.

**[PKCS #1]** K. Moriarty, B. Kaliski, J. Jonsson, A. Rusch. RFC 8017 “PKCS #1: RSA Cryptography Specifications Version 2.2”, November 2016  
URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc8017.txt.pdf>

**[PKCS #3]** RSA Laboratories. “Diffie-Hellman Key-Agreement Standard. v1.4”, November 1993.   
URL: <https://www.teletrust.de/fileadmin/files/oid/oid_pkcs-3v1-4.pdf>

**[PKCS #5]** K. Moriarty, B. Kaliski, A. Rusch. RFC 8018. “PKCS #5: Password-Based Cryptography Specification Version 2.1”, January 2017  
URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc8018.txt.pdf>

**[PKCS #7]** B.Kaliski. “PKCS #7 Cryptographic Message Syntax Version 1.5*”*, March 1998  
URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc2315.txt.pdf>

**[PKCS #8]** B. Kaliski. RFC 5208 “Public-Key Cryptography Standards (PKCS) #8: Private-Key Information Syntax Specification Version 1.2”, May 2008, obsoleted by RFC 5258 S.Turner “Asymmetric Key Packages”, August 2010   
URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc5958.txt.pdf>

**[PKCS #12]** K. Moriarty, M. Nystrom, S. Parkinson, A. Rusch, M. Scott. “PKCS #12 Personal Information Exchange Syntax v1.1”, July 2014.  
URL: <https://www.rfc-editor.org/rfc/pdfrfc/rfc7292.txt.pdf>

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

**[RFC 2409]** D. Harkins, D.Carrel. RFC 2409: “The Internet Key Exchange (IKE)”, November 1998.  
URL: <https://tools.ietf.org/html/rfc2409>

[RFC 2119] 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>.

**[RFC 2279]** F. Yergeau. RFC 2279: “UTF-8, a transformation format of ISO 10646 Alis Technologies”, January 1998.   
URL: [http://www.ietf.org/rfc/rfc2279.txt](http://ietf.org/rfc/rfc2279.txt)

**[RFC 2534]** Masinter, L., Wing, D., Mutz, A., and K. Holtman. RFC 2534: “Media Features for Display, Print, and Fax”. March 1999.   
URL: [http://www.ietf.org/rfc/rfc2534.txt](http://ietf.org/rfc/rfc2534.txt)

**[RFC 5652]** R. Housley. RFC 5652: “Cryptographic Message Syntax”, September 2009. URL:   
<http://www.ietf.org/rfc/rfc5652.txt>

**[RFC 5707]** Rescorla, E., “The Keying Material Exporters for Transport Layer Security (TLS)”, RFC 5705, March 2010.   
URL: http://www.ietf.org/rfc/rfc5705.txt

**[RFC 5996]** C. Kaufman, P. Hoffman, Y. Nir, P. Eronen. RFC 5996: “Internet Key Exchange Protocol Version 2 (IKEv2)”, September 2010.   
URL: <https://tools.ietf.org/html/rfc5996>

**[RFC 8554]** D. McGrew, m. Curcio, S. Fluhrer. RFC 8554: “Leighton-Micali Hash-Based Signatures”, April 2019.  
URL: <https://tools.ietf.org/html/rfc8554>

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

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

**[TLS]** [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999. URL: 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.   
URL**:** <http://www.ietf.org/rfc/rfc4346.txt> , which was superseded by [TLS12].

**[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>

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

**[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

## 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

**[CC/PP]** W3C. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies. World Wide Web Consortium, January 2004.   
URL: <http://www.w3.org/TR/CCPP-struct-vocab/>

**[CDPD**] Ameritech Mobile Communications et al. Cellular Digital Packet Data System Specifications: Part 406: Airlink Security. 1993.

**[CT-KIP]** RSA Laboratories. Cryptographic Token Key Initialization Protocol. Version 1.0, December 2005.

**[GCS-API]** X/Open Company Ltd. Generic Cryptographic Service API (GCS-API), Base - Draft 2. February 14, 1995.

**[ISO/IEC 7816-1]** ISO. Information Technology — Identification Cards — Integrated Circuit(s) with Contacts — Part 1: Physical Characteristics. 1998.

**[ISO/IEC 7816-4]** ISO. Information Technology — Identification Cards — Integrated Circuit(s) with Contacts — Part 4: Interindustry Commands for Interchange. 1995.

**[ISO/IEC 8824-1]** ISO. Information Technology-- Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. 2002.

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

**[ISO/IEC 9594-1]** ISO. Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts, Models and Services. 2001.

**[ISO/IEC 9594-8]** ISO. Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks. 2001

**[ISO/IEC 9796-2]** ISO. Information Technology — Security Techniques — Digital Signature Scheme Giving Message Recovery — Part 2: Integer factorization based mechanisms. 2002.

**[Java MIDP]** Java Community Process. Mobile Information Device Profile for Java 2 Micro Edition. November 2002.   
URL: <http://jcp.org/jsr/detail/118.jsp>

**[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

**[MeT-PTD]** MeT. MeT PTD Definition – Personal Trusted Device Definition, Version 1.0, February 2003.   
URL: <http://www.mobiletransaction.org>

**[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>

**[PCMCIA]** Personal Computer Memory Card International Association. *PC Card Standard,* Release 2.1,. July 1993.

**[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: <https://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.

**[WTLS]** WAP. Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a. April 2001.   
URL: <http://openmobilealliance.org/tech/affiliates/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

# Platform- and compiler-dependent directives for C or C++

There is a large array of Cryptoki-related data types that are defined in the Cryptoki header files. Certain packing and pointer-related aspects of these types are platform and compiler-dependent; these aspects are therefore resolved on a platform-by-platform (or compiler-by-compiler) basis outside of the Cryptoki header files by means of preprocessor directives.

This means that when writing C or C++ code, certain preprocessor directives MUST be issued before including a Cryptoki header file. These directives are described in the remainder of this section.

Plattform specific implementation hints can be found in the pkcs11.h header file.

## Structure packing

Cryptoki structures are packed to occupy as little space as is possible. Cryptoki structures SHALL be packed with 1-byte alignment.

## Pointer-related macros

Because different platforms and compilers have different ways of dealing with different types of pointers, the following 6 macros SHALL be set outside the scope of Cryptoki:

1. CK\_PTR

CK\_PTR is the “indirection string” a given platform and compiler uses to make a pointer to an object. It is used in the following fashion:

typedef CK\_BYTE CK\_PTR CK\_BYTE\_PTR;

1. CK\_DECLARE\_FUNCTION

CK\_DECLARE\_FUNCTION(returnType, name), when followed by a parentheses-enclosed list of arguments and a semicolon, declares a Cryptoki API function in a Cryptoki library. returnType is the return type of the function, and name is its name. It SHALL be used in the following fashion:

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Initialize)(

CK\_VOID\_PTR pReserved

);

1. CK\_DECLARE\_FUNCTION\_POINTER

CK\_DECLARE\_FUNCTION\_POINTER(returnType, name), when followed by a parentheses-enclosed list of arguments and a semicolon, declares a variable or type which is a pointer to a Cryptoki API function in a Cryptoki library. returnType is the return type of the function, and name is its name. It SHALL be used in either of the following fashions to define a function pointer variable, myC\_Initialize, which can point to a C\_Initialize function in a Cryptoki library (note that neither of the following code snippets actually assigns a value to myC\_Initialize):

CK\_DECLARE\_FUNCTION\_POINTER(CK\_RV, myC\_Initialize)(

CK\_VOID\_PTR pReserved

);

or:

typedef CK\_DECLARE\_FUNCTION\_POINTER(CK\_RV, myC\_InitializeType)(

CK\_VOID\_PTR pReserved

);

myC\_InitializeType myC\_Initialize;

1. CK\_CALLBACK\_FUNCTION

CK\_CALLBACK\_FUNCTION(returnType, name), when followed by a parentheses-enclosed list of arguments and a semicolon, declares a variable or type which is a pointer to an application callback function that can be used by a Cryptoki API function in a Cryptoki library. returnType is the return type of the function, and name is its name. It SHALL be used in either of the following fashions to define a function pointer variable, myCallback, which can point to an application callback which takes arguments args and returns a CK\_RV (note that neither of the following code snippets actually assigns a value to myCallback):

CK\_CALLBACK\_FUNCTION(CK\_RV, myCallback)(args);

or:

typedef CK\_CALLBACK\_FUNCTION(CK\_RV, myCallbackType)(args);

myCallbackType myCallback;

1. NULL\_PTR

NULL\_PTR is the value of a NULL pointer. In any ANSI C environment—and in many others as well—NULL\_PTR SHALL be defined simply as 0.

# General data types

The general Cryptoki data types are described in the following subsections. The data types for holding parameters for various mechanisms, and the pointers to those parameters, are not described here; these types are described with the information on the mechanisms themselves, in Section 6.

A C or C++ source file in a Cryptoki application or library can define all these types (the types described here and the types that are specifically used for particular mechanism parameters) by including the top-level Cryptoki include file, pkcs11.h. pkcs11.h, in turn, includes the other Cryptoki include files, pkcs11t.h and pkcs11f.h. A source file can also include just pkcs11t.h (instead of pkcs11.h); this defines most (but not all) of the types specified here.

When including either of these header files, a source file MUST specify the preprocessor directives indicated in Section 2.

## General information

Cryptoki represents general information with the following types:

1. CK\_VERSION; CK\_VERSION\_PTR

**CK\_VERSION** is a structure that describes the version of a Cryptoki interface, a Cryptoki library, or an SSL or TLS implementation, or the hardware or firmware version of a slot or token. It is defined as follows:

typedef struct CK\_VERSION {

CK\_BYTE major;

CK\_BYTE minor;

} CK\_VERSION;

The fields of the structure have the following meanings:

*major* major version number (the integer portion of the version)

*minor* minor version number (the hundredths portion of the version)

Example: For version 1.0, *major* = 1 and *minor* = 0. For version 2.10, *major* = 2 and *minor* = 10. Table 4 below lists the major and minor version values for the officially published Cryptoki specifications.

Table , Major and minor version values for published Cryptoki specifications

|  |  |  |
| --- | --- | --- |
| **Version** | **major** | **minor** |
| 1.0 | 0x01 | 0x00 |
| 2.01 | 0x02 | 0x01 |
| 2.10 | 0x02 | 0x0a |
| 2.11 | 0x02 | 0x0b |
| 2.20 | 0x02 | 0x14 |
| 2.30 | 0x02 | 0x1e |
| 2.40 | 0x02 | 0x28 |
| 3.0 | 0x03 | 0x00 |

Minor revisions of the Cryptoki standard are always upwardly compatible within the same major version number.

**CK\_VERSION\_PTR** is a pointer to a **CK\_VERSION**.

1. CK\_INFO; CK\_INFO\_PTR

**CK\_INFO** provides general information about Cryptoki. It is defined as follows:

typedef struct CK\_INFO {

CK\_VERSION cryptokiVersion;

CK\_UTF8CHAR manufacturerID[32];

CK\_FLAGS flags;

CK\_UTF8CHAR libraryDescription[32];

CK\_VERSION libraryVersion;

} CK\_INFO;

The fields of the structure have the following meanings:

*cryptokiVersion* Cryptoki interface version number, for compatibility with future revisions of this interface

*manufacturerID* ID of the Cryptoki library manufacturer. MUST be padded with the blank character (‘ ‘). Should *not* be null-terminated.

*flags* bit flags reserved for future versions. MUST be zero for this version

*libraryDescription* character-string description of the library. MUST be padded with the blank character (‘ ‘). Should *not* be null-terminated.

*libraryVersion* Cryptoki library version number

For libraries written to this document, the value of *cryptokiVersion* should match the version of this specification; the value of *libraryVersion* is the version number of the library software itself.

**CK\_INFO\_PTR** is a pointer to a **CK\_INFO**.

1. CK\_NOTIFICATION

**CK\_NOTIFICATION** holds the types of notifications that Cryptoki provides to an application. It is defined as follows:

typedef CK\_ULONG CK\_NOTIFICATION;

For this version of Cryptoki, the following types of notifications are defined:

CKN\_SURRENDER

The notifications have the following meanings:

*CKN\_SURRENDER* Cryptoki is surrendering the execution of a function executing in a session so that the application may perform other operations. After performing any desired operations, the application should indicate to Cryptoki whether to continue or cancel the function (see Section 5.21.1).

## Slot and token types

Cryptoki represents slot and token information with the following types:

1. CK\_SLOT\_ID; CK\_SLOT\_ID\_PTR

**CK\_SLOT\_ID** is a Cryptoki-assigned value that identifies a slot. It is defined as follows:

typedef CK\_ULONG CK\_SLOT\_ID;

A list of **CK\_SLOT\_ID**s is returned by **C\_GetSlotList**. A priori, *any* value of **CK\_SLOT\_ID** can be a valid slot identifier—in particular, a system may have a slot identified by the value 0. It need not have such a slot, however.

**CK\_SLOT\_ID\_PTR** is a pointer to a **CK\_SLOT\_ID**.

1. CK\_SLOT\_INFO; CK\_SLOT\_INFO\_PTR

**CK\_SLOT\_INFO** provides information about a slot. It is defined as follows:

typedef struct CK\_SLOT\_INFO {

CK\_UTF8CHAR slotDescription[64];

CK\_UTF8CHAR manufacturerID[32];

CK\_FLAGS flags;

CK\_VERSION hardwareVersion;

CK\_VERSION firmwareVersion;

} CK\_SLOT\_INFO;

The fields of the structure have the following meanings:

*slotDescription* character-string description of the slot. MUST be padded with the blank character (‘ ‘). MUST NOT be null-terminated.

*manufacturerID* ID of the slot manufacturer. MUST be padded with the blank character (‘ ‘). MUST NOT be null-terminated.

*flags* bits flags that provide capabilities of the slot. The flags are defined below

*hardwareVersion* version number of the slot’s hardware

*firmwareVersion* version number of the slot’s firmware

The following table defines the *flags* field:

Table , Slot Information Flags

| **Bit Flag** | **Mask** | **Meaning** |
| --- | --- | --- |
| CKF\_TOKEN\_PRESENT | 0x00000001 | True if a token is present in the slot (*e.g.*, a device is in the reader) |
| CKF\_REMOVABLE\_DEVICE | 0x00000002 | True if the reader supports removable devices |
| CKF\_HW\_SLOT | 0x00000004 | True if the slot is a hardware slot, as opposed to a software slot implementing a “soft token” |

For a given slot, the value of the **CKF\_REMOVABLE\_DEVICE** flag *never changes*. In addition, if this flag is not set for a given slot, then the **CKF\_TOKEN\_PRESENT** flag for that slot is *always* set. That is, if a slot does not support a removable device, then that slot always has a token in it.

**CK\_SLOT\_INFO\_PTR** is a pointer to a **CK\_SLOT\_INFO**.

1. CK\_TOKEN\_INFO; CK\_TOKEN\_INFO\_PTR

**CK\_TOKEN\_INFO** provides information about a token. It is defined as follows:

typedef struct CK\_TOKEN\_INFO {

CK\_UTF8CHAR label[32];

CK\_UTF8CHAR manufacturerID[32];

CK\_UTF8CHAR model[16];

CK\_CHAR serialNumber[16];

CK\_FLAGS flags;

CK\_ULONG ulMaxSessionCount;

CK\_ULONG ulSessionCount;

CK\_ULONG ulMaxRwSessionCount;

CK\_ULONG ulRwSessionCount;

CK\_ULONG ulMaxPinLen;

CK\_ULONG ulMinPinLen;

CK\_ULONG ulTotalPublicMemory;

CK\_ULONG ulFreePublicMemory;

CK\_ULONG ulTotalPrivateMemory;

CK\_ULONG ulFreePrivateMemory;

CK\_VERSION hardwareVersion;

CK\_VERSION firmwareVersion;

CK\_CHAR utcTime[16];

} CK\_TOKEN\_INFO;

The fields of the structure have the following meanings:

*label* application-defined label, assigned during token initialization. MUST be padded with the blank character (‘ ‘). MUST NOT be null-terminated.

*manufacturerID* ID of the device manufacturer. MUST be padded with the blank character (‘ ‘). MUST NOT be null-terminated.

*model* model of the device. MUST be padded with the blank character (‘ ‘). MUST NOT be null-terminated.

*serialNumber* character-string serial number of the device. MUST be padded with the blank character (‘ ‘). MUST NOT be null-terminated.

*flags* bit flags indicating capabilities and status of the device as defined below

*ulMaxSessionCount* maximum number of sessions that can be opened with the token at one time by a single application (see **CK\_TOKEN\_INFO Note** below)

*ulSessionCount* number of sessions that this application currently has open with the token (see **CK\_TOKEN\_INFO Note** below)

*ulMaxRwSessionCount* maximum number of read/write sessions that can be opened with the token at one time by a single application (see **CK\_TOKEN\_INFO Note** below)

*ulRwSessionCount* number of read/write sessions that this application currently has open with the token (see **CK\_TOKEN\_INFO Note** below)

*ulMaxPinLen* maximum length in bytes of the PIN

*ulMinPinLen* minimum length in bytes of the PIN

*ulTotalPublicMemory* the total amount of memory on the token in bytes in which public objects may be stored (see **CK\_TOKEN\_INFO Note** below)

*ulFreePublicMemory* the amount of free (unused) memory on the token in bytes for public objects (see **CK\_TOKEN\_INFO Note** below)

*ulTotalPrivateMemory* the total amount of memory on the token in bytes in which private objects may be stored (see **CK\_TOKEN\_INFO Note** below)

*ulFreePrivateMemory* the amount of free (unused) memory on the token in bytes for private objects (see **CK\_TOKEN\_INFO Note** below)

*hardwareVersion* version number of hardware

*firmwareVersion* version number of firmware

*utcTime* current time as a character-string of length 16, represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second; and 2 additional reserved ‘0’ characters). The value of this field only makes sense for tokens equipped with a clock, as indicated in the token information flags (see below)

The following table defines the *flags* field:

Table , Token Information Flags

| **Bit Flag** | **Mask** | **Meaning** |
| --- | --- | --- |
| CKF\_RNG | 0x00000001 | True if the token has its own random number generator |
| CKF\_WRITE\_PROTECTED | 0x00000002 | True if the token is write-protected (see below) |
| CKF\_LOGIN\_REQUIRED | 0x00000004 | True if there are some cryptographic functions that a user MUST be logged in to perform |
| CKF\_USER\_PIN\_INITIALIZED | 0x00000008 | True if the normal user’s PIN has been initialized |
| CKF\_RESTORE\_KEY\_NOT\_NEEDED | 0x00000020 | True if a successful save of a session’s cryptographic operations state *always* contains all keys needed to restore the state of the session |
| CKF\_CLOCK\_ON\_TOKEN | 0x00000040 | True if token has its own hardware clock |
| CKF\_PROTECTED\_AUTHENTICATION\_PATH | 0x00000100 | True if token has a “protected authentication path”, whereby a user can log into the token without passing a PIN through the Cryptoki library |
| CKF\_DUAL\_CRYPTO\_OPERATIONS | 0x00000200 | True if a single session with the token can perform dual cryptographic operations (see Section 5.14) |
| CKF\_TOKEN\_INITIALIZED | 0x00000400 | True if the token has been initialized using C\_InitToken or an equivalent mechanism outside the scope of this standard. Calling C\_InitToken when this flag is set will cause the token to be reinitialized. |
| CKF\_SECONDARY\_AUTHENTICATION | 0x00000800 | True if the token supports secondary authentication for private key objects. (Deprecated; new implementations MUST NOT set this flag) |
| CKF\_USER\_PIN\_COUNT\_LOW | 0x00010000 | True if an incorrect user login PIN has been entered at least once since the last successful authentication. |
| CKF\_USER\_PIN\_FINAL\_TRY | 0x00020000 | True if supplying an incorrect user PIN will cause it to become locked. |
| CKF\_USER\_PIN\_LOCKED | 0x00040000 | True if the user PIN has been locked. User login to the token is not possible. |
| CKF\_USER\_PIN\_TO\_BE\_CHANGED | 0x00080000 | True if the user PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card. |
| CKF\_SO\_PIN\_COUNT\_LOW | 0x00100000 | True if an incorrect SO login PIN has been entered at least once since the last successful authentication. |
| CKF\_SO\_PIN\_FINAL\_TRY | 0x00200000 | True if supplying an incorrect SO PIN will cause it to become locked. |
| CKF\_SO\_PIN\_LOCKED | 0x00400000 | True if the SO PIN has been locked. SO login to the token is not possible. |
| CKF\_SO\_PIN\_TO\_BE\_CHANGED | 0x00800000 | True if the SO PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card. |
| CKF\_ERROR\_STATE | 0x01000000 | True if the token failed a FIPS 140-2 self-test and entered an error state. |

Exactly what the **CKF\_WRITE\_PROTECTED** flag means is not specified in Cryptoki. An application may be unable to perform certain actions on a write-protected token; these actions can include any of the following, among others:

* Creating/modifying/deleting any object on the token.
* Creating/modifying/deleting a token object on the token.
* Changing the SO’s PIN.
* Changing the normal user’s PIN.

The token may change the value of the **CKF\_WRITE\_PROTECTED** flag depending on the session state to implement its object management policy. For instance, the token may set the **CKF\_WRITE\_PROTECTED** flag unless the session state is R/W SO or R/W User to implement a policy that does not allow any objects, public or private, to be created, modified, or deleted unless the user has successfully called C\_Login.

The **CKF\_USER\_PIN\_COUNT\_LOW**, **CKF\_USER\_PIN\_COUNT\_LOW**, **CKF\_USER\_PIN\_FINAL\_TRY**, and **CKF\_SO\_PIN\_FINAL\_TRY** flags may always be set to false if the token does not support the functionality or will not reveal the information because of its security policy.

The **CKF\_USER\_PIN\_TO\_BE\_CHANGED** and **CKF\_SO\_PIN\_TO\_BE\_CHANGED** flags may always be set to false if the token does not support the functionality. If a PIN is set to the default value, or has expired, the appropriate **CKF\_USER\_PIN\_TO\_BE\_CHANGED** or **CKF\_SO\_PIN\_TO\_BE\_CHANGED** flag is set to true. When either of these flags are true, logging in with the corresponding PIN will succeed, but only the C\_SetPIN function can be called. Calling any other function that required the user to be logged in will cause CKR\_PIN\_EXPIRED to be returned until C\_SetPIN is called successfully.

**CK\_TOKEN\_INFO Note**: The fields ulMaxSessionCount, ulSessionCount, ulMaxRwSessionCount, ulRwSessionCount, ulTotalPublicMemory, ulFreePublicMemory, ulTotalPrivateMemory, and ulFreePrivateMemory can have the special value CK\_UNAVAILABLE\_INFORMATION, which means that the token and/or library is unable or unwilling to provide that information. In addition, the fields ulMaxSessionCount and ulMaxRwSessionCount can have the special value CK\_EFFECTIVELY\_INFINITE, which means that there is no practical limit on the number of sessions (resp. R/W sessions) an application can have open with the token.

It is important to check these fields for these special values. This is particularly true for CK\_EFFECTIVELY\_INFINITE, since an application seeing this value in the ulMaxSessionCount or ulMaxRwSessionCount field would otherwise conclude that it can’t open any sessions with the token, which is far from being the case.

The upshot of all this is that the correct way to interpret (for example) the ulMaxSessionCount field is something along the lines of the following:

CK\_TOKEN\_INFO info;

.

.

if ((CK\_LONG) info.ulMaxSessionCount

== CK\_UNAVAILABLE\_INFORMATION) {

/\* Token refuses to give value of ulMaxSessionCount \*/

.

.

} else if (info.ulMaxSessionCount == CK\_EFFECTIVELY\_INFINITE) {

/\* Application can open as many sessions as it wants \*/

.

.

} else {

/\* ulMaxSessionCount really does contain what it should \*/

.

.

}

CK\_TOKEN\_INFO\_PTR is a pointer to a CK\_TOKEN\_INFO.

## Session types

Cryptoki represents session information with the following types:

1. CK\_SESSION\_HANDLE; CK\_SESSION\_HANDLE\_PTR

**CK\_SESSION\_HANDLE** is a Cryptoki-assigned value that identifies a session. It is defined as follows:

typedef CK\_ULONG CK\_SESSION\_HANDLE;

*Valid session handles in Cryptoki always have nonzero values.* For developers’ convenience, Cryptoki defines the following symbolic value:

CK\_INVALID\_HANDLE

CK\_SESSION\_HANDLE\_PTR is a pointer to a CK\_SESSION\_HANDLE.

1. CK\_USER\_TYPE

**CK\_USER\_TYPE** holds the types of Cryptoki users described in **[PKCS11-UG]** and, in addition, a context-specific type described in Section 4.9. It is defined as follows:

typedef CK\_ULONG CK\_USER\_TYPE;

For this version of Cryptoki, the following types of users are defined:

CKU\_SO

CKU\_USER

CKU\_CONTEXT\_SPECIFIC

1. CK\_STATE

**CK\_STATE** holds the session state, as described in **[PKCS11-UG]**. It is defined as follows:

typedef CK\_ULONG CK\_STATE;

For this version of Cryptoki, the following session states are defined:

CKS\_RO\_PUBLIC\_SESSION

CKS\_RO\_USER\_FUNCTIONS

CKS\_RW\_PUBLIC\_SESSION

CKS\_RW\_USER\_FUNCTIONS

CKS\_RW\_SO\_FUNCTIONS

1. CK\_SESSION\_INFO; CK\_SESSION\_INFO\_PTR

**CK\_SESSION\_INFO** provides information about a session. It is defined as follows:

typedef struct CK\_SESSION\_INFO {

CK\_SLOT\_ID slotID;

CK\_STATE state;

CK\_FLAGS flags;

CK\_ULONG ulDeviceError;

} CK\_SESSION\_INFO;

The fields of the structure have the following meanings:

*slotID* ID of the slot that interfaces with the token

*state* the state of the session

*flags* bit flags that define the type of session; the flags are defined below

*ulDeviceError* an error code defined by the cryptographic device. Used for errors not covered by Cryptoki.

The following table defines the *flags* field:

Table , Session Information Flags

| **Bit Flag** | **Mask** | **Meaning** |
| --- | --- | --- |
| CKF\_RW\_SESSION | 0x00000002 | True if the session is read/write; false if the session is read-only |
| CKF\_SERIAL\_SESSION | 0x00000004 | This flag is provided for backward compatibility, and should always be set to true |

CK\_SESSION\_INFO\_PTR is a pointer to a CK\_SESSION\_INFO.

## Object types

Cryptoki represents object information with the following types:

1. CK\_OBJECT\_HANDLE; CK\_OBJECT\_HANDLE\_PTR

**CK\_OBJECT\_HANDLE** is a token-specific identifier for an object. It is defined as follows:

typedef CK\_ULONG CK\_OBJECT\_HANDLE;

When an object is created or found on a token by an application, Cryptoki assigns it an object handle for that application’s sessions to use to access it. A particular object on a token does not necessarily have a handle which is fixed for the lifetime of the object; however, if a particular session can use a particular handle to access a particular object, then that session will continue to be able to use that handle to access that object as long as the session continues to exist, the object continues to exist, and the object continues to be accessible to the session.

*Valid object handles in Cryptoki always have nonzero values.* For developers’ convenience, Cryptoki defines the following symbolic value:

CK\_INVALID\_HANDLE

CK\_OBJECT\_HANDLE\_PTR is a pointer to a CK\_OBJECT\_HANDLE.

1. CK\_OBJECT\_CLASS; CK\_OBJECT\_CLASS\_PTR

CK\_OBJECT\_CLASS is a value that identifies the classes (or types) of objects that Cryptoki recognizes. It is defined as follows:

typedef CK\_ULONG CK\_OBJECT\_CLASS;

Object classes are defined with the objects that use them. The type is specified on an object through the CKA\_CLASS attribute of the object.

Vendor defined values for this type may also be specified.

CKO\_VENDOR\_DEFINED

Object classes **CKO\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their object classes through the PKCS process.

**CK\_OBJECT\_CLASS\_PTR** is a pointer to a **CK\_OBJECT\_CLASS**.

1. CK\_HW\_FEATURE\_TYPE

**CK\_HW\_FEATURE\_TYPE** is a value that identifies a hardware feature type of a device. It is defined as follows:

typedef CK\_ULONG CK\_HW\_FEATURE\_TYPE;

Hardware feature types are defined with the objects that use them. The type is specified on an object through the CKA\_HW\_FEATURE\_TYPE attribute of the object.

Vendor defined values for this type may also be specified.

CKH\_VENDOR\_DEFINED

Feature types **CKH\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their feature types through the PKCS process.

1. CK\_KEY\_TYPE

**CK\_KEY\_TYPE** is a value that identifies a key type. It is defined as follows:

typedef CK\_ULONG CK\_KEY\_TYPE;

Key types are defined with the objects and mechanisms that use them. The key type is specified on an object through the CKA\_KEY\_TYPE attribute of the object.

Vendor defined values for this type may also be specified.

CKK\_VENDOR\_DEFINED

Key types **CKK\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their key types through the PKCS process.

1. CK\_CERTIFICATE\_TYPE

**CK\_CERTIFICATE\_TYPE** is a value that identifies a certificate type. It is defined as follows:

typedef CK\_ULONG CK\_CERTIFICATE\_TYPE;

Certificate types are defined with the objects and mechanisms that use them. The certificate type is specified on an object through the CKA\_CERTIFICATE\_TYPE attribute of the object.

Vendor defined values for this type may also be specified.

CKC\_VENDOR\_DEFINED

Certificate types **CKC\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their certificate types through the PKCS process.

1. CK\_CERTIFICATE\_CATEGORY

**CK\_CERTIFICATE\_CATEGORY** is a value that identifies a certificate category. It is defined as follows:

typedef CK\_ULONG CK\_CERTIFICATE\_CATEGORY;

For this version of Cryptoki, the following certificate categories are defined:

| **Constant** | **Value** | **Meaning** |
| --- | --- | --- |
| CK\_CERTIFICATE\_CATEGORY\_UNSPECIFIED | 0x00000000UL | No category specified |
| CK\_CERTIFICATE\_CATEGORY\_TOKEN\_USER | 0x00000001UL | Certificate belongs to owner of the token |
| CK\_CERTIFICATE\_CATEGORY\_AUTHORITY | 0x00000002UL | Certificate belongs to a certificate authority |
| CK\_CERTIFICATE\_CATEGORY\_OTHER\_ENTITY | 0x00000003UL | Certificate belongs to an end entity (i.e.: not a CA) |

1. CK\_ATTRIBUTE\_TYPE

**CK\_ATTRIBUTE\_TYPE** is a value that identifies an attribute type. It is defined as follows:

typedef CK\_ULONG CK\_ATTRIBUTE\_TYPE;

Attributes are defined with the objects and mechanisms that use them. Attributes are specified on an object as a list of type, length value items. These are often specified as an attribute template.

Vendor defined values for this type may also be specified.

CKA\_VENDOR\_DEFINED

Attribute types **CKA\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their attribute types through the PKCS process.

1. CK\_ATTRIBUTE; CK\_ATTRIBUTE\_PTR

**CK\_ATTRIBUTE** is a structure that includes the type, value, and length of an attribute. It is defined as follows:

typedef struct CK\_ATTRIBUTE {

CK\_ATTRIBUTE\_TYPE type;

CK\_VOID\_PTR pValue;

CK\_ULONG ulValueLen;

} CK\_ATTRIBUTE;

The fields of the structure have the following meanings:

*type* the attribute type

*pValue* pointer to the value of the attribute

*ulValueLen* length in bytes of the value

If an attribute has no value, then *ulValueLen* = 0, and the value of *pValue* is irrelevant. An array of **CK\_ATTRIBUTE**s is called a “template” and is used for creating, manipulating and searching for objects. The order of the attributes in a template *never* matters, even if the template contains vendor-specific attributes. Note that *pValue* is a “void” pointer, facilitating the passing of arbitrary values. Both the application and Cryptoki library MUST ensure that the pointer can be safely cast to the expected type (*i.e.*, without word-alignment errors).

The constant CK\_UNAVAILABLE\_INFORMATION is used in the ulValueLen field to denote an invalid or unavailable value. See C\_GetAttributeValue for further details.

**CK\_ATTRIBUTE\_PTR** is a pointer to a **CK\_ATTRIBUTE**.

1. CK\_DATE

**CK\_DATE** is a structure that defines a date. It is defined as follows:

typedef struct CK\_DATE {

CK\_CHAR year[4];

CK\_CHAR month[2];

CK\_CHAR day[2];

} CK\_DATE;

The fields of the structure have the following meanings:

*year* the year (“1900” - “9999”)

*month* the month (“01” - “12”)

*day* the day (“01” - “31”)

The fields hold numeric characters from the character set in Table 3, not the literal byte values.

When a Cryptoki object carries an attribute of this type, and the default value of the attribute is specified to be "empty," then Cryptoki libraries SHALL set the attribute's *ulValueLen* to 0.

Note that implementations of previous versions of Cryptoki may have used other methods to identify an "empty" attribute of type CK\_DATE, and applications that needs to interoperate with these libraries therefore have to be flexible in what they accept as an empty value.

1. CK\_PROFILE\_ID; CK\_PROFILE\_ID\_PTR

**CK\_PROFILE\_ID** is an unsigend ulong value represting a specific token profile. It is defined as follows:

typedef CK\_ULONG CK\_PROFILE\_ID;

Profiles are defines in the PKCS #11 Cryptographic Token Interface Profiles document. s. ID's greater than 0xffffffff may cause compatibility issues on platforms that have CK\_ULONG values of 32 bits, and should be avoided.

Vendor defined values for this type may also be specified.

CKP\_VENDOR\_DEFINED

Profile IDs **CKP\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their object classes through the PKCS process.

*Valid Profile IDs in Cryptoki always have nonzero values.* For developers’ convenience, Cryptoki defines the following symbolic value:

CKP\_INVALID\_ID

CK\_PROFILE\_ID\_PTR is a pointer to a CK\_PROFILE\_ID.

1. CK\_JAVA\_MIDP\_SECURITY\_DOMAIN

CK\_JAVA\_MIDP\_SECURITY\_DOMAIN is a value that identifies the Java MIDP security domain of a certificate. It is defined as follows:

typedef CK\_ULONG CK\_JAVA\_MIDP\_SECURITY\_DOMAIN;

For this version of Cryptoki, the following security domains are defined. See the Java MIDP specification for further information:

| **Constant** | **Value** | **Meaning** |
| --- | --- | --- |
| CK\_SECURITY\_DOMAIN\_UNSPECIFIED | 0x00000000UL | No domain specified |
| CK\_SECURITY\_DOMAIN\_MANUFACTURER | 0x00000001UL | Manufacturer protection domain |
| CK\_SECURITY\_DOMAIN\_OPERATOR | 0x00000002UL | Operator protection domain |
| CK\_SECURITY\_DOMAIN\_THIRD\_PARTY | 0x00000003UL | Third party protection domain |

## Data types for mechanisms

Cryptoki supports the following types for describing mechanisms and parameters to them:

1. CK\_MECHANISM\_TYPE; CK\_MECHANISM\_TYPE\_PTR

**CK\_MECHANISM\_TYPE** is a value that identifies a mechanism type. It is defined as follows:

typedef CK\_ULONG CK\_MECHANISM\_TYPE;

Mechanism types are defined with the objects and mechanism descriptions that use them.

Vendor defined values for this type may also be specified.

CKM\_VENDOR\_DEFINED

Mechanism types **CKM\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their mechanism types through the PKCS process.

**CK\_MECHANISM\_TYPE\_PTR** is a pointer to a **CK\_MECHANISM\_TYPE**.

1. CK\_MECHANISM; CK\_MECHANISM\_PTR

**CK\_MECHANISM** is a structure that specifies a particular mechanism and any parameters it requires. It is defined as follows:

typedef struct CK\_MECHANISM {

CK\_MECHANISM\_TYPE mechanism;

CK\_VOID\_PTR pParameter;

CK\_ULONG ulParameterLen;

} CK\_MECHANISM;

The fields of the structure have the following meanings:

*mechanism* the type of mechanism

*pParameter* pointer to the parameter if required by the mechanism

*ulParameterLen* length in bytes of the parameter

Note that *pParameter* 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\_MECHANISM\_PTR** is a pointer to a **CK\_MECHANISM**.

1. CK\_MECHANISM\_INFO; CK\_MECHANISM\_INFO\_PTR

**CK\_MECHANISM\_INFO** is a structure that provides information about a particular mechanism. It is defined as follows:

typedef struct CK\_MECHANISM\_INFO {

CK\_ULONG ulMinKeySize;

CK\_ULONG ulMaxKeySize;

CK\_FLAGS flags;

} CK\_MECHANISM\_INFO;

The fields of the structure have the following meanings:

*ulMinKeySize* the minimum size of the key for the mechanism (whether this is measured in bits or in bytes is mechanism-dependent)

*ulMaxKeySize* the maximum size of the key for the mechanism (whether this is measured in bits or in bytes is mechanism-dependent)

*flags* bit flags specifying mechanism capabilities

For some mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields have meaningless values.

The following table defines the *flags* field:

Table , Mechanism Information Flags

| **Bit Flag** | **Mask** | **Meaning** |
| --- | --- | --- |
| CKF\_HW | 0x00000001 | True if the mechanism is performed by the device; false if the mechanism is performed in software |
| CKF\_MESSAGE\_ENCRYPT | 0x00000002 | True if the mechanism can be used with **C\_MessageEncryptInit** |
| CKF\_MESSAGE\_DECRYPT | 0x00000004 | True if the mechanism can be used with **C\_MessageDecryptInit** |
| CKF\_MESSAGE\_SIGN | 0x00000008 | True if the mechanism can be used with **C\_MessageSignInit** |
| CKF\_MESSAGE\_VERIFY | 0x00000010 | True if the mechanism can be used with **C\_MessageVerifyInit** |
| CKF\_MULTI\_MESSAGE | 0x00000020 | True if the mechanism can be used with **C\_\*MessageBegin**. One of CKF\_MESSAGE\_\* flag must also be set. |
| CKF\_FIND\_OBJECTS | 0x00000040 | This flag can be passed in as a parameter to **C\_SessionCancel** to cancel an active object search operation. Any other use of this flag is outside the scope of this standard. |
| CKF\_ENCRYPT | 0x00000100 | True if the mechanism can be used with **C\_EncryptInit** |
| CKF\_DECRYPT | 0x00000200 | True if the mechanism can be used with **C\_DecryptInit** |
| CKF\_DIGEST | 0x00000400 | True if the mechanism can be used with **C\_DigestInit** |
| CKF\_SIGN | 0x00000800 | True if the mechanism can be used with **C\_SignInit** |
| CKF\_SIGN\_RECOVER | 0x00001000 | True if the mechanism can be used with **C\_SignRecoverInit** |
| CKF\_VERIFY | 0x00002000 | True if the mechanism can be used with **C\_VerifyInit** |
| CKF\_VERIFY\_RECOVER | 0x00004000 | True if the mechanism can be used with **C\_VerifyRecoverInit** |
| CKF\_GENERATE | 0x00008000 | True if the mechanism can be used with **C\_GenerateKey** |
| CKF\_GENERATE\_KEY\_PAIR | 0x00010000 | True if the mechanism can be used with **C\_GenerateKeyPair** |
| CKF\_WRAP | 0x00020000 | True if the mechanism can be used with **C\_WrapKey** |
| CKF\_UNWRAP | 0x00040000 | True if the mechanism can be used with **C\_UnwrapKey** |
| CKF\_DERIVE | 0x00080000 | True if the mechanism can be used with **C\_DeriveKey** |
| CKF\_EXTENSION | 0x80000000 | True if there is an extension to the flags; false if no extensions. MUST be false for this version. |

CK\_MECHANISM\_INFO\_PTR is a pointer to a CK\_MECHANISM\_INFO.

## Function types

Cryptoki represents information about functions with the following data types:

1. CK\_RV

**CK\_RV** is a value that identifies the return value of a Cryptoki function. It is defined as follows:

typedef CK\_ULONG CK\_RV;

Vendor defined values for this type may also be specified.

CKR\_VENDOR\_DEFINED

Section 5.1 defines the meaning of each **CK\_RV** value. Return values **CKR\_VENDOR\_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their return values through the PKCS process.

1. CK\_NOTIFY

**CK\_NOTIFY** is the type of a pointer to a function used by Cryptoki to perform notification callbacks. It is defined as follows:

typedef CK\_CALLBACK\_FUNCTION(CK\_RV, CK\_NOTIFY)(

CK\_SESSION\_HANDLE hSession,

CK\_NOTIFICATION event,

CK\_VOID\_PTR pApplication

);

The arguments to a notification callback function have the following meanings:

*hSession* The handle of the session performing the callback

*event* The type of notification callback

*pApplication* An application-defined value. This is the same value as was passed to **C\_OpenSession** to open the session performing the callback

1. CK\_C\_XXX

Cryptoki also defines an entire family of other function pointer types. For each function **C\_XXX** in the Cryptoki API (see Section 4.12 for detailed information about each of them), Cryptoki defines a type **CK\_C\_XXX**, which is a pointer to a function with the same arguments and return value as **C\_XXX** has. An appropriately-set variable of type **CK\_C\_XXX** may be used by an application to call the Cryptoki function **C\_XXX**.

1. CK\_FUNCTION\_LIST; CK\_FUNCTION\_LIST\_PTR; CK\_FUNCTION\_LIST\_PTR\_PTR

**CK\_FUNCTION\_LIST** is a structure which contains a Cryptoki version and a function pointer to each function in the Cryptoki API. It is defined as follows:

typedef struct CK\_FUNCTION\_LIST {

CK\_VERSION version;

CK\_C\_Initialize C\_Initialize;

CK\_C\_Finalize C\_Finalize;

CK\_C\_GetInfo C\_GetInfo;

CK\_C\_GetFunctionList C\_GetFunctionList;

CK\_C\_GetSlotList C\_GetSlotList;

CK\_C\_GetSlotInfo C\_GetSlotInfo;

CK\_C\_GetTokenInfo C\_GetTokenInfo;

CK\_C\_GetMechanismList C\_GetMechanismList;

CK\_C\_GetMechanismInfo C\_GetMechanismInfo;

CK\_C\_InitToken C\_InitToken;

CK\_C\_InitPIN C\_InitPIN;

CK\_C\_SetPIN C\_SetPIN;

CK\_C\_OpenSession C\_OpenSession;

CK\_C\_CloseSession C\_CloseSession;

CK\_C\_CloseAllSessions C\_CloseAllSessions;

CK\_C\_GetSessionInfo C\_GetSessionInfo;

CK\_C\_GetOperationState C\_GetOperationState;

CK\_C\_SetOperationState C\_SetOperationState;

CK\_C\_Login C\_Login;

CK\_C\_Logout C\_Logout;

CK\_C\_CreateObject C\_CreateObject;

CK\_C\_CopyObject C\_CopyObject;

CK\_C\_DestroyObject C\_DestroyObject;

CK\_C\_GetObjectSize C\_GetObjectSize;

CK\_C\_GetAttributeValue C\_GetAttributeValue;

CK\_C\_SetAttributeValue C\_SetAttributeValue;

CK\_C\_FindObjectsInit C\_FindObjectsInit;

CK\_C\_FindObjects C\_FindObjects;

CK\_C\_FindObjectsFinal C\_FindObjectsFinal;

CK\_C\_EncryptInit C\_EncryptInit;

CK\_C\_Encrypt C\_Encrypt;

CK\_C\_EncryptUpdate C\_EncryptUpdate;

CK\_C\_EncryptFinal C\_EncryptFinal;

CK\_C\_DecryptInit C\_DecryptInit;

CK\_C\_Decrypt C\_Decrypt;

CK\_C\_DecryptUpdate C\_DecryptUpdate;

CK\_C\_DecryptFinal C\_DecryptFinal;

CK\_C\_DigestInit C\_DigestInit;

CK\_C\_Digest C\_Digest;

CK\_C\_DigestUpdate C\_DigestUpdate;

CK\_C\_DigestKey C\_DigestKey;

CK\_C\_DigestFinal C\_DigestFinal;

CK\_C\_SignInit C\_SignInit;

CK\_C\_Sign C\_Sign;

CK\_C\_SignUpdate C\_SignUpdate;

CK\_C\_SignFinal C\_SignFinal;

CK\_C\_SignRecoverInit C\_SignRecoverInit;

CK\_C\_SignRecover C\_SignRecover;

CK\_C\_VerifyInit C\_VerifyInit;

CK\_C\_Verify C\_Verify;

CK\_C\_VerifyUpdate C\_VerifyUpdate;

CK\_C\_VerifyFinal C\_VerifyFinal;

CK\_C\_VerifyRecoverInit C\_VerifyRecoverInit;

CK\_C\_VerifyRecover C\_VerifyRecover;

CK\_C\_DigestEncryptUpdate C\_DigestEncryptUpdate;

CK\_C\_DecryptDigestUpdate C\_DecryptDigestUpdate;

CK\_C\_SignEncryptUpdate C\_SignEncryptUpdate;

CK\_C\_DecryptVerifyUpdate C\_DecryptVerifyUpdate;

CK\_C\_GenerateKey C\_GenerateKey;

CK\_C\_GenerateKeyPair C\_GenerateKeyPair;

CK\_C\_WrapKey C\_WrapKey;

CK\_C\_UnwrapKey C\_UnwrapKey;

CK\_C\_DeriveKey C\_DeriveKey;

CK\_C\_SeedRandom C\_SeedRandom;

CK\_C\_GenerateRandom C\_GenerateRandom;

CK\_C\_GetFunctionStatus C\_GetFunctionStatus;

CK\_C\_CancelFunction C\_CancelFunction;

CK\_C\_WaitForSlotEvent C\_WaitForSlotEvent;

} CK\_FUNCTION\_LIST;

Each Cryptoki library has a static **CK\_FUNCTION\_LIST** structure, and a pointer to it (or to a copy of it which is also owned by the library) may be obtained by the **C\_GetFunctionList** function (see Section 5.2). The value that this pointer points to can be used by an application to quickly find out where the executable code for each function in the Cryptoki API is located. Every function in the Cryptoki API MUST have an entry point defined in the Cryptoki library’s **CK\_FUNCTION\_LIST** structure. If a particular function in the Cryptoki API is not supported by a library, then the function pointer for that function in the library’s **CK\_FUNCTION\_LIST** structure should point to a function stub which simply returns CKR\_FUNCTION\_NOT\_SUPPORTED.

In this structure ‘version’ is the cryptoki specification version number. The major and minor versions must be set to 0x02 and 0x28 indicating a version 2.40 compatible structure. The updated function list table for this version of the specification may be returned via **C\_GetInterfaceList** or **C\_GetInterface.**

An application may or may not be able to modify a Cryptoki library’s static **CK\_FUNCTION\_LIST** structure. Whether or not it can, it should never attempt to do so.

PKCS #11 modules must not add new functions at the end of the **CK\_FUNCTION\_LIST** that are not contained within the defined structure. If a PKCS#11 module needs to define additional functions, they should be placed within a vendor defined interface returned via **C\_GetInterfaceList** or **C\_GetInterface**.

**CK\_FUNCTION\_LIST\_PTR** is a pointer to a **CK\_FUNCTION\_LIST**.

**CK\_FUNCTION\_LIST\_PTR\_PTR** is a pointer to a **CK\_FUNCTION\_LIST\_PTR**.

1. CK\_FUNCTION\_LIST\_3\_0; CK\_FUNCTION\_LIST\_3\_0\_PTR; CK\_FUNCTION\_LIST\_3\_0\_PTR\_PTR

**CK\_FUNCTION\_LIST\_3\_0** is a structure which contains the same function pointers as in **CK\_FUNCTION\_LIST** and additional functions added to the end of the structure that were defined in Cryptoki version 3.0. It is defined as follows:

typedef struct CK\_FUNCTION\_LIST\_3\_0 {

CK\_VERSION version;

CK\_C\_Initialize C\_Initialize;

CK\_C\_Finalize C\_Finalize;

CK\_C\_GetInfo C\_GetInfo;

CK\_C\_GetFunctionList C\_GetFunctionList;

CK\_C\_GetSlotList C\_GetSlotList;

CK\_C\_GetSlotInfo C\_GetSlotInfo;

CK\_C\_GetTokenInfo C\_GetTokenInfo;

CK\_C\_GetMechanismList C\_GetMechanismList;

CK\_C\_GetMechanismInfo C\_GetMechanismInfo;

CK\_C\_InitToken C\_InitToken;

CK\_C\_InitPIN C\_InitPIN;

CK\_C\_SetPIN C\_SetPIN;

CK\_C\_OpenSession C\_OpenSession;

CK\_C\_CloseSession C\_CloseSession;

CK\_C\_CloseAllSessions C\_CloseAllSessions;

CK\_C\_GetSessionInfo C\_GetSessionInfo;

CK\_C\_GetOperationState C\_GetOperationState;

CK\_C\_SetOperationState C\_SetOperationState;

CK\_C\_Login C\_Login;

CK\_C\_Logout C\_Logout;

CK\_C\_CreateObject C\_CreateObject;

CK\_C\_CopyObject C\_CopyObject;

CK\_C\_DestroyObject C\_DestroyObject;

CK\_C\_GetObjectSize C\_GetObjectSize;

CK\_C\_GetAttributeValue C\_GetAttributeValue;

CK\_C\_SetAttributeValue C\_SetAttributeValue;

CK\_C\_FindObjectsInit C\_FindObjectsInit;

CK\_C\_FindObjects C\_FindObjects;

CK\_C\_FindObjectsFinal C\_FindObjectsFinal;

CK\_C\_EncryptInit C\_EncryptInit;

CK\_C\_Encrypt C\_Encrypt;

CK\_C\_EncryptUpdate C\_EncryptUpdate;

CK\_C\_EncryptFinal C\_EncryptFinal;

CK\_C\_DecryptInit C\_DecryptInit;

CK\_C\_Decrypt C\_Decrypt;

CK\_C\_DecryptUpdate C\_DecryptUpdate;

CK\_C\_DecryptFinal C\_DecryptFinal;

CK\_C\_DigestInit C\_DigestInit;

CK\_C\_Digest C\_Digest;

CK\_C\_DigestUpdate C\_DigestUpdate;

CK\_C\_DigestKey C\_DigestKey;

CK\_C\_DigestFinal C\_DigestFinal;

CK\_C\_SignInit C\_SignInit;

CK\_C\_Sign C\_Sign;

CK\_C\_SignUpdate C\_SignUpdate;

CK\_C\_SignFinal C\_SignFinal;

CK\_C\_SignRecoverInit C\_SignRecoverInit;

CK\_C\_SignRecover C\_SignRecover;

CK\_C\_VerifyInit C\_VerifyInit;

CK\_C\_Verify C\_Verify;

CK\_C\_VerifyUpdate C\_VerifyUpdate;

CK\_C\_VerifyFinal C\_VerifyFinal;

CK\_C\_VerifyRecoverInit C\_VerifyRecoverInit;

CK\_C\_VerifyRecover C\_VerifyRecover;

CK\_C\_DigestEncryptUpdate C\_DigestEncryptUpdate;

CK\_C\_DecryptDigestUpdate C\_DecryptDigestUpdate;

CK\_C\_SignEncryptUpdate C\_SignEncryptUpdate;

CK\_C\_DecryptVerifyUpdate C\_DecryptVerifyUpdate;

CK\_C\_GenerateKey C\_GenerateKey;

CK\_C\_GenerateKeyPair C\_GenerateKeyPair;

CK\_C\_WrapKey C\_WrapKey;

CK\_C\_UnwrapKey C\_UnwrapKey;

CK\_C\_DeriveKey C\_DeriveKey;

CK\_C\_SeedRandom C\_SeedRandom;

CK\_C\_GenerateRandom C\_GenerateRandom;

CK\_C\_GetFunctionStatus C\_GetFunctionStatus;

CK\_C\_CancelFunction C\_CancelFunction;

CK\_C\_WaitForSlotEvent C\_WaitForSlotEvent;

CK\_C\_GetInterfaceList C\_GetInterfaceList;

CK\_C\_GetInterface C\_GetInterface;

CK\_C\_LoginUser C\_LoginUser;

CK\_C\_SessionCancel C\_SessionCancel;

CK\_C\_MessageEncryptInit C\_MessageEncryptInit;

CK\_C\_EncryptMessage C\_EncryptMessage;

CK\_C\_EncryptMessageBegin C\_EncryptMessageBegin;

CK\_C\_EncryptMessageNext C\_EncryptMessageNext;

CK\_C\_MessageEncryptFinal C\_MessageEncryptFinal;

CK\_C\_MessageDecryptInit C\_MessageDecryptInit;

CK\_C\_DecryptMessage C\_DecryptMessage;

CK\_C\_DecryptMessageBegin C\_DecryptMessageBegin;

CK\_C\_DecryptMessageNext C\_DecryptMessageNext;

CK\_C\_MessageDecryptFinal C\_MessageDecryptFinal;

CK\_C\_MessageSignInit C\_MessageSignInit;

CK\_C\_SignMessage C\_SignMessage;

CK\_C\_SignMessageBegin C\_SignMessageBegin;

CK\_C\_SignMessageNext C\_SignMessageNext;

CK\_C\_MessageSignFinal C\_MessageSignFinal;

CK\_C\_MessageVerifyInit C\_MessageVerifyInit;

CK\_C\_VerifyMessage C\_VerifyMessage;

CK\_C\_VerifyMessageBegin C\_VerifyMessageBegin;

CK\_C\_VerifyMessageNext C\_VerifyMessageNext;

CK\_C\_MessageVerifyFinal C\_MessageVerifyFinal;

} CK\_FUNCTION\_LIST\_3\_0;

For a general description of **CK\_FUNCTION\_LIST\_3\_0** see **CK\_FUNCTION\_LIST**.

In this structure, *version* is the cryptoki specification version number. It should match the value of *cryptokiVersion* returned in the **CK\_INFO** structure, but must be 3.0 at minimum.

This function list may be returned via **C\_GetInterfaceList** or **C\_GetInterface**

**CK\_FUNCTION\_LIST\_3\_0\_PTR** is a pointer to a **CK\_FUNCTION\_LIST\_3\_0**.

**CK\_FUNCTION\_LIST\_3\_0\_PTR\_PTR** is a pointer to a **CK\_FUNCTION\_LIST\_3\_0\_PTR**.

1. CK\_INTERFACE; CK\_INTERFACE\_PTR; CK\_INTERFACE\_PTR\_PTR

**CK\_INTERFACE** is a structure which contains an interface name with a function list and flag.

It is defined as follows:

typedef struct CK\_INTERFACE {

CK\_UTF8CHAR\_PTR pInterfaceName;

CK\_VOID\_PTR pFunctionList;

CK\_FLAGS flags;

} CK\_INTERFACE;

The fields of the structure have the following meanings:

*pInterfaceName* the name of the interface

*pFunctionList* the interface function list which must always begin with a CK\_VERSION structure as the first field

*flags* bit flags specifying interface capabilities

The interface name “PKCS 11” is reserved for use by interfaces defined within the cryptoki specification.

Interfaces starting with the string: “Vendor ” are reserved for vendor use and will not oetherwise be defined as interfaces in the PKCS #11 specification. Vendors should supply new functions with interface names of “Vendor {vendor name}”. For example “Vendor ACME Inc”.

The following table defines the flags field:

Table , CK\_INTERFACE Flags

| **Bit Flag** | **Mask** | **Meaning** |
| --- | --- | --- |
| CKF\_INTERFACE\_FORK\_SAFE | 0x00000001 | The returned interface will have fork tolerant semantics. When the application forks, each process will get its own copy of all session objects, session states, login states, and encryption states. Each process will also maintain access to token objects with their previously supplied handles. |

**CK\_INTERFACE\_PTR** is a pointer to a **CK\_INTERFACE**.

**CK\_INTERFACE\_PTR\_PTR** is a pointer to a **CK\_INTERFACE\_PTR**.

## Locking-related types

The types in this section are provided solely for applications which need to access Cryptoki from multiple threads simultaneously. *Applications which will not do this need not use any of these types.*

1. CK\_CREATEMUTEX

**CK\_CREATEMUTEX** is the type of a pointer to an application-supplied function which creates a new mutex object and returns a pointer to it. It is defined as follows:

typedef CK\_CALLBACK\_FUNCTION(CK\_RV, CK\_CREATEMUTEX)(

CK\_VOID\_PTR\_PTR ppMutex

);

Calling a CK\_CREATEMUTEX function returns the pointer to the new mutex object in the location pointed to by ppMutex. Such a function should return one of the following values:

CKR\_OK, CKR\_GENERAL\_ERROR

CKR\_HOST\_MEMORY

1. CK\_DESTROYMUTEX

**CK\_DESTROYMUTEX** is the type of a pointer to an application-supplied function which destroys an existing mutex object. It is defined as follows:

typedef CK\_CALLBACK\_FUNCTION(CK\_RV, CK\_DESTROYMUTEX)(

CK\_VOID\_PTR pMutex

);

The argument to a CK\_DESTROYMUTEX function is a pointer to the mutex object to be destroyed. Such a function should return one of the following values:

CKR\_OK, CKR\_GENERAL\_ERROR

CKR\_HOST\_MEMORY

CKR\_MUTEX\_BAD

1. CK\_LOCKMUTEX and CK\_UNLOCKMUTEX

**CK\_LOCKMUTEX** is the type of a pointer to an application-supplied function which locks an existing mutex object. **CK\_UNLOCKMUTEX** is the type of a pointer to an application-supplied function which unlocks an existing mutex object. The proper behavior for these types of functions is as follows:

* If a CK\_LOCKMUTEX function is called on a mutex which is not locked, the calling thread obtains a lock on that mutex and returns.
* If a CK\_LOCKMUTEX function is called on a mutex which is locked by some thread other than the calling thread, the calling thread blocks and waits for that mutex to be unlocked.
* If a CK\_LOCKMUTEX function is called on a mutex which is locked by the calling thread, the behavior of the function call is undefined.
* If a CK\_UNLOCKMUTEX function is called on a mutex which is locked by the calling thread, that mutex is unlocked and the function call returns. Furthermore:
  + If exactly one thread was blocking on that particular mutex, then that thread stops blocking, obtains a lock on that mutex, and its CK\_LOCKMUTEX call returns.
  + If more than one thread was blocking on that particular mutex, then exactly one of the blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on the mutex, and its CK\_LOCKMUTEX call returns. All other threads blocking on that particular mutex continue to block.
* If a CK\_UNLOCKMUTEX function is called on a mutex which is not locked, then the function call returns the error code CKR\_MUTEX\_NOT\_LOCKED.
* If a CK\_UNLOCKMUTEX function is called on a mutex which is locked by some thread other than the calling thread, the behavior of the function call is undefined.

**CK\_LOCKMUTEX** is defined as follows:

typedef CK\_CALLBACK\_FUNCTION(CK\_RV, CK\_LOCKMUTEX)(

CK\_VOID\_PTR pMutex

);

The argument to a CK\_LOCKMUTEX function is a pointer to the mutex object to be locked. Such a function should return one of the following values:

CKR\_OK, CKR\_GENERAL\_ERROR

CKR\_HOST\_MEMORY,

CKR\_MUTEX\_BAD

**CK\_UNLOCKMUTEX** is defined as follows:

typedef CK\_CALLBACK\_FUNCTION(CK\_RV, CK\_UNLOCKMUTEX)(

CK\_VOID\_PTR pMutex

);

The argument to a CK\_UNLOCKMUTEX function is a pointer to the mutex object to be unlocked. Such a function should return one of the following values:

CKR\_OK, CKR\_GENERAL\_ERROR

CKR\_HOST\_MEMORY

CKR\_MUTEX\_BAD

CKR\_MUTEX\_NOT\_LOCKED

1. CK\_C\_INITIALIZE\_ARGS; CK\_C\_INITIALIZE\_ARGS\_PTR

**CK\_C\_INITIALIZE\_ARGS** is a structure containing the optional arguments for the **C\_Initialize** function. For this version of Cryptoki, these optional arguments are all concerned with the way the library deals with threads. **CK\_C\_INITIALIZE\_ARGS** is defined as follows:

typedef struct CK\_C\_INITIALIZE\_ARGS {

CK\_CREATEMUTEX CreateMutex;

CK\_DESTROYMUTEX DestroyMutex;

CK\_LOCKMUTEX LockMutex;

CK\_UNLOCKMUTEX UnlockMutex;

CK\_FLAGS flags;

CK\_VOID\_PTR pReserved;

} CK\_C\_INITIALIZE\_ARGS;

The fields of the structure have the following meanings:

*CreateMutex* pointer to a function to use for creating mutex objects

*DestroyMutex* pointer to a function to use for destroying mutex objects

*LockMutex* pointer to a function to use for locking mutex objects

*UnlockMutex* pointer to a function to use for unlocking mutex objects

*flags* bit flags specifying options for **C\_Initialize**; the flags are defined below

*pReserved* reserved for future use. Should be NULL\_PTR for this version of Cryptoki

The following table defines the flags field:

Table , C\_Initialize Parameter Flags

| **Bit Flag** | **Mask** | **Meaning** |
| --- | --- | --- |
| CKF\_LIBRARY\_CANT\_CREATE\_OS\_THREADS | 0x00000001 | True if application threads which are executing calls to the library may *not* use native operating system calls to spawn new threads; false if they may |
| CKF\_OS\_LOCKING\_OK | 0x00000002 | True if the library can use the native operation system threading model for locking; false otherwise |

CK\_C\_INITIALIZE\_ARGS\_PTR is a pointer to a CK\_C\_INITIALIZE\_ARGS.

# Objects

Cryptoki recognizes a number of classes of objects, as defined in the **CK\_OBJECT\_CLASS** data type. An object consists of a set of attributes, each of which has a given value. Each attribute that an object possesses has precisely one value. The following figure illustrates the high-level hierarchy of the Cryptoki objects and some of the attributes they support:



Figure , Object Attribute Hierarchy

Cryptoki provides functions for creating, destroying, and copying objects in general, and for obtaining and modifying the values of their attributes. Some of the cryptographic functions (*e.g.*, **C\_GenerateKey**) also create key objects to hold their results.

Objects are always “well-formed” in Cryptoki—that is, an object always contains all required attributes, and the attributes are always consistent with one another from the time the object is created. This contrasts with some object-based paradigms where an object has no attributes other than perhaps a class when it is created, and is uninitialized for some time. In Cryptoki, objects are always initialized.

Tables throughout most of Section 4 define each Cryptoki attribute in terms of the data type of the attribute value and the meaning of the attribute, which may include a default initial value. Some of the data types are defined explicitly by Cryptoki (*e.g.*, **CK\_OBJECT\_CLASS**). Attribute values may also take the following types:

Byte array an arbitrary string (array) of **CK\_BYTE**s

Big integer a string of **CK\_BYTE**s representing an unsigned integer of arbitrary size, most-significant byte first (*e.g.*, the integer 32768 is represented as the 2-byte string 0x80 0x00)

Local string an unpadded string of **CK\_CHAR**s (see Table 3) with no null-termination

RFC2279 string an unpadded string of **CK\_UTF8CHARs** with no null-termination

A token can hold several identical objects, *i.e.*, it is permissible for two or more objects to have exactly the same values for all their attributes.

In most cases each type of object in the Cryptoki specification possesses a completely well-defined set of Cryptoki attributes. Some of these attributes possess default values, and need not be specified when creating an object; some of these default values may even be the empty string (“”). Nonetheless, the object possesses these attributes. A given object has a single value for each attribute it possesses, even if the attribute is a vendor-specific attribute whose meaning is outside the scope of Cryptoki.

In addition to possessing Cryptoki attributes, objects may possess additional vendor-specific attributes whose meanings and values are not specified by Cryptoki.

## Creating, modifying, and copying objects

All Cryptoki functions that create, modify, or copy objects take a template as one of their arguments, where the template specifies attribute values. Cryptographic functions that create objects (see Section 5.18) may also contribute some additional attribute values themselves; which attributes have values contributed by a cryptographic function call depends on which cryptographic mechanism is being performed (see section 6 Mechanisms and [PKCS11-Hist] for specification of mechanisms for PKCS #11). In any case, all the required attributes supported by an object class that do not have default values MUST be specified when an object is created, either in the template or by the function itself.

### Creating objects

Objects may be created with the Cryptoki functions **C\_CreateObject** (see Section 5.7), **C\_GenerateKey**, **C\_GenerateKeyPair**, **C\_UnwrapKey**, and **C\_DeriveKey** (see Section 5.18). In addition, copying an existing object (with the function **C\_CopyObject**) also creates a new object, but we consider this type of object creation separately in Section 4.1.3.

Attempting to create an object with any of these functions requires an appropriate template to be supplied.

1. If the supplied template specifies a value for an invalid attribute, then the attempt should fail with the error code CKR\_ATTRIBUTE\_TYPE\_INVALID. An attribute is valid if it is either one of the attributes described in the Cryptoki specification or an additional vendor-specific attribute supported by the library and token.
2. If the supplied template specifies an invalid value for a valid attribute, then the attempt should fail with the error code CKR\_ATTRIBUTE\_VALUE\_INVALID. The valid values for Cryptoki attributes are described in the Cryptoki specification.
3. If the supplied template specifies a value for a read-only attribute, then the attempt should fail with the error code CKR\_ATTRIBUTE\_READ\_ONLY. Whether or not a given Cryptoki attribute is read-only is explicitly stated in the Cryptoki specification; however, a particular library and token may be even more restrictive than Cryptoki specifies. In other words, an attribute which Cryptoki says is not read-only may nonetheless be read-only under certain circumstances (*i.e.*, in conjunction with some combinations of other attributes) for a particular library and token. Whether or not a given non-Cryptoki attribute is read-only is obviously outside the scope of Cryptoki.
4. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are insufficient to fully specify the object to create, then the attempt should fail with the error code CKR\_TEMPLATE\_INCOMPLETE.
5. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are inconsistent, then the attempt should fail with the error code CKR\_TEMPLATE\_INCONSISTENT. A set of attribute values is inconsistent if not all of its members can be satisfied simultaneously *by the token*, although each value individually is valid in Cryptoki. One example of an inconsistent template would be using a template which specifies two different values for the same attribute. Another example would be trying to create a secret key object with an attribute which is appropriate for various types of public keys or private keys, but not for secret keys. A final example would be a template with an attribute that violates some token specific requirement. Note that this final example of an inconsistent template is token-dependent—on a different token, such a template might *not* be inconsistent.
6. If the supplied template specifies the same value for a particular attribute more than once (or the template specifies the same value for a particular attribute that the object-creation function itself contributes to the object), then the behavior of Cryptoki is not completely specified. The attempt to create an object can either succeed—thereby creating the same object that would have been created if the multiply-specified attribute had only appeared once—or it can fail with error code CKR\_TEMPLATE\_INCONSISTENT. Library developers are encouraged to make their libraries behave as though the attribute had only appeared once in the template; application developers are strongly encouraged never to put a particular attribute into a particular template more than once.

If more than one of the situations listed above applies to an attempt to create an object, then the error code returned from the attempt can be any of the error codes from above that applies.

### Modifying objects

Objects may be modified with the Cryptoki function **C\_SetAttributeValue** (see Section 5.7). The template supplied to **C\_SetAttributeValue** can contain new values for attributes which the object already possesses; values for attributes which the object does not yet possess; or both.

Some attributes of an object may be modified after the object has been created, and some may not. In addition, attributes which Cryptoki specifies are modifiable may actually *not* be modifiable on some tokens. That is, if a Cryptoki attribute is described as being modifiable, that really means only that it is modifiable *insofar as the Cryptoki specification is concerned*. A particular token might not actually support modification of some such attributes. Furthermore, whether or not a particular attribute of an object on a particular token is modifiable might depend on the values of certain attributes of the object. For example, a secret key object’s **CKA\_SENSITIVE** attribute can be changed from CK\_FALSE to CK\_TRUE, but not the other way around.

All the scenarios in Section 4.1.1—and the error codes they return—apply to modifying objects with **C\_SetAttributeValue**, except for the possibility of a template being incomplete.

### Copying objects

Unless an object's CKA\_COPYABLE (see Table 17) attribute is set to CK\_FALSE, it may be copied with the Cryptoki function **C\_CopyObject** (see Section 5.7). In the process of copying an object, **C\_CopyObject** also modifies the attributes of the newly-created copy according to an application-supplied template.

The Cryptoki attributes which can be modified during the course of a **C\_CopyObject** operation are the same as the Cryptoki attributes which are described as being modifiable, plus the four special attributes **CKA\_TOKEN**, **CKA\_PRIVATE**, **CKA\_MODIFIABLE and CKA\_DESTROYABLE**. To be more precise, these attributes are modifiable during the course of a **C\_CopyObject** operation *insofar as the Cryptoki specification is concerned*. A particular token might not actually support modification of some such attributes during the course of a **C\_CopyObject** operation. Furthermore, whether or not a particular attribute of an object on a particular token is modifiable during the course of a **C\_CopyObject** operation might depend on the values of certain attributes of the object. For example, a secret key object’s **CKA\_SENSITIVE** attribute can be changed from CK\_FALSE to CK\_TRUE during the course of a **C\_CopyObject** operation, but not the other way around.

If the CKA\_COPYABLE attribute of the object to be copied is set to CK\_FALSE, C\_CopyObject returns CKR\_ACTION\_PROHIBITED. Otherwise, the scenarios described in 10.1.1 - and the error codes they return - apply to copying objects with C\_CopyObject, except for the possibility of a template being incomplete.

## Common attributes

Table , Common footnotes for object attribute tables

|  |
| --- |
| 1 MUST be specified when object is created with **C\_CreateObject**.  2 MUST *not* be specified when object is created with **C\_CreateObject**.  3 MUST be specified when object is generated with **C\_GenerateKey** or **C\_GenerateKeyPair**.  4 MUST *not* be specified when object is generated with **C\_GenerateKey** or **C\_GenerateKeyPair**.  5 MUST be specified when object is unwrapped with **C\_UnwrapKey**.  6 MUST *not* be specified when object is unwrapped with **C\_UnwrapKey**.  7 Cannot be revealed if object has its **CKA\_SENSITIVE** attribute set to CK\_TRUE or its **CKA\_EXTRACTABLE** attribute set to CK\_FALSE.  8 May be modified after object is created with a **C\_SetAttributeValue** call, or in the process of copying object with a **C\_CopyObject** call. However, it is possible that a particular token may not permit modification of the attribute during the course of a **C\_CopyObject** call.  9 Default value is token-specific, and may depend on the values of other attributes.  10 Can only be set to CK\_TRUE by the SO user.  11 Attribute cannot be changed once set to CK\_TRUE. It becomes a read only attribute.  12 Attribute cannot be changed once set to CK\_FALSE. It becomes a read only attribute. |

Table , Common Object Attributes

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_CLASS1 | CK\_OBJECT\_CLASS | Object class (type) |

Refer to Table 11 for footnotes

The above table defines the attributes common to all objects.

## Hardware Feature Objects

### Definitions

This section defines the object class CKO\_HW\_FEATURE for type CK\_OBJECT\_CLASS as used in the CKA\_CLASS attribute of objects.

### Overview

Hardware feature objects (**CKO\_HW\_FEATURE**) represent features of the device. They provide an easily expandable method for introducing new value-based features to the Cryptoki interface.

When searching for objects using **C\_FindObjectsInit** and **C\_FindObjects**, hardware feature objects are not returned unless the **CKA\_CLASS** attribute in the template has the value **CKO\_HW\_FEATURE**. This protects applications written to previous versions of Cryptoki from finding objects that they do not understand.

Table , Hardware Feature Common Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Meaning** |
| CKA\_HW\_FEATURE\_TYPE1 | CK\_HW\_FEATURE\_TYPE | Hardware feature (type) |

- Refer to Table 11 for footnotes

### Clock

#### Definition

The CKA\_HW\_FEATURE\_TYPE attribute takes the value CKH\_CLOCK of type CK\_HW\_FEATURE\_TYPE.

#### Description

Clock objects represent real-time clocks that exist on the device. This represents the same clock source as the **utcTime** field in the **CK\_TOKEN\_INFO** structure.

Table , Clock Object Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Meaning** |
| CKA\_VALUE | CK\_CHAR[16] | Current time as a character-string of length 16, represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second; and 2 additional reserved ‘0’ characters). |

The **CKA\_VALUE** attribute may be set using the **C\_SetAttributeValue** function if permitted by the device. The session used to set the time MUST be logged in. The device may require the SO to be the user logged in to modify the time value. **C\_SetAttributeValue** will return the error CKR\_USER\_NOT\_LOGGED\_IN to indicate that a different user type is required to set the value.

### Monotonic Counter Objects

#### Definition

The CKA\_HW\_FEATURE\_TYPE attribute takes the value CKH\_MONOTONIC\_COUNTER of type CK\_HW\_FEATURE\_TYPE.

#### Description

Monotonic counter objects represent hardware counters that exist on the device. The counter is guaranteed to increase each time its value is read, but not necessarily by one. This might be used by an application for generating serial numbers to get some assurance of uniqueness per token.

Table , Monotonic Counter Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Meaning** |
| CKA\_RESET\_ON\_INIT1 | CK\_BBOOL | The value of the counter will reset to a previously returned value if the token is initialized using **C\_InitToken**. |
| CKA\_HAS\_RESET1 | CK\_BBOOL | The value of the counter has been reset at least once at some point in time. |
| CKA\_VALUE1 | Byte Array | The current version of the monotonic counter. The value is returned in big endian order. |

1Read Only

The **CKA\_VALUE** attribute may not be set by the client.

### User Interface Objects

#### Definition

The CKA\_HW\_FEATURE\_TYPE attribute takes the value CKH\_USER\_INTERFACE of type CK\_HW\_FEATURE\_TYPE.

#### Description

User interface objects represent the presentation capabilities of the device.

Table , User Interface Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PIXEL\_X | CK\_ULONG | Screen resolution (in pixels) in X-axis (e.g. 1280) |
| CKA\_PIXEL\_Y | CK\_ULONG | Screen resolution (in pixels) in Y-axis (e.g. 1024) |
| CKA\_RESOLUTION | CK\_ULONG | DPI, pixels per inch |
| CKA\_CHAR\_ROWS | CK\_ULONG | For character-oriented displays; number of character rows (e.g. 24) |
| CKA\_CHAR\_COLUMNS | CK\_ULONG | For character-oriented displays: number of character columns (e.g. 80). If display is of proportional-font type, this is the width of the display in “em”-s (letter “M”), see CC/PP Struct. |
| CKA\_COLOR | CK\_BBOOL | Color support |
| CKA\_BITS\_PER\_PIXEL | CK\_ULONG | The number of bits of color or grayscale information per pixel. |
| CKA\_CHAR\_SETS | RFC 2279 string | String indicating supported character sets, as defined by IANA MIBenum sets ([www.iana.org](http://www.iana.org)). Supported character sets are separated with “;”. E.g. a token supporting iso-8859-1 and US-ASCII would set the attribute value to “4;3”. |
| CKA\_ENCODING\_METHODS | RFC 2279 string | String indicating supported content transfer encoding methods, as defined by IANA ([www.iana.org](http://www.iana.org)). Supported methods are separated with “;”. E.g. a token supporting 7bit, 8bit and base64 could set the attribute value to “7bit;8bit;base64”. |
| CKA\_MIME\_TYPES | RFC 2279 string | String indicating supported (presentable) MIME-types, as defined by IANA ([www.iana.org](http://www.iana.org)). Supported types are separated with “;”. E.g. a token supporting MIME types "a/b", "a/c" and "a/d" would set the attribute value to “a/b;a/c;a/d”. |

The selection of attributes, and associated data types, has been done in an attempt to stay as aligned with RFC 2534 and CC/PP Struct as possible. The special value CK\_UNAVAILABLE\_INFORMATION may be used for CK\_ULONG-based attributes when information is not available or applicable.

None of the attribute values may be set by an application.

The value of the **CKA\_ENCODING\_METHODS** attribute may be used when the application needs to send MIME objects with encoded content to the token.

## Storage Objects

This is not an object class; hence no CKO\_ definition is required. It is a category of object classes with common attributes for the object classes that follow.

Table , Common Storage Object Attributes

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_TOKEN | CK\_BBOOL | CK\_TRUE if object is a token object; CK\_FALSE if object is a session object. Default is CK\_FALSE. |
| CKA\_PRIVATE | CK\_BBOOL | CK\_TRUE if object is a private object; CK\_FALSE if object is a public object. Default value is token-specific, and may depend on the values of other attributes of the object. |
| CKA\_MODIFIABLE | CK\_BBOOL | CK\_TRUE if object can be modified Default is CK\_TRUE. |
| CKA\_LABEL | RFC2279 string | Description of the object (default empty). |
| CKA\_COPYABLE | CK\_BBOOL | CK\_TRUE if object can be copied using C\_CopyObject. Defaults to CK\_TRUE. Can’t be set to TRUE once it is set to FALSE. |
| CKA\_DESTROYABLE | CK\_BBOOL | CK\_TRUE if the object can be destroyed using C\_DestroyObject. Default is CK\_TRUE. |
| CKA\_UNIQUE\_ID246 | RFC2279 string | The unique identifier assigned to the object. |

Only the **CKA\_LABEL** attribute can be modified after the object is created. (The **CKA\_TOKEN**, **CKA\_PRIVATE**, and **CKA\_MODIFIABLE** attributes can be changed in the process of copying an object, however.)

The **CKA\_TOKEN** attribute identifies whether the object is a token object or a session object.

When the **CKA\_PRIVATE** attribute is CK\_TRUE, a user may not access the object until the user has been authenticated to the token.

The value of the **CKA\_MODIFIABLE** attribute determines whether or not an object is read-only.

The **CKA\_LABEL** attribute is intended to assist users in browsing.

The value of the CKA\_COPYABLE attribute determines whether or not an object can be copied. This attribute can be used in conjunction with CKA\_MODIFIABLE to prevent changes to the permitted usages of keys and other objects.

The value of the CKA\_DESTROYABLE attribute determines whether the object can be destroyed using C\_DestroyObject.

### The CKA\_UNIQUE\_ID attribute

Any time a new object is created, a value for CKA\_UNIQUE\_ID MUST be generated by the token and stored with the object. The specific algorithm used to generate unique ID values for objects is token-specific, but values generated MUST be unique across all objects visible to any particular session, and SHOULD be unique across all objects created by the token. Reinitializing the token, such as by calling C\_InitToken, MAY cause reuse of CKA\_UNIQUE\_ID values.

Any attempt to modify the CKA\_UNIQUE\_ID attribute of an existing object or to specify the value of the CKA\_UNIQUE\_ID attribute in the template for an operation that creates one or more objects MUST fail. Operations failing for this reason return the error code CKR\_ATTRIBUTE\_READ\_ONLY.

## Data objects

### Definitions

This section defines the object class CKO\_DATA for type CK\_OBJECT\_CLASS as used in the CKA\_CLASS attribute of objects.

### Overview

Data objects (object class **CKO\_DATA**) hold information defined by an application. Other than providing access to it, Cryptoki does not attach any special meaning to a data object. The following table lists the attributes supported by data objects, in addition to the common attributes defined for this object class:

Table , Data Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_APPLICATION | RFC2279 string | Description of the application that manages the object (default empty) |
| CKA\_OBJECT\_ID | Byte Array | DER-encoding of the object identifier indicating the data object type (default empty) |
| CKA\_VALUE | Byte array | Value of the object (default empty) |

The **CKA\_APPLICATION** attribute provides a means for applications to indicate ownership of the data objects they manage. Cryptoki does not provide a means of ensuring that only a particular application has access to a data object, however.

The **CKA\_OBJECT\_ID** attribute provides an application independent and expandable way to indicate the type of the data object value. Cryptoki does not provide a means of insuring that the data object identifier matches the data value.

The following is a sample template containing attributes for creating a data object:

CK\_OBJECT\_CLASS class = CKO\_DATA;

CK\_UTF8CHAR label[] = “A data object”;

CK\_UTF8CHAR application[] = “An application”;

CK\_BYTE data[] = “Sample data”;

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

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

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

{CKA\_APPLICATION, application, sizeof(application)-1},

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

};

## Certificate objects

### Definitions

This section defines the object class CKO\_CERTIFICATE for type CK\_OBJECT\_CLASS as used in the CKA\_CLASS attribute of objects.

### Overview

Certificate objects (object class **CKO\_CERTIFICATE**) hold public-key or attribute certificates. Other than providing access to certificate objects, Cryptoki does not attach any special meaning to certificates. The following table defines the common certificate object attributes, in addition to the common attributes defined for this object class:

Table , Common Certificate Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_CERTIFICATE\_TYPE1 | CK\_CERTIFICATE\_TYPE | Type of certificate |
| CKA\_TRUSTED10 | CK\_BBOOL | The certificate can be trusted for the application that it was created. |
| CKA\_CERTIFICATE\_CATEGORY | CKA\_CERTIFICATE\_CATEGORY | (default CK\_CERTIFICATE\_CATEGORY\_UNSPECIFIED) |
| CKA\_CHECK\_VALUE | Byte array | Checksum |
| CKA\_START\_DATE | CK\_DATE | Start date for the certificate (default empty) |
| CKA\_END\_DATE | CK\_DATE | End date for the certificate (default empty) |
| CKA\_PUBLIC\_KEY\_INFO | Byte Array | DER-encoding of the SubjectPublicKeyInfo for the public key contained in this certificate (default empty) |

- Refer to Table 11 for footnotes

Cryptoki does not enforce the relationship of the CKA\_PUBLIC\_KEY\_INFO to the public key in the certificate, but does recommend that the key be extracted from the certificate to create this value.

The **CKA\_CERTIFICATE\_TYPE** attribute may not be modified after an object is created. This version of Cryptoki supports the following certificate types:

* X.509 public key certificate
* WTLS public key certificate
* X.509 attribute certificate

The **CKA\_TRUSTED** attribute cannot be set to CK\_TRUE by an application. It MUST be set by a token initialization application or by the token’s SO. Trusted certificates cannot be modified.

The **CKA\_CERTIFICATE\_CATEGORY** attribute is used to indicate if a stored certificate is a user certificate for which the corresponding private key is available on the token (“token user”), a CA certificate (“authority”), or another end-entity certificate (“other entity”). This attribute may not be modified after an object is created.

The **CKA\_CERTIFICATE\_CATEGORY** and **CKA\_TRUSTED** attributes will together be used to map to the categorization of the certificates.

**CKA\_CHECK\_VALUE**: The value of this attribute is derived from the certificate by taking the first three bytes of the SHA-1 hash of the certificate object’s CKA\_VALUE attribute.

The **CKA\_START\_DATE** and **CKA\_END\_DATE** attributes are for reference only; Cryptoki does not attach any special meaning to them. When present, the application is responsible to set them to values that match the certificate’s encoded “not before” and “not after” fields (if any).

### X.509 public key certificate objects

X.509 certificate objects (certificate type **CKC\_X\_509**) hold X.509 public key certificates. The following table defines the X.509 certificate object attributes, in addition to the common attributes defined for this object class:

Table , X.509 Certificate Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_SUBJECT1 | Byte array | DER-encoding of the certificate subject name |
| CKA\_ID | Byte array | Key identifier for public/private key pair (default empty) |
| CKA\_ISSUER | Byte array | DER-encoding of the certificate issuer name (default empty) |
| CKA\_SERIAL\_NUMBER | Byte array | DER-encoding of the certificate serial number (default empty) |
| CKA\_VALUE2 | Byte array | BER-encoding of the certificate |
| CKA\_URL3 | RFC2279 string | If not empty this attribute gives the URL where the complete certificate can be obtained (default empty) |
| CKA\_HASH\_OF\_SUBJECT\_PUBLIC\_KEY4 | Byte array | Hash of the subject public key (default empty). Hash algorithm is defined by CKA\_NAME\_HASH\_ALGORITHM |
| CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY4 | Byte array | Hash of the issuer public key (default empty). Hash algorithm is defined by CKA\_NAME\_HASH\_ALGORITHM |
| CKA\_JAVA\_MIDP\_SECURITY\_DOMAIN | CK\_JAVA\_MIDP\_SECURITY\_DOMAIN | Java MIDP security domain. (default CK\_SECURITY\_DOMAIN\_UNSPECIFIED) |
| CKA\_NAME\_HASH\_ALGORITHM | CK\_MECHANISM\_TYPE | Defines the mechanism used to calculate CKA\_HASH\_OF\_SUBJECT\_PUBLIC\_KEY and CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY. If the attribute is not present then the type defaults to SHA-1. |

1MUST be specified when the object is created. 2MUST be specified when the object is created. MUST be non-empty if CKA\_URL is empty.

3MUST be non-empty if CKA\_VALUE is empty.

4Can only be empty if CKA\_URL is empty.

Only the **CKA\_ID**, **CKA\_ISSUER**, and **CKA\_SERIAL\_NUMBER** attributes may be modified after the object is created.

The **CKA\_ID** attribute is intended as a means of distinguishing multiple public-key/private-key pairs held by the same subject (whether stored in the same token or not). (Since the keys are distinguished by subject name as well as identifier, it is possible that keys for different subjects may have the same **CKA\_ID** value without introducing any ambiguity.)

It is intended in the interests of interoperability that the subject name and key identifier for a certificate will be the same as those for the corresponding public and private keys (though it is not required that all be stored in the same token). However, Cryptoki does not enforce this association, or even the uniqueness of the key identifier for a given subject; in particular, an application may leave the key identifier empty.

The **CKA\_ISSUER** and **CKA\_SERIAL\_NUMBER** attributes are for compatibility with PKCS #7 and Privacy Enhanced Mail (RFC1421). Note that with the version 3 extensions to X.509 certificates, the key identifier may be carried in the certificate. It is intended that the **CKA\_ID** value be identical to the key identifier in such a certificate extension, although this will not be enforced by Cryptoki.

The **CKA\_URL** attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.

The **CKA\_HASH\_OF\_SUBJECT\_PUBLIC\_KEY** and **CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY** attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer. The hash algorithm is defined by CKA\_NAME\_HASH\_ALGORITHM.

The **CKA\_JAVA\_MIDP\_SECURITY\_DOMAIN** attribute associates a certificate with a Java MIDP security domain.

The following is a sample template for creating an X.509 certificate object:

CK\_OBJECT\_CLASS class = CKO\_CERTIFICATE;

CK\_CERTIFICATE\_TYPE certType = CKC\_X\_509;

CK\_UTF8CHAR label[] = “A certificate object”;

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

CK\_BYTE id[] = {123};

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

{CKA\_CERTIFICATE\_TYPE, &certType, sizeof(certType)};

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

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

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

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

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

};

### WTLS public key certificate objects

WTLS certificate objects (certificate type **CKC\_WTLS**) hold WTLS public key certificates. The following table defines the WTLS certificate object attributes, in addition to the common attributes defined for this object class.

Table : WTLS Certificate Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_SUBJECT1 | Byte array | WTLS-encoding (Identifier type) of the certificate subject |
| CKA\_ISSUER | Byte array | WTLS-encoding (Identifier type) of the certificate issuer (default empty) |
| CKA\_VALUE2 | Byte array | WTLS-encoding of the certificate |
| CKA\_URL3 | RFC2279 string | If not empty this attribute gives the URL where the complete certificate can be obtained |
| CKA\_HASH\_OF\_SUBJECT\_PUBLIC\_KEY4 | Byte array | SHA-1 hash of the subject public key (default empty). Hash algorithm is defined by CKA\_NAME\_HASH\_ALGORITHM |
| CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY4 | Byte array | SHA-1 hash of the issuer public key (default empty). Hash algorithm is defined by CKA\_NAME\_HASH\_ALGORITHM |
| CKA\_NAME\_HASH\_ALGORITHM | CK\_MECHANISM\_TYPE | Defines the mechanism used to calculate CKA\_HASH\_OF\_SUBJECT\_PUBLIC\_KEY and CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY. If the attribute is not present then the type defaults to SHA-1. |

1MUST be specified when the object is created. Can only be empty if CKA\_VALUE is empty.

2MUST be specified when the object is created. MUST be non-empty if CKA\_URL is empty.

3MUST be non-empty if CKA\_VALUE is empty.

4Can only be empty if CKA\_URL is empty.

Only the **CKA\_ISSUER** attribute may be modified after the object has been created.

The encoding for the **CKA\_SUBJECT**, **CKA\_ISSUER**, and **CKA\_VALUE** attributes can be found in [WTLS].

The **CKA\_URL** attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.

The **CKA\_HASH\_OF\_SUBJECT\_PUBLIC\_KEY** and **CKA\_HASH\_OF\_ISSUER\_PUBLIC\_KEY** attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer. The hash algorithm is defined by CKA\_NAME\_HASH\_ALGORITHM.

The following is a sample template for creating a WTLS certificate object:

CK\_OBJECT\_CLASS class = CKO\_CERTIFICATE;

CK\_CERTIFICATE\_TYPE certType = CKC\_WTLS;

CK\_UTF8CHAR label[] = “A certificate object”;

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] =

{

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

{CKA\_CERTIFICATE\_TYPE, &certType, sizeof(certType)};

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

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

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

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

};

### X.509 attribute certificate objects

X.509 attribute certificate objects (certificate type **CKC\_X\_509\_ATTR\_CERT**) hold X.509 attribute certificates. The following table defines the X.509 attribute certificate object attributes, in addition to the common attributes defined for this object class:

Table , X.509 Attribute Certificate Object Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Meaning** |
| CKA\_OWNER1 | Byte Array | DER-encoding of the attribute certificate's subject field. This is distinct from the CKA\_SUBJECT attribute contained in CKC\_X\_509 certificates because the ASN.1 syntax and encoding are different. |
| CKA\_AC\_ISSUER | Byte Array | DER-encoding of the attribute certificate's issuer field. This is distinct from the CKA\_ISSUER attribute contained in CKC\_X\_509 certificates because the ASN.1 syntax and encoding are different. (default empty) |
| CKA\_SERIAL\_NUMBER | Byte Array | DER-encoding of the certificate serial number. (default empty) |
| CKA\_ATTR\_TYPES | Byte Array | BER-encoding of a sequence of object identifier values corresponding to the attribute types contained in the certificate. When present, this field offers an opportunity for applications to search for a particular attribute certificate without fetching and parsing the certificate itself. (default empty) |
| CKA\_VALUE1 | Byte Array | BER-encoding of the certificate. |

1MUST be specified when the object is created

Only the **CKA\_AC\_ISSUER**, **CKA\_SERIAL\_NUMBER** and **CKA\_ATTR\_TYPES** attributes may be modified after the object is created.

The following is a sample template for creating an X.509 attribute certificate object:

CK\_OBJECT\_CLASS class = CKO\_CERTIFICATE;

CK\_CERTIFICATE\_TYPE certType = CKC\_X\_509\_ATTR\_CERT;

CK\_UTF8CHAR label[] = "An attribute certificate object";

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

{CKA\_CERTIFICATE\_TYPE, &certType, sizeof(certType)};

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

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

{CKA\_OWNER, owner, sizeof(owner)},

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

};

## Key objects

### Definitions

There is no CKO\_ definition for the base key object class, only for the key types derived from it.

This section defines the object class CKO\_PUBLIC\_KEY, CKO\_PRIVATE\_KEY and CKO\_SECRET\_KEY for type CK\_OBJECT\_CLASS as used in the CKA\_CLASS attribute of objects.

### Overview

Key objects hold encryption or authentication keys, which can be public keys, private keys, or secret keys. The following common footnotes apply to all the tables describing attributes of keys:

The following table defines the attributes common to public key, private key and secret key classes, in addition to the common attributes defined for this object class:

Table , Common Key Attributes

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_KEY\_TYPE1,5 | CK\_KEY\_TYPE | Type of key |
| CKA\_ID8 | Byte array | Key identifier for key (default empty) |
| CKA\_START\_DATE8 | CK\_DATE | Start date for the key (default empty) |
| CKA\_END\_DATE8 | CK\_DATE | End date for the key (default empty) |
| CKA\_DERIVE8 | CK\_BBOOL | CK\_TRUE if key supports key derivation (*i.e.*, if other keys can be derived from this one (default CK\_FALSE) |
| CKA\_LOCAL2,4,6 | CK\_BBOOL | CK\_TRUE only if key was either   1. generated locally (*i.e.*, on the token) with a **C\_GenerateKey** or **C\_GenerateKeyPair** call 2. created with a **C\_CopyObject** call as a copy of a key which had its **CKA\_LOCAL** attribute set to CK\_TRUE |
| CKA\_KEY\_GEN\_ MECHANISM2,4,6 | CK\_MECHANISM\_TYPE | Identifier of the mechanism used to generate the key material. |
| CKA\_ALLOWED\_MECHANISMS | CK\_MECHANISM\_TYPE \_PTR, pointer to a CK\_MECHANISM\_TYPE array | A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the *ulValueLen* component of the attribute divided by the size  of CK\_MECHANISM\_TYPE. |

- Refer to Table 11 for footnotes

The **CKA\_ID** field is intended to distinguish among multiple keys. In the case of public and private keys, this field assists in handling multiple keys held by the same subject; the key identifier for a public key and its corresponding private key should be the same. The key identifier should also be the same as for the corresponding certificate, if one exists. Cryptoki does not enforce these associations, however. (See Section 4.6 for further commentary.)

In the case of secret keys, the meaning of the **CKA\_ID** attribute is up to the application.

Note that the **CKA\_START\_DATE** and **CKA\_END\_DATE** attributes are for reference only; Cryptoki does not attach any special meaning to them. In particular, it does not restrict usage of a key according to the dates; doing this is up to the application.

The **CKA\_DERIVE** attribute has the value CK\_TRUE if and only if it is possible to derive other keys from the key.

The **CKA\_LOCAL** attribute has the value CK\_TRUE if and only if the value of the key was originally generated on the token by a **C\_GenerateKey** or **C\_GenerateKeyPair** call.

The **CKA\_KEY\_GEN\_MECHANISM** attribute identifies the key generation mechanism used to generate the key material. It contains a valid value only if the **CKA\_LOCAL** attribute has the value CK\_TRUE. If **CKA\_LOCAL** has the value CK\_FALSE, the value of the attribute is CK\_UNAVAILABLE\_INFORMATION.

## Public key objects

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

Table , Common Public Key Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_SUBJECT8 | Byte array | DER-encoding of the key subject name (default empty) |
| CKA\_ENCRYPT8 | CK\_BBOOL | CK\_TRUE if key supports encryption9 |
| CKA\_VERIFY8 | CK\_BBOOL | CK\_TRUE if key supports verification where the signature is an appendix to the data9 |
| CKA\_VERIFY\_RECOVER8 | CK\_BBOOL | CK\_TRUE if key supports verification where the data is recovered from the signature9 |
| CKA\_WRAP8 | CK\_BBOOL | CK\_TRUE if key supports wrapping (*i.e.*, can be used to wrap other keys)9 |
| CKA\_TRUSTED10 | CK\_BBOOL | The key can be trusted for the application that it was created.  The wrapping key can be used to wrap keys with CKA\_WRAP\_WITH\_TRUSTED set to CK\_TRUE. |
| CKA\_WRAP\_TEMPLATE | CK\_ATTRIBUTE\_PTR | For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the *ulValueLen* component of the attribute divided by the size of CK\_ATTRIBUTE. |
| CKA\_PUBLIC\_KEY\_INFO | Byte array | DER-encoding of the SubjectPublicKeyInfo for this public key. (MAY be empty, DEFAULT derived from the underlying public key data) |

- Refer to Table 11 for footnotes

It is intended in the interests of interoperability that the subject name and key identifier for a public key will be the same as those for the corresponding certificate and private key. However, Cryptoki does not enforce this, and it is not required that the certificate and private key also be stored on the token.

To map between ISO/IEC 9594-8 (X.509) **keyUsage** flags for public keys and the PKCS #11 attributes for public keys, use the following table.

Table , Mapping of X.509 key usage flags to Cryptoki attributes for public keys

|  |  |
| --- | --- |
| **Key usage flags for public keys in X.509 public key certificates** | **Corresponding cryptoki attributes for public keys.** |
| dataEncipherment | CKA\_ENCRYPT |
| digitalSignature, keyCertSign, cRLSign | CKA\_VERIFY |
| digitalSignature, keyCertSign, cRLSign | CKA\_VERIFY\_RECOVER |
| keyAgreement | CKA\_DERIVE |
| keyEncipherment | CKA\_WRAP |
| nonRepudiation | CKA\_VERIFY |
| nonRepudiation | CKA\_VERIFY\_RECOVER |

The value of the CKA\_PUBLIC\_KEY\_INFO attribute is the DER encoded value of SubjectPublicKeyInfo:

SubjectPublicKeyInfo **::**= SEQUENCE {

algorithm AlgorithmIdentifier,

subjectPublicKey BIT\_STRING }

The encodings for the subjectPublicKey field are specified in the description of the public key types in the appropriate sections for the key types defined within this specification.

## Private key objects

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

Table , Common Private Key Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_SUBJECT8 | Byte array | DER-encoding of certificate subject name (default empty) |
| CKA\_SENSITIVE8,11 | CK\_BBOOL | CK\_TRUE if key is sensitive9 |
| CKA\_DECRYPT8 | CK\_BBOOL | CK\_TRUE if key supports decryption9 |
| CKA\_SIGN8 | CK\_BBOOL | CK\_TRUE if key supports signatures where the signature is an appendix to the data9 |
| CKA\_SIGN\_RECOVER8 | CK\_BBOOL | CK\_TRUE if key supports signatures where the data can be recovered from the signature9 |
| CKA\_UNWRAP8 | CK\_BBOOL | CK\_TRUE if key supports unwrapping (*i.e.*, can be used to unwrap other keys)9 |
| CKA\_EXTRACTABLE8,12 | CK\_BBOOL | CK\_TRUE if key is extractable and can be wrapped 9 |
| CKA\_ALWAYS\_SENSITIVE2,4,6 | CK\_BBOOL | CK\_TRUE if key has *always* had the CKA\_SENSITIVE attribute set to CK\_TRUE |
| CKA\_NEVER\_EXTRACTABLE2,4,6 | CK\_BBOOL | CK\_TRUE if key has *never* had the CKA\_EXTRACTABLE attribute set to CK\_TRUE |
| CKA\_WRAP\_WITH\_TRUSTED11 | CK\_BBOOL | CK\_TRUE if the key can only be wrapped with a wrapping key that has CKA\_TRUSTED set to CK\_TRUE.  Default is CK\_FALSE. |
| CKA\_UNWRAP\_TEMPLATE | CK\_ATTRIBUTE\_PTR | For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the *ulValueLen* component of the attribute divided by the size of  CK\_ATTRIBUTE. |
| CKA\_ALWAYS\_AUTHENTICATE | CK\_BBOOL | If CK\_TRUE, the user has to supply the PIN for each use (sign or decrypt) with the key. Default is CK\_FALSE. |
| CKA\_PUBLIC\_KEY\_INFO8 | Byte Array | DER-encoding of the SubjectPublicKeyInfo for the associated public key (MAY be empty; DEFAULT derived from the underlying private key data; MAY be manually set for specific key types; if set; MUST be consistent with the underlying private key data) |
| CKA\_DERIVE\_TEMPLATE | CK\_ATTRIBUTE\_PTR | For deriving keys. The attribute template to match against any keys derived using this derivation key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK\_ATTRIBUTE. |

- Refer to Table 11 for footnotes

It is intended in the interests of interoperability that the subject name and key identifier for a private key will be the same as those for the corresponding certificate and public key. However, this is not enforced by Cryptoki, and it is not required that the certificate and public key also be stored on the token.

If the **CKA\_SENSITIVE** attribute is CK\_TRUE, or if the **CKA\_EXTRACTABLE** attribute is CK\_FALSE, then certain attributes of the private key cannot be revealed in plaintext outside the token. Which attributes these are is specified for each type of private key in the attribute table in the section describing that type of key.

The **CKA\_ALWAYS\_AUTHENTICATE** attribute can be used to force re-authentication (i.e. force the user to provide a PIN) for each use of a private key. “Use” in this case means a cryptographic operation such as sign or decrypt. This attribute may only be set to CK\_TRUE when **CKA\_PRIVATE** is also CK\_TRUE.

Re-authentication occurs by calling **C\_Login** with *userType* set to **CKU\_CONTEXT\_SPECIFIC** immediately after a cryptographic operation using the key has been initiated (e.g. after **C\_SignInit**). In this call, the actual user type is implicitly given by the usage requirements of the active key. If **C\_Login** returns CKR\_OK the user was successfully authenticated and this sets the active key in an authenticated state that lasts until the cryptographic operation has successfully or unsuccessfully been completed (e.g. by **C\_Sign**, **C\_SignFinal**,..). A return value CKR\_PIN\_INCORRECT from **C\_Login** means that the user was denied permission to use the key and continuing the cryptographic operation will result in a behavior as if **C\_Login** had not been called. In both of these cases the session state will remain the same, however repeated failed re-authentication attempts may cause the PIN to be locked. **C\_Login** returns in this case CKR\_PIN\_LOCKED and this also logs the user out from the token. Failing or omitting to re-authenticate when CKA\_ALWAYS\_AUTHENTICATE is set to CK\_TRUE will result in CKR\_USER\_NOT\_LOGGED\_IN to be returned from calls using the key. **C\_Login** will return CKR\_OPERATION\_NOT\_INITIALIZED, but the active cryptographic operation will not be affected, if an attempt is made to re-authenticate when CKA\_ALWAYS\_AUTHENTICATE is set to CK\_FALSE.

The **CKA\_PUBLIC\_KEY\_INFO** attribute represents the public key associated with this private key. The data it represents may either be stored as part of the private key data, or regenerated as needed from the private key.

If this attribute is supplied as part of a template for **C\_CreateObject, C\_CopyObject** or C**\_SetAttributeValue** for a private key, the token MUST verify correspondence between the private key data and the public key data as supplied in **CKA\_PUBLIC\_KEY\_INFO**. This can be done either by deriving a public key from the private key and comparing the values, or by doing a sign and verify operation. If there is a mismatch, the command SHALL return **CKR\_ATTRIBUTE\_VALUE\_INVALID.** A token MAY choose not to support the **CKA\_PUBLIC\_KEY\_INFO** attribute for commands which create new private keys. If it does not support the attribute, the command SHALL return **CKR\_ATTRIBUTE\_TYPE\_INVALID**.

As a general guideline, private keys of any type SHOULD store sufficient information to retrieve the public key information. In particular, the RSA private key description has been modified in PKCS #11 V2.40 to add the CKA\_PUBLIC\_EXPONENT to the list of attributes required for an RSA private key. All other private key types described in this specification contain sufficient information to recover the associated public key.

## Secret key objects

Secret key objects (object class **CKO\_SECRET\_KEY**) hold secret keys. The following table defines the attributes common to all secret keys, in addition to the common attributes defined for this object class:

Table , Common Secret Key Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_SENSITIVE8,11 | CK\_BBOOL | CK\_TRUE if object is sensitive (default CK\_FALSE) |
| CKA\_ENCRYPT8 | CK\_BBOOL | CK\_TRUE if key supports encryption9 |
| CKA\_DECRYPT8 | CK\_BBOOL | CK\_TRUE if key supports decryption9 |
| CKA\_SIGN8 | CK\_BBOOL | CK\_TRUE if key supports signatures (*i.e.*, authentication codes) where the signature is an appendix to the data9 |
| CKA\_VERIFY8 | CK\_BBOOL | CK\_TRUE if key supports verification (*i.e.*, of authentication codes) where the signature is an appendix to the data9 |
| CKA\_WRAP8 | CK\_BBOOL | CK\_TRUE if key supports wrapping (*i.e.*, can be used to wrap other keys)9 |
| CKA\_UNWRAP8 | CK\_BBOOL | CK\_TRUE if key supports unwrapping (*i.e.*, can be used to unwrap other keys)9 |
| CKA\_EXTRACTABLE8,12 | CK\_BBOOL | CK\_TRUE if key is extractable and can be wrapped 9 |
| CKA\_ALWAYS\_SENSITIVE2,4,6 | CK\_BBOOL | CK\_TRUE if key has *always* had the CKA\_SENSITIVE attribute set to CK\_TRUE |
| CKA\_NEVER\_EXTRACTABLE2,4,6 | CK\_BBOOL | CK\_TRUE if key has *never* had the CKA\_EXTRACTABLE attribute set to CK\_TRUE |
| CKA\_CHECK\_VALUE | Byte array | Key checksum |
| CKA\_WRAP\_WITH\_TRUSTED11 | CK\_BBOOL | CK\_TRUE if the key can only be wrapped with a wrapping key that has CKA\_TRUSTED set to CK\_TRUE.  Default is CK\_FALSE. |
| CKA\_TRUSTED10 | CK\_BBOOL | The wrapping key can be used to wrap keys with CKA\_WRAP\_WITH\_TRUSTED set to CK\_TRUE. |
| CKA\_WRAP\_TEMPLATE | CK\_ATTRIBUTE\_PTR | For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the  *ulValueLen* component of the attribute divided by the size of  CK\_ATTRIBUTE |
| CKA\_UNWRAP\_TEMPLATE | CK\_ATTRIBUTE\_PTR | For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the *ulValueLen* component of the attribute divided by the size of  CK\_ATTRIBUTE. |
| A\_DERIVE\_TEMPLATE | CK\_ATTRIBUTE\_PTR | For deriving keys. The attribute template to match against any keys derived using this derivation key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the ulValueLen component of the attribute divided by the size of CK\_ATTRIBUTE. |

- Refer to Table 11 for footnotes

If the **CKA\_SENSITIVE** attribute is CK\_TRUE, or if the **CKA\_EXTRACTABLE** attribute is CK\_FALSE, then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which attributes these are is specified for each type of secret key in the attribute table in the section describing that type of key.

The key check value (KCV) attribute for symmetric key objects to be called **CKA\_CHECK\_VALUE,** of type byte array, length 3 bytes, operates like a fingerprint, or checksum of the key. They are intended to be used to cross-check symmetric keys against other systems where the same key is shared, and as a validity check after manual key entry or restore from backup. Refer to object definitions of specific key types for KCV algorithms.

Properties:

1. For two keys that are cryptographically identical the value of this attribute should be identical.
2. CKA\_CHECK\_VALUE should not be usable to obtain any part of the key value.
3. Non-uniqueness. Two different keys can have the same CKA\_CHECK\_VALUE. This is unlikely (the probability can easily be calculated) but possible.

The attribute is optional, but if supported, regardless of how the key object is created or derived, the value of the attribute is always supplied. It SHALL be supplied even if the encryption operation for the key is forbidden (i.e. when CKA\_ENCRYPT is set to CK\_FALSE).

If a value is supplied in the application template (allowed but never necessary) then, if supported, it MUST match what the library calculates it to be or the library returns a CKR\_ATTRIBUTE\_VALUE\_INVALID. If the library does not support the attribute then it should ignore it. Allowing the attribute in the template this way does no harm and allows the attribute to be treated like any other attribute for the purposes of key wrap and unwrap where the attributes are preserved also.

The generation of the KCV may be prevented by the application supplying the attribute in the template as a no-value (0 length) entry. The application can query the value at any time like any other attribute using C\_GetAttributeValue. C\_SetAttributeValue may be used to destroy the attribute, by supplying no-value.

Unless otherwise specified for the object definition, the value of this attribute is derived from the key object by taking the first three bytes of an encryption of a single block of null (0x00) bytes, using the default cipher and mode (e.g. ECB) associated with the key type of the secret key object.

## Domain parameter objects

### Definitions

This section defines the object class CKO\_DOMAIN\_PARAMETERS for type CK\_OBJECT\_CLASS as used in the CKA\_CLASS attribute of objects.

### Overview

This object class was created to support the storage of certain algorithm's extended parameters. DSA and DH both use domain parameters in the key-pair generation step. In particular, some libraries support the generation of domain parameters (originally out of scope for PKCS11) so the object class was added.

To use a domain parameter object you MUST extract the attributes into a template and supply them (still in the template) to the corresponding key-pair generation function.

Domain parameter objects (object class **CKO\_DOMAIN\_PARAMETERS**) hold public domain parameters.

The following table defines the attributes common to domain parameter objects in addition to the common attributes defined for this object class:

Table , Common Domain Parameter Attributes

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_KEY\_TYPE1 | CK\_KEY\_TYPE | Type of key the domain parameters can be used to generate. |
| CKA\_LOCAL2,4 | CK\_BBOOL | CK\_TRUE only if domain parameters were either   1. generated locally (*i.e.*, on the token) with a **C\_GenerateKey** 2. created with a **C\_CopyObject** call as a copy of domain parameters which had its **CKA\_LOCAL** attribute set to CK\_TRUE |

- Refer to Table 11 for footnotes

The **CKA\_LOCAL** attribute has the value CK\_TRUE if and only if the values of the domain parameters were originally generated on the token by a **C\_GenerateKey** call.

## Mechanism objects

### Definitions

This section defines the object class CKO\_MECHANISM for type CK\_OBJECT\_CLASS as used in the CKA\_CLASS attribute of objects.

### Overview

Mechanism objects provide information about mechanisms supported by a device beyond that given by the **CK\_MECHANISM\_INFO** structure.

When searching for objects using **C\_FindObjectsInit** and **C\_FindObjects**, mechanism objects are not returned unless the **CKA\_CLASS** attribute in the template has the value **CKO\_MECHANISM**. This protects applications written to previous versions of Cryptoki from finding objects that they do not understand.

Table , Common Mechanism Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Data Type** | **Meaning** |
| CKA\_MECHANISM\_TYPE | CK\_MECHANISM\_TYPE | The type of mechanism object |

The **CKA\_MECHANISM\_TYPE** attribute may not be set.

## Profile objects

### Definitions

This section defines the object class CKO\_PROFILE for type CK\_OBJECT\_CLASS as used in the CKA\_CLASS attribute of objects.

### Overview

Profile objects (object class CKO\_PROFILE) describe which PKCS #11 profiles the token implements. Profiles are defined in the OASIS PKCS #11 Cryptographic Token Interface Profiles document. A given token can contain more than one profile ID. The following table lists the attributes supported by profile objects, in addition to the common attributes defined for this object class:

Table , Profile Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_PROFILE\_ID | CK\_PROFILE\_ID | ID of the supported profile. |

The **CKA\_PROFILE\_ID** attribute identifies a profile that the token supports.

# Functions

Cryptoki's functions are organized into the following categories:

* general-purpose functions (4 functions)
* slot and token management functions (9 functions)
* session management functions (8 functions)
* object management functions (9 functions)
* encryption functions (4 functions)
* message-based encryption functions (5 functions)
* decryption functions (4 functions)
* message digesting functions (5 functions)
* signing and MACing functions (6 functions)
* functions for verifying signatures and MACs (6 functions)
* dual-purpose cryptographic functions (4 functions)
* key management functions (5 functions)
* random number generation functions (2 functions)
* parallel function management functions (2 functions)

In addition to these functions, Cryptoki can use application-supplied callback functions to notify an application of certain events, and can also use application-supplied functions to handle mutex objects for safe multi-threaded library access.

The Cryptoki API functions are presented in the following table:

Table , Summary of Cryptoki Functions

| **Category** | **Function** | **Description** |
| --- | --- | --- |
| General | C\_Initialize | initializes Cryptoki |
| purpose functions | C\_Finalize | clean up miscellaneous Cryptoki-associated resources |
|  | C\_GetInfo | obtains general information about Cryptoki |
|  | C\_GetFunctionList | obtains entry points of Cryptoki library functions |
|  | C\_GetInterfaceList | obtains list of interfaces supported by Cryptoki library |
|  | C\_GetInterface | obtains interface specific entry points to Cryptoki library functions |
| Slot and token | C\_GetSlotList | obtains a list of slots in the system |
| management | C\_GetSlotInfo | obtains information about a particular slot |
| functions | C\_GetTokenInfo | obtains information about a particular token |
|  | C\_WaitForSlotEvent | waits for a slot event (token insertion, removal, etc.) to occur |
|  | C\_GetMechanismList | obtains a list of mechanisms supported by a token |
|  | C\_GetMechanismInfo | obtains information about a particular mechanism |
|  | C\_InitToken | initializes a token |
|  | C\_InitPIN | initializes the normal user’s PIN |
|  | C\_SetPIN | modifies the PIN of the current user |
| Session management functions | C\_OpenSession | opens a connection between an application and a particular token or sets up an application callback for token insertion |
|  | C\_CloseSession | closes a session |
|  | C\_CloseAllSessions | closes all sessions with a token |
|  | C\_GetSessionInfo | obtains information about the session |
|  | C\_SessionCancel | terminates active session based operations |
|  | C\_GetOperationState | obtains the cryptographic operations state of a session |
|  | C\_SetOperationState | sets the cryptographic operations state of a session |
|  | C\_Login | logs into a token |
|  | C\_LoginUser | logs into a token with explicit user name |
|  | C\_Logout | logs out from a token |
| Object | C\_CreateObject | creates an object |
| management | C\_CopyObject | creates a copy of an object |
| functions | C\_DestroyObject | destroys an object |
|  | C\_GetObjectSize | obtains the size of an object in bytes |
|  | C\_GetAttributeValue | obtains an attribute value of an object |
|  | C\_SetAttributeValue | modifies an attribute value of an object |
|  | C\_FindObjectsInit | initializes an object search operation |
|  | C\_FindObjects | continues an object search operation |
|  | C\_FindObjectsFinal | finishes an object search operation |
| Encryption | C\_EncryptInit | initializes an encryption operation |
| functions | C\_Encrypt | encrypts single-part data |
|  | C\_EncryptUpdate | continues a multiple-part encryption operation |
|  | C\_EncryptFinal | finishes a multiple-part encryption operation |
| Message-based Encryption Functions | C\_MessageEncryptInit | initializes a message-based encryption process |
| C\_EncryptMessage | encrypts a single-part message |
| C\_EncryptMessageBegin | begins a multiple-part message encryption operation |
| C\_EncryptMessageNext | continues or finishes a multiple-part message encryption operation |
| C\_MessageEncryptFinal | finishes a message-based encryption process |
| Decryption | C\_DecryptInit | initializes a decryption operation |
| Functions | C\_Decrypt | decrypts single-part encrypted data |
|  | C\_DecryptUpdate | continues a multiple-part decryption operation |
|  | C\_DecryptFinal | finishes a multiple-part decryption operation |
| Message-based  Decryption  Functions | C\_MessageDecryptInit | initializes a message decryption operation |
| C\_DecryptMessage | decrypts single-part data |
| C\_DecryptMessageBegin | starts a multiple-part message decryption operation |
| C\_DecryptMessageNext | Continues and finishes a multiple-part message decryption operation |
| C\_MessageDecryptFinal | finishes a message decryption operation |
| Message | C\_DigestInit | initializes a message-digesting operation |
| Digesting | C\_Digest | digests single-part data |
| Functions | C\_DigestUpdate | continues a multiple-part digesting operation |
|  | C\_DigestKey | digests a key |
|  | C\_DigestFinal | finishes a multiple-part digesting operation |
| Signing | C\_SignInit | initializes a signature operation |
| and MACing | C\_Sign | signs single-part data |
| functions | C\_SignUpdate | continues a multiple-part signature operation |
|  | C\_SignFinal | finishes a multiple-part signature operation |
|  | C\_SignRecoverInit | initializes a signature operation, where the data can be recovered from the signature |
|  | C\_SignRecover | signs single-part data, where the data can be recovered from the signature |
| Message-based Signature functions | C\_MessageSignInit | initializes a message signature operation |
| C\_SignMessage | signs single-part data |
| C\_SignMessageBegin | starts a multiple-part message signature operation |
| C\_SignMessageNext | continues and finishes a multiple-part message signature operation |
| C\_MessageSignFinal | finishes a message signature operation |
| Functions for verifying | C\_VerifyInit | initializes a verification operation |
| signatures | C\_Verify | verifies a signature on single-part data |
| and MACs | C\_VerifyUpdate | continues a multiple-part verification operation |
|  | C\_VerifyFinal | finishes a multiple-part verification operation |
|  | C\_VerifyRecoverInit | initializes a verification operation where the data is recovered from the signature |
|  | C\_VerifyRecover | verifies a signature on single-part data, where the data is recovered from the signature |
| Message-based Functions for verifying signatures and MACs | C\_MessageVerifyInit | initializes a message verification operation |
| C\_VerifyMessage | verifies single-part data |
| C\_VerifyMessageBegin | starts a multiple-part message verification operation |
| C\_VerifyMessageNext | continues and finishes a multiple-part message verification operation |
| C\_MessageVerifyFinal | finishes a message verification operation |
| Dual-purpose cryptographic | C\_DigestEncryptUpdate | continues simultaneous multiple-part digesting and encryption operations |
| functions | C\_DecryptDigestUpdate | continues simultaneous multiple-part decryption and digesting operations |
|  | C\_SignEncryptUpdate | continues simultaneous multiple-part signature and encryption operations |
|  | C\_DecryptVerifyUpdate | continues simultaneous multiple-part decryption and verification operations |
| Key | C\_GenerateKey | generates a secret key |
| management | C\_GenerateKeyPair | generates a public-key/private-key pair |
| functions | C\_WrapKey | wraps (encrypts) a key |
|  | C\_UnwrapKey | unwraps (decrypts) a key |
|  | C\_DeriveKey | derives a key from a base key |
| Random number generation | C\_SeedRandom | mixes in additional seed material to the random number generator |
| functions | C\_GenerateRandom | generates random data |
| Parallel function management | C\_GetFunctionStatus | legacy function which always returns CKR\_FUNCTION\_NOT\_PARALLEL |
| functions | C\_CancelFunction | legacy function which always returns CKR\_FUNCTION\_NOT\_PARALLEL |
| Callback function |  | application-supplied function to process notifications from Cryptoki |

Execution of a Cryptoki function call is in general an all-or-nothing affair, *i.e.*, a function call accomplishes either its entire goal, or nothing at all.

* If a Cryptoki function executes successfully, it returns the value CKR\_OK.
* If a Cryptoki function does not execute successfully, it returns some value other than CKR\_OK, and the token is in the same state as it was in prior to the function call. If the function call was supposed to modify the contents of certain memory addresses on the host computer, these memory addresses may have been modified, despite the failure of the function.
* In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value CKR\_GENERAL\_ERROR. When this happens, the token and/or host computer may be in an inconsistent state, and the goals of the function may have been partially achieved.

There are a small number of Cryptoki functions whose return values do not behave precisely as described above; these exceptions are documented individually with the description of the functions themselves.

A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported function MUST have a “stub” in the library which simply returns the value CKR\_FUNCTION\_NOT\_SUPPORTED. The function’s entry in the library’s **CK\_FUNCTION\_LIST** structure (as obtained by **C\_GetFunctionList**) should point to this stub function (see Section 3.6).

## Function return values

The Cryptoki interface possesses a large number of functions and return values. In Section 5.1, we enumerate the various possible return values for Cryptoki functions; most of the remainder of Section 5.1 details the behavior of Cryptoki functions, including what values each of them may return.

Because of the complexity of the Cryptoki specification, it is recommended that Cryptoki applications attempt to give some leeway when interpreting Cryptoki functions’ return values. We have attempted to specify the behavior of Cryptoki functions as completely as was feasible; nevertheless, there are presumably some gaps. For example, it is possible that a particular error code which might apply to a particular Cryptoki function is unfortunately not actually listed in the description of that function as a possible error code. It is conceivable that the developer of a Cryptoki library might nevertheless permit his/her implementation of that function to return that error code. It would clearly be somewhat ungraceful if a Cryptoki application using that library were to terminate by abruptly dumping core upon receiving that error code for that function. It would be far preferable for the application to examine the function’s return value, see that it indicates some sort of error (even if the application doesn’t know precisely *what* kind of error), and behave accordingly.

See Section 5.1.8 for some specific details on how a developer might attempt to make an application that accommodates a range of behaviors from Cryptoki libraries.

### Universal Cryptoki function return values

Any Cryptoki function can return any of the following values:

* CKR\_GENERAL\_ERROR: Some horrible, unrecoverable error has occurred. In the worst case, it is possible that the function only partially succeeded, and that the computer and/or token is in an inconsistent state.
* CKR\_HOST\_MEMORY: The computer that the Cryptoki library is running on has insufficient memory to perform the requested function.
* CKR\_FUNCTION\_FAILED: The requested function could not be performed, but detailed information about why not is not available in this error return. If the failed function uses a session, it is possible that the **CK\_SESSION\_INFO** structure that can be obtained by calling **C\_GetSessionInfo** will hold useful information about what happened in its *ulDeviceError* field. In any event, although the function call failed, the situation is not necessarily totally hopeless, as it is likely to be when CKR\_GENERAL\_ERROR is returned. Depending on what the root cause of the error actually was, it is possible that an attempt to make the exact same function call again would succeed.
* CKR\_OK: The function executed successfully. Technically, CKR\_OK is not *quite* a “universal” return value; in particular, the legacy functions **C\_GetFunctionStatus** and **C\_CancelFunction** (see Section 5.20) cannot return CKR\_OK.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR\_GENERAL\_ERROR or CKR\_HOST\_MEMORY would be an appropriate error return, then CKR\_GENERAL\_ERROR should be returned.

### Cryptoki function return values for functions that use a session handle

Any Cryptoki function that takes a session handle as one of its arguments (i.e., any Cryptoki function except for C\_Initialize, C\_Finalize, C\_GetInfo, C\_GetFunctionList, C\_GetSlotList, C\_GetSlotInfo, C\_GetTokenInfo, C\_WaitForSlotEvent, C\_GetMechanismList, C\_GetMechanismInfo, C\_InitToken, C\_OpenSession, and C\_CloseAllSessions) can return the following values:

* CKR\_SESSION\_HANDLE\_INVALID: The specified session handle was invalid *at the time that the function was invoked*. Note that this can happen if the session’s token is removed before the function invocation, since removing a token closes all sessions with it.
* CKR\_DEVICE\_REMOVED: The token was removed from its slot *during the execution of the function*.
* CKR\_SESSION\_CLOSED: The session was closed *during the execution of the function*. Note that, as stated in **[PKCS11-UG]**, the behavior of Cryptoki is *undefined* if multiple threads of an application attempt to access a common Cryptoki session simultaneously. Therefore, there is actually no guarantee that a function invocation could ever return the value CKR\_SESSION\_CLOSED. An example of multiple threads accessing a common session simultaneously is where one thread is using a session when another thread closes that same session.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR\_SESSION\_HANDLE\_INVALID or CKR\_DEVICE\_REMOVED would be an appropriate error return, then CKR\_SESSION\_HANDLE\_INVALID should be returned.

In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction between a token being removed *before* a function invocation and a token being removed *during* a function execution.

### Cryptoki function return values for functions that use a token

Any Cryptoki function that uses a particular token (*i.e.*, any Cryptoki function except for **C\_Initialize**, **C\_Finalize**, **C\_GetInfo**, **C\_GetFunctionList**, **C\_GetSlotList**, **C\_GetSlotInfo**, or **C\_WaitForSlotEvent**) can return any of the following values:

* CKR\_DEVICE\_MEMORY: The token does not have sufficient memory to perform the requested function.
* CKR\_DEVICE\_ERROR: Some problem has occurred with the token and/or slot. This error code can be returned by more than just the functions mentioned above; in particular, it is possible for **C\_GetSlotInfo** to return CKR\_DEVICE\_ERROR.
* CKR\_TOKEN\_NOT\_PRESENT: The token was not present in its slot *at the time that the function was invoked*.
* CKR\_DEVICE\_REMOVED: The token was removed from its slot *during the execution of the function*.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of CKR\_DEVICE\_MEMORY or CKR\_DEVICE\_ERROR would be an appropriate error return, then CKR\_DEVICE\_MEMORY should be returned.

In practice, it is often not critical (or possible) for a Cryptoki library to be able to make a distinction between a token being removed *before* a function invocation and a token being removed *during* a function execution.

### Special return value for application-supplied callbacks

There is a special-purpose return value which is not returned by any function in the actual Cryptoki API, but which may be returned by an application-supplied callback function. It is:

* CKR\_CANCEL: When a function executing in serial with an application decides to give the application a chance to do some work, it calls an application-supplied function with a CKN\_SURRENDER callback (see Section 5.21). If the callback returns the value CKR\_CANCEL, then the function aborts and returns CKR\_FUNCTION\_CANCELED.

### Special return values for mutex-handling functions

There are two other special-purpose return values which are not returned by any actual Cryptoki functions. These values may be returned by application-supplied mutex-handling functions, and they may safely be ignored by application developers who are not using their own threading model. They are:

* CKR\_MUTEX\_BAD: This error code can be returned by mutex-handling functions that are passed a bad mutex object as an argument. Unfortunately, it is possible for such a function not to recognize a bad mutex object. There is therefore no guarantee that such a function will successfully detect bad mutex objects and return this value.
* CKR\_MUTEX\_NOT\_LOCKED: This error code can be returned by mutex-unlocking functions. It indicates that the mutex supplied to the mutex-unlocking function was not locked.

### All other Cryptoki function return values

Descriptions of the other Cryptoki function return values follow. Except as mentioned in the descriptions of particular error codes, there are in general no particular priorities among the errors listed below, *i.e.*, if more than one error code might apply to an execution of a function, then the function may return any applicable error code.

* CKR\_ACTION\_PROHIBITED: This value can only be returned by C\_CopyObject, C\_SetAttributeValue and C\_DestroyObject. It denotes that the action may not be taken, either because of underlying policy restrictions on the token, or because the object has the relevant CKA\_COPYABLE, CKA\_MODIFIABLE or CKA\_DESTROYABLE policy attribute set to CK\_FALSE.
* CKR\_ARGUMENTS\_BAD: This is a rather generic error code which indicates that the arguments supplied to the Cryptoki function were in some way not appropriate.
* CKR\_ATTRIBUTE\_READ\_ONLY: An attempt was made to set a value for an attribute which may not be set by the application, or which may not be modified by the application. See Section 4.1 for more information.
* CKR\_ATTRIBUTE\_SENSITIVE: An attempt was made to obtain the value of an attribute of an object which cannot be satisfied because the object is either sensitive or un-extractable.
* CKR\_ATTRIBUTE\_TYPE\_INVALID: An invalid attribute type was specified in a template. See Section 4.1 for more information.
* CKR\_ATTRIBUTE\_VALUE\_INVALID: An invalid value was specified for a particular attribute in a template. See Section 4.1 for more information.
* CKR\_BUFFER\_TOO\_SMALL: The output of the function is too large to fit in the supplied buffer.
* CKR\_CANT\_LOCK: This value can only be returned by **C\_Initialize**. It means that the type of locking requested by the application for thread-safety is not available in this library, and so the application cannot make use of this library in the specified fashion.
* CKR\_CRYPTOKI\_ALREADY\_INITIALIZED: This value can only be returned by **C\_Initialize**. It means that the Cryptoki library has already been initialized (by a previous call to **C\_Initialize** which did not have a matching **C\_Finalize** call).
* CKR\_CRYPTOKI\_NOT\_INITIALIZED: This value can be returned by any function other than **C\_Initialize,**  **C\_GetFunctionList, C\_GetInterfaceList** and **C\_GetInterface**. It indicates that the function cannot be executed because the Cryptoki library has not yet been initialized by a call to **C\_Initialize**.
* CKR\_CURVE\_NOT\_SUPPORTED: This curve is not supported by this token. Used with Elliptic Curve mechanisms.
* CKR\_DATA\_INVALID: The plaintext input data to a cryptographic operation is invalid. This return value has lower priority than CKR\_DATA\_LEN\_RANGE.
* CKR\_DATA\_LEN\_RANGE: The plaintext input data to a cryptographic operation has a bad length. Depending on the operation’s mechanism, this could mean that the plaintext data is too short, too long, or is not a multiple of some particular block size. This return value has higher priority than CKR\_DATA\_INVALID.
* CKR\_DOMAIN\_PARAMS\_INVALID: Invalid or unsupported domain parameters were supplied to the function. Which representation methods of domain parameters are supported by a given mechanism can vary from token to token.
* CKR\_ENCRYPTED\_DATA\_INVALID: The encrypted input to a decryption operation has been determined to be invalid ciphertext. This return value has lower priority than CKR\_ENCRYPTED\_DATA\_LEN\_RANGE.
* CKR\_ENCRYPTED\_DATA\_LEN\_RANGE: The ciphertext input to a decryption operation has been determined to be invalid ciphertext solely on the basis of its length. Depending on the operation’s mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some particular block size. This return value has higher priority than CKR\_ENCRYPTED\_DATA\_INVALID.
* CKR\_EXCEEDED\_MAX\_ITERATIONS: An iterative algorithm (for key pair generation, domain parameter generation etc.) failed because we have exceeded the maximum number of iterations. This error code has precedence over CKR\_FUNCTION\_FAILED. Examples of iterative algorithms include DSA signature generation (retry if either r = 0 or s = 0) and generation of DSA primes p and q specified in FIPS 186-4.
* CKR\_FIPS\_SELF\_TEST\_FAILED: A FIPS 140-2 power-up self-test or conditional self-test failed. The token entered an error state. Future calls to cryptographic functions on the token will return CKR\_GENERAL\_ERROR. CKR\_FIPS\_SELF\_TEST\_FAILED has a higher precedence over CKR\_GENERAL\_ERROR. This error may be returned by C\_Initialize, if a power-up self-test failed, by C\_GenerateRandom or C\_SeedRandom, if the continuous random number generator test failed, or by C\_GenerateKeyPair, if the pair-wise consistency test failed.
* CKR\_FUNCTION\_CANCELED: The function was canceled in mid-execution. This happens to a cryptographic function if the function makes a **CKN\_SURRENDER** application callback which returns CKR\_CANCEL (see CKR\_CANCEL). It also happens to a function that performs PIN entry through a protected path. The method used to cancel a protected path PIN entry operation is device dependent.
* CKR\_FUNCTION\_NOT\_PARALLEL: There is currently no function executing in parallel in the specified session. This is a legacy error code which is only returned by the legacy functions **C\_GetFunctionStatus** and **C\_CancelFunction**.
* CKR\_FUNCTION\_NOT\_SUPPORTED: The requested function is not supported by this Cryptoki library. Even unsupported functions in the Cryptoki API should have a “stub” in the library; this stub should simply return the value CKR\_FUNCTION\_NOT\_SUPPORTED.
* CKR\_FUNCTION\_REJECTED: The signature request is rejected by the user.
* CKR\_INFORMATION\_SENSITIVE: The information requested could not be obtained because the token considers it sensitive, and is not able or willing to reveal it.
* CKR\_KEY\_CHANGED: This value is only returned by **C\_SetOperationState**. It indicates that one of the keys specified is not the same key that was being used in the original saved session.
* CKR\_KEY\_FUNCTION\_NOT\_PERMITTED: An attempt has been made to use a key for a cryptographic purpose that the key’s attributes are not set to allow it to do. For example, to use a key for performing encryption, that key MUST have its **CKA\_ENCRYPT** attribute set to CK\_TRUE (the fact that the key MUST have a **CKA\_ENCRYPT** attribute implies that the key cannot be a private key). This return value has lower priority than CKR\_KEY\_TYPE\_INCONSISTENT.
* CKR\_KEY\_HANDLE\_INVALID: The specified key handle is not valid. It may be the case that the specified handle is a valid handle for an object which is not a key. We reiterate here that 0 is never a valid key handle.
* CKR\_KEY\_INDIGESTIBLE: This error code can only be returned by **C\_DigestKey**. It indicates that the value of the specified key cannot be digested for some reason (perhaps the key isn’t a secret key, or perhaps the token simply can’t digest this kind of key).
* CKR\_KEY\_NEEDED: This value is only returned by **C\_SetOperationState**. It indicates that the session state cannot be restored because **C\_SetOperationState** needs to be supplied with one or more keys that were being used in the original saved session.
* CKR\_KEY\_NOT\_NEEDED: An extraneous key was supplied to **C\_SetOperationState**. For example, an attempt was made to restore a session that had been performing a message digesting operation, and an encryption key was supplied.
* CKR\_KEY\_NOT\_WRAPPABLE: Although the specified private or secret key does not have its CKA\_EXTRACTABLE attribute set to CK\_FALSE, Cryptoki (or the token) is unable to wrap the key as requested (possibly the token can only wrap a given key with certain types of keys, and the wrapping key specified is not one of these types). Compare with CKR\_KEY\_UNEXTRACTABLE.
* CKR\_KEY\_SIZE\_RANGE: Although the requested keyed cryptographic operation could in principle be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key‘s size is outside the range of key sizes that it can handle.
* CKR\_KEY\_TYPE\_INCONSISTENT: The specified key is not the correct type of key to use with the specified mechanism. This return value has a higher priority than CKR\_KEY\_FUNCTION\_NOT\_PERMITTED.
* CKR\_KEY\_UNEXTRACTABLE: The specified private or secret key can’t be wrapped because its CKA\_EXTRACTABLE attribute is set to CK\_FALSE. Compare with CKR\_KEY\_NOT\_WRAPPABLE.
* CKR\_LIBRARY\_LOAD\_FAILED: The Cryptoki library could not load a dependent shared library.
* CKR\_MECHANISM\_INVALID: An invalid mechanism was specified to the cryptographic operation. This error code is an appropriate return value if an unknown mechanism was specified or if the mechanism specified cannot be used in the selected token with the selected function.
* CKR\_MECHANISM\_PARAM\_INVALID: Invalid parameters were supplied to the mechanism specified to the cryptographic operation. Which parameter values are supported by a given mechanism can vary from token to token.
* CKR\_NEED\_TO\_CREATE\_THREADS: This value can only be returned by **C\_Initialize**. It is returned when two conditions hold:

1. The application called **C\_Initialize** in a way which tells the Cryptoki library that application threads executing calls to the library cannot use native operating system methods to spawn new threads.
2. The library cannot function properly without being able to spawn new threads in the above fashion.

* CKR\_NO\_EVENT: This value can only be returned by **C\_WaitForSlotEvent**. It is returned when **C\_WaitForSlotEvent** is called in non-blocking mode and there are no new slot events to return.
* CKR\_OBJECT\_HANDLE\_INVALID: The specified object handle is not valid. We reiterate here that 0 is never a valid object handle.
* CKR\_OPERATION\_ACTIVE: There is already an active operation (or combination of active operations) which prevents Cryptoki from activating the specified operation. For example, an active object-searching operation would prevent Cryptoki from activating an encryption operation with **C\_EncryptInit**. Or, an active digesting operation and an active encryption operation would prevent Cryptoki from activating a signature operation. Or, on a token which doesn’t support simultaneous dual cryptographic operations in a session (see the description of the **CKF\_DUAL\_CRYPTO\_OPERATIONS** flag in the **CK\_TOKEN\_INFO** structure), an active signature operation would prevent Cryptoki from activating an encryption operation.
* CKR\_OPERATION\_NOT\_INITIALIZED: There is no active operation of an appropriate type in the specified session. For example, an application cannot call **C\_Encrypt** in a session without having called **C\_EncryptInit** first to activate an encryption operation.
* CKR\_PIN\_EXPIRED: The specified PIN has expired, and the requested operation cannot be carried out unless C\_SetPIN is called to change the PIN value. Whether or not the normal user’s PIN on a token ever expires varies from token to token.
* CKR\_PIN\_INCORRECT: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the token. More generally-- when authentication to the token involves something other than a PIN-- the attempt to authenticate the user has failed.
* CKR\_PIN\_INVALID: The specified PIN has invalid characters in it. This return code only applies to functions which attempt to set a PIN.
* CKR\_PIN\_LEN\_RANGE: The specified PIN is too long or too short. This return code only applies to functions which attempt to set a PIN.
* CKR\_PIN\_LOCKED: The specified PIN is “locked”, and cannot be used. That is, because some particular number of failed authentication attempts has been reached, the token is unwilling to permit further attempts at authentication. Depending on the token, the specified PIN may or may not remain locked indefinitely.
* CKR\_PIN\_TOO\_WEAK: The specified PIN is too weak so that it could be easy to guess. If the PIN is too short, CKR\_PIN\_LEN\_RANGE should be returned instead. This return code only applies to functions which attempt to set a PIN.
* CKR\_PUBLIC\_KEY\_INVALID: The public key fails a public key validation. For example, an EC public key fails the public key validation specified in Section 5.2.2 of ANSI X9.62. This error code may be returned by C\_CreateObject, when the public key is created, or by C\_VerifyInit or C\_VerifyRecoverInit, when the public key is used. It may also be returned by C\_DeriveKey, in preference to CKR\_MECHANISM\_PARAM\_INVALID, if the other party's public key specified in the mechanism's parameters is invalid.
* CKR\_RANDOM\_NO\_RNG: This value can be returned by **C\_SeedRandom** and **C\_GenerateRandom**. It indicates that the specified token doesn’t have a random number generator. This return value has higher priority than CKR\_RANDOM\_SEED\_NOT\_SUPPORTED.
* CKR\_RANDOM\_SEED\_NOT\_SUPPORTED: This value can only be returned by **C\_SeedRandom**. It indicates that the token’s random number generator does not accept seeding from an application. This return value has lower priority than CKR\_RANDOM\_NO\_RNG.
* CKR\_SAVED\_STATE\_INVALID: This value can only be returned by **C\_SetOperationState**. It indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be restored to the specified session.
* CKR\_SESSION\_COUNT: This value can only be returned by **C\_OpenSession**. It indicates that the attempt to open a session failed, either because the token has too many sessions already open, or because the token has too many read/write sessions already open.
* CKR\_SESSION\_EXISTS: This value can only be returned by **C\_InitToken**. It indicates that a session with the token is already open, and so the token cannot be initialized.
* CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED: The specified token does not support parallel sessions. This is a legacy error code—in Cryptoki Version 2.01 and up, *no* token supports parallel sessions. CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED can only be returned by **C\_OpenSession**, and it is only returned when **C\_OpenSession** is called in a particular [deprecated] way.
* CKR\_SESSION\_READ\_ONLY: The specified session was unable to accomplish the desired action because it is a read-only session. This return value has lower priority than CKR\_TOKEN\_WRITE\_PROTECTED.
* CKR\_SESSION\_READ\_ONLY\_EXISTS: A read-only session already exists, and so the SO cannot be logged in.
* CKR\_SESSION\_READ\_WRITE\_SO\_EXISTS: A read/write SO session already exists, and so a read-only session cannot be opened.
* CKR\_SIGNATURE\_LEN\_RANGE: The provided signature/MAC can be seen to be invalid solely on the basis of its length. This return value has higher priority than CKR\_SIGNATURE\_INVALID.
* CKR\_SIGNATURE\_INVALID: The provided signature/MAC is invalid. This return value has lower priority than CKR\_SIGNATURE\_LEN\_RANGE.
* CKR\_SLOT\_ID\_INVALID: The specified slot ID is not valid.
* CKR\_STATE\_UNSAVEABLE: The cryptographic operations state of the specified session cannot be saved for some reason (possibly the token is simply unable to save the current state). This return value has lower priority than CKR\_OPERATION\_NOT\_INITIALIZED.
* CKR\_TEMPLATE\_INCOMPLETE: The template specified for creating an object is incomplete, and lacks some necessary attributes. See Section 4.1 for more information.
* CKR\_TEMPLATE\_INCONSISTENT: The template specified for creating an object has conflicting attributes. See Section 4.1 for more information.
* CKR\_TOKEN\_NOT\_RECOGNIZED: The Cryptoki library and/or slot does not recognize the token in the slot.
* CKR\_TOKEN\_WRITE\_PROTECTED: The requested action could not be performed because the token is write-protected. This return value has higher priority than CKR\_SESSION\_READ\_ONLY.
* CKR\_UNWRAPPING\_KEY\_HANDLE\_INVALID: This value can only be returned by **C\_UnwrapKey**. It indicates that the key handle specified to be used to unwrap another key is not valid.
* CKR\_UNWRAPPING\_KEY\_SIZE\_RANGE: This value can only be returned by **C\_UnwrapKey**. It indicates that although the requested unwrapping operation could in principle be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key’s size is outside the range of key sizes that it can handle.
* CKR\_UNWRAPPING\_KEY\_TYPE\_INCONSISTENT: This value can only be returned by **C\_UnwrapKey**. It indicates that the type of the key specified to unwrap another key is not consistent with the mechanism specified for unwrapping.
* CKR\_USER\_ALREADY\_LOGGED\_IN: This value can only be returned by **C\_Login**. It indicates that the specified user cannot be logged into the session, because it is already logged into the session. For example, if an application has an open SO session, and it attempts to log the SO into it, it will receive this error code.
* CKR\_USER\_ANOTHER\_ALREADY\_LOGGED\_IN: This value can only be returned by **C\_Login**. It indicates that the specified user cannot be logged into the session, because another user is already logged into the session. For example, if an application has an open SO session, and it attempts to log the normal user into it, it will receive this error code.
* CKR\_USER\_NOT\_LOGGED\_IN: The desired action cannot be performed because the appropriate user (or *an* appropriate user) is not logged in. One example is that a session cannot be logged out unless it is logged in. Another example is that a private object cannot be created on a token unless the session attempting to create it is logged in as the normal user. A final example is that cryptographic operations on certain tokens cannot be performed unless the normal user is logged in.
* CKR\_USER\_PIN\_NOT\_INITIALIZED: This value can only be returned by **C\_Login**. It indicates that the normal user’s PIN has not yet been initialized with **C\_InitPIN**.
* CKR\_USER\_TOO\_MANY\_TYPES: An attempt was made to have more distinct users simultaneously logged into the token than the token and/or library permits. For example, if some application has an open SO session, and another application attempts to log the normal user into a session, the attempt may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be supported does **C\_Login** have to return this value. Note that this error code generalizes to true multi-user tokens.
* CKR\_USER\_TYPE\_INVALID: An invalid value was specified as a **CK\_USER\_TYPE**. Valid types are **CKU\_SO**, **CKU\_USER**, and **CKU\_CONTEXT\_SPECIFIC**.
* CKR\_WRAPPED\_KEY\_INVALID: This value can only be returned by **C\_UnwrapKey**. It indicates that the provided wrapped key is not valid. If a call is made to **C\_UnwrapKey** to unwrap a particular type of key (*i.e.*, some particular key type is specified in the template provided to **C\_UnwrapKey**), and the wrapped key provided to **C\_UnwrapKey** is recognizably not a wrapped key of the proper type, then **C\_UnwrapKey** should return CKR\_WRAPPED\_KEY\_INVALID. This return value has lower priority than CKR\_WRAPPED\_KEY\_LEN\_RANGE.
* CKR\_WRAPPED\_KEY\_LEN\_RANGE: This value can only be returned by **C\_UnwrapKey**. It indicates that the provided wrapped key can be seen to be invalid solely on the basis of its length. This return value has higher priority than CKR\_WRAPPED\_KEY\_INVALID.
* CKR\_WRAPPING\_KEY\_HANDLE\_INVALID: This value can only be returned by **C\_WrapKey**. It indicates that the key handle specified to be used to wrap another key is not valid.
* CKR\_WRAPPING\_KEY\_SIZE\_RANGE: This value can only be returned by **C\_WrapKey**. It indicates that although the requested wrapping operation could in principle be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied wrapping key’s size is outside the range of key sizes that it can handle.
* CKR\_WRAPPING\_KEY\_TYPE\_INCONSISTENT: This value can only be returned by **C\_WrapKey**. It indicates that the type of the key specified to wrap another key is not consistent with the mechanism specified for wrapping.
* CKR\_OPERATION\_CANCEL\_FAILED: This value can only be returned by **C\_SessionCancel**. It means that one or more of the requested operations could not be cancelled for implementation or vendor-specific reasons.

### More on relative priorities of Cryptoki errors

In general, when a Cryptoki call is made, error codes from Section 5.1.1 (other than CKR\_OK) take precedence over error codes from Section 5.1.2, which take precedence over error codes from Section 5.1.3, which take precedence over error codes from Section 5.1.6. One minor implication of this is that functions that use a session handle (*i.e.*, *most* functions!) never return the error code CKR\_TOKEN\_NOT\_PRESENT (they return CKR\_SESSION\_HANDLE\_INVALID instead). Other than these precedences, if more than one error code applies to the result of a Cryptoki call, any of the applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the descriptions of functions.

### Error code “gotchas”

Here is a short list of a few particular things about return values that Cryptoki developers might want to be aware of:

1. As mentioned in Sections 5.1.2 and 5.1.3, a Cryptoki library may not be able to make a distinction between a token being removed *before* a function invocation and a token being removed *during* a function invocation.
2. As mentioned in Section 5.1.2, an application should never count on getting a CKR\_SESSION\_CLOSED error.
3. The difference between CKR\_DATA\_INVALID and CKR\_DATA\_LEN\_RANGE can be somewhat subtle. Unless an application *needs* to be able to distinguish between these return values, it is best to always treat them equivalently.
4. Similarly, the difference between CKR\_ENCRYPTED\_DATA\_INVALID and CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, and between CKR\_WRAPPED\_KEY\_INVALID and CKR\_WRAPPED\_KEY\_LEN\_RANGE, can be subtle, and it may be best to treat these return values equivalently.
5. Even with the guidance of Section 4.1, it can be difficult for a Cryptoki library developer to know which of CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_TEMPLATE\_INCOMPLETE, or CKR\_TEMPLATE\_INCONSISTENT to return. When possible, it is recommended that application developers be generous in their interpretations of these error codes.

## Conventions for functions returning output in a variable-length buffer

A number of the functions defined in Cryptoki return output produced by some cryptographic mechanism. The amount of output returned by these functions is returned in a variable-length application-supplied buffer. An example of a function of this sort is **C\_Encrypt**, which takes some plaintext as an argument, and outputs a buffer full of ciphertext.

These functions have some common calling conventions, which we describe here. Two of the arguments to the function are a pointer to the output buffer (say *pBuf*) and a pointer to a location which will hold the length of the output produced (say *pulBufLen*). There are two ways for an application to call such a function:

1. If *pBuf* is NULL\_PTR, then all that the function does is return (in \**pulBufLen*) a number of bytes which would suffice to hold the cryptographic output produced from the input to the function. This number may somewhat exceed the precise number of bytes needed, but should not exceed it by a large amount. CKR\_OK is returned by the function.
2. If *pBuf* is not NULL\_PTR, then \**pulBufLen* MUST contain the size in bytes of the buffer pointed to by *pBuf*. If that buffer is large enough to hold the cryptographic output produced from the input to the function, then that cryptographic output is placed there, and CKR\_OK is returned by the function and \*pulBufLen is set to the exact number of bytes returned. If the buffer is not large enough, then CKR\_BUFFER\_TOO\_SMALL is returned and \**pulBufLen* is set to at least the number of bytes needed to hold the cryptographic output produced from the input to the function.

NOTE: This is a change from previous specs. The problem is that in some decrypt cases, the token doesn’t know how big a buffer is needed until the decrypt completes. The act of doing decrypt can mess up the internal encryption state. Many tokens already implement this relaxed behavior, tokens which implement the more precise behavior are still compliant. The one corner case is applications using a token that knows exactly how big the decryption is (through some out of band means), could get CKR\_BUFFER\_TOO\_SMALL returned when it supplied a buffer exactly big enough to hold the decrypted value when it may previously have succeeded.

All functions which use the above convention will explicitly say so.

Cryptographic functions which return output in a variable-length buffer should always return as much output as can be computed from what has been passed in to them thus far. As an example, consider a session which is performing a multiple-part decryption operation with DES in cipher-block chaining mode with PKCS padding. Suppose that, initially, 8 bytes of ciphertext are passed to the **C\_DecryptUpdate** function. The block size of DES is 8 bytes, but the PKCS padding makes it unclear at this stage whether the ciphertext was produced from encrypting a 0-byte string, or from encrypting some string of length at least 8 bytes. Hence the call to **C\_DecryptUpdate** should return 0 bytes of plaintext. If a single additional byte of ciphertext is supplied by a subsequent call to **C\_DecryptUpdate**, then that call should return 8 bytes of plaintext (one full DES block).

## Disclaimer concerning sample code

For the remainder of this section, we enumerate the various functions defined in Cryptoki. Most functions will be shown in use in at least one sample code snippet. For the sake of brevity, sample code will frequently be somewhat incomplete. In particular, sample code will generally ignore possible error returns from C library functions, and also will not deal with Cryptoki error returns in a realistic fashion.

## General-purpose functions

Cryptoki provides the following general-purpose functions:

### **C\_Initialize**

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Initialize) {

CK\_VOID\_PTR pInitArgs

);

**C\_Initialize** initializes the Cryptoki library. *pInitArgs* either has the value NULL\_PTR or points to a **CK\_C\_INITIALIZE\_ARGS** structure containing information on how the library should deal with multi-threaded access. If an application will not be accessing Cryptoki through multiple threads simultaneously, it can generally supply the value NULL\_PTR to **C\_Initialize** (the consequences of supplying this value will be explained below).

If *pInitArgs* is non-NULL\_PTR, **C\_Initialize** should cast it to a **CK\_C\_INITIALIZE\_ARGS\_PTR** and then dereference the resulting pointer to obtain the **CK\_C\_INITIALIZE\_ARGS** fields *CreateMutex*, *DestroyMutex*, *LockMutex*, *UnlockMutex*, *flags*, and *pReserved*. For this version of Cryptoki, the value of *pReserved* thereby obtained MUST be NULL\_PTR; if it’s not, then **C\_Initialize** should return with the value CKR\_ARGUMENTS\_BAD.

If the **CKF\_LIBRARY\_CANT\_CREATE\_OS\_THREADS** flag in the *flags* field is set, that indicates that application threads which are executing calls to the Cryptoki library are not permitted to use the native operation system calls to spawn off new threads. In other words, the library’s code may not create its own threads. If the library is unable to function properly under this restriction, **C\_Initialize** should return with the value CKR\_NEED\_TO\_CREATE\_THREADS.

A call to **C\_Initialize** specifies one of four different ways to support multi-threaded access via the value of the **CKF\_OS\_LOCKING\_OK** flag in the *flags* field and the values of the *CreateMutex*, *DestroyMutex*, *LockMutex*, and *UnlockMutex* function pointer fields:

1. If the flag *isn’t* set, and the function pointer fields *aren’t* supplied (*i.e.*, they all have the value NULL\_PTR), that means that the application *won’t* be accessing the Cryptoki library from multiple threads simultaneously.
2. If the flag *is* set, and the function pointer fields *aren’t* supplied (*i.e.*, they all have the value NULL\_PTR), that means that the application *will* be performing multi-threaded Cryptoki access, and the library needs to use the native operating system primitives to ensure safe multi-threaded access. If the library is unable to do this, **C\_Initialize** should return with the value CKR\_CANT\_LOCK.
3. If the flag *isn’t* set, and the function pointer fields *are* supplied (*i.e.*, they all have non-NULL\_PTR values), that means that the application *will* be performing multi-threaded Cryptoki access, and the library needs to use the supplied function pointers for mutex-handling to ensure safe multi-threaded access. If the library is unable to do this, **C\_Initialize** should return with the value CKR\_CANT\_LOCK.
4. If the flag *is* set, and the function pointer fields *are* supplied (*i.e.*, they all have non-NULL\_PTR values), that means that the application *will* be performing multi-threaded Cryptoki access, and the library needs to use either the native operating system primitives or the supplied function pointers for mutex-handling to ensure safe multi-threaded access. If the library is unable to do this, **C\_Initialize** should return with the value CKR\_CANT\_LOCK.

If some, but not all, of the supplied function pointers to **C\_Initialize** are non-NULL\_PTR, then **C\_Initialize** should return with the value CKR\_ARGUMENTS\_BAD.

A call to **C\_Initialize** with *pInitArgs* set to NULL\_PTR is treated like a call to **C\_Initialize** with *pInitArgs* pointing to a **CK\_C\_INITIALIZE\_ARGS** which has the *CreateMutex*, *DestroyMutex*, *LockMutex*, *UnlockMutex*, and *pReserved* fields set to NULL\_PTR, and has the *flags* field set to 0.

**C\_Initialize** should be the first Cryptoki call made by an application, except for calls to **C\_GetFunctionList**, **C\_GetInterfaceList**, or **C\_GetInterface**. What this function actually does is implementation-dependent; typically, it might cause Cryptoki to initialize its internal memory buffers, or any other resources it requires.

If several applications are using Cryptoki, each one should call **C\_Initialize**. Every call to **C\_Initialize** should (eventually) be succeeded by a single call to **C\_Finalize**. See **[PKCS11-UG]** for further details.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CANT\_LOCK, CKR\_CRYPTOKI\_ALREADY\_INITIALIZED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_NEED\_TO\_CREATE\_THREADS, CKR\_OK.

Example: see **C\_GetInfo**.

### C\_Finalize

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Finalize)(

CK\_VOID\_PTR pReserved

);

**C\_Finalize** is called to indicate that an application is finished with the Cryptoki library. It should be the last Cryptoki call made by an application. The *pReserved* parameter is reserved for future versions; for this version, it should be set to NULL\_PTR (if **C\_Finalize** is called with a non-NULL\_PTR value for *pReserved*, it should return the value CKR\_ARGUMENTS\_BAD.

If several applications are using Cryptoki, each one should call **C\_Finalize**. Each application’s call to **C\_Finalize** should be preceded by a single call to **C\_Initialize**; in between the two calls, an application can make calls to other Cryptoki functions. See **[PKCS11-UG]** for further details.

*Despite the fact that the parameters supplied to* ***C\_Initialize*** *can in general allow for safe multi-threaded access to a Cryptoki library, the behavior of* ***C\_Finalize*** *is nevertheless undefined if it is called by an application while other threads of the application are making Cryptoki calls. The exception to this exceptional behavior of* ***C\_Finalize*** *occurs when a thread calls* ***C\_Finalize*** *while another of the application’s threads is blocking on Cryptoki’s* ***C\_WaitForSlotEvent*** *function. When this happens, the blocked thread becomes unblocked and returns the value CKR\_CRYPTOKI\_NOT\_INITIALIZED. See* ***C\_WaitForSlotEvent*** *for more information.*

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

Example: see **C\_GetInfo**.

### C\_GetInfo

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetInfo)(

CK\_INFO\_PTR pInfo

);

**C\_GetInfo** returns general information about Cryptoki. *pInfo* points to the location that receives the information.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

Example:

CK\_INFO info;

CK\_RV rv;

CK\_C\_INITIALIZE\_ARGS InitArgs;

InitArgs.CreateMutex = &MyCreateMutex;

InitArgs.DestroyMutex = &MyDestroyMutex;

InitArgs.LockMutex = &MyLockMutex;

InitArgs.UnlockMutex = &MyUnlockMutex;

InitArgs.flags = CKF\_OS\_LOCKING\_OK;

InitArgs.pReserved = NULL\_PTR;

rv = C\_Initialize((CK\_VOID\_PTR)&InitArgs);

assert(rv == CKR\_OK);

rv = C\_GetInfo(&info);

assert(rv == CKR\_OK);

if(info.cryptokiVersion.major == 2) {

/\* Do lots of interesting cryptographic things with the token \*/

.

.

}

rv = C\_Finalize(NULL\_PTR);

assert(rv == CKR\_OK);

### C\_GetFunctionList

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetFunctionList)(

CK\_FUNCTION\_LIST\_PTR\_PTR ppFunctionList

);

**C\_GetFunctionList** obtains a pointer to the Cryptoki library’s list of function pointers. *ppFunctionList* points to a value which will receive a pointer to the library’s **CK\_FUNCTION\_LIST** structure, which in turn contains function pointers for all the Cryptoki API routines in the library. *The pointer thus obtained may point into memory which is owned by the Cryptoki library, and which may or may not be writable*. Whether or not this is the case, no attempt should be made to write to this memory.

**C\_GetFunctionList**, **C\_GetInterfaceList**, and **C\_GetInterface** are the only Cryptoki functions which an application may call before calling **C\_Initialize**. It is provided to make it easier and faster for applications to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

Example:

CK\_FUNCTION\_LIST\_PTR pFunctionList;

CK\_C\_Initialize pC\_Initialize;

CK\_RV rv;

/\* It’s OK to call C\_GetFunctionList before calling C\_Initialize \*/

rv = C\_GetFunctionList(&pFunctionList);

assert(rv == CKR\_OK);

pC\_Initialize = pFunctionList -> C\_Initialize;

/\* Call the C\_Initialize function in the library \*/

rv = (\*pC\_Initialize)(NULL\_PTR);

### C\_GetInterfaceList

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetInterfaceList)(

CK\_INTERFACE\_PTR pInterfaceList,

CK\_ULONG\_PTR pulCount

);

**C\_GetInterfaceList** is used to obtain a list of interfaces supported by a Cryptoki library. *pulCount* points to the location that receives the number of interfaces.

There are two ways for an application to call **C\_GetInterfaceList**:

1. If *pInterfaceList* is NULL\_PTR, then all that **C\_GetInterfaceList** does is return (in \**pulCount*) the number of interfaces, without actually returning a list of interfaces. The contents of \**pulCount* on entry to **C\_GetInterfaceList** has no meaning in this case, and the call returns the value CKR\_OK.
2. If *pIntrerfaceList* is not NULL\_PTR, then \**pulCount* MUST contain the size (in terms of **CK\_INTERFACE** elements) of the buffer pointed to by *pInterfaceList*. If that buffer is large enough to hold the list of interfaces, then the list is returned in it, and CKR\_OK is returned. If not, then the call to **C\_GetInterfaceList** returns the value CKR\_BUFFER\_TOO\_SMALL. In either case, the value \**pulCount* is set to hold the number of interfaces.

Because **C\_GetInterfaceList** does not allocate any space of its own, an application will often call **C\_GetInterfaceList** twice. However, this behavior is by no means required.

**C\_GetInterfaceList** obtains (in \**pFunctionList* of each interface) a pointer to the Cryptoki library’s list of function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki library, and which may or may not be writable*. Whether or not this is the case, no attempt should be made to write to this memory. The same caveat applies to the interface names returned.

**C\_GetFunctionList**, **C\_GetInterfaceList**, and **C\_GetInterface** are the only Cryptoki functions which an application may call before calling **C\_Initialize**. It is provided to make it easier and faster for applications to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

Return values: CKR\_BUFFER\_TOO\_SMALL, CKR\_ARGUMENTS\_BAD, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

Example:

CK\_ULONG ulCount=0;

CK\_INTERFACE\_PTR interfaceList=NULL;

CK\_RV rv;

int I;

/\* get number of interfaces \*/

rv = C\_GetInterfaceList(NULL,&ulCount);

if (rv == CKR\_OK) {

/\* get copy of interfaces \*/

interfaceList = (CK\_INTERFACE\_PTR)malloc(ulCount\*sizeof(CK\_INTERFACE));

rv = C\_GetInterfaceList(interfaceList,&ulCount);

for(i=0;i<ulCount;i++) {

printf("interface %s version %d.%d funcs %p flags 0x%lu\n",

interfaceList[i].pInterfaceName,

((CK\_VERSION \*)interfaceList[i].pFunctionList)->major,

((CK\_VERSION \*)interfaceList[i].pFunctionList)->minor,

interfaceList[i].pFunctionList,

interfaceList[i].flags);

}

}

### C\_GetInterface

CK\_DECLARE\_FUNCTION(CK\_RV,C\_GetInterface)(

CK\_UTF8CHAR\_PTR pInterfaceName,

CK\_VERSION\_PTR pVersion,

CK\_INTERFACE\_PTR\_PTR ppInterface,

CK\_FLAGS flags

);

**C\_GetInterface** is used to obtain an interface supported by a Cryptoki library. *pInterfaceName* specifies the name of the interface, *pVersion* specifies the interface version, *ppInterface* points to the location that receives the interface, *flags* specifies the required interface flags.

There are multiple ways for an application to specify a particular interface when calling **C\_GetInterface**:

1. If *pInterfaceName* is not NULL\_PTR, the name of the interface returned must match. If *pInterfaceName* is NULL\_PTR, the cryptoki library can return a default interface of its choice
2. If *pVersion* is not NULL\_PTR, the version of the interface returned must match. If *pVersion* is NULL\_PTR, the cryptoki library can return an interface of any version
3. If *flags* is non-zero, the interface returned must match all of the supplied flag values (but may include additional flags not specified). If *flags* is 0, the cryptoki library can return an interface with any flags

**C\_GetInterface** obtains (in \**pFunctionList* of each interface) a pointer to the Cryptoki library’s list of function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki library, and which may or may not be writable*. Whether or not this is the case, no attempt should be made to write to this memory. The same caveat applies to the interface names returned.

**C\_GetFunctionList**, **C\_GetInterfaceList**, and **C\_GetInterface** are the only Cryptoki functions which an application may call before calling **C\_Initialize**. It is provided to make it easier and faster for applications to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

Return values: CKR\_BUFFER\_TOO\_SMALL, CKR\_ARGUMENTS\_BAD, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

Example:

CK\_INTERFACE\_PTR interface;

CK\_RV rv;

CK\_VERSION version;

CK\_FLAGS flags=CKF\_ INTERFACE\_FORK\_SAFE;

/\* get default interface \*/

rv = C\_GetInterface(NULL,NULL,&interface,flags);

if (rv == CKR\_OK) {

printf("interface %s version %d.%d funcs %p flags 0x%lu\n",

interface->pInterfaceName,

((CK\_VERSION \*)interface->pFunctionList)->major,

((CK\_VERSION \*)interface->pFunctionList)->minor,

interface->pFunctionList,

interface->flags);

}

/\* get default standard interface \*/

rv = C\_GetInterface((CK\_UTF8CHAR\_PTR)"PKCS 11",NULL,&interface,flags);

if (rv == CKR\_OK) {

printf("interface %s version %d.%d funcs %p flags 0x%lu\n",

interface->pInterfaceName,

((CK\_VERSION \*)interface->pFunctionList)->major,

((CK\_VERSION \*)interface->pFunctionList)->minor,

interface->pFunctionList,

interface->flags);

}

/\* get specific standard version interface \*/

version.major=3;

version.minor=0;

rv = C\_GetInterface((CK\_UTF8CHAR\_PTR)"PKCS 11",&version,&interface,flags);

if (rv == CKR\_OK) {

CK\_FUNCTION\_LIST\_3\_0\_PTR pkcs11=interface->pFunctionList;

/\* ... use the new functions \*/

pkcs11->C\_LoginUser(hSession,userType,pPin,ulPinLen,  
 pUsername,ulUsernameLen);

}

/\* get specific vendor version interface \*/

version.major=1;

version.minor=0;

rv = C\_GetInterface((CK\_UTF8CHAR\_PTR)

"Vendor VendorName",&version,&interface,flags);

if (rv == CKR\_OK) {

CK\_FUNCTION\_LIST\_VENDOR\_1\_0\_PTR pkcs11=interface->pFunctionList;

/\* ... use vendor specific functions \*/

pkcs11->C\_VendorFunction1(param1,param2,param3);

}

## Slot and token management functions

Cryptoki provides the following functions for slot and token management:

### C\_GetSlotList

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetSlotList)(

CK\_BBOOL tokenPresent,

CK\_SLOT\_ID\_PTR pSlotList,

CK\_ULONG\_PTR pulCount

);

**C\_GetSlotList** is used to obtain a list of slots in the system. *tokenPresent* indicates whether the list obtained includes only those slots with a token present (CK\_TRUE), or all slots (CK\_FALSE); *pulCount* points to the location that receives the number of slots.

There are two ways for an application to call **C\_GetSlotList**:

1. If *pSlotList* is NULL\_PTR, then all that **C\_GetSlotList** does is return (in \**pulCount*) the number of slots, without actually returning a list of slots. The contents of the buffer pointed to by *pulCount* on entry to **C\_GetSlotList** has no meaning in this case, and the call returns the value CKR\_OK.
2. If *pSlotList* is not NULL\_PTR, then \**pulCount* MUST contain the size (in terms of **CK\_SLOT\_ID** elements) of the buffer pointed to by *pSlotList*. If that buffer is large enough to hold the list of slots, then the list is returned in it, and CKR\_OK is returned. If not, then the call to **C\_GetSlotList** returns the value CKR\_BUFFER\_TOO\_SMALL. In either case, the value \**pulCount* is set to hold the number of slots.

Because **C\_GetSlotList** does not allocate any space of its own, an application will often call **C\_GetSlotList** twice (or sometimes even more times—if an application is trying to get a list of all slots with a token present, then the number of such slots can (unfortunately) change between when the application asks for how many such slots there are and when the application asks for the slots themselves). However, multiple calls to **C\_GetSlotList** are by no means *required*.

All slots which **C\_GetSlotList** reports MUST be able to be queried as valid slots by **C\_GetSlotInfo**. Furthermore, the set of slots accessible through a Cryptoki library is checked at the time that **C\_GetSlotList,** for list length prediction (NULL pSlotList argument) is called. If an application calls **C\_GetSlotList** with a non-NULL pSlotList, and *then* the user adds or removes a hardware device, the changed slot list will only be visible and effective if **C\_GetSlotList** is called again with NULL. Even if **C\_ GetSlotList** is successfully called this way, it may or may not be the case that the changed slot list will be successfully recognized depending on the library implementation. On some platforms, or earlier PKCS11 compliant libraries, it may be necessary to successfully call **C\_Initialize** or to restart the entire system.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK.

Example:

CK\_ULONG ulSlotCount, ulSlotWithTokenCount;

CK\_SLOT\_ID\_PTR pSlotList, pSlotWithTokenList;

CK\_RV rv;

/\* Get list of all slots \*/

rv = C\_GetSlotList(CK\_FALSE, NULL\_PTR, &ulSlotCount);

if (rv == CKR\_OK) {

pSlotList =

(CK\_SLOT\_ID\_PTR) malloc(ulSlotCount\*sizeof(CK\_SLOT\_ID));

rv = C\_GetSlotList(CK\_FALSE, pSlotList, &ulSlotCount);

if (rv == CKR\_OK) {

/\* Now use that list of all slots \*/

.

.

}

free(pSlotList);

}

/\* Get list of all slots with a token present \*/

pSlotWithTokenList = (CK\_SLOT\_ID\_PTR) malloc(0);

ulSlotWithTokenCount = 0;

while (1) {

rv = C\_GetSlotList(

CK\_TRUE, pSlotWithTokenList, &ulSlotWithTokenCount);

if (rv != CKR\_BUFFER\_TOO\_SMALL)

break;

pSlotWithTokenList = realloc(

pSlotWithTokenList,

ulSlotWithTokenList\*sizeof(CK\_SLOT\_ID));

}

if (rv == CKR\_OK) {

/\* Now use that list of all slots with a token present \*/

.

.

}

free(pSlotWithTokenList);

### C\_GetSlotInfo

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetSlotInfo)(

CK\_SLOT\_ID slotID,

CK\_SLOT\_INFO\_PTR pInfo

);

**C\_GetSlotInfo** obtains information about a particular slot in the system. *slotID* is the ID of the slot; *pInfo* points to the location that receives the slot information.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SLOT\_ID\_INVALID.

Example: see **C\_GetTokenInfo.**

### C\_GetTokenInfo

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetTokenInfo)(

CK\_SLOT\_ID slotID,

CK\_TOKEN\_INFO\_PTR pInfo

);

**C\_GetTokenInfo** obtains information about a particular token in the system. *slotID* is the ID of the token’s slot; *pInfo* points to the location that receives the token information.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SLOT\_ID\_INVALID, CKR\_TOKEN\_NOT\_PRESENT, CKR\_TOKEN\_NOT\_RECOGNIZED, CKR\_ARGUMENTS\_BAD.

Example:

CK\_ULONG ulCount;

CK\_SLOT\_ID\_PTR pSlotList;

CK\_SLOT\_INFO slotInfo;

CK\_TOKEN\_INFO tokenInfo;

CK\_RV rv;

rv = C\_GetSlotList(CK\_FALSE, NULL\_PTR, &ulCount);

if ((rv == CKR\_OK) && (ulCount > 0)) {

pSlotList = (CK\_SLOT\_ID\_PTR) malloc(ulCount\*sizeof(CK\_SLOT\_ID));

rv = C\_GetSlotList(CK\_FALSE, pSlotList, &ulCount);

assert(rv == CKR\_OK);

/\* Get slot information for first slot \*/

rv = C\_GetSlotInfo(pSlotList[0], &slotInfo);

assert(rv == CKR\_OK);

/\* Get token information for first slot \*/

rv = C\_GetTokenInfo(pSlotList[0], &tokenInfo);

if (rv == CKR\_TOKEN\_NOT\_PRESENT) {

.

.

}

.

.

free(pSlotList);

}

### C\_WaitForSlotEvent

CK\_DECLARE\_FUNCTION(CK\_RV, C\_WaitForSlotEvent)(

CK\_FLAGS flags,

CK\_SLOT\_ID\_PTR pSlot,

CK\_VOID\_PTR pReserved

);

**C\_WaitForSlotEvent** waits for a slot event, such as token insertion or token removal, to occur. *flags* determines whether or not the **C\_WaitForSlotEvent** call blocks (*i.e.*, waits for a slot event to occur); *pSlot* points to a location which will receive the ID of the slot that the event occurred in. *pReserved* is reserved for future versions; for this version of Cryptoki, it should be NULL\_PTR.

At present, the only flag defined for use in the *flags* argument is **CKF\_DONT\_BLOCK**:

Internally, each Cryptoki application has a flag for each slot which is used to track whether or not any unrecognized events involving that slot have occurred. When an application initially calls **C\_Initialize**, every slot’s event flag is cleared. Whenever a slot event occurs, the flag corresponding to the slot in which the event occurred is set.

If **C\_WaitForSlotEvent** is called with the **CKF\_DONT\_BLOCK** flag set in the *flags* argument, and some slot’s event flag is set, then that event flag is cleared, and the call returns with the ID of that slot in the location pointed to by *pSlot*. If more than one slot’s event flag is set at the time of the call, one such slot is chosen by the library to have its event flag cleared and to have its slot ID returned.

If **C\_WaitForSlotEvent** is called with the **CKF\_DONT\_BLOCK** flag set in the *flags* argument, and no slot’s event flag is set, then the call returns with the value CKR\_NO\_EVENT. In this case, the contents of the location pointed to by *pSlot* when **C\_WaitForSlotEvent** are undefined.

If **C\_WaitForSlotEvent** is called with the **CKF\_DONT\_BLOCK** flag clear in the *flags* argument, then the call behaves as above, except that it will block. That is, if no slot’s event flag is set at the time of the call, **C\_WaitForSlotEvent** will wait until some slot’s event flag becomes set. If a thread of an application has a **C\_WaitForSlotEvent** call blocking when another thread of that application calls **C\_Finalize**, the **C\_WaitForSlotEvent** call returns with the value CKR\_CRYPTOKI\_NOT\_INITIALIZED.

*Although the parameters supplied to* ***C\_Initialize*** *can in general allow for safe multi-threaded access to a Cryptoki library,* ***C\_WaitForSlotEvent*** *is exceptional in that the behavior of Cryptoki is undefined if multiple threads of a single application make simultaneous calls to* ***C\_WaitForSlotEvent****.*

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_NO\_EVENT, CKR\_OK.

Example:

CK\_FLAGS flags = 0;

CK\_SLOT\_ID slotID;

CK\_SLOT\_INFO slotInfo;

CK\_RV rv;

.

.

/\* Block and wait for a slot event \*/

rv = C\_WaitForSlotEvent(flags, &slotID, NULL\_PTR);

assert(rv == CKR\_OK);

/\* See what’s up with that slot \*/

rv = C\_GetSlotInfo(slotID, &slotInfo);

assert(rv == CKR\_OK);

### C\_GetMechanismList

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetMechanismList)(

CK\_SLOT\_ID slotID,

CK\_MECHANISM\_TYPE\_PTR pMechanismList,

CK\_ULONG\_PTR pulCount

);

**C\_GetMechanismList** is used to obtain a list of mechanism types supported by a token. *SlotID* is the ID of the token’s slot; *pulCount* points to the location that receives the number of mechanisms.

There are two ways for an application to call **C\_GetMechanismList**:

1. If *pMechanismList* is NULL\_PTR, then all that **C\_GetMechanismList** does is return (in \**pulCount*) the number of mechanisms, without actually returning a list of mechanisms. The contents of \**pulCount* on entry to **C\_GetMechanismList** has no meaning in this case, and the call returns the value CKR\_OK.
2. If *pMechanismList* is not NULL\_PTR, then \**pulCount* MUST contain the size (in terms of **CK\_MECHANISM\_TYPE** elements) of the buffer pointed to by *pMechanismList*. If that buffer is large enough to hold the list of mechanisms, then the list is returned in it, and CKR\_OK is returned. If not, then the call to **C\_GetMechanismList** returns the value CKR\_BUFFER\_TOO\_SMALL. In either case, the value \**pulCount* is set to hold the number of mechanisms.

Because **C\_GetMechanismList** does not allocate any space of its own, an application will often call **C\_GetMechanismList** twice. However, this behavior is by no means required.

Return values: CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SLOT\_ID\_INVALID, CKR\_TOKEN\_NOT\_PRESENT, CKR\_TOKEN\_NOT\_RECOGNIZED, CKR\_ARGUMENTS\_BAD.

Example:

CK\_SLOT\_ID slotID;

CK\_ULONG ulCount;

CK\_MECHANISM\_TYPE\_PTR pMechanismList;

CK\_RV rv;

.

.

rv = C\_GetMechanismList(slotID, NULL\_PTR, &ulCount);

if ((rv == CKR\_OK) && (ulCount > 0)) {

pMechanismList =

(CK\_MECHANISM\_TYPE\_PTR)

malloc(ulCount\*sizeof(CK\_MECHANISM\_TYPE));

rv = C\_GetMechanismList(slotID, pMechanismList, &ulCount);

if (rv == CKR\_OK) {

.

.

}

free(pMechanismList);

}

### C\_GetMechanismInfo

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetMechanismInfo)(

CK\_SLOT\_ID slotID,

CK\_MECHANISM\_TYPE type,

CK\_MECHANISM\_INFO\_PTR pInfo

);

**C\_GetMechanismInfo** obtains information about a particular mechanism possibly supported by a token. *slotID* is the ID of the token’s slot; *type* is the type of mechanism; *pInfo* points to the location that receives the mechanism information.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_MECHANISM\_INVALID, CKR\_OK, CKR\_SLOT\_ID\_INVALID, CKR\_TOKEN\_NOT\_PRESENT, CKR\_TOKEN\_NOT\_RECOGNIZED, CKR\_ARGUMENTS\_BAD.

Example:

CK\_SLOT\_ID slotID;

CK\_MECHANISM\_INFO info;

CK\_RV rv;

.

.

/\* Get information about the CKM\_MD2 mechanism for this token \*/

rv = C\_GetMechanismInfo(slotID, CKM\_MD2, &info);

if (rv == CKR\_OK) {

if (info.flags & CKF\_DIGEST) {

.

.

}

}

### C\_InitToken

CK\_DECLARE\_FUNCTION(CK\_RV, C\_InitToken)(

CK\_SLOT\_ID slotID,

CK\_UTF8CHAR\_PTR pPin,

CK\_ULONG ulPinLen,

CK\_UTF8CHAR\_PTR pLabel

);

**C\_InitToken** initializes a token. *slotID* is the ID of the token’s slot; *pPin* points to the SO’s initial PIN (which need *not* be null-terminated); *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-byte label of the token (which MUST be padded with blank characters, and which MUST *not* be null-terminated). This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

If the token has not been initialized (i.e. new from the factory), then the *pPin* parameter becomes the initial value of the SO PIN. If the token is being reinitialized, the *pPin* parameter is checked against the existing SO PIN to authorize the initialization operation. In both cases, the SO PIN is the value *pPin* after the function completes successfully. If the SO PIN is lost, then the card MUST be reinitialized using a mechanism outside the scope of this standard. The **CKF\_TOKEN\_INITIALIZED** flag in the **CK\_TOKEN\_INFO** structure indicates the action that will result from calling **C\_InitToken**. If set, the token will be reinitialized, and the client MUST supply the existing SO password in *pPin*.

When a token is initialized, all objects that can be destroyed are destroyed (*i.e.*, all except for “indestructible” objects such as keys built into the token). Also, access by the normal user is disabled until the SO sets the normal user’s PIN. Depending on the token, some “default” objects may be created, and attributes of some objects may be set to default values.

If the token has a “protected authentication path”, as indicated by the **CKF\_PROTECTED\_AUTHENTICATION\_PATH** flag in its **CK\_TOKEN\_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having the application send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin* parameter to **C\_InitToken** should be NULL\_PTR. During the execution of **C\_InitToken**, the SO’s PIN will be entered through the protected authentication path.

If the token has a protected authentication path other than a PINpad, then it is token-dependent whether or not **C\_InitToken** can be used to initialize the token.

A token cannot be initialized if Cryptoki detects that *any* application has an open session with it; when a call to **C\_InitToken** is made under such circumstances, the call fails with error CKR\_SESSION\_EXISTS. Unfortunately, it may happen when **C\_InitToken** is called that some other application *does* have an open session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other applications using the token. If this is the case, then the consequences of the **C\_InitToken** call are undefined.

The **C\_InitToken** function may not be sufficient to properly initialize complex tokens. In these situations, an initialization mechanism outside the scope of Cryptoki MUST be employed. The definition of “complex token” is product specific.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_PIN\_INCORRECT, CKR\_PIN\_LOCKED, CKR\_SESSION\_EXISTS, CKR\_SLOT\_ID\_INVALID, CKR\_TOKEN\_NOT\_PRESENT, CKR\_TOKEN\_NOT\_RECOGNIZED, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_ARGUMENTS\_BAD.

Example:

CK\_SLOT\_ID slotID;

CK\_UTF8CHAR pin[] = {“MyPIN”};

CK\_UTF8CHAR label[32];

CK\_RV rv;

.

.

memset(label, ‘ ’, sizeof(label));

memcpy(label, “My first token”, strlen(“My first token”));

rv = C\_InitToken(slotID, pin, strlen(pin), label);

if (rv == CKR\_OK) {

.

.

}

### C\_InitPIN

CK\_DECLARE\_FUNCTION(CK\_RV, C\_InitPIN)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_UTF8CHAR\_PTR pPin,  
 CK\_ULONG ulPinLen  
);

**C\_InitPIN** initializes the normal user’s PIN. *hSession* is the session’s handle; *pPin* points to the normal user’s PIN; *ulPinLen* is the length in bytes of the PIN. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

**C\_InitPIN** can only be called in the “R/W SO Functions” state. An attempt to call it from a session in any other state fails with error CKR\_USER\_NOT\_LOGGED\_IN.

If the token has a “protected authentication path”, as indicated by the CKF\_PROTECTED\_AUTHENTICATION\_PATH flag in its **CK\_TOKEN\_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having to send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or on the slot device. To initialize the normal user’s PIN on a token with such a protected authentication path, the *pPin* parameter to **C\_InitPIN** should be NULL\_PTR. During the execution of **C\_InitPIN**, the SO will enter the new PIN through the protected authentication path.

If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether or not **C\_InitPIN** can be used to initialize the normal user’s token access.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_PIN\_INVALID, CKR\_PIN\_LEN\_RANGE, CKR\_SESSION\_CLOSED, CKR\_SESSION\_READ\_ONLY, CKR\_SESSION\_HANDLE\_INVALID, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_ARGUMENTS\_BAD.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_UTF8CHAR newPin[]= {“NewPIN”};

CK\_RV rv;

rv = C\_InitPIN(hSession, newPin, sizeof(newPin)-1);

if (rv == CKR\_OK) {

.

.

}

### C\_SetPIN

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SetPIN)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_UTF8CHAR\_PTR pOldPin,  
 CK\_ULONG ulOldLen,  
 CK\_UTF8CHAR\_PTR pNewPin,  
 CK\_ULONG ulNewLen  
);

**C\_SetPIN** modifies the PIN of the user that is currently logged in, or the CKU\_USER PIN if the session is not logged in. *hSession* is the session’s handle; *pOldPin* points to the old PIN; *ulOldLen* is the length in bytes of the old PIN; *pNewPin* points to the new PIN; *ulNewLen* is the length in bytes of the new PIN. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

**C\_SetPIN** can only be called in the “R/W Public Session” state, “R/W SO Functions” state, or “R/W User Functions” state. An attempt to call it from a session in any other state fails with error CKR\_SESSION\_READ\_ONLY.

If the token has a “protected authentication path”, as indicated by the CKF\_PROTECTED\_AUTHENTICATION\_PATH flag in its **CK\_TOKEN\_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having to send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or on the slot device. To modify the current user’s PIN on a token with such a protected authentication path, the *pOldPin* and *pNewPin* parameters to **C\_SetPIN** should be NULL\_PTR. During the execution of **C\_SetPIN**, the current user will enter the old PIN and the new PIN through the protected authentication path. It is not specified how the PIN pad should be used to enter *two* PINs; this varies.

If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether or not **C\_SetPIN** can be used to modify the current user’s PIN.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_PIN\_INCORRECT, CKR\_PIN\_INVALID, CKR\_PIN\_LEN\_RANGE, CKR\_PIN\_LOCKED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_ARGUMENTS\_BAD.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_UTF8CHAR oldPin[] = {“OldPIN”};

CK\_UTF8CHAR newPin[] = {“NewPIN”};

CK\_RV rv;

rv = C\_SetPIN(

hSession, oldPin, sizeof(oldPin)-1, newPin, sizeof(newPin)-1);

if (rv == CKR\_OK) {

.

.

}

## Session management functions

A typical application might perform the following series of steps to make use of a token (note that there are other reasonable sequences of events that an application might perform):

1. Select a token.
2. Make one or more calls to **C\_OpenSession** to obtain one or more sessions with the token.
3. Call **C\_Login** to log the user into the token. Since all sessions an application has with a token have a shared login state, **C\_Login** only needs to be called for one of the sessions.
4. Perform cryptographic operations using the sessions with the token.
5. Call **C\_CloseSession** once for each session that the application has with the token, or call **C\_CloseAllSessions** to close all the application’s sessions simultaneously.

As has been observed, an application may have concurrent sessions with more than one token. It is also possible for a token to have concurrent sessions with more than one application.

Cryptoki provides the following functions for session management:

### C\_OpenSession

CK\_DECLARE\_FUNCTION(CK\_RV, C\_OpenSession)(  
 CK\_SLOT\_ID slotID,  
 CK\_FLAGS flags,  
 CK\_VOID\_PTR pApplication,  
 CK\_NOTIFY Notify,  
 CK\_SESSION\_HANDLE\_PTR phSession  
);

**C\_OpenSession** opens a session between an application and a token in a particular slot. *slotID* is the slot’s ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to the notification callback; *Notify* is the address of the notification callback function (see Section 5.21); *phSession* points to the location that receives the handle for the new session.

When opening a session with **C\_OpenSession**, the *flags* parameter consists of the logical OR of zero or more bit flags defined in the **CK\_SESSION\_INFO** data type. For legacy reasons, the **CKF\_SERIAL\_SESSION** bit MUST always be set; if a call to **C\_OpenSession** does not have this bit set, the call should return unsuccessfully with the error code CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED.

There may be a limit on the number of concurrent sessions an application may have with the token, which may depend on whether the session is “read-only” or “read/write”. An attempt to open a session which does not succeed because there are too many existing sessions of some type should return CKR\_SESSION\_COUNT.

If the token is write-protected (as indicated in the **CK\_TOKEN\_INFO** structure), then only read-only sessions may be opened with it.

If the application calling **C\_OpenSession** already has a R/W SO session open with the token, then any attempt to open a R/O session with the token fails with error code CKR\_SESSION\_READ\_WRITE\_SO\_EXISTS (see **[PKCS11-UG]** for further details).

The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the application does not wish to support callbacks, it should pass a value of NULL\_PTR as the *Notify* parameter. See Section 5.21 for more information about application callbacks.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_COUNT, CKR\_SESSION\_PARALLEL\_NOT\_SUPPORTED, CKR\_SESSION\_READ\_WRITE\_SO\_EXISTS, CKR\_SLOT\_ID\_INVALID, CKR\_TOKEN\_NOT\_PRESENT, CKR\_TOKEN\_NOT\_RECOGNIZED, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_ARGUMENTS\_BAD.

Example: see **C\_CloseSession**.

### C\_CloseSession

CK\_DECLARE\_FUNCTION(CK\_RV, C\_CloseSession)(  
 CK\_SESSION\_HANDLE hSession  
);

**C\_CloseSession** closes a session between an application and a token. *hSession* is the session’s handle.

When a session is closed, all session objects created by the session are destroyed automatically, even if the application has other sessions “using” the objects (see **[PKCS11-UG]** for further details).

If this function is successful and it closes the last session between the application and the token, the login state of the token for the application returns to public sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W Public sessions.

Depending on the token, when the last open session any application has with the token is closed, the token may be “ejected” from its reader (if this capability exists).

Despite the fact this **C\_CloseSession** is supposed to close a session, the return value CKR\_SESSION\_CLOSED is an *error* return. It actually indicates the (probably somewhat unlikely) event that while this function call was executing, another call was made to **C\_CloseSession** to close this particular session, and that call finished executing first. Such uses of sessions are a bad idea, and Cryptoki makes little promise of what will occur in general if an application indulges in this sort of behavior.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

CK\_SLOT\_ID slotID;

CK\_BYTE application;

CK\_NOTIFY MyNotify;

CK\_SESSION\_HANDLE hSession;

CK\_RV rv;

.

.

application = 17;

MyNotify = &EncryptionSessionCallback;

rv = C\_OpenSession(

slotID, CKF\_SERIAL\_SESSION | CKF\_RW\_SESSION,

(CK\_VOID\_PTR) &application, MyNotify,

&hSession);

if (rv == CKR\_OK) {

.

.

C\_CloseSession(hSession);

}

### C\_CloseAllSessions

CK\_DECLARE\_FUNCTION(CK\_RV, C\_CloseAllSessions)(  
 CK\_SLOT\_ID slotID  
);

**C\_CloseAllSessions** closes all sessions an application has with a token. *slotID* specifies the token’s slot.

When a session is closed, all session objects created by the session are destroyed automatically.

After successful execution of this function, the login state of the token for the application returns to public sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W Public sessions.

Depending on the token, when the last open session any application has with the token is closed, the token may be “ejected” from its reader (if this capability exists).

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SLOT\_ID\_INVALID, CKR\_TOKEN\_NOT\_PRESENT.

Example:

CK\_SLOT\_ID slotID;

CK\_RV rv;

.

.

rv = C\_CloseAllSessions(slotID);

### C\_GetSessionInfo

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetSessionInfo)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_SESSION\_INFO\_PTR pInfo  
);

**C\_GetSessionInfo** obtains information about a session. *hSession* is the session’s handle; *pInfo* points to the location that receives the session information.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_ARGUMENTS\_BAD.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_SESSION\_INFO info;

CK\_RV rv;

.

.

rv = C\_GetSessionInfo(hSession, &info);

if (rv == CKR\_OK) {

if (info.state == CKS\_RW\_USER\_FUNCTIONS) {

.

.

}

.

.

}

### C\_SessionCancel

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SessionCancel)(  
 CK\_SESSION\_HANDLE hSession   
 CK\_FLAGS flags   
);

**C\_SessionCancel** terminates active session based operations. *hSession* is the session’s handle; *flags* indicates the operations to cancel.

To identify which operation(s) should be terminated, the *flags* parameter should be assigned the logical bitwise OR of one or more of the bit flags defined in the **CK\_MECHANISM\_INFO** structure.

If no flags are set, the session state will not be modified and CKR\_OK will be returned.

If a flag is set for an operation that has not been initialized in the session, the operation flag will be ignored and **C\_SessionCancel** will behave as if the operation flag was not set.

If any of the operations indicated by the *flags* parameter cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be returned. If multiple operation flags were set and CKR\_OPERATION\_CANCEL\_FAILED is returned, this function does not provide any information about which operation(s) could not be cancelled. If an application desires to know if any single operation could not be cancelled, the application should not call **C\_SessionCancel** with multiple flags set.

If **C\_SessionCancel** is called from an application callback (see Section 5.21), no action will be taken by the library and CKR\_FUNCTION\_FAILED must be returned.

If **C\_SessionCancel** is used to cancel one half of a dual-function operation, the remaining operation should still be left in an active state. However, it is expected that some Cryptoki implementations may not support this and return CKR\_OPERATION\_CANCEL\_FAILED unless flags for both operations are provided.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_CANCEL\_FAILED, CKR\_TOKEN\_NOT\_PRESENT.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_RV rv;

rv = C\_EncryptInit(hSession, &mechanism, hKey);

if (rv != CKR\_OK)

{

.

.

}

rv = C\_SessionCancel (hSession, CKF\_ENCRYPT);

if (rv != CKR\_OK)

{

.

.

}

rv = C\_EncryptInit(hSession, &mechanism, hKey);

if (rv != CKR\_OK)

{

.

.

}

Below are modifications to existing API descriptions to allow an alternate method of cancelling individual operations. The additional text is highlighted.

### C\_GetOperationState

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetOperationState)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pOperationState,  
 CK\_ULONG\_PTR pulOperationStateLen  
);

**C\_GetOperationState** obtains a copy of the cryptographic operations state of a session, encoded as a string of bytes. *hSession* is the session’s handle; *pOperationState* points to the location that receives the state; *pulOperationStateLen* points to the location that receives the length in bytes of the state.

Although the saved state output by **C\_GetOperationState** is not really produced by a “cryptographic mechanism”, **C\_GetOperationState** nonetheless uses the convention described in Section 5.2 on producing output.

Precisely what the “cryptographic operations state” this function saves is varies from token to token; however, this state is what is provided as input to **C\_SetOperationState** to restore the cryptographic activities of a session.

Consider a session which is performing a message digest operation using SHA-1 (*i.e.*, the session is using the **CKM\_SHA\_1** mechanism). Suppose that the message digest operation was initialized properly, and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The application now wants to “save the state” of this digest operation, so that it can continue it later. In this particular case, since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic operations state of the session most likely consists of three distinct parts: the state of SHA-1’s 160-bit internal chaining variable; the 16 bytes of unprocessed input data; and some administrative data indicating that this saved state comes from a session which was performing SHA-1 hashing. Taken together, these three pieces of information suffice to continue the current hashing operation at a later time.

Consider next a session which is performing an encryption operation with DES (a block cipher with a block size of 64 bits) in CBC (cipher-block chaining) mode (*i.e.*, the session is using the **CKM\_DES\_CBC** mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already been produced and output. In this case, the cryptographic operations state of the session most likely consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipher-block chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some administrative data indicating that this saved state comes from a session which was performing DES encryption in CBC mode; and possibly the DES key being used for encryption (see **C\_SetOperationState** for more information on whether or not the key is present in the saved state).

If a session is performing two cryptographic operations simultaneously (see Section 5.14), then the cryptographic operations state of the session will contain all the necessary information to restore both operations.

An attempt to save the cryptographic operations state of a session which does not currently have some active savable cryptographic operation(s) (encryption, decryption, digesting, signing without message recovery, verification without message recovery, or some legal combination of two of these) should fail with the error CKR\_OPERATION\_NOT\_INITIALIZED.

An attempt to save the cryptographic operations state of a session which is performing an appropriate cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain necessary state information and/or key information can’t leave the token, for example) should fail with the error CKR\_STATE\_UNSAVEABLE.

Return values: CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_STATE\_UNSAVEABLE, CKR\_ARGUMENTS\_BAD.

Example: see **C\_SetOperationState**.

### C\_SetOperationState

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SetOperationState)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pOperationState,  
 CK\_ULONG ulOperationStateLen,  
 CK\_OBJECT\_HANDLE hEncryptionKey,  
 CK\_OBJECT\_HANDLE hAuthenticationKey  
);

**C\_SetOperationState** restores the cryptographic operations state of a session from a string of bytes obtained with **C\_GetOperationState**. *hSession* is the session’s handle; *pOperationState* points to the location holding the saved state; *ulOperationStateLen* holds the length of the saved state; *hEncryptionKey* holds a handle to the key which will be used for an ongoing encryption or decryption operation in the restored session (or 0 if no encryption or decryption key is needed, either because no such operation is ongoing in the stored session or because all the necessary key information is present in the saved state); *hAuthenticationKey* holds a handle to the key which will be used for an ongoing signature, MACing, or verification operation in the restored session (or 0 if no such key is needed, either because no such operation is ongoing in the stored session or because all the necessary key information is present in the saved state).

The state need not have been obtained from the same session (the “source session”) as it is being restored to (the “destination session”). However, the source session and destination session should have a common session state (*e.g.*, CKS\_RW\_USER\_FUNCTIONS), and should be with a common token. There is also no guarantee that cryptographic operations state may be carried across logins, or across different Cryptoki implementations.

If **C\_SetOperationState** is supplied with alleged saved cryptographic operations state which it can determine is not valid saved state (or is cryptographic operations state from a session with a different session state, or is cryptographic operations state from a different token), it fails with the error CKR\_SAVED\_STATE\_INVALID.

Saved state obtained from calls to **C\_GetOperationState** may or may not contain information about keys in use for ongoing cryptographic operations. If a saved cryptographic operations state has an ongoing encryption or decryption operation, and the key in use for the operation is not saved in the state, then it MUST be supplied to **C\_SetOperationState** in the *hEncryptionKey* argument. If it is not, then **C\_SetOperationState** will fail and return the error CKR\_KEY\_NEEDED. If the key in use for the operation *is* saved in the state, then it *can* be supplied in the *hEncryptionKey* argument, but this is not required.

Similarly, if a saved cryptographic operations state has an ongoing signature, MACing, or verification operation, and the key in use for the operation is not saved in the state, then it MUST be supplied to **C\_SetOperationState** in the *hAuthenticationKey* argument. If it is not, then **C\_SetOperationState** will fail with the error CKR\_KEY\_NEEDED. If the key in use for the operation *is* saved in the state, then it *can* be supplied in the *hAuthenticationKey* argument, but this is not required.

If an *irrelevant* key is supplied to **C\_SetOperationState** call (*e.g.*, a nonzero key handle is submitted in the *hEncryptionKey* argument, but the saved cryptographic operations state supplied does not have an ongoing encryption or decryption operation, then **C\_SetOperationState** fails with the error CKR\_KEY\_NOT\_NEEDED.

If a key is supplied as an argument to **C\_SetOperationState**, and **C\_SetOperationState** can somehow detect that this key was not the key being used in the source session for the supplied cryptographic operations state (it may be able to detect this if the key or a hash of the key is present in the saved state, for example), then **C\_SetOperationState** fails with the error CKR\_KEY\_CHANGED.

An application can look at the **CKF\_RESTORE\_KEY\_NOT\_NEEDED** flag in the flags field of the **CK\_TOKEN\_INFO** field for a token to determine whether or not it needs to supply key handles to **C\_SetOperationState** calls. If this flag is true, then a call to **C\_SetOperationState** *never* needs a key handle to be supplied to it. If this flag is false, then at least some of the time, **C\_SetOperationState** requires a key handle, and so the application should probably *always* pass in any relevant key handles when restoring cryptographic operations state to a session.

**C\_SetOperationState** can successfully restore cryptographic operations state to a session even if that session has active cryptographic or object search operations when **C\_SetOperationState** is called (the ongoing operations are abruptly cancelled).

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_CHANGED, CKR\_KEY\_NEEDED, CKR\_KEY\_NOT\_NEEDED, CKR\_OK, CKR\_SAVED\_STATE\_INVALID, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_ARGUMENTS\_BAD.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_MECHANISM digestMechanism;

CK\_BYTE\_PTR pState;

CK\_ULONG ulStateLen;

CK\_BYTE data1[] = {0x01, 0x03, 0x05, 0x07};

CK\_BYTE data2[] = {0x02, 0x04, 0x08};

CK\_BYTE data3[] = {0x10, 0x0F, 0x0E, 0x0D, 0x0C};

CK\_BYTE pDigest[20];

CK\_ULONG ulDigestLen;

CK\_RV rv;

.

.

/\* Initialize hash operation \*/

rv = C\_DigestInit(hSession, &digestMechanism);

assert(rv == CKR\_OK);

/\* Start hashing \*/

rv = C\_DigestUpdate(hSession, data1, sizeof(data1));

assert(rv == CKR\_OK);

/\* Find out how big the state might be \*/

rv = C\_GetOperationState(hSession, NULL\_PTR, &ulStateLen);

assert(rv == CKR\_OK);

/\* Allocate some memory and then get the state \*/

pState = (CK\_BYTE\_PTR) malloc(ulStateLen);

rv = C\_GetOperationState(hSession, pState, &ulStateLen);

/\* Continue hashing \*/

rv = C\_DigestUpdate(hSession, data2, sizeof(data2));

assert(rv == CKR\_OK);

/\* Restore state. No key handles needed \*/

rv = C\_SetOperationState(hSession, pState, ulStateLen, 0, 0);

assert(rv == CKR\_OK);

/\* Continue hashing from where we saved state \*/

rv = C\_DigestUpdate(hSession, data3, sizeof(data3));

assert(rv == CKR\_OK);

/\* Conclude hashing operation \*/

ulDigestLen = sizeof(pDigest);

rv = C\_DigestFinal(hSession, pDigest, &ulDigestLen);

if (rv == CKR\_OK) {

/\* pDigest[] now contains the hash of 0x01030507100F0E0D0C \*/

.

.

}

### C\_Login

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Login)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_USER\_TYPE userType,  
 CK\_UTF8CHAR\_PTR pPin,  
 CK\_ULONG ulPinLen  
);

**C\_Login** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to the user’s PIN; *ulPinLen* is the length of the PIN. This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

When the user type is either CKU\_SO or CKU\_USER, if the call succeeds, each of the application's sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User Functions" state. If the user type is CKU\_CONTEXT\_SPECIFIC, the behavior of C\_Login depends on the context in which it is called. Improper use of this user type will result in a return value CKR\_OPERATION\_NOT\_INITIALIZED..

If the token has a “protected authentication path”, as indicated by the **CKF\_PROTECTED\_AUTHENTICATION\_PATH** flag in its **CK\_TOKEN\_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having to send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or on the slot device. Or the user might not even use a PIN—authentication could be achieved by some fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin* parameter to **C\_Login** should be NULL\_PTR. When **C\_Login** returns, whatever authentication method supported by the token will have been performed; a return value of CKR\_OK means that the user was successfully authenticated, and a return value of CKR\_PIN\_INCORRECT means that the user was denied access.

If there are any active cryptographic or object finding operations in an application’s session, and then **C\_Login** is successfully executed by that application, it may or may not be the case that those operations are still active. Therefore, before logging in, any active operations should be finished.

If the application calling **C\_Login** has a R/O session open with the token, then it will be unable to log the SO into a session (see **[PKCS11-UG]** for further details). An attempt to do this will result in the error code CKR\_SESSION\_READ\_ONLY\_EXISTS.

C\_Login may be called repeatedly, without intervening **C\_Logout** calls, if (and only if) a key with the CKA\_ALWAYS\_AUTHENTICATE attribute set to CK\_TRUE exists, and the user needs to do cryptographic operation on this key. See further Section 4.9.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_PIN\_INCORRECT, CKR\_PIN\_LOCKED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY\_EXISTS, CKR\_USER\_ALREADY\_LOGGED\_IN, CKR\_USER\_ANOTHER\_ALREADY\_LOGGED\_IN, CKR\_USER\_PIN\_NOT\_INITIALIZED, CKR\_USER\_TOO\_MANY\_TYPES, CKR\_USER\_TYPE\_INVALID.

Example: see **C\_Logout**.

### C\_LoginUser

CK\_DECLARE\_FUNCTION(CK\_RV, C\_LoginUser)(  
 CK\_SESSION\_HANDLE hSession,

CK\_USER\_TYPE userType,

CK\_UTF8CHAR\_PTR pPin,

CK\_ULONG ulPinLen,

CK\_UTF8CHAR\_PTR pUsername,

CK\_ULONG ulUsernameLen  
);

**C\_LoginUser** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to the user’s PIN; *ulPinLen* is the length of the PIN, *pUsername* points to the user name, *ulUsernameLen* is the length of the user name. This standard allows PIN and user name values to contain any valid UTF8 character, but the token may impose subset restrictions.

When the user type is either CKU\_SO or CKU\_USER, if the call succeeds, each of the application's sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User Functions" state. If the user type is CKU\_CONTEXT\_SPECIFIC, the behavior of **C\_LoginUser** depends on the context in which it is called. Improper use of this user type will result in a return value CKR\_OPERATION\_NOT\_INITIALIZED.

If the token has a “protected authentication path”, as indicated by the CKF\_PROTECTED\_AUTHENTICATION\_PATH flag in its CK\_TOKEN\_INFO being set, then that means that there is some way for a user to be authenticated to the token without having to send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or on the slot device. The user might not even use a PIN—authentication could be achieved by some fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin* parameter to **C\_LoginUser** should be NULL\_PTR. When **C\_LoginUser** returns, whatever authentication method supported by the token will have been performed; a return value of CKR\_OK means that the user was successfully authenticated, and a return value of CKR\_PIN\_INCORRECT means that the user was denied access.

If there are any active cryptographic or object finding operations in an application’s session, and then **C\_LoginUser** is successfully executed by that application, it may or may not be the case that those operations are still active. Therefore, before logging in, any active operations should be finished.

If the application calling **C\_LoginUser** has a R/O session open with the token, then it will be unable to log the SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error code CKR\_SESSION\_READ\_ONLY\_EXISTS.

**C\_LoginUser** may be called repeatedly, without intervening **C\_Logout** calls, if (and only if) a key with the CKA\_ALWAYS\_AUTHENTICATE attribute set to CK\_TRUE exists, and the user needs to do cryptographic operation on this key. See further Section 4.9.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_PIN\_INCORRECT, CKR\_PIN\_LOCKED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY\_EXISTS, CKR\_USER\_ALREADY\_LOGGED\_IN, CKR\_USER\_ANOTHER\_ALREADY\_LOGGED\_IN, CKR\_USER\_PIN\_NOT\_INITIALIZED, CKR\_USER\_TOO\_MANY\_TYPES, CKR\_USER\_TYPE\_INVALID.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_UTF8CHAR userPin[] = {“MyPIN”};

CK\_UTF8CHAR userName[] = {“MyUserName”};

CK\_RV rv;

rv = C\_LoginUser(hSession, CKU\_USER, userPin, sizeof(userPin)-1, userName,

sizeof(userName)-1);

if (rv == CKR\_OK) {

.

.

rv = C\_Logout(hSession);

if (rv == CKR\_OK) {

.

.

}

}

### C\_Logout

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Logout)(  
 CK\_SESSION\_HANDLE hSession  
);

**C\_Logout** logs a user out from a token. *hSession* is the session’s handle.

Depending on the current user type, if the call succeeds, each of the application’s sessions will enter either the “R/W Public Session” state or the “R/O Public Session” state.

When **C\_Logout** successfully executes, any of the application’s handles to private objects become invalid (even if a user is later logged back into the token, those handles remain invalid). In addition, all private session objects from sessions belonging to the application are destroyed.

If there are any active cryptographic or object-finding operations in an application’s session, and then **C\_Logout** is successfully executed by that application, it may or may not be the case that those operations are still active. Therefore, before logging out, any active operations should be finished.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_UTF8CHAR userPin[] = {“MyPIN”};

CK\_RV rv;

rv = C\_Login(hSession, CKU\_USER, userPin, sizeof(userPin)-1);

if (rv == CKR\_OK) {

.

.

rv = C\_Logout(hSession);

if (rv == CKR\_OK) {

.

.

}

}

## Object management functions

Cryptoki provides the following functions for managing objects. Additional functions provided specifically for managing key objects are described in Section 5.18.

### C\_CreateObject

CK\_DECLARE\_FUNCTION(CK\_RV, C\_CreateObject)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_ATTRIBUTE\_PTR pTemplate,  
 CK\_ULONG ulCount,  
 CK\_OBJECT\_HANDLE\_PTR phObject  
);

**C\_CreateObject** creates a new object. *hSession* is the session’s handle; *pTemplate* points to the object’s template; *ulCount* is the number of attributes in the template; *phObject* points to the location that receives the new object’s handle.

If a call to **C\_CreateObject** cannot support the precise template supplied to it, it will fail and return without creating any object.

If **C\_CreateObject** is used to create a key object, the key object will have its **CKA\_LOCAL** attribute set to CK\_FALSE. If that key object is a secret or private key then the new key will have the **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, and the **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE.

Only session objects can be created during a read-only session. Only public objects can be created unless the normal user is logged in.

Whenever an object is created, a value for CKA\_UNIQUE\_ID is generated and assigned to the new object (See Section 4.4.1).

Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_READ\_ONLY, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_CURVE\_NOT\_SUPPORTED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_DOMAIN\_PARAMS\_INVALID, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCOMPLETE, CKR\_TEMPLATE\_INCONSISTENT, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE

hData,

hCertificate,

hKey;

CK\_OBJECT\_CLASS

dataClass = CKO\_DATA,

certificateClass = CKO\_CERTIFICATE,

keyClass = CKO\_PUBLIC\_KEY;

CK\_KEY\_TYPE keyType = CKK\_RSA;

CK\_UTF8CHAR application[] = {“My Application”};

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

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

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

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

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE dataTemplate[] = {

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

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

{CKA\_APPLICATION, application, sizeof(application)-1},

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

};

CK\_ATTRIBUTE certificateTemplate[] = {

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

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

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

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

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

};

CK\_ATTRIBUTE keyTemplate[] = {

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

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

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

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

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

};

CK\_RV rv;

.

.

/\* Create a data object \*/

rv = C\_CreateObject(hSession, dataTemplate, 4, &hData);

if (rv == CKR\_OK) {

.

.

}

/\* Create a certificate object \*/

rv = C\_CreateObject(

hSession, certificateTemplate, 5, &hCertificate);

if (rv == CKR\_OK) {

.

.

}

/\* Create an RSA public key object \*/

rv = C\_CreateObject(hSession, keyTemplate, 5, &hKey);

if (rv == CKR\_OK) {

.

.

}

### C\_CopyObject

CK\_DECLARE\_FUNCTION(CK\_RV, C\_CopyObject)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_OBJECT\_HANDLE hObject,  
 CK\_ATTRIBUTE\_PTR pTemplate,  
 CK\_ULONG ulCount,  
 CK\_OBJECT\_HANDLE\_PTR phNewObject  
);

**C\_CopyObject** copies an object, creating a new object for the copy. *hSession* is the session’s handle; *hObject* is the object’s handle; *pTemplate* points to the template for the new object; *ulCount* is the number of attributes in the template; *phNewObject* points to the location that receives the handle for the copy of the object.

The template may specify new values for any attributes of the object that can ordinarily be modified (*e.g.*, in the course of copying a secret key, a key’s **CKA\_EXTRACTABLE** attribute may be changed from CK\_TRUE to CK\_FALSE, but not the other way around. If this change is made, the new key’s **CKA\_NEVER\_EXTRACTABLE** attribute will have the value CK\_FALSE. Similarly, the template may specify that the new key’s **CKA\_SENSITIVE** attribute be CK\_TRUE; the new key will have the same value for its **CKA\_ALWAYS\_SENSITIVE** attribute as the original key). It may also specify new values of the **CKA\_TOKEN** and **CKA\_PRIVATE** attributes (*e.g.*, to copy a session object to a token object). If the template specifies a value of an attribute which is incompatible with other existing attributes of the object, the call fails with the return code CKR\_TEMPLATE\_INCONSISTENT.

If a call to **C\_CopyObject** cannot support the precise template supplied to it, it will fail and return without creating any object. If the object indicated by hObject has its CKA\_COPYABLE attribute set to CK\_FALSE, C\_CopyObject will return CKR\_ACTION\_PROHIBITED.

Whenever an object is copied, a new value for CKA\_UNIQUE\_ID is generated and assigned to the new object (See Section 4.4.1).

Only session objects can be created during a read-only session. Only public objects can be created unless the normal user is logged in.

Return values: , CKR\_ACTION\_PROHIBITED, CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_READ\_ONLY, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OBJECT\_HANDLE\_INVALID, CKR\_OK, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCONSISTENT, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey, hNewKey;

CK\_OBJECT\_CLASS keyClass = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DES;

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

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

CK\_BBOOL false = CK\_FALSE;

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE keyTemplate[] = {

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

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

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

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

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

};

CK\_ATTRIBUTE copyTemplate[] = {

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

};

CK\_RV rv;

.

.

/\* Create a DES secret key session object \*/

rv = C\_CreateObject(hSession, keyTemplate, 5, &hKey);

if (rv == CKR\_OK) {

/\* Create a copy which is a token object \*/

rv = C\_CopyObject(hSession, hKey, copyTemplate, 1, &hNewKey);

.

.

}

### C\_DestroyObject

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DestroyObject)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_OBJECT\_HANDLE hObject  
);

**C\_DestroyObject** destroys an object. *hSession* is the session’s handle; and *hObject* is the object’s handle.

Only session objects can be destroyed during a read-only session. Only public objects can be destroyed unless the normal user is logged in.

Certain objects may not be destroyed. Calling C\_DestroyObject on such objects will result in the CKR\_ACTION\_PROHIBITED error code. An application can consult the object's CKA\_DESTROYABLE attribute to determine if an object may be destroyed or not.

Return values: CKR\_ACTION\_PROHIBITED, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OBJECT\_HANDLE\_INVALID, CKR\_OK, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TOKEN\_WRITE\_PROTECTED.

Example: see **C\_GetObjectSize**.

### C\_GetObjectSize

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetObjectSize)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_OBJECT\_HANDLE hObject,  
 CK\_ULONG\_PTR pulSize  
);

**C\_GetObjectSize** gets the size of an object in bytes. *hSession* is the session’s handle; *hObject* is the object’s handle; *pulSize* points to the location that receives the size in bytes of the object.

Cryptoki does not specify what the precise meaning of an object’s size is. Intuitively, it is some measure of how much token memory the object takes up. If an application deletes (say) a private object of size S, it might be reasonable to assume that the *ulFreePrivateMemory* field of the token’s **CK\_TOKEN\_INFO** structure increases by approximately S.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_INFORMATION\_SENSITIVE, CKR\_OBJECT\_HANDLE\_INVALID, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hObject;

CK\_OBJECT\_CLASS dataClass = CKO\_DATA;

CK\_UTF8CHAR application[] = {“My Application”};

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

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

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

{CKA\_APPLICATION, application, sizeof(application)-1},

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

};

CK\_ULONG ulSize;

CK\_RV rv;

.

.

rv = C\_CreateObject(hSession, template, 4, &hObject);

if (rv == CKR\_OK) {

rv = C\_GetObjectSize(hSession, hObject, &ulSize);

if (rv != CKR\_INFORMATION\_SENSITIVE) {

.

.

}

rv = C\_DestroyObject(hSession, hObject);

.

.

}

### C\_GetAttributeValue

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetAttributeValue)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_OBJECT\_HANDLE hObject,  
 CK\_ATTRIBUTE\_PTR pTemplate,  
 CK\_ULONG ulCount  
);

**C\_GetAttributeValue** obtains the value of one or more attributes of an object. *hSession* is the session’s handle; *hObject* is the object’s handle; *pTemplate* points to a template that specifies which attribute values are to be obtained, and receives the attribute values; *ulCount* is the number of attributes in the template.

For each (*type*, *pValue*, *ulValueLen*) triple in the template, **C\_GetAttributeValue** performs the following algorithm:

1. If the specified attribute (i.e., the attribute specified by the type field) for the object cannot be revealed because the object is sensitive or unextractable, then the ulValueLen field in that triple is modified to hold the value CK\_UNAVAILABLE\_INFORMATION.
2. Otherwise, if the specified value for the object is invalid (the object does not possess such an attribute), then the ulValueLen field in that triple is modified to hold the value CK\_UNAVAILABLE\_INFORMATION.
3. Otherwise, if the *pValue* field has the value NULL\_PTR, then the *ulValueLen* field is modified to hold the exact length of the specified attribute for the object.
4. Otherwise, if the length specified in *ulValueLen* is large enough to hold the value of the specified attribute for the object, then that attribute is copied into the buffer located at *pValue*, and the *ulValueLen* field is modified to hold the exact length of the attribute.
5. Otherwise, the ulValueLen field is modified to hold the value CK\_UNAVAILABLE\_INFORMATION.

If case 1 applies to any of the requested attributes, then the call should return the value CKR\_ATTRIBUTE\_SENSITIVE. If case 2 applies to any of the requested attributes, then the call should return the value CKR\_ATTRIBUTE\_TYPE\_INVALID. If case 5 applies to any of the requested attributes, then the call should return the value CKR\_BUFFER\_TOO\_SMALL. As usual, if more than one of these error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the requested attributes will CKR\_OK be returned.

In the special case of an attribute whose value is an array of attributes, for example CKA\_WRAP\_TEMPLATE, where it is passed in with pValue not NULL, the length specified in ulValueLen MUST be large enough to hold all attributes in the array. If the pValue of elements within the array is NULL\_PTR then the ulValueLen of elements within the array will be set to the required length. If the pValue of elements within the array is not NULL\_PTR, then the ulValueLen element of attributes within the array MUST reflect the space that the corresponding pValue points to, and pValue is filled in if there is sufficient room. Therefore it is important to initialize the contents of a buffer before calling C\_GetAttributeValue to get such an array value. Note that the type element of attributes within the array MUST be ignored on input and MUST be set on output. If any ulValueLen within the array isn't large enough, it will be set to CK\_UNAVAILABLE\_INFORMATION and the function will return CKR\_BUFFER\_TOO\_SMALL, as it does if an attribute in the pTemplate argument has ulValueLen too small. Note that any attribute whose value is an array of attributes is identifiable by virtue of the attribute type having the CKF\_ARRAY\_ATTRIBUTE bit set.

Note that the error codes CKR\_ATTRIBUTE\_SENSITIVE, CKR\_ATTRIBUTE\_TYPE\_INVALID, and CKR\_BUFFER\_TOO\_SMALL do not denote true errors for **C\_GetAttributeValue**. If a call to **C\_GetAttributeValue** returns any of these three values, then the call MUST nonetheless have processed *every* attribute in the template supplied to **C\_GetAttributeValue**. Each attribute in the template whose value *can be* returned by the call to **C\_GetAttributeValue** *will be* returned by the call to **C\_GetAttributeValue**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_SENSITIVE, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OBJECT\_HANDLE\_INVALID, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hObject;

CK\_BYTE\_PTR pModulus, pExponent;

CK\_ATTRIBUTE template[] = {

{CKA\_MODULUS, NULL\_PTR, 0},

{CKA\_PUBLIC\_EXPONENT, NULL\_PTR, 0}

};

CK\_RV rv;

.

.

rv = C\_GetAttributeValue(hSession, hObject, template, 2);

if (rv == CKR\_OK) {

pModulus = (CK\_BYTE\_PTR) malloc(template[0].ulValueLen);

template[0].pValue = pModulus;

/\* template[0].ulValueLen was set by C\_GetAttributeValue \*/

pExponent = (CK\_BYTE\_PTR) malloc(template[1].ulValueLen);

template[1].pValue = pExponent;

/\* template[1].ulValueLen was set by C\_GetAttributeValue \*/

rv = C\_GetAttributeValue(hSession, hObject, template, 2);

if (rv == CKR\_OK) {

.

.

}

free(pModulus);

free(pExponent);

}

### C\_SetAttributeValue

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SetAttributeValue)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_OBJECT\_HANDLE hObject,  
 CK\_ATTRIBUTE\_PTR pTemplate,  
 CK\_ULONG ulCount  
);

**C\_SetAttributeValue** modifies the value of one or more attributes of an object. *hSession* is the session’s handle; *hObject* is the object’s handle; *pTemplate* points to a template that specifies which attribute values are to be modified and their new values; *ulCount* is the number of attributes in the template.

Certain objects may not be modified. Calling C\_SetAttributeValue on such objects will result in the CKR\_ACTION\_PROHIBITED error code. An application can consult the object's CKA\_MODIFIABLE attribute to determine if an object may be modified or not.

Only session objects can be modified during a read-only session.

The template may specify new values for any attributes of the object that can be modified. If the template specifies a value of an attribute which is incompatible with other existing attributes of the object, the call fails with the return code CKR\_TEMPLATE\_INCONSISTENT.

Not all attributes can be modified; see Section 4.1.2 for more details.

Return values: CKR\_ACTION\_PROHIBITED, CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_READ\_ONLY, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OBJECT\_HANDLE\_INVALID, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCONSISTENT, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hObject;

CK\_UTF8CHAR label[] = {“New label”};

CK\_ATTRIBUTE template[] = {

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

};

CK\_RV rv;

.

.

rv = C\_SetAttributeValue(hSession, hObject, template, 1);

if (rv == CKR\_OK) {

.

.

}

### C\_FindObjectsInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_FindObjectsInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_ATTRIBUTE\_PTR pTemplate,  
 CK\_ULONG ulCount  
);

**C\_FindObjectsInit** initializes a search for token and session objects that match a template. *hSession* is the session’s handle; *pTemplate* points to a search template that specifies the attribute values to match; *ulCount* is the number of attributes in the search template. The matching criterion is an exact byte-for-byte match with all attributes in the template. To find all objects, set *ulCount* to 0.

After calling **C\_FindObjectsInit**, the application may call **C\_FindObjects** one or more times to obtain handles for objects matching the template, and then eventually call **C\_FindObjectsFinal** to finish the active search operation. At most one search operation may be active at a given time in a given session.

The object search operation will only find objects that the session can view. For example, an object search in an “R/W Public Session” will not find any private objects (even if one of the attributes in the search template specifies that the search is for private objects).

If a search operation is active, and objects are created or destroyed which fit the search template for the active search operation, then those objects may or may not be found by the search operation. Note that this means that, under these circumstances, the search operation may return invalid object handles.

Even though **C\_FindObjectsInit** can return the values CKR\_ATTRIBUTE\_TYPE\_INVALID and CKR\_ATTRIBUTE\_VALUE\_INVALID, it is not required to. For example, if it is given a search template with nonexistent attributes in it, it can return CKR\_ATTRIBUTE\_TYPE\_INVALID, or it can initialize a search operation which will match no objects and return CKR\_OK.

If the CKA\_UNIQUE\_ID attribute is present in the search template, either zero or one objects will be found, since at most one object can have any particular CKA\_UNIQUE\_ID value.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example: see **C\_FindObjectsFinal**.

### C\_FindObjects

CK\_DECLARE\_FUNCTION(CK\_RV, C\_FindObjects)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_OBJECT\_HANDLE\_PTR phObject,  
 CK\_ULONG ulMaxObjectCount,  
 CK\_ULONG\_PTR pulObjectCount  
);

**C\_FindObjects** continues a search for token and session objects that match a template, obtaining additional object handles. *hSession* is the session’s handle; *phObject* points to the location that receives the list (array) of additional object handles; *ulMaxObjectCount* is the maximum number of object handles to be returned; *pulObjectCount* points to the location that receives the actual number of object handles returned.

If there are no more objects matching the template, then the location that *pulObjectCount* points to receives the value 0.

The search MUST have been initialized with **C\_FindObjectsInit**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example: see **C\_FindObjectsFinal**.

### C\_FindObjectsFinal

CK\_DECLARE\_FUNCTION(CK\_RV, C\_FindObjectsFinal)(  
 CK\_SESSION\_HANDLE hSession  
);

**C\_FindObjectsFinal** terminates a search for token and session objects. *hSession* is the session’s handle.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hObject;

CK\_ULONG ulObjectCount;

CK\_RV rv;

.

.

rv = C\_FindObjectsInit(hSession, NULL\_PTR, 0);

assert(rv == CKR\_OK);

while (1) {

rv = C\_FindObjects(hSession, &hObject, 1, &ulObjectCount);

if (rv != CKR\_OK || ulObjectCount == 0)

break;

.

.

}

rv = C\_FindObjectsFinal(hSession);

assert(rv == CKR\_OK);

## Encryption functions

Cryptoki provides the following functions for encrypting data:

### C\_EncryptInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_EncryptInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hKey  
);

**C\_EncryptInit** initializes an encryption operation. *hSession* is the session’s handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key.

The **CKA\_ENCRYPT** attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK\_TRUE.

After calling **C\_EncryptInit**, the application can either call **C\_Encrypt** to encrypt data in a single part; or call **C\_EncryptUpdate** zero or more times, followed by **C\_EncryptFinal**, to encrypt data in multiple parts. The encryption operation is active until the application uses a call to **C\_Encrypt** or **C\_EncryptFinal** *to actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application MUST call **C\_EncryptInit** again.

**C\_EncryptInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active encryption operation. If an active operation operations cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be returned.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

Example: see **C\_EncryptFinal**.

### C\_Encrypt

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Encrypt)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pData,  
 CK\_ULONG ulDataLen,  
 CK\_BYTE\_PTR pEncryptedData,  
 CK\_ULONG\_PTR pulEncryptedDataLen  
);

**C\_Encrypt** encrypts single-part data. *hSession* is the session’s handle; *pData* points to the data; *ulDataLen* is the length in bytes of the data; *pEncryptedData* points to the location that receives the encrypted data; *pulEncryptedDataLen* points to the location that holds the length in bytes of the encrypted data.

**C\_Encrypt** uses the convention described in Section 5.2 on producing output.

The encryption operation MUST have been initialized with **C\_EncryptInit**. A call to **C\_Encrypt** always terminates the active encryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the ciphertext.

**C\_Encrypt**cannot be used to terminate a multi-part operation, and MUST be called after **C\_EncryptInit** without intervening **C\_EncryptUpdate** calls.

For some encryption mechanisms, the input plaintext data has certain length constraints (either because the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism’s input data MUST consist of an integral number of blocks). If these constraints are not satisfied, then **C\_Encrypt** will fail with return code CKR\_DATA\_LEN\_RANGE.

The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pData* and *pEncryptedData* point to the same location.

For most mechanisms, **C\_Encrypt** is equivalent to a sequence of **C\_EncryptUpdate** operations followed by **C\_EncryptFinal**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example: see **C\_EncryptFinal** for an example of similar functions.

### C\_EncryptUpdate

CK\_DECLARE\_FUNCTION(CK\_RV, C\_EncryptUpdate)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pPart,  
 CK\_ULONG ulPartLen,  
 CK\_BYTE\_PTR pEncryptedPart,  
 CK\_ULONG\_PTR pulEncryptedPartLen  
);

**C\_EncryptUpdate** continues a multiple-part encryption operation, processing another data part. *hSession* is the session’s handle; *pPart* points to the data part; *ulPartLen* is the length of the data part; *pEncryptedPart* points to the location that receives the encrypted data part; *pulEncryptedPartLen* points to the location that holds the length in bytes of the encrypted data part.

**C\_EncryptUpdate** uses the convention described in Section 5.2 on producing output.

The encryption operation MUST have been initialized with **C\_EncryptInit**. This function may be called any number of times in succession. A call to **C\_EncryptUpdate** which results in an error other than CKR\_BUFFER\_TOO\_SMALL terminates the current encryption operation.

The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pPart* and *pEncryptedPart* point to the same location.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example: see **C\_EncryptFinal.**

### C\_EncryptFinal

CK\_DECLARE\_FUNCTION(CK\_RV, C\_EncryptFinal)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pLastEncryptedPart,  
 CK\_ULONG\_PTR pulLastEncryptedPartLen  
);

**C\_EncryptFinal** finishes a multiple-part encryption operation. *hSession* is the session’s handle; *pLastEncryptedPart* points to the location that receives the last encrypted data part, if any; *pulLastEncryptedPartLen* points to the location that holds the length of the last encrypted data part.

**C\_EncryptFinal** uses the convention described in Section 5.2 on producing output.

The encryption operation MUST have been initialized with **C\_EncryptInit**. A call to **C\_EncryptFinal** always terminates the active encryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the ciphertext.

For some multi-part encryption mechanisms, the input plaintext data has certain length constraints, because the mechanism’s input data MUST consist of an integral number of blocks. If these constraints are not satisfied, then **C\_EncryptFinal** will fail with return code CKR\_DATA\_LEN\_RANGE.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

#define PLAINTEXT\_BUF\_SZ 200

#define CIPHERTEXT\_BUF\_SZ 256

CK\_ULONG firstPieceLen, secondPieceLen;

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_BYTE iv[8];

CK\_MECHANISM mechanism = {

CKM\_DES\_CBC\_PAD, iv, sizeof(iv)

};

CK\_BYTE data[PLAINTEXT\_BUF\_SZ];

CK\_BYTE encryptedData[CIPHERTEXT\_BUF\_SZ];

CK\_ULONG ulEncryptedData1Len;

CK\_ULONG ulEncryptedData2Len;

CK\_ULONG ulEncryptedData3Len;

CK\_RV rv;

.

.

firstPieceLen = 90;

secondPieceLen = PLAINTEXT\_BUF\_SZ-firstPieceLen;

rv = C\_EncryptInit(hSession, &mechanism, hKey);

if (rv == CKR\_OK) {

/\* Encrypt first piece \*/

ulEncryptedData1Len = sizeof(encryptedData);

rv = C\_EncryptUpdate(

hSession,

&data[0], firstPieceLen,

&encryptedData[0], &ulEncryptedData1Len);

if (rv != CKR\_OK) {

.

.

}

/\* Encrypt second piece \*/

ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;

rv = C\_EncryptUpdate(

hSession,

&data[firstPieceLen], secondPieceLen,

&encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);

if (rv != CKR\_OK) {

.

.

}

/\* Get last little encrypted bit \*/

ulEncryptedData3Len =

sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;

rv = C\_EncryptFinal(

hSession,

&encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],

&ulEncryptedData3Len);

if (rv != CKR\_OK) {

.

.

}

}

## Message-based encryption functions

Message-based encryption refers to the process of encrypting multiple messages using the same encryption mechanism and encryption key. The encryption mechanism can be either an authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.

Cryptoki provides the following functions for message-based encryption:

### C\_MessageEncryptInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_MessageEncryptInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hKey  
);

**C\_MessageEncryptInit** prepares a session for one or more encryption operations that use the same encryption mechanism and encryption key. hSession is the session’s handle; pMechanism points to the encryption mechanism; hKey is the handle of the encryption key.

The CKA\_ENCRYPT attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK\_TRUE.

After calling **C\_MessageEncryptInit**, the application can either call **C\_EncryptMessage** to encrypt a message in a single part, or call **C\_EncryptMessageBegin**, followed by **C\_EncryptMessageNext** one or more times, to encrypt a message in multiple parts. This may be repeated several times. The message-based encryption process is active until the application calls **C\_MessageEncryptFinal** to finish the message-based encryption process.

**C\_MessageEncryptInit** can be called with *pMechanism* set to NULL\_PTR to terminate a message-based encryption process. If a multi-part message encryption operation is active, it will also be terminated. If an active operation has been initialized and it cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be returned.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

### C\_EncryptMessage

CK\_DECLARE\_FUNCTION(CK\_RV, C\_EncryptMessage)(  
 CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pAssociatedData,

CK\_ULONG ulAssociatedDataLen,

CK\_BYTE\_PTR pPlaintext,

CK\_ULONG ulPlaintextLen,

CK\_BYTE\_PTR pCiphertext,

CK\_ULONG\_PTR pulCiphertextLen

);

**C\_EncryptMessage** encrypts a message in a single part. *hSession* is the session’s handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism; *pPlaintext* points to the plaintext data; *ulPlaintextLen* is the length in bytes of the plaintext data; *pCiphertext* points to the location that receives the encrypted data; *pulCiphertextLen* points to the location that holds the length in bytes of the encrypted data.

Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter passed to **C\_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV generator will be output to the *pParameter* buffer.

If the encryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be set to (NULL, 0).

**C\_EncryptMessage** uses the convention described in Section 5.2 on producing output.

The message-based encryption process MUST have been initialized with **C\_MessageEncryptInit**. A call to **C\_EncryptMessage** begins and terminates a message encryption operation.

**C\_EncryptMessage** cannot be called in the middle of a multi-part message encryption operation.

For some encryption mechanisms, the input plaintext data has certain length constraints (either because the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism’s input data MUST consist of an integral number of blocks). If these constraints are not satisfied, then **C\_EncryptMessage** will fail with return code CKR\_DATA\_LEN\_RANGE. The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintext* and *pCiphertext* point to the same location.

For most mechanisms, **C\_EncryptMessage** is equivalent to **C\_EncryptMessageBegin** followed by a sequence of **C\_EncryptMessageNext** operations.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

### C\_EncryptMessageBegin

CK\_DECLARE\_FUNCTION(CK\_RV, C\_EncryptMessageBegin)(

CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pAssociatedData,

CK\_ULONG ulAssociatedDataLen

);

**C\_EncryptMessageBegin** begins a multiple-part message encryption operation. *hSession* is the session’s handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message encryption operation*; pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism.

Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter passed to **C\_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV generator will be output to the *pParameter* buffer.

If the mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be set to (NULL, 0).

After calling **C\_EncryptMessageBegin**, the application should call **C\_EncryptMessageNext** one or more times to encrypt the message in multiple parts. The message encryption operation is active until the application uses a call to **C\_EncryptMessageNext** with flags=CKF\_END\_OF\_MESSAGE to actually obtain the final piece of ciphertext. To process additional messages (in single or multiple parts), the application MUST call **C\_EncryptMessage** or **C\_EncryptMessageBegin** again.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### C\_EncryptMessageNext

CK\_DECLARE\_FUNCTION(CK\_RV, C\_EncryptMessageNext)(

CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pPlaintextPart,

CK\_ULONG ulPlaintextPartLen,

CK\_BYTE\_PTR pCiphertextPart,

CK\_ULONG\_PTR pulCiphertextPartLen,

CK\_FLAGS flags

);

**C\_EncryptMessageNext** continues a multiple-part message encryption operation, processing another message part. *hSession* is the session’s handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message encryption operation; *pPlaintextPart* points to the plaintext message part; *ulPlaintextPartLen* is the length of the plaintext message part; *pCiphertextPart* points to the location that receives the encrypted message part; *pulCiphertextPartLen* points to the location that holds the length in bytes of the encrypted message part;flags is set to 0 if there is more plaintext data to follow, or set to CKF\_END\_OF\_MESSAGE if this is the last plaintext part.

Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter passed to **C\_EncryptMessageNext**, *pParameter* may be either an input or an output parameter. For example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV generator will be output to the *pParameter* buffer.

**C\_EncryptMessageNext** uses the convention described in Section 5.2 on producing output.

The message encryption operation MUST have been started with **C\_EncryptMessageBegin**. This function may be called any number of times in succession. A call to C\_EncryptMessageNext with flags=0 which results in an error other than CKR\_BUFFER\_TOO\_SMALL terminates the current message encryption operation. A call to **C\_EncryptMessageNext** with flags=CKF\_END\_OF\_MESSAGE always terminates the active message encryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (i.e., one which returns **CKR\_OK**) to determine the length of the buffer needed to hold the ciphertext.

Although the last **C\_EncryptMessageNext** call ends the encryption of a message, it does not finish the message-based encryption process. Additional **C\_EncryptMessage** or **C\_EncryptMessageBegin** and **C\_EncryptMessageNext** calls may be made on the session.

The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintextPart* and *pCiphertextPart* point to the same location.

For some multi-part encryption mechanisms, the input plaintext data has certain length constraints, because the mechanism’s input data MUST consist of an integral number of blocks. If these constraints are not satisfied when the final message part is supplied (i.e., with flags=CKF\_END\_OF\_MESSAGE), then **C\_EncryptMessageNext** will fail with return code CKR\_DATA\_LEN\_RANGE.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

### C\_MessageEncryptFinal

CK\_DECLARE\_FUNCTION(CK\_RV, C\_MessageEncryptFinal)(

CK\_SESSION\_HANDLE hSession

);

**C\_MessageEncryptFinal** finishes a message-based encryption process. hSession is the session’s handle.

The message-based encryption process MUST have been initialized with **C\_MessageEncryptInit**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR,

CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

#define PLAINTEXT\_BUF\_SZ 200

#define AUTH\_BUF\_SZ 100

#define CIPHERTEXT\_BUF\_SZ 256

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_BYTE iv[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

CK\_BYTE tag[16];

CK\_GCM\_MESSAGE\_PARAMS gcmParams = {

iv,

sizeof(iv) \* 8,

0,

CKG\_NO\_GENERATE,

tag,

sizeof(tag) \* 8

};

CK\_MECHANISM mechanism = {

CKM\_AES\_GCM, &gcmParams, sizeof(gcmParams)

};

CK\_BYTE data[2][PLAINTEXT\_BUF\_SZ];

CK\_BYTE auth[2][AUTH\_BUF\_SZ];

CK\_BYTE encryptedData[2][CIPHERTEXT\_BUF\_SZ];

CK\_ULONG ulEncryptedDataLen, ulFirstEncryptedDataLen;

CK\_ULONG firstPieceLen = PLAINTEXT\_BUF\_SZ / 2;

/\* error handling is omitted for better readability \*/

.

.

C\_MessageEncryptInit(hSession, &mechanism, hKey);

/\* encrypt message en bloc with given IV \*/

ulEncryptedDataLen = sizeof(encryptedData[0]);

C\_EncryptMessage(hSession,

&gcmParams, sizeof(gcmParams),

&auth[0][0], sizeof(auth[0]),

&data[0][0], sizeof(data[0]),

&encryptedData[0][0], &ulEncryptedDataLen);

/\* iv and tag are set now for message \*/

/\* encrypt message in two steps with generated IV \*/

gcmParams.ivGenerator = CKG\_GENERATE;

C\_EncryptMessageBegin(hSession,

&gcmParams, sizeof(gcmParams),

&auth[1][0], sizeof(auth[1])

);

/\* encrypt first piece \*/

ulFirstEncryptedDataLen = sizeof(encryptedData[1]);

C\_EncryptMessageNext(hSession,

&gcmParams, sizeof(gcmParams),

&data[1][0], firstPieceLen,

&encryptedData[1][0], &ulFirstEncryptedDataLen,

0

);

/\* encrypt second piece \*/

ulEncryptedDataLen = sizeof(encryptedData[1]) - ulFirstEncryptedDataLen;

C\_EncryptMessageNext(hSession,

&gcmParams, sizeof(gcmParams),

&data[1][firstPieceLen], sizeof(data[1])-firstPieceLen,

&encryptedData[1][ulFirstEncryptedDataLen], &ulEncryptedDataLen,

CKF\_END\_OF\_MESSAGE

);

/\* tag is set now for message \*/

/\* finalize \*/

C\_MessageEncryptFinal(hSession);

## Decryption functions

Cryptoki provides the following functions for decrypting data:

### C\_DecryptInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DecryptInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hKey  
);

**C\_DecryptInit** initializes a decryption operation. *hSession* is the session’s handle; *pMechanism* points to the decryption mechanism; *hKey* is the handle of the decryption key.

The **CKA\_DECRYPT** attribute of the decryption key, which indicates whether the key supports decryption, MUST be CK\_TRUE.

After calling **C\_DecryptInit**, the application can either call **C\_Decrypt** to decrypt data in a single part; or call **C\_DecryptUpdate** zero or more times, followed by **C\_DecryptFinal**, to decrypt data in multiple parts. The decryption operation is active until the application uses a call to **C\_Decrypt** or **C\_DecryptFinal** *to actually obtain* the final piece of plaintext. To process additional data (in single or multiple parts), the application MUST call **C\_DecryptInit** again.

**C\_DecryptInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active decryption operation. If an active operation cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be returned.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

Example: see **C\_DecryptFinal**.

### C\_Decrypt

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Decrypt)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pEncryptedData,  
 CK\_ULONG ulEncryptedDataLen,  
 CK\_BYTE\_PTR pData,  
 CK\_ULONG\_PTR pulDataLen  
);

**C\_Decrypt** decrypts encrypted data in a single part. *hSession* is the session’s handle; *pEncryptedData* points to the encrypted data; *ulEncryptedDataLen* is the length of the encrypted data; *pData* points to the location that receives the recovered data; *pulDataLen* points to the location that holds the length of the recovered data.

**C\_Decrypt** uses the convention described in Section 5.2 on producing output.

The decryption operation MUST have been initialized with **C\_DecryptInit**. A call to **C\_Decrypt** always terminates the active decryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the plaintext.

**C\_Decrypt**cannot be used to terminate a multi-part operation, and MUST be called after **C\_DecryptInit** without intervening **C\_DecryptUpdate** calls.

The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedData* and *pData* point to the same location.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either CKR\_ENCRYPTED\_DATA\_INVALID or CKR\_ENCRYPTED\_DATA\_LEN\_RANGE may be returned.

For most mechanisms, **C\_Decrypt** is equivalent to a sequence of **C\_DecryptUpdate** operations followed by **C\_DecryptFinal**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID, CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

Example: see **C\_DecryptFinal** for an example of similar functions.

### C\_DecryptUpdate

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DecryptUpdate)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pEncryptedPart,  
 CK\_ULONG ulEncryptedPartLen,  
 CK\_BYTE\_PTR pPart,  
 CK\_ULONG\_PTR pulPartLen  
);

**C\_DecryptUpdate** continues a multiple-part decryption operation, processing another encrypted data part. *hSession* is the session’s handle; *pEncryptedPart* points to the encrypted data part; *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

**C\_DecryptUpdate** uses the convention described in Section 5.2 on producing output.

The decryption operation MUST have been initialized with **C\_DecryptInit**. This function may be called any number of times in succession. A call to **C\_DecryptUpdate** which results in an error other than CKR\_BUFFER\_TOO\_SMALL terminates the current decryption operation.

The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedPart* and *pPart* point to the same location.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID, CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

Example: See **C\_DecryptFinal**.

### C\_DecryptFinal

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DecryptFinal)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pLastPart,  
 CK\_ULONG\_PTR pulLastPartLen  
);

**C\_DecryptFinal** finishes a multiple-part decryption operation. *hSession* is the session’s handle; *pLastPart* points to the location that receives the last recovered data part, if any; *pulLastPartLen* points to the location that holds the length of the last recovered data part.

**C\_DecryptFinal** uses the convention described in Section 5.2 on producing output.

The decryption operation MUST have been initialized with **C\_DecryptInit**. A call to **C\_DecryptFinal** always terminates the active decryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the plaintext.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either CKR\_ENCRYPTED\_DATA\_INVALID or CKR\_ENCRYPTED\_DATA\_LEN\_RANGE may be returned.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID, CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

#define CIPHERTEXT\_BUF\_SZ 256

#define PLAINTEXT\_BUF\_SZ 256

CK\_ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_BYTE iv[8];

CK\_MECHANISM mechanism = {

CKM\_DES\_CBC\_PAD, iv, sizeof(iv)

};

CK\_BYTE data[PLAINTEXT\_BUF\_SZ];

CK\_BYTE encryptedData[CIPHERTEXT\_BUF\_SZ];

CK\_ULONG ulData1Len, ulData2Len, ulData3Len;

CK\_RV rv;

.

.

firstEncryptedPieceLen = 90;

secondEncryptedPieceLen = CIPHERTEXT\_BUF\_SZ-firstEncryptedPieceLen;

rv = C\_DecryptInit(hSession, &mechanism, hKey);

if (rv == CKR\_OK) {

/\* Decrypt first piece \*/

ulData1Len = sizeof(data);

rv = C\_DecryptUpdate(

hSession,

&encryptedData[0], firstEncryptedPieceLen,

&data[0], &ulData1Len);

if (rv != CKR\_OK) {

.

.

}

/\* Decrypt second piece \*/

ulData2Len = sizeof(data)-ulData1Len;

rv = C\_DecryptUpdate(

hSession,

&encryptedData[firstEncryptedPieceLen],

secondEncryptedPieceLen,

&data[ulData1Len], &ulData2Len);

if (rv != CKR\_OK) {

.

.

}

/\* Get last little decrypted bit \*/

ulData3Len = sizeof(data)-ulData1Len-ulData2Len;

rv = C\_DecryptFinal(

hSession,

&data[ulData1Len+ulData2Len], &ulData3Len);

if (rv != CKR\_OK) {

.

.

}

}

## Message-based decryption functions

Message-based decryption refers to the process of decrypting multiple encrypted messages using the same decryption mechanism and decryption key. The decryption mechanism can be either an authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.

Cryptoki provides the following functions for message-based decryption.

### C\_MessageDecryptInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_MessageDecryptInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hKey  
);

**C\_MessageDecryptInit** initializes a message-based decryption process, preparing a session for one or more decryption operations that use the same decryption mechanism and decryption key. *hSession* is the session’s handle; *pMechanism* points to the decryption mechanism; *hKey* is the handle of the decryption key.

The CKA\_DECRYPT attribute of the decryption key, which indicates whether the key supports decryption, MUST be CK\_TRUE.

After calling **C\_MessageDecryptInit**, the application can either call **C\_DecryptMessage** to decrypt an encrypted message in a single part; or call **C\_DecryptMessageBegin**, followed by **C\_DecryptMessageNext** one or more times, to decrypt an encrypted message in multiple parts. This may be repeated several times. The message-based decryption process is active until the application uses a call to **C\_MessageDecryptFinal** to finish the message-based decryption process.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

### C\_DecryptMessage

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DecryptMessage)(  
 CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pAssociatedData,

CK\_ULONG ulAssociatedDataLen,

CK\_BYTE\_PTR pCiphertext,

CK\_ULONG ulCiphertextLen,

CK\_BYTE\_PTR pPlaintext,

CK\_ULONG\_PTR pulPlaintextLen

);

**C\_DecryptMessage** decrypts an encrypted message in a single part. *hSession* is the session’s handle;*pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism; *pCiphertext* points to the encrypted message; *ulCiphertextLen* is the length of the encrypted message; *pPlaintext* points to the location that receives the recovered message; *pulPlaintextLen* points to the location that holds the length of the recovered message.

Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of **C\_EncryptMessage**, *pParameter* is always an input parameter.

If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be set to (NULL, 0).

**C\_DecryptMessage** uses the convention described in Section 5.2 on producing output.

The message-based decryption process MUST have been initialized with **C\_MessageDecryptInit**. A call to **C\_DecryptMessage** begins and terminates a message decryption operation.

**C\_DecryptMessage** cannot be called in the middle of a multi-part message decryption operation.

The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertext* and *pPlaintext* point to the same location.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either CKR\_ENCRYPTED\_DATA\_INVALID or CKR\_ENCRYPTED\_DATA\_LEN\_RANGE may be returned.

If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or ciphertext cannot be verified, then CKR\_AEAD\_DECRYPT\_FAILED is returned.

For most mechanisms, **C\_DecryptMessage** is equivalent to **C\_DecryptMessageBegin** followed by a sequence of **C\_DecryptMessageNext** operations.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID, CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_AEAD\_DECRYPT\_FAILED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

### C\_DecryptMessageBegin

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DecryptMessageBegin)(  
 CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pAssociatedData,

CK\_ULONG ulAssociatedDataLen

);

**C\_DecryptMessageBegin** begins a multiple-part message decryption operation. *hSession* is the session’s handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism.

Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of **C\_EncryptMessageBegin**, *pParameter* is always an input parameter.

If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be set to (NULL, 0).

After calling **C\_DecryptMessageBegin**, the application should call **C\_DecryptMessageNext** one or more times to decrypt the encrypted message in multiple parts. The message decryption operation is active until the application uses a call to **C\_DecryptMessageNext** with flags=CKF\_END\_OF\_MESSAGE to actually obtain the final piece of plaintext. To process additional encrypted messages (in single or multiple parts), the application MUST call **C\_DecryptMessage** or **C\_DecryptMessageBegin** again.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### C\_DecryptMessageNext

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DecryptMessageNext)(  
 CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pCiphertextPart,

CK\_ULONG ulCiphertextPartLen,

CK\_BYTE\_PTR pPlaintextPart,

CK\_ULONG\_PTR pulPlaintextPartLen,

CK\_FLAGS flags

);

**C\_DecryptMessageNext** continues a multiple-part message decryption operation, processing another encrypted message part. *hSession* is the session’s handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption operation; *pCiphertextPart* points to the encrypted message part; *ulCiphertextPartLen* is the length of the encrypted message part; *pPlaintextPart* points to the location that receives the recovered message part; *pulPlaintextPartLen* points to the location that holds the length of the recovered message part;flags is set to 0 if there is more ciphertext data to follow, or set to CKF\_END\_OF\_MESSAGE if this is the last ciphertext part.

Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of **C\_EncryptMessageNext**, *pParameter* is always an input parameter.

**C\_DecryptMessageNext** uses the convention described in Section 5.2 on producing output.

The message decryption operation MUST have been started with **C\_DecryptMessageBegin.** This function may be called any number of times in succession. A call to **C\_DecryptMessageNext** with flags=0 which results in an error other than CKR\_BUFFER\_TOO\_SMALL terminates the current message decryption operation. A call to **C\_DecryptMessageNext** with flags=CKF\_END\_OF\_MESSAGE always terminates the active message decryption operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (i.e., one which returns CKR\_OK) to determine the length of the buffer needed to hold the plaintext.

The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertextPart* and *pPlaintextPart* point to the same location.

Although the last **C\_DecryptMessageNext** call ends the decryption of a message, it does not finish the message-based decryption process. Additional **C\_DecryptMessage** or **C\_DecryptMessageBegin** and **C\_DecryptMessageNext c**alls may be made on the session.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either CKR\_ENCRYPTED\_DATA\_INVALID or CKR\_ENCRYPTED\_DATA\_LEN\_RANGE may be returned by the last **C\_DecryptMessageNext** call.

If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or ciphertext cannot be verified, then CKR\_AEAD\_DECRYPT\_FAILED is returned by the last **C\_DecryptMessageNext** call.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID, CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_AEAD\_DECRYPT\_FAILED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### C\_MessageDecryptFinal

CK\_DECLARE\_FUNCTION(CK\_RV, C\_MessageDecryptFinal)(  
 CK\_SESSION\_HANDLE hSession   
);

**C\_MessageDecryptFinal** finishes a message-based decryption process. *hSession* is the session’s handle.

The message-based decryption process MUST have been initialized with **C\_MessageDecryptInit.**

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

## Message digesting functions

Cryptoki provides the following functions for digesting data:

### C\_DigestInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DigestInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism  
);

**C\_DigestInit** initializes a message-digesting operation. *hSession* is the session’s handle; *pMechanism* points to the digesting mechanism.

After calling **C\_DigestInit**, the application can either call **C\_Digest** to digest data in a single part; or call **C\_DigestUpdate** zero or more times, followed by **C\_DigestFinal**, to digest data in multiple parts. The message-digesting operation is active until the application uses a call to **C\_Digest** or **C\_DigestFinal** *to actually obtain* the message digest. To process additional data (in single or multiple parts), the application MUST call **C\_DigestInit** again.

**C\_DigestInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active message-digesting operation. If an operation has been initialized and it cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be returned.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

Example: see **C\_DigestFinal**.

### C\_Digest

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Digest)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pData,  
 CK\_ULONG ulDataLen,  
 CK\_BYTE\_PTR pDigest,  
 CK\_ULONG\_PTR pulDigestLen  
);

**C\_Digest** digests data in a single part. *hSession* is the session’s handle, *pData* points to the data; *ulDataLen* is the length of the data; *pDigest* points to the location that receives the message digest; *pulDigestLen* points to the location that holds the length of the message digest.

**C\_Digest** uses the convention described in Section 5.2 on producing output.

The digest operation MUST have been initialized with **C\_DigestInit**. A call to **C\_Digest** always terminates the active digest operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the message digest.

**C\_Digest**cannot be used to terminate a multi-part operation, and MUST be called after **C\_DigestInit** without intervening **C\_DigestUpdate** calls.

The input data and digest output can be in the same place, *i.e.*, it is OK if *pData* and *pDigest* point to the same location.

**C\_Digest** is equivalent to a sequence of **C\_DigestUpdate** operations followed by **C\_DigestFinal**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example: see **C\_DigestFinal** for an example of similar functions.

### C\_DigestUpdate

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DigestUpdate)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pPart,  
 CK\_ULONG ulPartLen  
);

**C\_DigestUpdate** continues a multiple-part message-digesting operation, processing another data part. *hSession* is the session’s handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

The message-digesting operation MUST have been initialized with **C\_DigestInit**. Calls to this function and **C\_DigestKey** may be interspersed any number of times in any order. A call to **C\_DigestUpdate** which results in an error terminates the current digest operation.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example: see **C\_DigestFinal**.

### C\_DigestKey

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DigestKey)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_OBJECT\_HANDLE hKey  
);

**C\_DigestKey** continues a multiple-part message-digesting operation by digesting the value of a secret key. *hSession* is the session’s handle; *hKey* is the handle of the secret key to be digested.

The message-digesting operation MUST have been initialized with **C\_DigestInit**. Calls to this function and **C\_DigestUpdate** may be interspersed any number of times in any order.

If the value of the supplied key cannot be digested purely for some reason related to its length, **C\_DigestKey** should return the error code CKR\_KEY\_SIZE\_RANGE.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_INDIGESTIBLE, CKR\_KEY\_SIZE\_RANGE, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example: see **C\_DigestFinal**.

### C\_DigestFinal

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DigestFinal)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pDigest,  
 CK\_ULONG\_PTR pulDigestLen  
);

**C\_DigestFinal** finishes a multiple-part message-digesting operation, returning the message digest. *hSession* is the session’s handle; *pDigest* points to the location that receives the message digest; *pulDigestLen* points to the location that holds the length of the message digest.

**C\_DigestFinal** uses the convention described in Section 5.2 on producing output.

The digest operation MUST have been initialized with **C\_DigestInit**. A call to **C\_DigestFinal** always terminates the active digest operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the message digest.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_MECHANISM mechanism = {

CKM\_MD5, NULL\_PTR, 0

};

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

CK\_BYTE digest[16];

CK\_ULONG ulDigestLen;

CK\_RV rv;

.

.

rv = C\_DigestInit(hSession, &mechanism);

if (rv != CKR\_OK) {

.

.

}

rv = C\_DigestUpdate(hSession, data, sizeof(data));

if (rv != CKR\_OK) {

.

.

}

rv = C\_DigestKey(hSession, hKey);

if (rv != CKR\_OK) {

.

.

}

ulDigestLen = sizeof(digest);

rv = C\_DigestFinal(hSession, digest, &ulDigestLen);

.

.

## Signing and MACing functions

Cryptoki provides the following functions for signing data (for the purposes of Cryptoki, these operations also encompass message authentication codes).

### C\_SignInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SignInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hKey  
);

**C\_SignInit** initializes a signature operation, where the signature is an appendix to the data. *hSession* is the session’s handle; *pMechanism* points to the signature mechanism; *hKey* is the handle of the signature key.

The **CKA\_SIGN** attribute of the signature key, which indicates whether the key supports signatures with appendix, MUST be CK\_TRUE.

After calling **C\_SignInit**, the application can either call **C\_Sign** to sign in a single part; or call **C\_SignUpdate** one or more times, followed by **C\_SignFinal,** to sign data in multiple parts. The signature operation is active until the application uses a call to **C\_Sign** or **C\_SignFinal** *to actually obtain* the signature. To process additional data (in single or multiple parts), the application MUST call **C\_SignInit** again.

**C\_SignInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active signature operation. If an operation has been initialized and it cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be returned.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED,CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

Example: see **C\_SignFinal**.

### C\_Sign

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Sign)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pData,  
 CK\_ULONG ulDataLen,  
 CK\_BYTE\_PTR pSignature,  
 CK\_ULONG\_PTR pulSignatureLen  
);

**C\_Sign** signs data in a single part, where the signature is an appendix to the data. *hSession* is the session’s handle; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

**C\_Sign** uses the convention described in Section 5.2 on producing output.

The signing operation MUST have been initialized with **C\_SignInit**. A call to **C\_Sign** always terminates the active signing operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the signature.

**C\_Sign**cannot be used to terminate a multi-part operation, and MUST be called after **C\_SignInit** without intervening **C\_SignUpdate** calls.

For most mechanisms, **C\_Sign** is equivalent to a sequence of **C\_SignUpdate** operations followed by **C\_SignFinal**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

Example: see **C\_SignFinal** for an example of similar functions.

### C\_SignUpdate

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SignUpdate)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pPart,  
 CK\_ULONG ulPartLen  
);

**C\_SignUpdate** continues a multiple-part signature operation, processing another data part. *hSession* is the session’s handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

The signature operation MUST have been initialized with **C\_SignInit**. This function may be called any number of times in succession. A call to **C\_SignUpdate** which results in an error terminates the current signature operation.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

Example: see **C\_SignFinal**.

### C\_SignFinal

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SignFinal)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pSignature,  
 CK\_ULONG\_PTR pulSignatureLen  
);

**C\_SignFinal** finishes a multiple-part signature operation, returning the signature. *hSession* is the session’s handle; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

**C\_SignFinal** uses the convention described in Section 5.2 on producing output.

The signing operation MUST have been initialized with **C\_SignInit**. A call to **C\_SignFinal** always terminates the active signing operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the signature.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_MECHANISM mechanism = {

CKM\_DES\_MAC, NULL\_PTR, 0

};

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

CK\_BYTE mac[4];

CK\_ULONG ulMacLen;

CK\_RV rv;

.

.

rv = C\_SignInit(hSession, &mechanism, hKey);

if (rv == CKR\_OK) {

rv = C\_SignUpdate(hSession, data, sizeof(data));

.

.

ulMacLen = sizeof(mac);

rv = C\_SignFinal(hSession, mac, &ulMacLen);

.

.

}

### C\_SignRecoverInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SignRecoverInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hKey  
);

**C\_SignRecoverInit** initializes a signature operation, where the data can be recovered from the signature. *hSession* is the session’s handle; *pMechanism* points to the structure that specifies the signature mechanism; *hKey* is the handle of the signature key.

The **CKA\_SIGN\_RECOVER** attribute of the signature key, which indicates whether the key supports signatures where the data can be recovered from the signature, MUST be CK\_TRUE.

After calling **C\_SignRecoverInit**, the application may call **C\_SignRecover** to sign in a single part. The signature operation is active until the application uses a call to **C\_SignRecover** *to actually obtain* the signature. To process additional data in a single part, the application MUST call **C\_SignRecoverInit** again.

**C\_SignRecoverInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active signature with data recovery operation. If an active operation has been initialized and it cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be returned.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

Example: see **C\_SignRecover**.

### C\_SignRecover

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SignRecover)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pData,  
 CK\_ULONG ulDataLen,  
 CK\_BYTE\_PTR pSignature,  
 CK\_ULONG\_PTR pulSignatureLen  
);

**C\_SignRecover** signs data in a single operation, where the data can be recovered from the signature. *hSession* is the session’s handle; *pData* points to the data; *uLDataLen* is the length of the data; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

**C\_SignRecover** uses the convention described in Section 5.2 on producing output.

The signing operation MUST have been initialized with **C\_SignRecoverInit**. A call to **C\_SignRecover** always terminates the active signing operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the signature.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_MECHANISM mechanism = {

CKM\_RSA\_9796, NULL\_PTR, 0

};

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

CK\_BYTE signature[128];

CK\_ULONG ulSignatureLen;

CK\_RV rv;

.

.

rv = C\_SignRecoverInit(hSession, &mechanism, hKey);

if (rv == CKR\_OK) {

ulSignatureLen = sizeof(signature);

rv = C\_SignRecover(

hSession, data, sizeof(data), signature, &ulSignatureLen);

if (rv == CKR\_OK) {

.

.

}

}

## Message-based signing and MACing functions

Message-based signature refers to the process of signing multiple messages using the same signature mechanism and signature key.

Cryptoki provides the following functions for for signing messages (for the purposes of Cryptoki, these operations also encompass message authentication codes).

### C\_MessageSignInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_MessageSignInit)(

CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hKey

);

**C\_MessageSignInit** initializes a message-based signature process, preparing a session for one or more signature operations (where the signature is an appendix to the data) that use the same signature mechanism and signature key. *hSession* is the session’s handle; *pMechanism* points to the signature mechanism; *hKey* is the handle of the signature key.

The **CKA\_SIGN** attribute of the signature key, which indicates whether the key supports signatures with appendix, MUST be CK\_TRUE.

After calling **C\_MessageSignInit**, the application can either call **C\_SignMessage** to sign a message in a single part; or call **C\_SignMessageBegin**, followed by **C\_SignMessageNext** one or more times, to sign a message in multiple parts. This may be repeated several times. The message-based signature process is active until the application calls **C\_MessageSignFinal** to finish the message-based signature process.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED,CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### C\_SignMessage

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SignMessage)(

CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pData,

CK\_ULONG ulDataLen,

CK\_BYTE\_PTR pSignature,

CK\_ULONG\_PTR pulSignatureLen

);

**C\_SignMessage** signs a message in a single part, where the signature is an appendix to the message. **C\_MessageSignInit** must previously been called on the session. *hSession* is the session’s handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

Depending on the mechanism parameter passed to **C\_MessageSignInit**, *pParameter* may be either an input or an output parameter.

**C\_SignMessage** uses the convention described in Section 5.2 on producing output.

The message-based signing process MUST have been initialized with **C\_MessageSignInit**. A call to **C\_SignMessage** begins and terminates a message signing operation unless it returns CKR\_BUFFER\_TOO\_SMALL to determine the length of the buffer needed to hold the signature, or is a successful call (i.e., one which returns CKR\_OK).

**C\_SignMessage** cannot be called in the middle of a multi-part message signing operation.

**C\_SignMessage** does not finish the message-based signing process. Additional **C\_SignMessage** or **C\_SignMessageBegin** and **C\_SignMessageNext** calls may be made on the session.

For most mechanisms, **C\_SignMessage** is equivalent to **C\_SignMessageBegin** followed by a sequence of **C\_SignMessageNext** operations.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

### C\_SignMessageBegin

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SignMessageBegin)(

CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen

);

**C\_SignMessageBegin** begins a multiple-part message signature operation, where the signature is an appendix to the message. **C\_MessageSignInit** must previously been called on the session. *hSession* is the session’s handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature operation.

Depending on the mechanism parameter passed to **C\_MessageSignInit**, *pParameter* may be either an input or an output parameter.

After calling **C\_SignMessageBegin**, the application should call **C\_SignMessageNext** one or more times to sign the message in multiple parts. The message signature operation is active until the application uses a call to **C\_SignMessageNext** with a non-NULL *pulSignatureLen* to actually obtain the signature. To process additional messages (in single or multiple parts), the application MUST call **C\_SignMessage** or **C\_SignMessageBegin** again.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

### C\_SignMessageNext

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SignMessageNext)(

CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pDataPart,

CK\_ULONG ulDataPartLen,

CK\_BYTE\_PTR pSignature,

CK\_ULONG\_PTR pulSignatureLen

);

**C\_SignMessageNext** continues a multiple-part message signature operation, processing another data part, or finishes a multiple-part message signature operation, returning the signature. *hSession* is the session’s handle, *pDataPart* points to the data part; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature operation; *ulDataPartLen* is the length of the data part; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

The *pulSignatureLen* argument is set to NULL if there is more data part to follow, or set to a non-NULL value (to receive the signature length) if this is the last data part.

**C\_SignMessageNext** uses the convention described in Section 5.2 on producing output.

The message signing operation MUST have been started with **C\_SignMessageBegin**. This function may be called any number of times in succession. A call to **C\_SignMessageNext** with a NULL *pulSignatureLen* which results in an error terminates the current message signature operation. A call to **C\_SignMessageNext** with a non-NULL *pulSignatureLen* always terminates the active message signing operation unless it returns CKR\_BUFFER\_TOO\_SMALL to determine the length of the buffer needed to hold the signature, or is a successful call (i.e., one which returns CKR\_OK).

Although the last **C\_SignMessageNext** call ends the signing of a message, it does not finish the message-based signing process. Additional **C\_SignMessage** or **C\_SignMessageBegin** and **C\_SignMessageNext** calls may be made on the session.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

### C\_MessageSignFinal

CK\_DECLARE\_FUNCTION(CK\_RV, C\_MessageSignFinal)(

CK\_SESSION\_HANDLE hSession

);

**C\_MessageSignFinal** finishes a message-based signing process. *hSession* is the session’s handle.

The message-based signing process MUST have been initialized with **C\_MessageSignInit**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_FUNCTION\_REJECTED, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

## Functions for verifying signatures and MACs

Cryptoki provides the following functions for verifying signatures on data (for the purposes of Cryptoki, these operations also encompass message authentication codes):

### C\_VerifyInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_VerifyInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hKey  
);

**C\_VerifyInit** initializes a verification operation, where the signature is an appendix to the data. *hSession* is the session’s handle; *pMechanism* points to the structure that specifies the verification mechanism; *hKey* is the handle of the verification key.

The **CKA\_VERIFY** attribute of the verification key, which indicates whether the key supports verification where the signature is an appendix to the data, MUST be CK\_TRUE.

After calling **C\_VerifyInit**, the application can either call **C\_Verify** to verify a signature on data in a single part; or call **C\_VerifyUpdate** one or more times, followed by **C\_VerifyFinal,** to verify a signature on data in multiple parts. The verification operation is active until the application calls **C\_Verify** or **C\_VerifyFinal**. To process additional data (in single or multiple parts), the application MUST call **C\_VerifyInit** again.

**C\_VerifyInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active verification operation. If an active operation has been initialized and it cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be returned.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

Example: see **C\_VerifyFinal**.

### C\_Verify

CK\_DECLARE\_FUNCTION(CK\_RV, C\_Verify)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pData,  
 CK\_ULONG ulDataLen,  
 CK\_BYTE\_PTR pSignature,  
 CK\_ULONG ulSignatureLen  
);

**C\_Verify** verifies a signature in a single-part operation, where the signature is an appendix to the data. *hSession* is the session’s handle; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

The verification operation MUST have been initialized with **C\_VerifyInit**. A call to **C\_Verify** always terminates the active verification operation.

A successful call to **C\_Verify** should return either the value CKR\_OK (indicating that the supplied signature is valid) or CKR\_SIGNATURE\_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR\_SIGNATURE\_LEN\_RANGE should be returned. In any of these cases, the active signing operation is terminated.

**C\_Verify**cannot be used to terminate a multi-part operation, and MUST be called after **C\_VerifyInit** without intervening **C\_VerifyUpdate** calls.

For most mechanisms, **C\_Verify** is equivalent to a sequence of **C\_VerifyUpdate** operations followed by **C\_VerifyFinal**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SIGNATURE\_INVALID, CKR\_SIGNATURE\_LEN\_RANGE, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

Example: see **C\_VerifyFinal** for an example of similar functions.

### C\_VerifyUpdate

CK\_DECLARE\_FUNCTION(CK\_RV, C\_VerifyUpdate)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pPart,  
 CK\_ULONG ulPartLen  
);

**C\_VerifyUpdate** continues a multiple-part verification operation, processing another data part. *hSession* is the session’s handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

The verification operation MUST have been initialized with **C\_VerifyInit**. This function may be called any number of times in succession. A call to **C\_VerifyUpdate** which results in an error terminates the current verification operation.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

Example: see **C\_VerifyFinal**.

### C\_VerifyFinal

CK\_DECLARE\_FUNCTION(CK\_RV, C\_VerifyFinal)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pSignature,  
 CK\_ULONG ulSignatureLen  
);

**C\_VerifyFinal** finishes a multiple-part verification operation, checking the signature. *hSession* is the session’s handle; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

The verification operation MUST have been initialized with **C\_VerifyInit**. A call to **C\_VerifyFinal** always terminates the active verification operation.

A successful call to **C\_VerifyFinal** should return either the value CKR\_OK (indicating that the supplied signature is valid) or CKR\_SIGNATURE\_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR\_SIGNATURE\_LEN\_RANGE should be returned. In any of these cases, the active verifying operation is terminated.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SIGNATURE\_INVALID, CKR\_SIGNATURE\_LEN\_RANGE, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_MECHANISM mechanism = {

CKM\_DES\_MAC, NULL\_PTR, 0

};

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

CK\_BYTE mac[4];

CK\_RV rv;

.

.

rv = C\_VerifyInit(hSession, &mechanism, hKey);

if (rv == CKR\_OK) {

rv = C\_VerifyUpdate(hSession, data, sizeof(data));

.

.

rv = C\_VerifyFinal(hSession, mac, sizeof(mac));

.

.

}

### C\_VerifyRecoverInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_VerifyRecoverInit)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hKey  
);

**C\_VerifyRecoverInit** initializes a signature verification operation, where the data is recovered from the signature. *hSession* is the session’s handle; *pMechanism* points to the structure that specifies the verification mechanism; *hKey* is the handle of the verification key.

The **CKA\_VERIFY\_RECOVER** attribute of the verification key, which indicates whether the key supports verification where the data is recovered from the signature, MUST be CK\_TRUE.

After calling **C\_VerifyRecoverInit**, the application may call **C\_VerifyRecover** to verify a signature on data in a single part. The verification operation is active until the application uses a call to **C\_VerifyRecover** *to actually obtain* the recovered message.

**C\_VerifyRecoverInit** can be called with *pMechanism* set to NULL\_PTR to terminate an active verification with data recovery operation. If an active operations has been initialized and it cannot be cancelled, CKR\_OPERATION\_CANCEL\_FAILED must be returned.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_OPERATION\_CANCEL\_FAILED.

Example: see **C\_VerifyRecover**.

### C\_VerifyRecover

CK\_DECLARE\_FUNCTION(CK\_RV, C\_VerifyRecover)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pSignature,  
 CK\_ULONG ulSignatureLen,  
 CK\_BYTE\_PTR pData,  
 CK\_ULONG\_PTR pulDataLen  
);

**C\_VerifyRecover** verifies a signature in a single-part operation, where the data is recovered from the signature. *hSession* is the session’s handle; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature; *pData* points to the location that receives the recovered data; and *pulDataLen* points to the location that holds the length of the recovered data.

**C\_VerifyRecover** uses the convention described in Section 5.2 on producing output.

The verification operation MUST have been initialized with **C\_VerifyRecoverInit**. A call to **C\_VerifyRecover** always terminates the active verification operation unless it returns CKR\_BUFFER\_TOO\_SMALL or is a successful call (*i.e.*, one which returns CKR\_OK) to determine the length of the buffer needed to hold the recovered data.

A successful call to **C\_VerifyRecover** should return either the value CKR\_OK (indicating that the supplied signature is valid) or CKR\_SIGNATURE\_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR\_SIGNATURE\_LEN\_RANGE should be returned. The return codes CKR\_SIGNATURE\_INVALID and CKR\_SIGNATURE\_LEN\_RANGE have a higher priority than the return code CKR\_BUFFER\_TOO\_SMALL, *i.e.*, if **C\_VerifyRecover** is supplied with an invalid signature, it will never return CKR\_BUFFER\_TOO\_SMALL.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SIGNATURE\_LEN\_RANGE, CKR\_SIGNATURE\_INVALID, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_MECHANISM mechanism = {

CKM\_RSA\_9796, NULL\_PTR, 0

};

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

CK\_ULONG ulDataLen;

CK\_BYTE signature[128];

CK\_RV rv;

.

.

rv = C\_VerifyRecoverInit(hSession, &mechanism, hKey);

if (rv == CKR\_OK) {

ulDataLen = sizeof(data);

rv = C\_VerifyRecover(

hSession, signature, sizeof(signature), data, &ulDataLen);

.

.

}

## Message-based functions for verifying signatures and MACs

Message-based verification refers to the process of verifying signatures on multiple messages using the same verification mechanism and verification key.

Cryptoki provides the following functions for verifying signatures on messages (for the purposes of Cryptoki, these operations also encompass message authentication codes).

### C\_MessageVerifyInit

CK\_DECLARE\_FUNCTION(CK\_RV, C\_MessageVerifyInit)(

CK\_SESSION\_HANDLE hSession,

CK\_MECHANISM\_PTR pMechanism,

CK\_OBJECT\_HANDLE hKey

);

**C\_MessageVerifyInit** initializes a message-based verification process, preparing a session for one or more verification operations (where the signature is an appendix to the data) that use the same verification mechanism and verification key. *hSession* is the session’s handle; *pMechanism* points to the structure that specifies the verification mechanism; *hKey* is the handle of the verification key.

The **CKA\_VERIFY** attribute of the verification key, which indicates whether the key supports verification where the signature is an appendix to the data, MUST be CK\_TRUE.

After calling **C\_MessageVerifyInit**, the application can either call **C\_VerifyMessage** to verify a signature on a message in a single part; or call **C\_VerifyMessageBegin**, followed by **C\_VerifyMessageNext** one or more times, to verify a signature on a message in multiple parts. This may be repeated several times. The message-based verification process is active until the application calls **C\_MessageVerifyFinal** to finish the message-based verification process.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_FUNCTION\_NOT\_PERMITTED, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### C\_VerifyMessage

CK\_DECLARE\_FUNCTION(CK\_RV, C\_VerifyMessage)(

CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pData,

CK\_ULONG ulDataLen,

CK\_BYTE\_PTR pSignature,

CK\_ULONG ulSignatureLen

);

**C\_VerifyMessage** verifies a signature on a message in a single part operation, where the signature is an appendix to the data. **C\_MessageVerifyInit** must previously been called on the session. *hSession* is the session’s handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message verification operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

Unlike the *pParameter* parameter of **C\_SignMessage**, *pParameter* is always an input parameter.

The message-based verification process MUST have been initialized with **C\_MessageVerifyInit**. A call to **C\_VerifyMessage** starts and terminates a message verification operation.

A successful call to **C\_VerifyMessage** should return either the value CKR\_OK (indicating that the supplied signature is valid) or CKR\_SIGNATURE\_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR\_SIGNATURE\_LEN\_RANGE should be returned.

**C\_VerifyMessage** does not finish the message-based verification process. Additional **C\_VerifyMessage** or **C\_VerifyMessageBegin** and **C\_VerifyMessageNext** calls may be made on the session.

For most mechanisms, **C\_VerifyMessage** is equivalent to **C\_VerifyMessageBegin** followed by a sequence of **C\_VerifyMessageNext** operations.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_INVALID, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SIGNATURE\_INVALID, CKR\_SIGNATURE\_LEN\_RANGE, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

### C\_VerifyMessageBegin

CK\_DECLARE\_FUNCTION(CK\_RV, C\_VerifyMessageBegin)(

CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen

);

**C\_VerifyMessageBegin** begins a multiple-part message verification operation, where the signature is an appendix to the message. **C\_MessageVerifyInit** must previously been called on the session. *hSession* is the session’s handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message verification operation.

Unlike the *pParameter* parameter of **C\_SignMessageBegin**, *pParameter* is always an input parameter.

After calling **C\_VerifyMessageBegin**, the application should call **C\_VerifyMessageNext** one or more times to verify a signature on a message in multiple parts. The message verification operation is active until the application calls **C\_VerifyMessageNext** with a non-NULL *pSignature*. To process additional messages (in single or multiple parts), the application MUST call **C\_VerifyMessage** or **C\_VerifyMessageBegin** again.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

### C\_VerifyMessageNext

CK\_DECLARE\_FUNCTION(CK\_RV, C\_VerifyMessageNext)(

CK\_SESSION\_HANDLE hSession,

CK\_VOID\_PTR pParameter,

CK\_ULONG ulParameterLen,

CK\_BYTE\_PTR pDataPart,

CK\_ULONG ulDataPartLen,

CK\_BYTE\_PTR pSignature,

CK\_ULONG ulSignatureLen

);

**C\_VerifyMessageNext** continues a multiple-part message verification operation, processing another data part, or finishes a multiple-part message verification operation, checking the signature. *hSession* is the session’s handle, *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message verification operation, *pPart* points to the data part; *ulPartLen* is the length of the data part; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

The *pSignature* argument is set to NULL if there is more data part to follow, or set to a non-NULL value (pointing to the signature to verify) if this is the last data part.

The message verification operation MUST have been started with **C\_VerifyMessageBegin**. This function may be called any number of times in succession. A call to **C\_VerifyMessageNext** with a NULL *pSignature* which results in an error terminates the current message verification operation. A call to **C\_VerifyMessageNext** with a non-NULL *pSignature* always terminates the active message verification operation.

A successful call to **C\_VerifyMessageNext** with a non-NULL *pSignature* should return either the value CKR\_OK (indicating that the supplied signature is valid) or CKR\_SIGNATURE\_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR\_SIGNATURE\_LEN\_RANGE should be returned. In any of these cases, the active message verifying operation is terminated.

Although the last **C\_VerifyMessageNext** call ends the verification of a message, it does not finish the message-based verification process. Additional **C\_VerifyMessage** or **C\_VerifyMessageBegin** and **C\_VerifyMessageNext** calls may be made on the session.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SIGNATURE\_INVALID, CKR\_SIGNATURE\_LEN\_RANGE, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

### C\_MessageVerifyFinal

CK\_DECLARE\_FUNCTION(CK\_RV,C\_MessageVerifyFinal)(

CK\_SESSION\_HANDLE hSession

);

**C\_MessageVerifyFinal** finishes a message-based verification process. *hSession* is the session’s handle.

The message-based verification process MUST have been initialized with **C\_MessageVerifyInit**.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_TOKEN\_RESOURCE\_EXCEEDED.

## Dual-function cryptographic functions

Cryptoki provides the following functions to perform two cryptographic operations “simultaneously” within a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and from a token.

### C\_DigestEncryptUpdate

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DigestEncryptUpdate)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pPart,  
 CK\_ULONG ulPartLen,  
 CK\_BYTE\_PTR pEncryptedPart,  
 CK\_ULONG\_PTR pulEncryptedPartLen  
);

**C\_DigestEncryptUpdate** continues multiple-part digest and encryption operations, processing another data part. *hSession* is the session’s handle; *pPart* points to the data part; *ulPartLen* is the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part; *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

**C\_DigestEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a **C\_DigestEncryptUpdate** call does not produce encrypted output (because an error occurs, or because *pEncryptedPart* has the value NULL\_PTR, or because *pulEncryptedPartLen* is too small to hold the entire encrypted part output), then no plaintext is passed to the active digest operation.

Digest and encryption operations MUST both be active (they MUST have been initialized with **C\_DigestInit** and **C\_EncryptInit,** respectively). This function may be called any number of times in succession, and may be interspersed with **C\_DigestUpdate**, **C\_DigestKey**, and **C\_EncryptUpdate** calls (it would be somewhat unusual to intersperse calls to **C\_DigestEncryptUpdate** with calls to **C\_DigestKey**, however).

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

#define BUF\_SZ 512

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_BYTE iv[8];

CK\_MECHANISM digestMechanism = {

CKM\_MD5, NULL\_PTR, 0

};

CK\_MECHANISM encryptionMechanism = {

CKM\_DES\_ECB, iv, sizeof(iv)

};

CK\_BYTE encryptedData[BUF\_SZ];

CK\_ULONG ulEncryptedDataLen;

CK\_BYTE digest[16];

CK\_ULONG ulDigestLen;

CK\_BYTE data[(2\*BUF\_SZ)+8];

CK\_RV rv;

int i;

.

.

memset(iv, 0, sizeof(iv));

memset(data, ‘A’, ((2\*BUF\_SZ)+5));

rv = C\_EncryptInit(hSession, &encryptionMechanism, hKey);

if (rv != CKR\_OK) {

.

.

}

rv = C\_DigestInit(hSession, &digestMechanism);

if (rv != CKR\_OK) {

.

.

}

ulEncryptedDataLen = sizeof(encryptedData);

rv = C\_DigestEncryptUpdate(

hSession,

&data[0], BUF\_SZ,

encryptedData, &ulEncryptedDataLen);

.

.

ulEncryptedDataLen = sizeof(encryptedData);

rv = C\_DigestEncryptUpdate(

hSession,

&data[BUF\_SZ], BUF\_SZ,

encryptedData, &ulEncryptedDataLen);

.

.

/\*

\* The last portion of the buffer needs to be

\* handled with separate calls to deal with

\* padding issues in ECB mode

\*/

/\* First, complete the digest on the buffer \*/

rv = C\_DigestUpdate(hSession, &data[BUF\_SZ\*2], 5);

.

.

ulDigestLen = sizeof(digest);

rv = C\_DigestFinal(hSession, digest, &ulDigestLen);

.

.

/\* Then, pad last part with 3 0x00 bytes, and complete encryption \*/

for(i=0;i<3;i++)

data[((BUF\_SZ\*2)+5)+i] = 0x00;

/\* Now, get second-to-last piece of ciphertext \*/

ulEncryptedDataLen = sizeof(encryptedData);

rv = C\_EncryptUpdate(

hSession,

&data[BUF\_SZ\*2], 8,

encryptedData, &ulEncryptedDataLen);

.

.

/\* Get last piece of ciphertext (should have length 0, here) \*/

ulEncryptedDataLen = sizeof(encryptedData);

rv = C\_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);

.

.

### C\_DecryptDigestUpdate

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DecryptDigestUpdate)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pEncryptedPart,  
 CK\_ULONG ulEncryptedPartLen,  
 CK\_BYTE\_PTR pPart,  
 CK\_ULONG\_PTR pulPartLen  
);

**C\_DecryptDigestUpdate** continues a multiple-part combined decryption and digest operation, processing another data part. *hSession* is the session’s handle; *pEncryptedPart* points to the encrypted data part; *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

**C\_DecryptDigestUpdate** uses the convention described in Section 5.2 on producing output. If a **C\_DecryptDigestUpdate** call does not produce decrypted output (because an error occurs, or because *pPart* has the value NULL\_PTR, or because *pulPartLen* is too small to hold the entire decrypted part output), then no plaintext is passed to the active digest operation.

Decryption and digesting operations MUST both be active (they MUST have been initialized with **C\_DecryptInit** and **C\_DigestInit,** respectively). This function may be called any number of times in succession, and may be interspersed with **C\_DecryptUpdate**, **C\_DigestUpdate**, and **C\_DigestKey** calls (it would be somewhat unusual to intersperse calls to **C\_DigestEncryptUpdate** with calls to **C\_DigestKey**, however).

Use of **C\_DecryptDigestUpdate** involves a pipelining issue that does not arise when using **C\_DigestEncryptUpdate**, the “inverse function” of **C\_DecryptDigestUpdate**. This is because when **C\_DigestEncryptUpdate** is called, precisely the same input is passed to both the active digesting operation and the active encryption operation; however, when **C\_DecryptDigestUpdate** is called, the input passed to the active digesting operation is the *output of* the active decryption operation. This issue comes up only when the mechanism used for decryption performs padding.

In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this ciphertext and digest the original plaintext thereby obtained.

After initializing decryption and digesting operations, the application passes the 24-byte ciphertext (3 DES blocks) into **C\_DecryptDigestUpdate**. **C\_DecryptDigestUpdate** returns exactly 16 bytes of plaintext, since at this point, Cryptoki doesn’t know if there’s more ciphertext coming, or if the last block of ciphertext held any padding. These 16 bytes of plaintext are passed into the active digesting operation.

Since there is no more ciphertext, the application calls **C\_DecryptFinal**. This tells Cryptoki that there’s no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active decryption and digesting operations are linked *only* through the **C\_DecryptDigestUpdate** call, these 2 bytes of plaintext are *not* passed on to be digested.

A call to **C\_DigestFinal**, therefore, would compute the message digest of *the first 16 bytes of the plaintext*, not the message digest of the entire plaintext. It is crucial that, before **C\_DigestFinal** is called, the last 2 bytes of plaintext get passed into the active digesting operation via a **C\_DigestUpdate** call.

Because of this, it is critical that when an application uses a padded decryption mechanism with **C\_DecryptDigestUpdate**, it knows exactly how much plaintext has been passed into the active digesting operation. *Extreme caution is warranted when using a padded decryption mechanism with* ***C\_DecryptDigestUpdate****.*

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID, CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

#define BUF\_SZ 512

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_BYTE iv[8];

CK\_MECHANISM decryptionMechanism = {

CKM\_DES\_ECB, iv, sizeof(iv)

};

CK\_MECHANISM digestMechanism = {

CKM\_MD5, NULL\_PTR, 0

};

CK\_BYTE encryptedData[(2\*BUF\_SZ)+8];

CK\_BYTE digest[16];

CK\_ULONG ulDigestLen;

CK\_BYTE data[BUF\_SZ];

CK\_ULONG ulDataLen, ulLastUpdateSize;

CK\_RV rv;

.

.

memset(iv, 0, sizeof(iv));

memset(encryptedData, ‘A’, ((2\*BUF\_SZ)+8));

rv = C\_DecryptInit(hSession, &decryptionMechanism, hKey);

if (rv != CKR\_OK) {

.

.

}

rv = C\_DigestInit(hSession, &digestMechanism);

if (rv != CKR\_OK){

.

.

}

ulDataLen = sizeof(data);

rv = C\_DecryptDigestUpdate(

hSession,

&encryptedData[0], BUF\_SZ,

data, &ulDataLen);

.

.

ulDataLen = sizeof(data);

rv = C\_DecryptDigestUpdate(

hSession,

&encryptedData[BUF\_SZ], BUF\_SZ,

data, &ulDataLen);

.

.

/\*

\* The last portion of the buffer needs to be handled with

\* separate calls to deal with padding issues in ECB mode

\*/

/\* First, complete the decryption of the buffer \*/

ulLastUpdateSize = sizeof(data);

rv = C\_DecryptUpdate(

hSession,

&encryptedData[BUF\_SZ\*2], 8,

data, &ulLastUpdateSize);

.

.

/\* Get last piece of plaintext (should have length 0, here) \*/

ulDataLen = sizeof(data)-ulLastUpdateSize;

rv = C\_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);

if (rv != CKR\_OK) {

.

.

}

/\* Digest last bit of plaintext \*/

rv = C\_DigestUpdate(hSession, data, 5);

if (rv != CKR\_OK) {

.

.

}

ulDigestLen = sizeof(digest);

rv = C\_DigestFinal(hSession, digest, &ulDigestLen);

if (rv != CKR\_OK) {

.

.

}

### C\_SignEncryptUpdate

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SignEncryptUpdate)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pPart,  
 CK\_ULONG ulPartLen,  
 CK\_BYTE\_PTR pEncryptedPart,  
 CK\_ULONG\_PTR pulEncryptedPartLen  
);

**C\_SignEncryptUpdate** continues a multiple-part combined signature and encryption operation, processing another data part. *hSession* is the session’s handle; *pPart* points to the data part; *ulPartLen* is the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part; and *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

**C\_SignEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a **C\_SignEncryptUpdate** call does not produce encrypted output (because an error occurs, or because *pEncryptedPart* has the value NULL\_PTR, or because *pulEncryptedPartLen* is too small to hold the entire encrypted part output), then no plaintext is passed to the active signing operation.

Signature and encryption operations MUST both be active (they MUST have been initialized with **C\_SignInit** and **C\_EncryptInit,** respectively). This function may be called any number of times in succession, and may be interspersed with **C\_SignUpdate** and **C\_EncryptUpdate** calls.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

#define BUF\_SZ 512

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hEncryptionKey, hMacKey;

CK\_BYTE iv[8];

CK\_MECHANISM signMechanism = {

CKM\_DES\_MAC, NULL\_PTR, 0

};

CK\_MECHANISM encryptionMechanism = {

CKM\_DES\_ECB, iv, sizeof(iv)

};

CK\_BYTE encryptedData[BUF\_SZ];

CK\_ULONG ulEncryptedDataLen;

CK\_BYTE MAC[4];

CK\_ULONG ulMacLen;

CK\_BYTE data[(2\*BUF\_SZ)+8];

CK\_RV rv;

int i;

.

.

memset(iv, 0, sizeof(iv));

memset(data, ‘A’, ((2\*BUF\_SZ)+5));

rv = C\_EncryptInit(hSession, &encryptionMechanism, hEncryptionKey);

if (rv != CKR\_OK) {

.

.

}

rv = C\_SignInit(hSession, &signMechanism, hMacKey);

if (rv != CKR\_OK) {

.

.

}

ulEncryptedDataLen = sizeof(encryptedData);

rv = C\_SignEncryptUpdate(

hSession,

&data[0], BUF\_SZ,

encryptedData, &ulEncryptedDataLen);

.

.

ulEncryptedDataLen = sizeof(encryptedData);

rv = C\_SignEncryptUpdate(

hSession,

&data[BUF\_SZ], BUF\_SZ,

encryptedData, &ulEncryptedDataLen);

.

.

/\*

\* The last portion of the buffer needs to be handled with

\* separate calls to deal with padding issues in ECB mode

\*/

/\* First, complete the signature on the buffer \*/

rv = C\_SignUpdate(hSession, &data[BUF\_SZ\*2], 5);

.

.

ulMacLen = sizeof(MAC);

rv = C\_SignFinal(hSession, MAC, &ulMacLen);

.

.

/\* Then pad last part with 3 0x00 bytes, and complete encryption \*/

for(i=0;i<3;i++)

data[((BUF\_SZ\*2)+5)+i] = 0x00;

/\* Now, get second-to-last piece of ciphertext \*/

ulEncryptedDataLen = sizeof(encryptedData);

rv = C\_EncryptUpdate(

hSession,

&data[BUF\_SZ\*2], 8,

encryptedData, &ulEncryptedDataLen);

.

.

/\* Get last piece of ciphertext (should have length 0, here) \*/

ulEncryptedDataLen = sizeof(encryptedData);

rv = C\_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);

.

.

### C\_DecryptVerifyUpdate

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DecryptVerifyUpdate)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pEncryptedPart,  
 CK\_ULONG ulEncryptedPartLen,  
 CK\_BYTE\_PTR pPart,  
 CK\_ULONG\_PTR pulPartLen  
);

**C\_DecryptVerifyUpdate** continues a multiple-part combined decryption and verification operation, processing another data part. *hSession* is the session’s handle; *pEncryptedPart* points to the encrypted data; *ulEncryptedPartLen* is the length of the encrypted data; *pPart* points to the location that receives the recovered data; and *pulPartLen* points to the location that holds the length of the recovered data.

**C\_DecryptVerifyUpdate** uses the convention described in Section 5.2 on producing output. If a **C\_DecryptVerifyUpdate** call does not produce decrypted output (because an error occurs, or because *pPart* has the value NULL\_PTR, or because *pulPartLen* is too small to hold the entire encrypted part output), then no plaintext is passed to the active verification operation.

Decryption and signature operations MUST both be active (they MUST have been initialized with **C\_DecryptInit** and **C\_VerifyInit,** respectively). This function may be called any number of times in succession, and may be interspersed with **C\_DecryptUpdate** and **C\_VerifyUpdate** calls.

Use of **C\_DecryptVerifyUpdate** involves a pipelining issue that does not arise when using **C\_SignEncryptUpdate**, the “inverse function” of **C\_DecryptVerifyUpdate**. This is because when **C\_SignEncryptUpdate** is called, precisely the same input is passed to both the active signing operation and the active encryption operation; however, when **C\_DecryptVerifyUpdate** is called, the input passed to the active verifying operation is the *output of* the active decryption operation. This issue comes up only when the mechanism used for decryption performs padding.

In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this ciphertext and verify a signature on the original plaintext thereby obtained.

After initializing decryption and verification operations, the application passes the 24-byte ciphertext (3 DES blocks) into **C\_DecryptVerifyUpdate**. **C\_DecryptVerifyUpdate** returns exactly 16 bytes of plaintext, since at this point, Cryptoki doesn’t know if there’s more ciphertext coming, or if the last block of ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation.

Since there is no more ciphertext, the application calls **C\_DecryptFinal**. This tells Cryptoki that there’s no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active decryption and verification operations are linked *only* through the **C\_DecryptVerifyUpdate** call, these 2 bytes of plaintext are *not* passed on to the verification mechanism.

A call to **C\_VerifyFinal**, therefore, would verify whether or not the signature supplied is a valid signature on *the first 16 bytes of the plaintext*, not on the entire plaintext. It is crucial that, before **C\_VerifyFinal** is called, the last 2 bytes of plaintext get passed into the active verification operation via a **C\_VerifyUpdate** call.

Because of this, it is critical that when an application uses a padded decryption mechanism with **C\_DecryptVerifyUpdate**, it knows exactly how much plaintext has been passed into the active verification operation. *Extreme caution is warranted when using a padded decryption mechanism with* ***C\_DecryptVerifyUpdate****.*

Return values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DATA\_LEN\_RANGE, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_ENCRYPTED\_DATA\_INVALID, CKR\_ENCRYPTED\_DATA\_LEN\_RANGE, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_NOT\_INITIALIZED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID.

Example:

#define BUF\_SZ 512

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hDecryptionKey, hMacKey;

CK\_BYTE iv[8];

CK\_MECHANISM decryptionMechanism = {

CKM\_DES\_ECB, iv, sizeof(iv)

};

CK\_MECHANISM verifyMechanism = {

CKM\_DES\_MAC, NULL\_PTR, 0

};

CK\_BYTE encryptedData[(2\*BUF\_SZ)+8];

CK\_BYTE MAC[4];

CK\_ULONG ulMacLen;

CK\_BYTE data[BUF\_SZ];

CK\_ULONG ulDataLen, ulLastUpdateSize;

CK\_RV rv;

.

.

memset(iv, 0, sizeof(iv));

memset(encryptedData, ‘A’, ((2\*BUF\_SZ)+8));

rv = C\_DecryptInit(hSession, &decryptionMechanism, hDecryptionKey);

if (rv != CKR\_OK) {

.

.

}

rv = C\_VerifyInit(hSession, &verifyMechanism, hMacKey);

if (rv != CKR\_OK){

.

.

}

ulDataLen = sizeof(data);

rv = C\_DecryptVerifyUpdate(

hSession,

&encryptedData[0], BUF\_SZ,

data, &ulDataLen);

.

.

ulDataLen = sizeof(data);

rv = C\_DecryptVerifyUpdate(

hSession,

&encryptedData[BUF\_SZ], BUF\_SZ,

data, &ulDataLen);

.

.

/\*

\* The last portion of the buffer needs to be handled with

\* separate calls to deal with padding issues in ECB mode

\*/

/\* First, complete the decryption of the buffer \*/

ulLastUpdateSize = sizeof(data);

rv = C\_DecryptUpdate(

hSession,

&encryptedData[BUF\_SZ\*2], 8,

data, &ulLastUpdateSize);

.

.

/\* Get last little piece of plaintext. Should have length 0 \*/

ulDataLen = sizeof(data)-ulLastUpdateSize;

rv = C\_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);

if (rv != CKR\_OK) {

.

.

}

/\* Send last bit of plaintext to verification operation \*/

rv = C\_VerifyUpdate(hSession, data, 5);

if (rv != CKR\_OK) {

.

.

}

rv = C\_VerifyFinal(hSession, MAC, ulMacLen);

if (rv == CKR\_SIGNATURE\_INVALID) {

.

.

}

## Key management functions

Cryptoki provides the following functions for key management:

### C\_GenerateKey

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GenerateKey)(  
 CK\_SESSION\_HANDLE hSession  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_ATTRIBUTE\_PTR pTemplate,  
 CK\_ULONG ulCount,  
 CK\_OBJECT\_HANDLE\_PTR phKey  
);

**C\_GenerateKey** generates a secret key or set of domain parameters, creating a new object. *hSession* is the session’s handle; *pMechanism* points to the generation mechanism; *pTemplate* points to the template for the new key or set of domain parameters; *ulCount* is the number of attributes in the template; *phKey* points to the location that receives the handle of the new key or set of domain parameters.

If the generation mechanism is for domain parameter generation, the **CKA\_CLASS** attribute will have the value CKO\_DOMAIN\_PARAMETERS; otherwise, it will have the value CKO\_SECRET\_KEY.

Since the type of key or domain parameters to be generated is implicit in the generation mechanism, the template does not need to supply a key type. If it does supply a key type which is inconsistent with the generation mechanism, **C\_GenerateKey** fails and returns the error code CKR\_TEMPLATE\_INCONSISTENT. The CKA\_CLASS attribute is treated similarly.

If a call to **C\_GenerateKey** cannot support the precise template supplied to it, it will fail and return without creating an object.

The object created by a successful call to **C\_GenerateKey** will have its **CKA\_LOCAL** attribute set to CK\_TRUE. In addition, the object created will have a value for CKA\_UNIQUE\_ID generated and assigned (See Section 4.4.1).

Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_READ\_ONLY, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_CURVE\_NOT\_SUPPORTED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCOMPLETE, CKR\_TEMPLATE\_INCONSISTENT, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hKey;

CK\_MECHANISM mechanism = {

CKM\_DES\_KEY\_GEN, NULL\_PTR, 0

};

CK\_RV rv;

.

.

rv = C\_GenerateKey(hSession, &mechanism, NULL\_PTR, 0, &hKey);

if (rv == CKR\_OK) {

.

.

}

### C\_GenerateKeyPair

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GenerateKeyPair)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_ATTRIBUTE\_PTR pPublicKeyTemplate,  
 CK\_ULONG ulPublicKeyAttributeCount,  
 CK\_ATTRIBUTE\_PTR pPrivateKeyTemplate,  
 CK\_ULONG ulPrivateKeyAttributeCount,  
 CK\_OBJECT\_HANDLE\_PTR phPublicKey,  
 CK\_OBJECT\_HANDLE\_PTR phPrivateKey  
);

**C\_GenerateKeyPair** generates a public/private key pair, creating new key objects. *hSession* is the session’s handle; *pMechanism* points to the key generation mechanism; *pPublicKeyTemplate* points to the template for the public key; *ulPublicKeyAttributeCount* is the number of attributes in the public-key template; *pPrivateKeyTemplate* points to the template for the private key; *ulPrivateKeyAttributeCount* is the number of attributes in the private-key template; *phPublicKey* points to the location that receives the handle of the new public key; *phPrivateKey* points to the location that receives the handle of the new private key.

Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates do not need to supply key types. If one of the templates does supply a key type which is inconsistent with the key generation mechanism, **C\_GenerateKeyPair** fails and returns the error code CKR\_TEMPLATE\_INCONSISTENT. The CKA\_CLASS attribute is treated similarly.

If a call to **C\_GenerateKeyPair** cannot support the precise templates supplied to it, it will fail and return without creating any key objects.

A call to **C\_GenerateKeyPair** will never create just one key and return. A call can fail, and create no keys; or it can succeed, and create a matching public/private key pair.

The key objects created by a successful call to **C\_GenerateKeyPair** will have their **CKA\_LOCAL** attributes set to CK\_TRUE. In addition, the key objects created will both have values for CKA\_UNIQUE\_ID generated and assigned (See Section 4.4.1).

*Note carefully the order of the arguments to* ***C\_GenerateKeyPair****. The last two arguments do not have the same order as they did in the original Cryptoki Version 1.0 document. The order of these two arguments has caused some unfortunate confusion.*

Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_READ\_ONLY, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_CURVE\_NOT\_SUPPORTED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_DOMAIN\_PARAMS\_INVALID, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCOMPLETE, CKR\_TEMPLATE\_INCONSISTENT, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hPublicKey, hPrivateKey;

CK\_MECHANISM mechanism = {

CKM\_RSA\_PKCS\_KEY\_PAIR\_GEN, NULL\_PTR, 0

};

CK\_ULONG modulusBits = 3072;

CK\_BYTE publicExponent[] = { 3 };

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

CK\_BYTE id[] = {123};

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE publicKeyTemplate[] = {

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

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

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

{CKA\_MODULUS\_BITS, &modulusBits, sizeof(modulusBits)},

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

};

CK\_ATTRIBUTE privateKeyTemplate[] = {

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

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

{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\_UNWRAP, &true, sizeof(true)}

};

CK\_RV rv;

rv = C\_GenerateKeyPair(

hSession, &mechanism,

publicKeyTemplate, 5,

privateKeyTemplate, 8,

&hPublicKey, &hPrivateKey);

if (rv == CKR\_OK) {

.

.

}

### C\_WrapKey

CK\_DECLARE\_FUNCTION(CK\_RV, C\_WrapKey)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hWrappingKey,  
 CK\_OBJECT\_HANDLE hKey,  
 CK\_BYTE\_PTR pWrappedKey,  
 CK\_ULONG\_PTR pulWrappedKeyLen  
);

**C\_WrapKey** wraps (*i.e.*, encrypts) a private or secret key. *hSession* is the session’s handle; *pMechanism* points to the wrapping mechanism; *hWrappingKey* is the handle of the wrapping key; *hKey* is the handle of the key to be wrapped; *pWrappedKey* points to the location that receives the wrapped key; and *pulWrappedKeyLen* points to the location that receives the length of the wrapped key.

**C\_WrapKey** uses the convention described in Section 5.2 on producing output.

The **CKA\_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping, MUST be CK\_TRUE. The **CKA\_EXTRACTABLE** attribute of the key to be wrapped MUST also be CK\_TRUE.

If the key to be wrapped cannot be wrapped for some token-specific reason, despite its having its **CKA\_EXTRACTABLE** attribute set to CK\_TRUE, then **C\_WrapKey** fails with error code CKR\_KEY\_NOT\_WRAPPABLE. If it cannot be wrapped with the specified wrapping key and mechanism solely because of its length, then **C\_WrapKey** fails with error code CKR\_KEY\_SIZE\_RANGE.

**C\_WrapKey** can be used in the following situations:

* To wrap any secret key with a public key that supports encryption and decryption.
* To wrap any secret key with any other secret key. Consideration MUST be given to key size and mechanism strength or the token may not allow the operation.
* To wrap a private key with any secret key.

Of course, tokens vary in which types of keys can actually be wrapped with which mechanisms.

To partition the wrapping keys so they can only wrap a subset of extractable keys the attribute CKA\_WRAP\_TEMPLATE can be used on the wrapping key to specify an attribute set that will be compared against the attributes of the key to be wrapped. If all attributes match according to the C\_FindObject rules of attribute matching then the wrap will proceed. The value of this attribute is an attribute template and the size is the number of items in the template times the size of CK\_ATTRIBUTE. If this attribute is not supplied then any template is acceptable. If an attribute is not present, it will not be checked. If any attribute mismatch occurs on an attempt to wrap a key then the function SHALL return CKR\_KEY\_HANDLE\_INVALID.

Return Values: CKR\_ARGUMENTS\_BAD, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_NOT\_WRAPPABLE, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_UNEXTRACTABLE, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_WRAPPING\_KEY\_HANDLE\_INVALID, CKR\_WRAPPING\_KEY\_SIZE\_RANGE, CKR\_WRAPPING\_KEY\_TYPE\_INCONSISTENT.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hWrappingKey, hKey;

CK\_MECHANISM mechanism = {

CKM\_DES3\_ECB, NULL\_PTR, 0

};

CK\_BYTE wrappedKey[8];

CK\_ULONG ulWrappedKeyLen;

CK\_RV rv;

.

.

ulWrappedKeyLen = sizeof(wrappedKey);

rv = C\_WrapKey(

hSession, &mechanism,

hWrappingKey, hKey,

wrappedKey, &ulWrappedKeyLen);

if (rv == CKR\_OK) {

.

.

}

### C\_UnwrapKey

CK\_DECLARE\_FUNCTION(CK\_RV, C\_UnwrapKey)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hUnwrappingKey,  
 CK\_BYTE\_PTR pWrappedKey,  
 CK\_ULONG ulWrappedKeyLen,  
 CK\_ATTRIBUTE\_PTR pTemplate,  
 CK\_ULONG ulAttributeCount,  
 CK\_OBJECT\_HANDLE\_PTR phKey  
);

**C\_UnwrapKey** unwraps (*i.e.* decrypts) a wrapped key, creating a new private key or secret key object. *hSession* is the session’s handle; *pMechanism* points to the unwrapping mechanism; *hUnwrappingKey* is the handle of the unwrapping key; *pWrappedKey* points to the wrapped key; *ulWrappedKeyLen* is the length of the wrapped key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number of attributes in the template; *phKey* points to the location that receives the handle of the recovered key.

The **CKA\_UNWRAP** attribute of the unwrapping key, which indicates whether the key supports unwrapping, MUST be CK\_TRUE.

The new key will have the **CKA\_ALWAYS\_SENSITIVE** attribute set to CK\_FALSE, and the **CKA\_NEVER\_EXTRACTABLE** attribute set to CK\_FALSE. The **CKA\_EXTRACTABLE** attribute is by default set to CK\_TRUE.

Some mechanisms may modify, or attempt to modify. the contents of the pMechanism structure at the same time that the key is unwrapped.

If a call to **C\_UnwrapKey** cannot support the precise template supplied to it, it will fail and return without creating any key object.

The key object created by a successful call to **C\_UnwrapKey** will have its **CKA\_LOCAL** attribute set to CK\_FALSE. In addition, the object created will have a value for CKA\_UNIQUE\_ID generated and assigned (See Section 4.4.1).

To partition the unwrapping keys so they can only unwrap a subset of keys the attribute CKA\_UNWRAP\_TEMPLATE can be used on the unwrapping key to specify an attribute set that will be added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied attribute template, in ‘pTemplate’, then the unwrap will proceed. The value of this attribute is an attribute template and the size is the number of items in the template times the size of CK\_ATTRIBUTE. If this attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute conflict occurs on an attempt to unwrap a key then the function SHALL return CKR\_TEMPLATE\_INCONSISTENT.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_READ\_ONLY, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_BUFFER\_TOO\_SMALL, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_CURVE\_NOT\_SUPPORTED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_DOMAIN\_PARAMS\_INVALID, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCOMPLETE, CKR\_TEMPLATE\_INCONSISTENT, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_UNWRAPPING\_KEY\_HANDLE\_INVALID, CKR\_UNWRAPPING\_KEY\_SIZE\_RANGE, CKR\_UNWRAPPING\_KEY\_TYPE\_INCONSISTENT, CKR\_USER\_NOT\_LOGGED\_IN, CKR\_WRAPPED\_KEY\_INVALID, CKR\_WRAPPED\_KEY\_LEN\_RANGE.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hUnwrappingKey, hKey;

CK\_MECHANISM mechanism = {

CKM\_DES3\_ECB, NULL\_PTR, 0

};

CK\_BYTE wrappedKey[8] = {...};

CK\_OBJECT\_CLASS keyClass = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DES;

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE template[] = {

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

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

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

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

};

CK\_RV rv;

.

.

rv = C\_UnwrapKey(

hSession, &mechanism, hUnwrappingKey,

wrappedKey, sizeof(wrappedKey), template, 4, &hKey);

if (rv == CKR\_OK) {

.

.

}

### C\_DeriveKey

CK\_DECLARE\_FUNCTION(CK\_RV, C\_DeriveKey)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_MECHANISM\_PTR pMechanism,  
 CK\_OBJECT\_HANDLE hBaseKey,  
 CK\_ATTRIBUTE\_PTR pTemplate,  
 CK\_ULONG ulAttributeCount,  
 CK\_OBJECT\_HANDLE\_PTR phKey  
);

**C\_DeriveKey** derives a key from a base key, creating a new key object. *hSession* is the session’s handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the handle of the base key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.

The values of the **CKA\_SENSITIVE**, **CKA\_ALWAYS\_SENSITIVE**, **CKA\_EXTRACTABLE**, and **CKA\_NEVER\_EXTRACTABLE** attributes for the base key affect the values that these attributes can hold for the newly-derived key. See the description of each particular key-derivation mechanism in Section 5.21.2 for any constraints of this type.

If a call to **C\_DeriveKey** cannot support the precise template supplied to it, it will fail and return without creating any key object.

The key object created by a successful call to **C\_DeriveKey** will have its **CKA\_LOCAL** attribute set to CK\_FALSE. In addition, the object created will have a value for CKA\_UNIQUE\_ID generated and assigned (See Section 4.4.1).

To partition the derivation keys so they can only derive a subset of keys the attribute CKA\_DERIVE\_TEMPLATE can be used on the derivation keys to specify an attribute set that will be added to attributes of the key to be derived. If the attributes do not conflict with the user supplied attribute template, in ‘pTemplate’, then the derivation will proceed. The value of this attribute is an attribute template and the size is the number of items in the template times the size of CK\_ATTRIBUTE. If this attribute is not present on the base derivation keys then no additional attributes will be added. If any attribute conflict occurs on an attempt to derive a key then the function SHALL return CKR\_TEMPLATE\_INCONSISTENT.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_ATTRIBUTE\_READ\_ONLY, CKR\_ATTRIBUTE\_TYPE\_INVALID, CKR\_ATTRIBUTE\_VALUE\_INVALID, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_CURVE\_NOT\_SUPPORTED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_DOMAIN\_PARAMS\_INVALID, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_KEY\_HANDLE\_INVALID, CKR\_KEY\_SIZE\_RANGE, CKR\_KEY\_TYPE\_INCONSISTENT, CKR\_MECHANISM\_INVALID, CKR\_MECHANISM\_PARAM\_INVALID, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_PIN\_EXPIRED, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_READ\_ONLY, CKR\_TEMPLATE\_INCOMPLETE, CKR\_TEMPLATE\_INCONSISTENT, CKR\_TOKEN\_WRITE\_PROTECTED, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

CK\_SESSION\_HANDLE hSession;

CK\_OBJECT\_HANDLE hPublicKey, hPrivateKey, hKey;

CK\_MECHANISM keyPairMechanism = {

CKM\_DH\_PKCS\_KEY\_PAIR\_GEN, NULL\_PTR, 0

};

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

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

CK\_BYTE publicValue[128];

CK\_BYTE otherPublicValue[128];

CK\_MECHANISM mechanism = {

CKM\_DH\_PKCS\_DERIVE, otherPublicValue, sizeof(otherPublicValue)

};

CK\_ATTRIBUTE template[] = {

{CKA\_VALUE, &publicValue, sizeof(publicValue)}

};

CK\_OBJECT\_CLASS keyClass = CKO\_SECRET\_KEY;

CK\_KEY\_TYPE keyType = CKK\_DES;

CK\_BBOOL true = CK\_TRUE;

CK\_ATTRIBUTE publicKeyTemplate[] = {

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

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

};

CK\_ATTRIBUTE privateKeyTemplate[] = {

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

};

CK\_ATTRIBUTE derivedKeyTemplate[] = {

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

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

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

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

};

CK\_RV rv;

.

.

rv = C\_GenerateKeyPair(

hSession, &keyPairMechanism,

publicKeyTemplate, 2,

privateKeyTemplate, 1,

&hPublicKey, &hPrivateKey);

if (rv == CKR\_OK) {

rv = C\_GetAttributeValue(hSession, hPublicKey, template, 1);

if (rv == CKR\_OK) {

/\* Put other guy’s public value in otherPublicValue \*/

.

.

rv = C\_DeriveKey(

hSession, &mechanism,

hPrivateKey, derivedKeyTemplate, 4, &hKey);

if (rv == CKR\_OK) {

.

.

}

}

}

## Random number generation functions

Cryptoki provides the following functions for generating random numbers:

### C\_SeedRandom

CK\_DECLARE\_FUNCTION(CK\_RV, C\_SeedRandom)(  
 CK\_SESSION\_HANDLE hSession,  
 CK\_BYTE\_PTR pSeed,  
 CK\_ULONG ulSeedLen  
);

**C\_SeedRandom** mixes additional seed material into the token’s random number generator. *hSession* is the session’s handle; *pSeed* points to the seed material; and *ulSeedLen* is the length in bytes of the seed material.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_RANDOM\_SEED\_NOT\_SUPPORTED, CKR\_RANDOM\_NO\_RNG, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

Example: see **C\_GenerateRandom**.

### C\_GenerateRandom

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GenerateRandom)(

CK\_SESSION\_HANDLE hSession,

CK\_BYTE\_PTR pRandomData,

CK\_ULONG ulRandomLen

);

**C\_GenerateRandom** generates random or pseudo-random data. *hSession* is the session’s handle; *pRandomData* points to the location that receives the random data; and *ulRandomLen* is the length in bytes of the random or pseudo-random data to be generated.

Return values: CKR\_ARGUMENTS\_BAD, CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_DEVICE\_ERROR, CKR\_DEVICE\_MEMORY, CKR\_DEVICE\_REMOVED, CKR\_FUNCTION\_CANCELED, CKR\_FUNCTION\_FAILED, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_OK, CKR\_OPERATION\_ACTIVE, CKR\_RANDOM\_NO\_RNG, CKR\_SESSION\_CLOSED, CKR\_SESSION\_HANDLE\_INVALID, CKR\_USER\_NOT\_LOGGED\_IN.

Example:

CK\_SESSION\_HANDLE hSession;

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

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

CK\_RV rv;

.

.

rv = C\_SeedRandom(hSession, seed, sizeof(seed));

if (rv != CKR\_OK) {

.

.

}

rv = C\_GenerateRandom(hSession, randomData, sizeof(randomData));

if (rv == CKR\_OK) {

.

.

}

## Parallel function management functions

Cryptoki provides the following functions for managing parallel execution of cryptographic functions. These functions exist only for backwards compatibility.

### C\_GetFunctionStatus

CK\_DECLARE\_FUNCTION(CK\_RV, C\_GetFunctionStatus)(  
 CK\_SESSION\_HANDLE hSession  
);

In previous versions of Cryptoki, **C\_GetFunctionStatus** obtained the status of a function running in parallel with an application. Now, however, **C\_GetFunctionStatus** is a legacy function which should simply return the value CKR\_FUNCTION\_NOT\_PARALLEL.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_FUNCTION\_FAILED, CKR\_FUNCTION\_NOT\_PARALLEL, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_CLOSED.

### C\_CancelFunction

CK\_DECLARE\_FUNCTION(CK\_RV, C\_CancelFunction)(  
 CK\_SESSION\_HANDLE hSession  
);

In previous versions of Cryptoki, **C\_CancelFunction** cancelled a function running in parallel with an application. Now, however, **C\_CancelFunction** is a legacy function which should simply return the value CKR\_FUNCTION\_NOT\_PARALLEL.

Return values: CKR\_CRYPTOKI\_NOT\_INITIALIZED, CKR\_FUNCTION\_FAILED, CKR\_FUNCTION\_NOT\_PARALLEL, CKR\_GENERAL\_ERROR, CKR\_HOST\_MEMORY, CKR\_SESSION\_HANDLE\_INVALID, CKR\_SESSION\_CLOSED.

## Callback functions

Cryptoki sessions can use function pointers of type **CK\_NOTIFY** to notify the application of certain events.

### Surrender callbacks

Cryptographic functions (*i.e.*, any functions falling under one of these categories: encryption functions; decryption functions; message digesting functions; signing and MACing functions; functions for verifying signatures and MACs; dual-purpose cryptographic functions; key management functions; random number generation functions) executing in Cryptoki sessions can periodically surrender control to the application who called them if the session they are executing in had a notification callback function associated with it when it was opened. They do this by calling the session’s callback with the arguments (hSession, CKN\_SURRENDER, pApplication), where hSession is the session’s handle and pApplication was supplied to **C\_OpenSession** when the session was opened. Surrender callbacks should return either the value CKR\_OK (to indicate that Cryptoki should continue executing the function) or the value CKR\_CANCEL (to indicate that Cryptoki should abort execution of the function). Of course, before returning one of these values, the callback function can perform some computation, if desired.

A typical use of a surrender callback might be to give an application user feedback during a lengthy key pair generation operation. Each time the application receives a callback, it could display an additional “.” to the user. It might also examine the keyboard’s activity since the last surrender callback, and abort the key pair generation operation (probably by returning the value CKR\_CANCEL) if the user hit <ESCAPE>.

A Cryptoki library is not *required* to make *any* surrender callbacks.

### Vendor-defined callbacks

Library vendors can also define additional types of callbacks. Because of this extension capability, application-supplied notification callback routines should examine each callback they receive, and if they are unfamiliar with the type of that callback, they should immediately give control back to the library by returning with the value CKR\_OK.

# Mechanisms

## RSA

*Table 32, 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\_SHA224\_RSA\_PKCS |  | ✓ |  |  |  |  |  |
| CKM\_SHA256\_RSA\_PKCS |  | ✓ |  |  |  |  |  |
| CKM\_SHA384\_RSA\_PKCS |  | ✓ |  |  |  |  |  |
| CKM\_SHA512\_RSA\_PKCS |  | ✓ |  |  |  |  |  |
| CKM\_SHA1\_RSA\_PKCS\_PSS |  | ✓ |  |  |  |  |  |
| CKM\_SHA224\_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 33, 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 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 , RSA Private Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_MODULUS1,4,6 | Big integer | Modulus *n* |
| CKA\_PUBLIC\_EXPONENT1,4,6 | 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 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 34 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 34 for which a Cryptoki implementation is *required* to be able to return values are **CKA\_MODULUS, CKA\_PUBLIC\_EXPONENT** and **CKA\_PRIVATE\_EXPONENT**. A token SHOULD also be able to return **CKA\_PUBLIC\_KEY\_INFO** for an RSA private key.

If an RSA private key object is created on a token, and more attributes from Table 34 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 , 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 Mask 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 , 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 , 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**.

### 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 , 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 , 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 , 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 , 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 , 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 , 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\_ PKCS\_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 , 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\_PKCS\_PSS**, **CKM\_SHA3\_256\_RSA\_PKCS\_PSS**, **CKM\_SHA3\_384\_RSA\_PKCS\_PSS**, and **CKM\_SHA3\_512\_RSA\_PKCS\_PSS** respectively, performs similarly as the other **CKM\_SHA*X*\_RSA\_PKCS\_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 , 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 , 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 , 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**.
* 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 private target key will be encoded as defined in section 6.7.

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**.
* Zeroizes the temporary AES key.
* Returns the handle to the newly unwrapped target key.

*Table 48, 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 49, 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\_PROBABILISTIC\_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\_PROBABILISTIC\_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.

CK\_DSA\_PARAMETER\_GEN\_PARAM\_PTR is a pointer to a CK\_DSA\_PARAMETER\_GEN\_PARAM.

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\_SHA\_1, CKM\_SHA224, CKM\_SHA256, CKM\_SHA384, CKM\_SHA512.

pSeed Seed value used to generate PQ and G. This value is returned by CKM\_DSA\_PROBABILISTIC\_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\_PROBABILISTIC\_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 50, 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 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 51, 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 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 52, 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 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\_PROBABILISTIC\_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 , DSA: Key And Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | DSA private key | 20, 28, 32, 48, or 64 bytes | 2\*length of subprime |
| C\_Verify1 | DSA public key | (20, 28, 32, 48, or 64 bytes), (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 54, 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-224 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 55, 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-256 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 56, 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-384 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 57, 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-512 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 58, 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 59, 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 60, 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 61, 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 62, 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 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 63, 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 64, 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 types “CKK\_EC”, “CKK\_EC\_EDWARDS” and “CKK\_EC\_MONTGOMERY” 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

### Short Weierstrass Elliptic Curve public key objects

Short Weierstrass EC 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 65, 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 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 short Weierstrass EC 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)}

};

### Short Weierstrass Elliptic Curve private key objects

Short Weierstrass EC private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_EC**) hold EC private keys. See Section 6.3 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 66, 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 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 short Weierstrass EC 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 67, 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 | Public key bytes in little endian order as defined in RFC 8032 |

- Refer to 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\_EDWARDS;

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 6.3 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 68, 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 | Private key bytes in little endian order as defined in RFC 8032 |

- Refer to 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\_EDWARDS;

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 69, 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 | Public key bytes in little endian order as defined in RFC 7748 |

- Refer to 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\_MONTGOMERY;

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 6.3 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 70, 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 bytes in little endian order as defined in RFC 7748 |

- Refer to 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 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\_MONTGOMERY;

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 short Weierstrass ECkey 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 short Weierstrass EC 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 6.3.1 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 71, ECDSA without hashing: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | CKK\_EC private key | any3 | 2*nLen* |
| C\_Verify1 | CKK\_EC 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 6.3.1 for signature encoding.

The ECDSA with SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanism, denoted **CKM\_ECDSA\_[SHA1|SHA224|SHA256|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-256, 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 72, ECDSA with hashing: Key and Data Length

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign | CKK\_EC private key | any | 2*nLen* |
| C\_Verify | CKK\_EC 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. The following table enumerates the five signature schemes defined in RFC 8032 and all supported permutations of the mechanism parameter and its content.

Table 73, 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 74, 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 CKR\_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 75, 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 76, 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 77, 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. For short Weierstrass EC keys: 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.   
For Montgomery keys: the public key is provided as bytes in little endian order as defined in RFC 7748.

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 for short Weierstrass EC keys and RFC 7748 for Montgomery keys, 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**.
* 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 private target key will be encoded as defined in section 6.7.

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**.
* Zeroizes the temporary AES key.

*Table 81, 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\_Wrap / C\_Unwrap | 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 83, 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\_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\_PARAMETER\_GEN

CKM\_DH\_PKCS\_DERIVE

CKM\_X9\_42\_DH\_KEY\_PAIR\_GEN

CKM\_X9\_42\_DH\_PARAMETER\_GEN

CKM\_X9\_42\_DH\_DERIVE

CKM\_X9\_42\_DH\_HYBRID\_DERIVE

CKM\_X9\_42\_MQV\_DERIVE

### 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 84, 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 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 85, 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 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 86, 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 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 87, 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 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 88, 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 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 89, 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 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 90, 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 91, 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\_INITIALIZE |  |  |  |  |  |  |  |
| CKM\_X3DH\_RESPOND |  |  |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_X3DH\_INITIALIZE

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 XEDDSA 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\_INITIALIZE, 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\_INITIALIZE** 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 92, 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 93, 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 94, 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 95, Double Ratchet Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_X2RATCHET\_INITIALIZE |  |  |  |  |  |  | ✓ |
| 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\_INITIALIZE

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\_INITIALIZE 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 96, 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\_NHKS | 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

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

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 97, 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, short Weierstrass EC private keys and DSA private keys. For wrapping, a private key is BER-encoded according to [PKCS #8] 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\_EXPONENT\_2**, 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.
* Short Weierstrass EC 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 is defined in the section for the respective key wrapping mechanism.

Unwrapping a wrapped private key undoes the above procedure. The ciphertext is decrypted as defined for the respective key unwrapping mechanism. The data thereby obtained are parsed as a PrivateKeyInfo type. An error will result if the original wrapped key does not decrypt properly, or if the decrypted 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 98, 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 99, 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 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 100, 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 101, 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 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 102, 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 103, 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, short Weierstrass EC and DSA private keys (see Section 6.7 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 104, 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 105, 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 106, 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 107, 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 108, 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 109, 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 110, 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 111, 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 112, 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 113, 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 114, 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

CKG\_GENERATE\_COUNTER\_XOR

### 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, the initial IV counter is zero.

CKG\_GENERATE\_COUNTER\_XOR means that the non-fixed portion of the IV is xored with a counter. The value of the non-fixed portion passed must not vary from call to call. Like CKG\_GENERATE\_COUNTER, the counter starts at zero.

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, the initial IV counter is zero.

CKG\_GENERATE\_COUNTER\_XOR means that the non-fixed portion of the IV is xored with a counter. The value of the non-fixed portion passed must not vary from call to call. Like CKG\_GENERATE\_COUNTER, the counter starts at zero.

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\_GCM\_PARAMS field *ulTagBits*.

The IV length is determined by the CK\_GCM\_PARAMS field *ulIvLen*.

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

Table 115, 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\_FUNCTION;

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\_ULONG ulIvBits;

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.

ulIvBits length of initialization vector in bits. Do no use ulIvBits to specify the length of the initialization vector, but ulIvLen instead.

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 116, 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 117, 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 118, 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 119, 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\_XTS\_KEY\_GEN

### AES-XTS secret key objects

Table 120, 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 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 121, 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 122, 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 | ✓ |  |  |  |  | ✓ |  |
| CKM\_AES\_KEY\_WRAP\_PKCS7 | ✓ |  |  |  |  | ✓ |  |
| 1SR = SignRecover, VR = VerifyRecover | | | | | | | |

### Definitions

Mechanisms:

CKM\_AES\_KEY\_WRAP

CKM\_AES\_KEY\_WRAP\_PAD

CKM\_AES\_KEY\_WRAP\_KWP

CKM\_AES\_KEY\_WRAP\_PKCS7

### 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\_PKCS7, 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 and CKM\_AES\_KEY\_WRAP\_PKCS7, resp. 4 for CKM\_AES\_KEY\_WRAP\_KWP, and the pointer non-NULL.

### AES Key Wrap

The mechanisms support only single-part operations, i.e. single part wrapping and unwrapping, and single-part encryption and decryption.

1. CKM\_AES\_KEY\_WRAP

The CKM\_AES\_KEY\_WRAP mechanism can wrap a key of any length. A secret key whose length is not a multiple of the AES Key Wrap semiblock size (8 bytes) will be zero padded to fit. Semiblock size is defined in Section 5.2 of [AES KEYWRAP]. A private key will be encoded as defined in section 6.7; the encoded private key will be zero padded to fit if necessary.

The CKM\_AES\_KEY\_WRAP mechanism can only encrypt a block of data whose size is an exact multiple of the AES Key Wrap algorithm semiblock size.

For unwrapping, the mechanism decrypts the wrapped key. In case of a secret key, it 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 length specified in the template must not be less than n-7 bytes, where n is the length of the wrapped key. In case of a private key, the mechanism parses the encoding as defined in section 6.7 and ignores trailing zero bytes.

1. CKM\_AES\_KEY\_WRAP\_PAD

The CKM\_AES\_KEY\_WRAP\_PAD mechanism is deprecated. CKM\_AES\_KEY\_WRAP\_KWP resp. CKM\_AES\_KEY\_WRAP\_PKCS7 shall be used instead.

1. CKM\_AES\_KEY\_WRAP\_KWP

The CKM\_AES\_KEY\_WRAP\_KWP mechanism can wrap a key or encrypt block of data of any length. The input is zero-padded and wrapped / encrypted as defined in Section 6.3 of [AES KEYWRAP], which produces same results as RFC 5649.

1. CKM\_AES\_KEY\_WRAP\_PKCS7

The CKM\_AES\_KEY\_WRAP\_PKCS7 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) up to a semiblock size to make it an exact multiple of AES Key Wrap algorithum semiblock size (8bytes), 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].

## 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 123, 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 6.43.2

Table 124, 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 6.20.5.

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 125, 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 126, DES2 Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (always 16 bytes long) |

- Refer to 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 127, DES3 Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (always 24 bytes long) |

- Refer to 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 128, 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 129, 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 130, 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 131, 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.

### 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 132, 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 133, 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 134, 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 , 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 136, 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 6.20.3.

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 137, 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\_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 FIPS PUB 180-4.

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 , 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 139, 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 6.20.5 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 140, 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 141, 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 6.20.3, 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 142, 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 6.22.3.

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 6.20.5, 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 143, 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 144, 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 6.20.3, 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 145, 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 6.20.5, 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 146, 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 147, 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 6.20.3, 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 148, 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 6.20.5, 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 149, 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 150, 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 6.20.3, 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 151, 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 6.24.5, 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 152, 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 153, 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 6.20.3, 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 154, 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 6.24.5, 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 155, 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

CKM\_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 156, 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 6.20.3, 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 6.24.5, 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 157, SHA3-224 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 , 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 6.20.4 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 159, 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 6.20.5 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 160, 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 161, 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 6.20.4, 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 162, 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.

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 6.20.5, 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 163, 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 164, 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 6.20.4, 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 165, 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 6.20.5, 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 166, 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 167, 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 6.20.4, 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 168, 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 6.20.5, 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 169, 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\_TEMPLATE\_INCOMPLETE** 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 170, 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 172, 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 6.20.5 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 173, 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 174, 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 175, 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 Section6.34.3.

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 6.20.5, 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 for 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 176, 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 177, 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 178, 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 6.20.5, except that it uses the BLAKE2B-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 179, 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 180, 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 181, 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 Section6.20.5, 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 182, 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 183, 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\_PARAMS2** structure defined below.

Table 184, 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\_PARAMS2; CK\_PKCS5\_PBKD2\_PARAMS2\_PTR

**CK\_PKCS5\_PBKD2\_PARAMS2** is a structure that provides the parameters to the **CKM\_PKCS5\_PBKD2** mechanism. The structure is defined as follows:

typedef struct CK\_PKCS5\_PBKD2\_PARAMS2 {

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 ulPasswordLen;

} CK\_PKCS5\_PBKD2\_PARAMS2;

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\_PARAMS2\_PTR** is a pointer to a **CK\_PKCS5\_PBKD2\_PARAMS2**.

### 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\_PARAMS2** 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 185,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\_TLS\_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\_TLS\_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.

**CKM\_TLS\_PRE\_MASTER\_KEY\_GEN** has identical functionality as **CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN.** It exists only for historical reasons, please use **CKM\_SSL3\_PRE\_MASTER\_KEY\_GEN** instead.

### 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 6.39.

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 6.39. 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 6.39.

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 186, 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 187, 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 188, 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

1. 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**.

1. 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**.

1. 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\_KDF\_PARAMS\_PTR** is a pointer to a **CK\_TLS\_KDF\_PARAMS**.

1. 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 prfHashMechanism;

CK\_ULONG ulMacLength;

CK\_ULONG ulServerOrClient;

} CK\_TLS\_MAC\_PARAMS;

The fields of the structure have the following meanings:

prfHashMechanism 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 189, 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 6.39.

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 6.39. 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 6.39.

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 190, 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\_ PRF\_DATA\_PARAM array that is part of the mechanism parameter.

*Table 191, 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 192, 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\_DES3\_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 193, 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\_LENGTH\_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\_LENGTH\_FORMAT\_PTR and *ulValueLen* must be set to sizeof(CK\_SP800\_108\_DKM\_LENGTH\_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 194, 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\_SP800\_108\_PRF\_TYPE prfType;

CK\_ULONG ulNumberOfDataParams;

CK\_PRF\_DATA\_PARAM\_PTR pDataParams;

CK\_ULONG ulAdditionalDerivedKeys;

CK\_DERIVED\_KEY\_PTR 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\_SP800\_108\_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\_PTR 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\_KEY 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 195, 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 196, 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 197, 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\_INVALID\_HANDLE 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\_LENGTH\_FORMAT dkmFormat

= {CK\_SP800\_108\_DKM\_LENGTH\_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 **=**

{

CKM\_AES\_CMAC,

DIM(dataParams),

&dataParams,

0, */\* no addition derived keys \*/*

NULL */\* no addition derived keys \*/*

};

CK\_MECHANISM **=** mechanism

{

CKM\_SP800\_108\_COUNTER\_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\_LENGTH\_FORMAT dkmFormat

= {CK\_SP800\_108\_DKM\_LENGTH\_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 **=**

{

CKM\_AES\_CMAC,

DIM(dataParams),

&dataParams,

0, */\* no addition derived keys \*/*

NULL */\* no addition derived keys \*/*

};

CK\_MECHANISM **=** mechanism

{

CKM\_SP800\_108\_COUNTER\_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\_LENGTH\_FORMAT dkmFormat

= {CK\_SP800\_108\_DKM\_LENGTH\_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 **=**

{

CKM\_AES\_CMAC,

DIM(dataParams),

&dataParams,

ulFeedbackIVLen,

baFeedbackIV,

0, */\* no addition derived keys \*/*

NULL */\* no addition derived keys \*/*

};

CK\_MECHANISM **=** mechanism

{

CKM\_SP800\_108\_FEEDBACK\_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\_LENGTH\_FORMAT dkmFormat

= {CK\_SP800\_108\_DKM\_LENGTH\_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 **=**

{

CKM\_AES\_CMAC,

DIM(dataParams),

&dataParams,

0, */\* no addition derived keys \*/*

NULL */\* no addition derived keys \*/*

};

CK\_MECHANISM **=** mechanism

{

CKM\_SP800\_108\_DOUBLE\_PIPELINE\_KDF,

&kdfParams,

sizeof(kdfParams)

};

hBaseKey **=** GetBaseKeyHandle(**.....**);

rv = C\_DeriveKey(

hSession,

&mechanism,

hBaseKey,

&derivedKeyTemplate,

DIM(derivedKeyTemplate),

&hDerivedKey);

## Miscellaneous simple key derivation mechanisms

*Table 198, 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 199, 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 200, 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. See [BLOWFISH] for details.

*Table 201, 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 202, 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 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 203, 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 204, 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

Twofish is a secret key block cipher. See [TWOFISH] for details.

### 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 205, 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 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 206, 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 207, 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 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 208, 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 209, 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, short Weierstrass EC and DSA private keys (see Section 6.7 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 210, 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 211, 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 212, 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 213, 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 214, 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 215, 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 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 216, 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 217, 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, short Weierstrass EC and DSA private keys (see Section 6.7 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 218, 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 219, 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 220, 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 221, 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 222, 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 223, SEED Secret Key Object Attributes

| **Attribute** | **Data type** | **Meaning** |
| --- | --- | --- |
| CKA\_VALUE1,4,6,7 | Byte array | Key value (always 16 bytes long) |

- Refer to 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.

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\_BYTE iv[16];

CK\_BYTE\_PTR pData;

CK\_ULONG length;

} CK\_SEED\_CBC\_ENCRYPT\_DATA\_PARAMS;

typedef CK\_SEED\_CBC\_ENCRYPT\_DATA\_PARAMS CK\_PTR CK\_SEED\_CBC\_ENCRYPT\_DATA\_PARAMS\_PTR;

### Mechanism Parameters

Table 224, 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\_SEED\_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 2 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 3 illustrates the server-side equivalent of the scenario depicted in Figure 2. 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 4 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 225: 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 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 226: 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\_OTP\_PARAM\_TYPE

**CK\_OTP\_PARAM\_TYPE** is a value that identifies an OTP parameter type. It is defined as follows:

typedef CK\_ULONG CK\_OTP\_PARAM\_TYPE;

The following **CK\_OTP\_PARAM\_TYPE** types are defined:

Table 227, 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\_OUTPUT\_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 228: 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\_OTP\_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\_REQUIREMENT, 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\_OUTPUT\_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 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 229, 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 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 : PKCS #11 and CT-KIP integration

Figure 5 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 230: 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)

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 231, 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 232, 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 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 233, 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 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 234, 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 235, 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 236, 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 237, 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 238, 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 239, 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 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 240, 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 241, 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 242, 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\_GOSTR3411 |  | ✓ |  |  |  |  |  |
| 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 243, 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 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 244, 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\_PARAMS4,6,8 | 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 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 245, 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 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\_KEY\_WRAP\_PARAMS\_PTR is a pointer to a CK\_GOSTR3410\_KEY\_WRAP\_PARAMS.

♦ **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.

CK\_GOSTR3410\_DERIVE\_PARAMS\_PTR is a pointer to a CK\_GOSTR3410\_DERIVE\_PARAMS.

### 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 246, 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 247, 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 6.57.5.

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 248, 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\_CHACHA20) 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 249, 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.

**CK\_CHACHA20\_PARAMS\_PTR** is a pointer to **CK\_CHACHA20\_PARAMS.**

### 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 252, 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\_SALSA20) 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 253, 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\_SALSA20\_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.

**CK\_SALSA20\_PARAMS\_PTR** is a pointer to **CK\_SALSA20\_PARAMS.**

### 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 256, 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

### 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 257, 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.

*Table 259, Poly1305 Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Functions** | | | | | | |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_CHACHA20\_POLY1305 | ✓ |  |  |  |  |  |  |
| CKM\_SALSA20\_POLY1305 | ✓ |  |  |  |  |  |  |

### 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\_POLY1305\_PARAMS for Encrypt, Decrypt and CK\_SALSA20\_CHACHA20\_POLY1305\_MSG\_PARAMS 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 260, 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\_BBOOL bExtract;

CK\_BBOOL bExpand;

CK\_MECHANISM\_TYPE prfHashMechanism;

CK\_ULONG ulSaltType;

CK\_BYTE\_PTR pSalt;

CK\_ULONG ulSaltLen;

CK\_OBJECT\_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 6.62.2.

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. Tokens may reject requests based on the *pInfo* values. Allowed *pInfo* values are 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\_KEY\_GEN generates a new random HKDF key. CKA\_VALUE\_LEN must be set in the template.

## NULL Mechanism

**CKM\_NULL** is a mechanism used to implement the trivial pass-through function.

*Table 261, 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.

## IKE Mechanisms

*Table 262, IKE Mechanisms vs. Functions*

|  | **Functions** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Encrypt**  **&**  **Decrypt** | **Sign**  **&**  **Verify** | **SR**  **&**  **VR**1 | **Digest** | **Gen.**  **Key/**  **Key**  **Pair** | **Wrap**  **&**  **Unwrap** | **Derive** |
| CKM\_IKE2\_PRF\_PLUS\_DERIVE |  |  |  |  |  |  |  |
| CKM\_IKE\_PRF\_DERIVE |  |  |  |  |  |  |  |
| CKM\_IKE1\_PRF\_DERIVE |  |  |  |  |  |  |  |
| CKM\_IKE1\_EXTENDED\_DERIVE |  |  |  |  |  |  |  |

### Definitions

Mechanisms:

CKM\_IKE2\_PRF\_PLUS\_DERIVE

CKM\_IKE\_PRF\_DERIVE

CKM\_IKE1\_PRF\_DERIVE

CKM\_IKE1\_EXTENDED\_DERIVE

### IKE mechanism parameters

1. CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS; CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS\_PTR

**CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_IKE2\_PRF\_PLUS\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS {

CK\_MECHANISM\_TYPE prfMechanism;

CK\_BBOOL bHasSeedKey;

CK\_OBJECT\_HANDLE hSeedKey;

CK\_BYTE\_PTR pSeedData;

CK\_ULONG ulSeedDataLen;

} CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

prfMechanism underlying MAC mechanism used to generate the prf

bHasSeedKey hSeed key is present

hSeedKey optional seed from key

pSeedData optional seed from data

ulSeedDataLen length of optional seed data. If no seed data is present this value is 0

**CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS.**

1. CK\_IKE\_PRF\_DERIVE\_PARAMS; CK\_IKE\_PRF\_DERIVE\_PARAMS\_PTR

**CK\_IKE\_PRF\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_IKE\_PRF\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_IKE\_PRF\_DERIVE\_PARAMS {

CK\_MECHANISM\_TYPE prfMechanism;

CK\_BBOOL bDataAsKey;

CK\_BBOOL bRekey;

CK\_BYTE\_PTR pNi;

CK\_ULONG ulNiLen;

CK\_BYTE\_PTR pNr;

CK\_ULONG ulNrLen;

CK\_OBJECT\_HANDLE hNewKey;

} CK\_IKE\_PRF\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

prfMechanism underlying MAC mechanism used to generate the prf

bDataAsKey Ni||Nr is used as the key for the prf rather than baseKey

bRekey rekey operation. hNewKey must be present

pNi Ni value

ulNiLen length of Ni

pNr Nr value

ulNrLen length of Nr

hNewKey New key value to drive the rekey.

**CK\_IKE\_PRF\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_IKE\_PRF\_DERIVE\_PARAMS**.

1. CK\_IKE1\_PRF\_DERIVE\_PARAMS; CK\_IKE1\_PRF\_DERIVE\_PARAMS\_PTR

**CK\_IKE1\_PRF\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_IKE1\_PRF\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_IKE1\_PRF\_DERIVE\_PARAMS {

CK\_MECHANISM\_TYPE prfMechanism;

CK\_BBOOL bHasPrevKey;

CK\_OBJECT\_HANDLE hKeygxy;

CK\_OBJECT\_HANDLE hPrevKey;

CK\_BYTE\_PTR pCKYi;

CK\_ULONG ulCKYiLen;

CK\_BYTE\_PTR pCKYr;

CK\_ULONG ulCKYrLen;

CK\_BYTE keyNumber;

} CK\_IKE1\_PRF\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

prfMechanism underlying MAC mechanism used to generate the prf

bHasPrevkey hPrevKey is present

hKeygxy handle to the exchanged g^xy key

hPrevKey handle to the previously derived key

pCKYi CKYi value

ulCKYiLen length of CKYi

pCKYr CKYr value

ulCKYrLen length of CKYr

keyNumber unique number for this key derivation

**CK\_IKE1\_PRF\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_IKE1\_PRF\_DERIVE\_PARAMS**.

1. CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS; CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS\_PTR

**CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS** is a structure that provides the parameters to the **CKM\_IKE1\_EXTENDED\_DERIVE** mechanism. It is defined as follows:

typedef struct CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS {

CK\_MECHANISM\_TYPE prfMechanism;

CK\_BBOOL bHasKeygxy;

CK\_OBJECT\_HANDLE hKeygxy;

CK\_BYTE\_PTR pExtraData;

CK\_ULONG ulExtraDataLen;

} CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS;

The fields of the structure have the following meanings:

prfMechanism underlying MAC mechanism used to generate the prf

bHasKeygxy hKeygxy key is present

hKeygxy optional key g^xy

pExtraData optional extra data

ulExtraDataLen length of optional extra data. If no extra data is present this value is 0

**CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS\_PTR** is a pointer to a **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS**.

### IKE PRF DERIVE

The IKE PRF Derive mechanism denoted **CKM\_IKE\_PRF\_DERIVE** is used in IPSEC both IKEv1 and IKEv2 to generate an initial key that is used to generate additional keys. It takes a **CK\_IKE\_PRF\_DERIVE\_PARAMS** as a mechanism parameter. *baseKey* is the base key passed into **C\_DeriveKey**. *baseKey* must be of type **CKK\_GENERIC\_SECRET** if *bDataAsKey* is TRUE and the key type of the underlying prf if *bDataAsKey* is FALSE. *hNewKey* must be of type **CKK\_GENERIC\_SECRET**. Depending on the parameter settings, it generates keys with a **CKA\_VALUE** of:

1. prf(pNi|pNr, baseKey); (bDataAsKey=TRUE, bRekey=FALSE)

2. prf(baseKey, pNi|pNr); (bDataAsKkey=FALSE, bRekey=FALSE)

3. prf(baseKey, ValueOf(hNewKey)| pNi | pNr); (bDataAsKey=FALSE, bRekey=TRUE)

The resulting output key is always the length of the underlying prf. The combination of *bDataAsKey*=TRUE and *bRekey*=TRUE is not allowed. If both are set, **CKR\_ARGUMENTS\_BAD** is returned.

Case 1 is used in

a. ikev2 (RFC 5996) *baseKey* is called g^ir, the output is called SKEYSEED

b. ikev1 (RFC 2409) *baseKey* is called g^ir, the output is called SKEYID

Case 2 is used in ikev1 (RFC 2409) inkey is called pre-shared-key, output is called SKEYID

Case 3 is used in ikev2 (RFC 5996) rekey case, *baseKey* is SK\_d, hNewKey is g^ir (new), the output is called SKEYSEED. The derived key will have a length of the length of the underlying prf. If **CKA\_VALUE\_LEN** is specified, it must equal the underlying prf or **CKR\_KEY\_SIZE\_RANGE** is returned. If **CKA\_KEY\_TYPE** is not specified in the template, it will be the underlying key type of the prf.

### IKEv1 PRF DERIVE

The IKEv1 PRF Derive mechanism denoted **CKM\_IKE1\_PRF\_DERIVE** is used in IPSEC IKEv1 to generate various additional keys from the initial SKEYID. It takes a **CK\_IKE1\_PRF\_DERIVE\_PARAMS** as a mechanism parameter. SKEYID is the base key passed into **C\_DeriveKey**.

This mechanism derives a key with **CKA\_VALUE** set to either:

prf(baseKey, ValueOf(hKeygxy) || pCKYi || pCKYr || key\_number)

or

prf(baseKey, ValueOf(hPrevKey) || ValueOf(hKeygxy) || pCKYi || pCKYr || key\_number)

depending on the state of *bHasPrevKey*.

The key type of *baseKey* must be the key type of the prf, and the key type of *hKeygxy* must be **CKK\_GENERIC\_SECRET**. The key type of *hPrevKey* can be any key type.

This is defined in RFC 2409. For each of the following keys.

*baseKey* is SKEYID, *hKeygxy* is g^xy

for outKey = SKEYID\_d, *bHasPrevKey* = false, key\_number = 0

for outKey = SKEYID\_a, *hPrevKey*= SKEYID\_d, key\_number = 1

for outKey = SKEYID\_e, *hPrevKey*= SKEYID\_a, key\_number = 2

If **CKA\_VALUE\_LEN** is not specified, the resulting key will be the length of the prf. If **CKA\_VALUE\_LEN** is greater then the prf, **CKR\_KEY\_SIZE\_RANGE** is returned. If it is less the key is truncated taking the left most bytes. The value **CKA\_KEY\_TYPE** must be specified in the template or **CKR\_TEMPLATE\_INCOMPLETE** is returned.

### IKEv2 PRF PLUS DERIVE

The IKEv2 PRF PLUS Derive mechanism denoted **CKM\_IKE2\_PRF\_PLUS\_DERIVE** is used in IPSEC IKEv2 to derive various additional keys from the initial SKEYSEED. It takes a **CK\_IKE2\_PRF\_PLUS\_DERIVE\_PARAMS** as a mechanism parameter. SKEYSEED is the base key passed into **C\_DeriveKey**. The key type of *baseKey* must be the key type of the underlying prf. This mechanism uses the base key and a feedback version of the prf to generate a single key with sufficient bytes to cover all additional keys. The application will then use **CKM\_EXTRACT\_KEY\_FROM\_KEY** several times to pull out the various keys. **CKA\_VALUE\_LEN** must be set in the template and its value must not be bigger than 255 times the size of the prf function output or **CKR\_KEY\_SIZE\_RANGE** will be returned. If **CKA\_KEY\_TYPE** is not specified, the output key type will be **CKK\_GENERIC\_SECRET**.

This mechanism derives a key with a **CKA\_VALUE** of (from RFC 5996):

prfplus = T1 | T2 | T3 | T4 |... Tn

where:

T1 = prf(K, S | 0x01)

T2 = prf(K, T1 | S | 0x02)

T3 = prf(K, T3 | S | 0x03)

T4 = prf(K, T4 | S | 0x04)

.

Tn = prf(K, T(n-1) | n)

K = *baseKey*, S = valueOf(*hSeedKey*) | *pSeedData*

### IKEv1 Extended Derive

The IKE Extended Derive mechanism denoted **CKM\_IKE1\_EXTENDED\_DERIVE** is used in IPSEC IKEv1 to derive longer keys than **CKM\_IKE1\_EXTENDED\_DERIVE** can from the initial SKEYID. It is used to support RFC 2409 appendix B and RFC 2409 section 5.5 (Quick Mode). It takes a **CK\_IKE1\_EXTENDED\_DERIVE\_PARAMS** as a mechanism parameter. SKEYID is the base key passed into **C\_DeriveKey**. **CKA\_VALUE\_LEN** must be set in the template and its value must not be bigger than 255 times the size of the prf function output or **CKR\_KEY\_SIZE\_RANGE** will be returned. If **CKA\_KEY\_TYPE** is not specified, the output key type will be **CKK\_GENERIC\_SECRET**. The key type of SKEYID must be the key type of the prf, and the key type of *hKeygxy* (if present) must be **CKK\_GENERIC\_SECRET**.

This mechanism derives a key with **CKA\_VALUE** (from RFC 2409 appendix B and section 5.5):

Ka = K1 | K2 | K3 | K4 |... Kn

where:

K1 = prf(K, valueOf(*hKeygxy*)|*pExtraData*) or prf(K,0x00) if *bHashKeygxy* is FALSE and *ulExtraData* is 0

K2 = prf(K, K1|valueOf(*hKeygxy*)|*pExtraData*)

K3 = prf(K, K2|valueOf(*hKeygxy*)|*pExtraData*)

K4 = prf(K, K3|valueOf(*hKeygxy*)|*pExtraData*)

.

Kn = prf(K, K(n-1)|valueOf(*hKeygxy*)|*pExtraData*)

K = *baseKey*

If **CKA\_VALUE\_LEN** is less then or equal to the prf length and bHasKeygxy is CK\_FALSE, then the new key is simply the base key truncated to **CKA\_VALUE\_LEN** (specified in RFC 2409 appendix B). Otherwise the prf is executed and the derived keys value is **CKA\_VALUE\_LEN** bytes of the resulting prf.

## HSS

HSS is a mechanism for single-part signatures and verification, following the digital signature algorithm defined in [RFC 8554] and [NIST 802-208].

*Table 263, HSS Mechanisms vs. Functions*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Mechanism** | **Functions** | | | | | | |
| **Encrypt & Decrypt** | **Sign & Verify** | **SR & VR** | **Digest** | **Gen. Key/ Key Pair** | **Wrap & Unwrap** | **Derive** |
| CKM\_HSS\_KEY\_PAIR\_GEN |  |  |  |  |  |  |  |
| CKM\_HSS |  | 1 |  |  |  |  |  |

1 Single-part operations only

### Definitions

This section defines the key type **CKK\_HSS** for type **CK\_KEY\_TYPE** as used in the **CKA\_KEY\_TYPE** attribute of key objects and domain parameter objects.

Mechanisms:

CKM\_HSS\_KEY\_PAIR\_GEN

CKM\_HSS

### HSS public key objects

HSS public key objects (object class **CKO\_PUBLIC\_KEY,** key type **CKK\_HSS**) hold HSS public keys.

The following table defines the HSS public key object attributes, in addition to the common attributes defined for this object class:

Table 264, *HSS Public Key Object Attributes*

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_HSS\_LEVELS2,4 | CK\_ULONG | The number of levels in the HSS scheme. |
| CKA\_HSS\_LMS\_TYPE2,4 | CK\_ULONG | The encoding for the Merkle tree heights of the top level LMS tree in the hierarchy. |
| CKA\_HSS\_LMOTS\_TYPE2,4 | CK\_ULONG | The encoding for the Winternitz parameter of the one-time-signature scheme of the top level LMS tree. |
| CKA\_VALUE1,4 | Byte array | XDR-encoded public key as defined in [RFC8554]. |

- Refer to Table 11 for footnotes

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

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

CK\_KEY\_TYPE keyType = CKK\_HSS;

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

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

CK\_BBOOL true = CK\_TRUE;

CK\_BBOOL false = CK\_FALSE;

CK\_ATTRIBUTE template[] = {

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

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

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

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

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

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

};

### HSS private key objects

HSS private key objects (object class **CKO\_PRIVATE\_KEY,** key type **CKK\_HSS**) hold HSS private keys.

The following table defines the HSS private key object attributes, in addition to the common attributes defined for this object class:

*Table 265, HSS Private Key Object Attributes*

| **Attribute** | **Data Type** | **Meaning** |
| --- | --- | --- |
| CKA\_HSS\_LEVELS1,3 | CK\_ULONG | The number of levels in the HSS scheme. |
| CKA\_HSS\_LMS\_TYPES1,3 | CK\_ULONG\_PTR | A list of encodings for the Merkle tree heights of the LMS trees in the hierarchy from top to bottom. The number of encodings in the array is the ulValueLen component of the attribute divided by the size of CK\_ULONG. This number must match the CKA\_HSS\_LEVELS attribute value. |
| CKA\_HSS\_LMOTS\_TYPES1,3 | CK\_ULONG\_PTR | A list of encodings for the Winternitz parameter of the one-time-signature scheme of the LMS trees in the hierarchy from top to bottom. The number of encodings in the array is the ulValueLen component of the attribute divided by the size of CK\_ULONG. This number must match the CKA\_HSS\_LEVELS attribute value. |
| CKA\_VALUE1,4,6,7 | Byte array | Vendor defined, must include state information.  Note that exporting this value is dangerous as it would allow key reuse. |
| CKA\_HSS\_KEYS\_REMAINING2,4 | CK\_ULONG | The minimum of the following two values: 1) The number of one-time private keys remaining; 2) 2^32-1 |

- Refer to Table 11 for footnotes

The encodings for CKA\_HSS\_LMOTS\_TYPES and CKA\_HSS\_LMS\_TYPES are defined in [RFC 8554] and [NIST 802-208].

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

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

CK\_KEY\_TYPE keyType = CKK\_HSS;

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

CK\_ULONG hssLevels = 123;

CK\_ULONG lmsTypes[] = {123,...};

CK\_ULONG lmotsTypes[] = {123,...};

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

CK\_BBOOL true = CK\_TRUE;

CK\_BBOOL false = CK\_FALSE;

CK\_ATTRIBUTE template[] = {

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

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

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

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

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

{CKA\_EXTRACTABLE, &false, sizeof(true)},

    {CKA\_HSS\_LEVELS, &hssLevels, sizeof(hssLevels)},

    {CKA\_HSS\_LMS\_TYPES, lmsTypes, sizeof(lmsTypes)},

    {CKA\_HSS\_LMOTS\_TYPES, lmotsTypes, sizeof(lmotsTypes)},

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

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

};

**CKA\_SENSITIVE** MUST be true, **CKA\_EXTRACTABLE** MUST be false, and **CKA\_COPYABLE** MUST be false for this key.

### HSS key pair generation

The HSS key pair generation mechanism, denoted **CKM\_HSS\_KEY\_PAIR\_GEN**, is a key pair generation mechanism for HSS.

This mechanism does not have a parameter.

The mechanism generates HSS public/private key pairs for the scheme specified by the **CKA\_HSS\_LEVELS**, **CKA\_HSS\_LMS\_TYPES**, and **CKA\_HSS\_LMOTS\_TYPES** attributes of the template for the private key.

The mechanism contributes the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_HSS\_LEVELS, CKA\_HSS\_LMS\_TYPE**, **CKA\_HSS\_LMOTS\_TYPE**, and **CKA\_VALUE** attributes to the new public key and the **CKA\_CLASS**, **CKA\_KEY\_TYPE**, **CKA\_VALUE**, and **CKA\_HSS\_KEYS\_REMAINING** attributes to the new private key.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used and must be set to 0.

### HSS without hashing

The HSS without hashing mechanism, denoted **CKM\_HSS**, is a mechanism for single-part signatures and verification for HSS. (This mechanism corresponds only to the part of LMS that processes the hash value, which may be of any length; it does not compute the hash value.)

This mechanism does not have a parameter.

For the purposes of these mechanisms, an HSS signature is a byte string with length depending on **CKA\_HSS\_LEVELS, CKA\_HSS\_LMS\_TYPES**, **CKA\_HSS\_LMOTS\_TYPES** as described in the following table.

Table 266, *HSS without hashing: Key and Data Length*

| **Function** | **Key type** | **Input length** | **Output length** |
| --- | --- | --- | --- |
| C\_Sign1 | HSS Private Key | any | 1296-749882 |
| C\_Verify1 | HSS Public Key | any, 1296-749882 | N/A |

1 Single-part operations only.  
2 4+(*levels*-1)\*56+*levels*\*(8+(36+32\**p*)+*h*\*32) where *p* has values (265, 133, 67, 34) for lmots type (W1, W2, W4, W8) and *h* is the number of levels in the LMS Merkle trees.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK\_MECHANISM\_INFO** structure are not used and must be set to 0.

If the number of signatures is exhausted, CKR\_KEY\_EXHAUSTED will be returned.

# PKCS #11 Implementation Conformance

## PKCS#11 Consumer Implementation Conformance

An implementation is a conforming PKCS#11 Consumer if the implementation meets the conditions specified in one or more consumer profiles specified in **[PKCS11-Prof]**.

A PKCS#11 consumer implementation SHALL be a conforming PKCS#11 Consumer.

If a PKCS#11 consumer implementation claims support for a particular consumer 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.

## PKCS#11 Provider Implementation Conformance

An implementation is a conforming PKCS#11 Provider if the implementation meets the conditions specified in one or more provider profiles specified in **[PKCS11-Prof]**.

A PKCS#11 provider implementation SHALL be a conforming PKCS#11 Provider.

1. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

| Salutation | First Name | Last Name | Company |
| --- | --- | --- | --- |
| Dr. | Warren | Armstrong | QuintessenceLabs Pty Ltd. |
|  | Anthony | Berglas | Cryptsoft Pty Ltd. |
| Mr. | Dieter | Bong | Utimaco IS GmbH |
| Mr. | Roland | Bramm | PrimeKey Solutions AB |
|  | Andrew | Byrne | Dell |
|  | Hamish | Cameron | nCipher |
|  | Kenli | Chong | QuintessenceLabs Pty Ltd. |
| Mr. | Justin | Corlett | Cryptsoft Pty Ltd. |
|  | Xuelei | Fan | Oracle |
| Mr. | Jan | Friedel | Oracle |
| Ms. | Susan | Gleeson | Oracle |
| Mr. | Thomas | Hardjono | M.I.T. |
| Mrs. | Jane | Harnad | OASIS |
|  | David | Horton | Dell |
|  | Tim | Hudson | Cryptsoft Pty Ltd. |
| Mr. | Gershon | Janssen | Individual |
| Mr. | Jakub | Jelen | Red Hat |
| Dr. | Mark | Joseph | P6R, Inc |
| Mr. | Paul | King | nCipher |
| Ms. | Dina | Kurktchi-Nimeh | Oracle |
|  | John | Leiseboer | QuintessenceLabs Pty Ltd. |
| Mr. | John | Leser | Oracle |
|  | Chris | Malafis | Red Hat |
| Dr. | Michael | Markowitz | Information Security Corporation |
| Mr. | Scott | Marshall | Cryptsoft Pty Ltd. |
| Mr. | Chris | Meyer | Utimaco IS GmbH |
| Mr. | Darren | Moffat | Oracle |
| Dr. | Florian | Poppa | QuintessenceLabs Pty Ltd. |
|  | Roland | Reichenberg | Utimaco IS GmbH |
| Mr. | Robert | Relyea | Red Hat |
| Mr. | Jonathan | Schulze-Hewett | Information Security Corporation |
| Mr. | Greg | Scott | Cryptsoft Pty Ltd. |
| Mr. | Martin | Shannon | QuintessenceLabs Pty Ltd. |
| Mr. | Oscar | So | Individual |
|  | Patrick | Steuer | IBM |
| Mr. | Gerald | Stueve | Fornetix |
|  | Jim | Susoy | P6R, Inc |
| Mr. | Sander | Temme | nCipher |
| Mr. | Manish | Upasani | Utimaco IS GmbH |
| Mr. | Charles | White | Fornetix |
| Ms. | Magda | Zdunkiewicz | Cryptsoft Pty Ltd. |

1. Manifest constants

The definitions for manifest constants specified in this document can be found in the Additional artifacts section.

1. Revision History

| **Revision** | **Date** | **Editor** | **Changes Made** |
| --- | --- | --- | --- |
| CSD02 WD01 | 12 May 2022 | Dieter Bong | Changes made compared to Committee Specification CSD01, as working draft of Committee Specification CSD02  - Editorial changes resolving comments by Paul Knight, OASIS, <https://lists.oasis-open.org/archives/pkcs11-comment/202203/msg00001.html>  - Reference [PKCS11-curr] replaced by reference within document  - Correction of typos |
|  |  |  |  |

1. Notices

Copyright © OASIS Open 2023. 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: [<https://www.oasis-open.org/policies-guidelines/ipr/>].

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 AND ITS MEMBERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THIS DOCUMENT OR ANY PART THEREOF.

As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards Final Deliverable documents (Committee Specifications, OASIS Standards, or Approved Errata).

[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 Standards Final Deliverable, 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 deliverable.]

[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 OASIS Standards Final Deliverable 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 OASIS Standards Final Deliverable. 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 OASIS Standards Final Deliverable 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 Standards Final Deliverable, 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 document, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, documents, while reserving the right to enforce its marks against misleading uses. Please see <https://www.oasis-open.org/policies-guidelines/trademark/> for above guidance.

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)