THALES

Thales Luna PCIe HSM 7

SDK REFERENCE

Document Information

Last Updated	2024-09-10 13:48:42 GMT-04:00

Trademarks, Copyrights, and Third-Party Software

Copyright 2001-2024 Thales Group. All rights reserved. Thales and the Thales logo are trademarks and service marks of Thales and/or its subsidiaries and are registered in certain countries. All other trademarks and service marks, whether registered or not in specific countries, are the property of their respective owners.

Disclaimer

All information herein is either public information or is the property of and owned solely by Thales Group and/or its subsidiaries who shall have and keep the sole right to file patent applications or any other kind of intellectual property protection in connection with such information.

Nothing herein shall be construed as implying or granting to you any rights, by license, grant or otherwise, under any intellectual and/or industrial property rights of or concerning any of Thales Group's information.

This document can be used for informational, non-commercial, internal, and personal use only provided that:

> The copyright notice, the confidentiality and proprietary legend and this full warning notice appear in all copies.

> This document shall not be posted on any publicly accessible network computer or broadcast in any media, and no modification of any part of this document shall be made.

Use for any other purpose is expressly prohibited and may result in severe civil and criminal liabilities.

The information contained in this document is provided "AS IS" without any warranty of any kind. Unless otherwise expressly agreed in writing, Thales Group makes no warranty as to the value or accuracy of information contained herein.

The document could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Furthermore, Thales Group reserves the right to make any change or improvement in the specifications data, information, and the like described herein, at any time.

Thales Group hereby disclaims all warranties and conditions with regard to the information contained herein, including all implied warranties of merchantability, fitness for a particular purpose, title and non-infringement. In no event shall Thales Group be liable, whether in contract, tort or otherwise, for any indirect, special or consequential damages or any damages whatsoever including but not limited to damages resulting from loss of use, data, profits, revenues, or customers, arising out of or in connection with the use or performance of information contained in this document.

Thales Group does not and shall not warrant that this product will be resistant to all possible attacks and shall not incur, and disclaims, any liability in this respect. Even if each product is compliant with current security standards in force on the date of their design, security mechanisms' resistance necessarily evolves according to the state of the art in security and notably under the emergence of new attacks. Under no circumstances, shall Thales Group be held liable for any third party actions and in particular in case of any successful attack against systems or equipment incorporating Thales products. Thales Group disclaims any liability with respect to security for direct, indirect, incidental or consequential damages that result from any use of its products. It is further stressed

that independent testing and verification by the person using the product is particularly encouraged, especially in any application in which defective, incorrect or insecure functioning could result in damage to persons or property, denial of service, or loss of privacy.

All intellectual property is protected by copyright. All trademarks and product names used or referred to are the copyright of their respective owners. No part of this document may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, chemical, photocopy, recording or otherwise without the prior written permission of Thales Group.

Regulatory Compliance

This product complies with the following regulatory regulations. To ensure compliancy, ensure that you install the products as specified in the installation instructions and use only Thales-supplied or approved accessories.

USA, FCC

This equipment has been tested and found to comply with the limits for a “Class B” digital device, pursuant to part 15 of the FCC rules.

Canada

This class B digital apparatus meets all requirements of the Canadian interference-causing equipment regulations.

Europe

This product is in conformity with the protection requirements of EC Council Directive 2014/30/EU. This product satisfies the CLASS B limits of EN55032.

CONTENTS

Preface: About the SDK Reference 17
- Customer Release Notes 17
- Audience 17
- Document Conventions 18
- Support Contacts 20

Chapter 1: Luna SDK Overview 21
- Supported Cryptographic Algorithms 21
- Application Programming Interface (API) Overview 21
  - Sample Application 23
  - A Note About RSA Key Attributes ‘p’ and ‘q’ 23
- Supported Integrations 23
  - Why Is an Integration Not Listed Here Or On the Website? 24

Chapter 2: PKCS#11 Support 25
- PKCS#11 Compliance 25
  - Supported PKCS#11 Services 25
  - Key Check Values 29
  - Additional Functions 29
- Using the PKCS#11 Sample 29
  - The SfntLibPath Environment Variable 30
  - What p11Sample Does 30

Chapter 3: Extensions to PKCS#11 32
- Luna Extensions to PKCS#11 32
  - Firmware Dependencies 32
  - Other APIs 33
  - Cryptoki Version Supported 33
  - Luna Extensions 33
- HSM Configuration Settings 43
- Secure PIN Port Authentication 44
- High Availability Indirect Login 44
  - Overview of High Availability Indirect Login 44
  - Supported HSM Roles 45
  - High Availability Indirect Login Functions Prior to Luna HSM Firmware 7.7.0 46
  - High Availability Indirect Login For HSM Firmware 7.7.0 and Newer 50
- MofN Secret Sharing (quorum or multi-person access control) 59
  - Setting up 59
  - Using 60
- Key Export Features 60
  - RSA Key Component Wrapping 60

Luna HSM Cloning API CPv1 - Extensions to PKCS #11 ... 63
Cloning on the Same Host System ... 63
Cloning between Host Systems ... 63
Luna HSM Cloning API CPv3 - Extensions to PKCS #11 ... 65
Cloning on the Same Host System ... 65
Cloning between Host Systems ... 66
IMPORTANT CONSIDERATIONS ... 68
Luna HSM Cloning API CPv4 Extensions to PKCS#11 ... 69
Top Level API ... 70
Low-Level APIs ... 71
New PKCS#11 Error Code Summary ... 77
Co-existence with Existing PKCS#11 APIs ... 79
PSK APIs ... 79
PKCS#11 Extension APIs ... 79
Error Codes ... 82
Secure External Scalable Key Storage Extensions ... 83
CA_SIMExtract ... 83
CA_SIMInsert ... 84
CA_SIM_MultiSign ... 84
CA_SMKRollover ... 85
Derivation of Symmetric Keys with 3DES_ECB ... 85
PKCS#11 Extension HA Status Call ... 85
Function Definition ... 86
PKCS#11 Extension HA Status Call - improved ... 86
Function Definition ... 87
ECIES Enhancement for HKDF ... 87
RULES FOR CK_ECIES_PARAMS_EXT3 USE WITH HKDF ... 88
CK_ECIES_PARAMS_EXT AND CK_ECIES_PARAMS ... 89
DECRYPTION ... 89
ENCRYPTION ... 90
TOOLS ... 90
BIP32 Mechanism Support and Implementation ... 91
Curve Support ... 91
Key Type and Form ... 91
Extended Keys and Hardened Keys ... 92
Key Derivation ... 92
Error Codes ... 93
Key Attributes ... 93
Public Key Import/Export ... 94
Private Key Import/Export ... 95
Key Backup and Cloning ... 95
Non-FIPS Algorithm ... 95
Host Tools ... 96
Code Samples ... 96
SLIP 10 ... 99
SLIP10 BIP32 Master Key derivation and Child Key derivation ... 99
Caveats ... 100

Curve Support 100
Hardened Keys 100
Key Derivation 100
3GPP Mechanisms for 5G Mobile Networks 103
MILENAGE 104
TUAK 106
Comp128 107
Storage Key (SK) 107
Luna Key Translation 107
Mechanism Description 108
CKM_KEY_TRANSLATE 108
Tooling 110
Derive Template 110
Examples 110
Counter Mode KDF Mechanisms 112
SM2/SM4 Mechanisms 113
SM2 113
SM4 114
SHA-3 Mechanisms 115
Digest Mechanisms 116
HMAC Mechanisms 116
Signature/Verification Mechanisms 117
Encrypt/Decrypt Mechanisms 118
Digest Key Derive Mechanisms 118
Key Derivation Function (KDF) Mechanisms 119

Chapter 4: Per-Key Authorization API 120
Design 120
The Luna use-case 120
The eIDAS use-case 120
New Assigned Key Attribute 121
New Authorization Data Attribute 122
Initializing the Authorization Data 122
Modifying the Authorization Data 122
Resetting the Authorization Data 122
Authorizing/Rescinding Authorization on a Key per Session 123
Authorizing/Rescinding Authorization on a Key per Access 124
Per-Key Authorization Failure Handling 124
Template Handling in the Cryptoki Library 124
V0 vs V1 Partitions 125
Cryptoki API 125
Library/Tool Considerations 128
Tool Changes 128
High Availability (HA) 129
Migration Scenarios for Per-Key Auth 129
Import via Cloning 130
Import via Legacy SKS 130

Import via Unwrapping 130
Import via V0 to V1 Partition Conversion 130
Summary of New PKA Commands and Capabilities 131
V0 PARTITIONS 133
Firmware Update 133
Partition Creation 133
Converting from V0 to V1 (changing the policy) 134
Converting from V1 to V0 (changing the policy) 134
G5 Backup HSM and Cloning Protocol 135
eIDAS partitions 135
Limited Crypto Officer (LCO) role 136

Chapter 5: Supported Mechanisms 139
RSA Mechanism Remap for FIPS Compliance 139
Effects of Remapping in FIPS Mode 140
Applying the Mechanism Remapping 140
CKM_AES_CBC 142
Firmware 7.4.2 and Older Summary 142
CKM_AES_CBC_ENCRYPT_DATA 144
CKM_AES_CBC_PAD 145
Firmware 7.4.2 and Older Summary 145
CKM_AES_CBC_PAD_IPSEC 147
CKM_AES_CFB8 148
CKM_AES_CFB128 149
CKM_AES_CMAC 150
CKM_AES_CMAC_GENERAL 151
CKM_AES_CTR 152
Firmware 7.4.2 and Older Summary 152
CKM_AES_ECB 154
Firmware 7.4.2 and Older Summary 154
CKM_AES_ECB_ENCRYPT_DATA 156
CKM_AES_GCM 157
CKM_AES_GMAC 160
CKM_AES_KEY_GEN 162
CKM_AES_KW 163
CKM_AES_KWP 165
CKM_AES_MAC 166
Firmware 7.4.2 and Older Summary 167
CKM_AES_MAC_GENERAL 169
Firmware 7.4.2 and Older Summary 170
CKM_AES_OFB 172
CKM_AES_XTS 173
CKM_ARIA_CBC 174
CKM_ARIA_CBC_ENCRYPT_DATA 175
CKM_ARIA_CBC_PAD 176
CKM_ARIA_CFB8 177
CKM_ARIA_CFB128 178

CKM_ARIA_CMAC 179
CKM_ARIA_CMAC_GENERAL 180
CKM_ARIA_CTR 181
CKM_ARIA_ECB 182
CKM_ARIA_ECB_ENCRYPT_DATA 183
CKM_ARIA_KEY_GEN 184
CKM_ARIA_L_CBC 185
CKM_ARIA_L_CBC_PAD 186
CKM_ARIA_L_ECB 187
CKM_ARIA_L_MAC 188
CKM_ARIA_L_MAC_GENERAL 189
CKM_ARIA_MAC 190
CKM_ARIA_MAC_GENERAL 191
CKM_ARIA_OFB 192
CKM_BIP32_CHILD_DERIVE 193
CKM_BIP32_MASTER_DERIVE 194
CKM_CAST3_CBC 195
CKM_CAST3_CBC_PAD 196
CKM_CAST3_ECB 197
CKM_CAST3_KEY_GEN 198
CKM_CAST3_MAC 199
CKM_CAST3_MAC_GENERAL 200
CKM_CAST5_CBC 201
CKM_CAST5_CBC_PAD 202
CKM_CAST5_ECB 203
CKM_CAST5_KEY_GEN 204
CKM_CAST5_MAC 205
CKM_CAST5_MAC_GENERAL 206
CKM_COMP128 207
CKM_DES_CBC 208
CKM_DES_CBC_ENCRYPT_DATA 209
CKM_DES_CBC_PAD 210
CKM_DES_CFB8 211
CKM_DES_CFB64 213
CKM_DES_ECB 215
CKM_DES_ECB_ENCRYPT_DATA 216
CKM_DES_KEY_GEN 217
CKM_DES_MAC 218
CKM_DES_MAC_GENERAL 219
CKM_DES_OFB64 220
CKM_DES2_DUKPT_DATA 222
CKM_DES2_DUKPT_DATA_RESP 224
CKM_DES2_DUKPT_IPEK 226
CKM_DES2_DUKPT_MAC 227
CKM_DES2_DUKPT_MAC_RESP 229
CKM_DES2_DUKPT_PIN 231
CKM_DES2_KEY_GEN 233

CKM_DES3_CBC 234
Firmware 7.4.2 and Older Summary 235
CKM_DES3_CBC_ENCRYPT_DATA 237
CKM_DES3_CBC_PAD 239
Firmware 7.4.2 and Older Summary 240
CKM_DES3_CBC_PAD_IPSEC 242
CKM_DES3_CMAC 243
CKM_DES3_CMAC_GENERAL 245
CKM_DES3_CTR 247
Firmware 7.2.0-7.4.2 Summary 248
CKM_DES3_ECB 250
Firmware 7.4.2 and Older Summary 251
CKM_DES3_ECB_ENCRYPT_DATA 253
CKM_DES3_KEY_GEN 255
CKM_DES3_MAC 256
Firmware 7.4.2 and Older Summary 258
CKM_DES3_MAC_GENERAL 259
Firmware 7.4.2 and Older Summary 261
CKM_DES3_X919_MAC 262
CKM_DH_PKCS_DERIVE 265
CKM_DH_PKCS_KEY_PAIR_GEN 266
CKM_DH_PKCS_PARAMETER_GEN 267
CKM_DSA 268
CKM_DSA_KEY_PAIR_GEN 270
CKM_DSA_PARAMETER_GEN 272
CKM_DSA_SHA1 274
Firmware 7.4.2 and Older Summary 274
CKM_DSA_SHA224 276
CKM_DSA_SHA256 278
CKM_EC_EDWARDS_KEY_PAIR_GEN 280
CKM_EC_KEY_PAIR_GEN 282
CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS 283
CKM_EC_MONTGOMERY_KEY_PAIR_GEN 284
CKM_ECDH1_COFACTOR_DERIVE 286
CKM_ECDH1_DERIVE 288
CKM_ECDSA 290
CKM_ECDSA_GBCS_SHA256 292
CKM_ECDSA_SHA1 294
CKM_ECDSA_SHA3_512 296
CKM_ECDSA_SHA3_256 298
CKM_ECDSA_SHA3_384 300
CKM_ECDSA_SHA3_224 302
CKM_ECDSA_SHA224 304
CKM_ECDSA_SHA256 306
CKM_ECDSA_SHA384 308
CKM_ECDSA_SHA512 310
CKM_ECIES 312

CKM_EDDSA 314
CKM_EDDSA_NACL 317
CKM_GENERIC_SECRET_KEY_GEN 319
CKM_HAS160 320
CKM_KCDSA_HAS160 321
CKM_KCDSA_HAS160_NO_PAD 322
CKM_KCDSA_KEY_PAIR_GEN 323
CKM_KCDSA_PARAMETER_GEN 324
CKM_KCDSA_SHA1 325
CKM_KCDSA_SHA1_NO_PAD 326
CKM_KCDSA_SHA224 327
CKM_KCDSA_SHA224_NO_PAD 328
CKM_KCDSA_SHA256 329
CKM_KCDSA_SHA256_NO_PAD 330
CKM_KCDSA_SHA384 331
CKM_KCDSA_SHA384_NO_PAD 332
CKM_KCDSA_SHA512 333
CKM_KCDSA_SHA512_NO_PAD 334
CKM_KECCAK_224 335
CKM_KECCAK_256 336
CKM_KECCAK_384 337
CKM_KECCAK_512 338
CKM_KEY_TRANSLATE 339
CKM_KEY_WRAP_SET_OAEP 340
CKM_MD2 341
CKM_MD2_KEY_DERIVATION 342
CKM_MD5_HMAC 343
CKM_MD5_HMAC_GENERAL 345
CKM_MD5_KEY_DERIVATION 347
CKM_MILENAGE 348
CKM_MILENAGE_AUTS 349
CKM_MILENAGE_RESYNC 350
CKM_NIST_PRF_KDF 351
CKM_PBE_MD2_DES_CBC 353
CKM_PBE_SHA1_CAST5_CBC 354
CKM_PBE_SHA1_DES2_EDE_CBC 355
CKM_PBE_SHA1_DES3_EDE_CBC 356
CKM_PBE_SHA1_RC2_40_CBC 357
CKM_PBE_SHA1_RC2_128_CBC 358
CKM_PBE_SHA1_RC4_40 359
CKM_PBE_SHA1_RC4_128 360
CKM_PKCS5_PBKD2 361
CKM_PRF_KDF 362
CKM_RC2_CBC 364
CKM_RC2_CBC_PAD 365
CKM_RC2_ECB 366
CKM_RC2_KEY_GEN 367

CKM_RC2_MAC 368
CKM_RC2_MAC_GENERAL 369
CKM_RC4 370
CKM_RC4_KEY_GEN 371
CKM_RC5_CBC 372
CKM_RC5_CBC_PAD 373
CKM_RC5_ECB 374
CKM_RC5_KEY_GEN 375
CKM_RC5_MAC 376
CKM_RC5_MAC_GENERAL 377
CKM_RSA_FIPS_186_3_AUX_PRIME_KEY_PAIR_GEN 378
CKM_RSA_FIPS_186_3_PRIME_KEY_PAIR_GEN 379
CKM_RSA_PKCS 380
Firmware 7.4.2 and Older Summary 382
CKM_RSA_PKCS_KEY_PAIR_GEN 384
CKM_RSA_PKCS_OAEP 385
CKM_RSA_PKCS_PSS 387
CKM_RSA_X_509 388
CKM_RSA_X9_31 389
CKM_RSA_X9_31_KEY_PAIR_GEN 391
CKM_RSA_X9_31_NON_FIPS 392
CKM_SEED_CBC 393
CKM_SEED_CBC_PAD 394
CKM_SEED_CMAC 395
CKM_SEED_CMAC_GENERAL 396
CKM_SEED_CTR 397
CKM_SEED_ECB 398
CKM_SEED_KEY_GEN 399
CKM_SEED_MAC 400
CKM_SEED_MAC_GENERAL 401
CKM_SHA_1 402
CKM_SHA_1_HMAC 403
CKM_SHA_1_HMAC_GENERAL 405
CKM_SHA1_EDDSA 407
CKM_SHA1_EDDSA_NACL 409
CKM_SHA1_KEY_DERIVATION 411
CKM_SHA1_RSA_PKCS 412
Firmware 7.4.2 and Older Summary 412
CKM_SHA1_RSA_PKCS_PSS 414
Firmware 7.4.2 and Older Summary 414
CKM_SHA1_RSA_X9_31 416
Firmware 7.4.2 and Older Summary 416
CKM_SHA1_RSA_X9_31_NON_FIPS 418
CKM_SHA1_SM2DSA 419
CKM_SHA3_224 420
CKM_SHA3_224_DSA 422
CKM_SHA3_224_EDDSA 424

CKM_SHA3_224_HMAC 426
CKM_SHA3_224_HMAC_GENERAL 428
CKM_SHA3_224_KEY_DERIVE 430
CKM_SHA3_224_RSA_PKCS 431
CKM_SHA3_224_RSA_PKCS_PSS 433
CKM_SHA3_256 435
CKM_SHA3_256_DSA 437
CKM_SHA3_256_EDDSA 439
CKM_SHA3_256_HMAC 441
CKM_SHA3_256_HMAC_GENERAL 443
CKM_SHA3_256_KEY_DERIVE 445
CKM_SHA3_256_RSA_PKCS 446
CKM_SHA3_256_RSA_PKCS_PSS 448
CKM_SHA3_384 450
CKM_SHA3_384_DSA 452
CKM_SHA3_384_EDDSA 454
CKM_SHA3_384_HMAC 456
CKM_SHA3_384_HMAC_GENERAL 458
CKM_SHA3_384_KEY_DERIVE 460
CKM_SHA3_384_RSA_PKCS 461
CKM_SHA3_384_RSA_PKCS_PSS 463
CKM_SHA3_512 465
CKM_SHA3_512_DSA 467
CKM_SHA3_512_EDDSA 469
CKM_SHA3_512_HMAC 471
CKM_SHA3_512_HMAC_GENERAL 473
CKM_SHA3_512_KEY_DERIVE 475
CKM_SHA3_512_RSA_PKCS 476
CKM_SHA3_512_RSA_PKCS_PSS 478
CKM_SHA224 480
CKM_SHA224_EDDSA 481
CKM_SHA224_EDDSA_NACL 483
CKM_SHA224_HMAC 485
CKM_SHA224_HMAC_GENERAL 487
CKM_SHA224_KEY_DERIVATION 489
CKM_SHA224_RSA_PKCS 490
CKM_SHA224_RSA_PKCS_PSS 491
CKM_SHA224_RSA_X9_31 492
CKM_SHA224_RSA_X9_31_NON_FIPS 494
CKM_SHA224_SM2DSA 495
CKM_SHA256 496
CKM_SHA256_EDDSA 497
CKM_SHA256_EDDSA_NACL 499
CKM_SHA256_HMAC 501
CKM_SHA256_HMAC_GENERAL 503
CKM_SHA256_KEY_DERIVATION 505
CKM_SHA256_RSA_PKCS 506

CKM_SHA256_RSA_PKCS_PSS 507
CKM_SHA256_RSA_X9_31 508
CKM_SHA256_RSA_X9_31_NON_FIPS 510
CKM_SHA256_SM2DSA 511
CKM_SHA384 512
CKM_SHA384_EDDSA 513
CKM_SHA384_EDDSA_NACL 515
CKM_SHA384_HMAC 517
CKM_SHA384_HMAC_GENERAL 519
CKM_SHA384_KEY_DERIVATION 521
CKM_SHA384_RSA_PKCS 522
CKM_SHA384_RSA_PKCS_PSS 523
CKM_SHA384_RSA_X9_31 524
CKM_SHA384_RSA_X9_31_NON_FIPS 526
CKM_SHA384_SM2DSA 527
CKM_SHA512 528
CKM_SHA512_EDDSA 529
CKM_SHA512_EDDSA_NACL 531
CKM_SHA512_HMAC 533
CKM_SHA512_HMAC_GENERAL 535
CKM_SHA512_KEY_DERIVATION 537
CKM_SHA512_RSA_PKCS 538
CKM_SHA512_RSA_PKCS_PSS 539
CKM_SHA512_RSA_X9_31 540
CKM_SHA512_RSA_X9_31_NON_FIPS 542
CKM_SHA512_SM2DSA 543
CKM_SHAKE_128 544
CKM_SHAKE_128_KEY_DERIVE 546
CKM_SHAKE_256 548
CKM_SHAKE_256_KEY_DERIVE 550
CKM_SM2_KEY_PAIR_GEN 551
CKM_SM2DSA 552
CKM_SM3 553
CKM_SM3_HMAC 554
CKM_SM3_HMAC_GENERAL 556
CKM_SM3_KEY_DERIVATION 558
CKM_SM3_SM2DSA 559
CKM_SM4_CBC 560
CKM_SM4_CBC_PAD 561
CKM_SM4_ECB 562
CKM_SM4_KEY_GEN 563
CKM_SSL3_KEY_AND_MAC_DERIVE 564
CKM_SSL3_MASTER_KEY_DERIVE 565
CKM_SSL3_MD5_MAC 566
CKM_SSL3_PRE_MASTER_KEY_GEN 567
CKM_SSL3_SHA1_MAC 568
CKM_TUAK 569

CKM_TUAK_AUTS 570
CKM_TUAK_RESYNC 571
CKM_X9_42_DH_DERIVE 572
CKM_X9_42_DH_HYBRID_DERIVE 573
CKM_X9_42_DH_KEY_PAIR_GEN 574
CKM_X9_42_DH_PARAMETER_GEN 575
CKM_XOR_BASE_AND_DATA_W_KDF 577

Chapter 6: Using the Luna SDK 578
Libraries and Applications 578
Luna SDK Applications General Information 578
Compiler Tools 580
Using CKlog 580
Application IDs 582
Compatibility Old with New 583
Shared Login State and Application IDs 584
Object handles and labels 587
Named Curves and User-Defined Parameters 588
Curve Validation Limitations 588
Storing Domain Parameters 588
Using Domain Parameters 589
User Friendly Encoder 589
Application Interfaces 589
Supported ECC Curves 595
ECDH with Key Derive Function 599
PKCS#11 standard KDFs supported in Luna HSM 599
Vendor defined KDFs 600
ECIES general 604
EC IES mechanism (X9.63) 604
Mechanism parameters for CKM_ECIES. 604
Parameter and values used with CKM_PRF_KDF and CKM_NIST_PRF_KDF 606
ECIES for 5G 607
Profiles Supported 607
Decryption 607
Luna 5G OP flags and message response TLV tag definitions 608
Summary 609
Tools 609
Capability and Policy Configuration Control Using the Luna API 610
HSM Capabilities and Policies 610
HSM Partition Capabilities and Policies 610
Policy Refinement 610
Policy Types 610
Querying and Modifying HSM Configuration 611
Connection Timeout 613
Linux and Unix Connection Timeout 614
Windows Connection Timeout 614

Chapter 7: Design Considerations 615
Multifactor Quorum-Authenticated HSMs 615
About CKDemo with PED key 615
Interchangeability 616
Startup 616
Cloning of Tokens 617
High Availability Implementations 617
Detecting the Failure of an HA Member 618
Key Attribute Defaults 619
Management Attributes 619
Key Usage Attributes 621
Vendor-defined key attributes 621
Object Usage Count 621
Migrating Keys From Software to a Luna PCIe HSM 7 624
Other Formats of Key Material 626
Sample Program 626
Audit Logging 647
Audit Log Records 648
Audit Log Message Format 649
Log External 650

Chapter 8: Java Interfaces 651
Luna JSP Overview and Installation 651
Installation 652
JSP Registration 653
Post-Installation Tasks 654
Luna JSP Configuration 657
Luna Java Security Provider 657
Keytool 659
Cleaning Up 660
PKCS#11/JCA Interaction 660
The JCPROV PKCS#11 Java Wrapper 661
JCPROV Overview 661
Installing JCPROV 662
JCPROV Sample Programs 663
JCPROV Sample Classes 664
DeleteKey 664
EncDec 665
GenerateKey 665
GetInfo 666
Threading 666
JCPROV API Documentation 667
Java or JSP Errors 667
Re-Establishing a Connection Between Your Java Application and Luna PCIe HSM 7 668
Recovering From the Loss of All HA Members 669
When to Use the reinitialize Method 669
Why the Method Must Be Used 669

What Happens on the HSM 669
Using Java Keytool with Luna PCIe HSM 7 671
Limitations 671
Keytool Usage and Examples 672
Import CA certificate 673
Generate private key 673
Create the CSR 674
Import client certificate 674
How to build a certificate with chain 675
Additional minor notes 676
LunaKeyStore Reference 676

Chapter 9: Microsoft Interfaces 678
Luna CSP Registration Utilities 678
register 678
ms2Luna 681
keymap 682
Luna KSP for CNG Registration Utilities 683
kspcmd 684
KspConfig 685
ms2Luna 687
ksputil 688
Algorithms Supported 688
Run a Windows CNG application as Crypto Officer limited to key handling ability at Crypto User level 689
Luna CSP Calls and Functions 692
Programming for Luna PCIe HSM 7 with Luna CSP 692
Algorithms 693

PREFACE: About the SDK Reference

This document describes how to use the Luna SDK to create applications that interact with Luna PCIe HSM 7s. It contains the following chapters:

> "Luna SDK Overview" on page 21

> "PKCS#11 Support" on page 25

> "Extensions to PKCS#11" on page 32

> "Supported Mechanisms" on page 139

> "Using the Luna SDK" on page 578

> "Design Considerations" on page 615

> "Java Interfaces" on page 651

> "Microsoft Interfaces" on page 678

The preface includes the following information about this document:

> "Customer Release Notes" below

> "Audience" below

> "Document Conventions" on the next page

> "Support Contacts" on page 20

For information regarding the document status and revision history, see "Document Information" on page 2.

Customer Release Notes

The Customer Release Notes (CRN) provide important information about specific releases. Read the CRN to fully understand the capabilities, limitations, and known issues for each release. You can view the latest version of the CRN at www.thalesdocs.com.

Audience

This document is intended for personnel responsible for maintaining your organization's security infrastructure. This includes Luna HSM users and security officers, key manager administrators, and network administrators.

All products manufactured and distributed by Thales are designed to be installed, operated, and maintained by personnel who have the knowledge, training, and qualifications required to safely perform the tasks assigned to them. The information, processes, and procedures contained in this document are intended for use by trained and qualified personnel only.

It is assumed that the users of this document are proficient with security concepts.

Document Conventions

This document uses standard conventions for describing the user interface and for alerting you to important information.

Notes

Notes are used to alert you to important or helpful information. They use the following format:

> **NOTE** Take note. Contains important or helpful information.

Cautions

Cautions are used to alert you to important information that may help prevent unexpected results or data loss. They use the following format:

> **CAUTION!** Exercise caution. Contains important information that may help prevent unexpected results or data loss.

Warnings

Warnings are used to alert you to the potential for catastrophic data loss or personal injury. They use the following format:

> ****WARNING** Be extremely careful and obey all safety and security measures. In this situation you might do something that could result in catastrophic data loss or personal injury.**

Command syntax and typeface conventions

Format	Convention
bold	The bold attribute is used to indicate the following: > Command-line commands and options (Type **dir /p**.) > Button names (Click **Save As**.) > Check box and radio button names (Select the **Print Duplex** check box.) > Dialog box titles (On the **Protect Document** dialog box, click **Yes**.) > Field names (**User Name**: Enter the name of the user.) > Menu names (On the **File** menu, click **Save**.) (Click **Menu** > **Go To** > **Folders**.) > User input (In the **Date** box, type **April 1**.)
italics	In type, the italic attribute is used for emphasis or to indicate a related document. (See the *Installation Guide* for more information.)

Format	Convention
<variable>	In command descriptions, angle brackets represent variables. You must substitute a value for command line arguments that are enclosed in angle brackets.
[**optional**] [<optional>]	Represent optional **keywords** or <variables> in a command line description. Optionally enter the keyword or <variable> that is enclosed in square brackets, if it is necessary or desirable to complete the task.
{**a**|**b**|**c**} {<a>|<b>|<c>}	Represent required alternate **keywords** or <variables> in a command line description. You must choose one command line argument enclosed within the braces. Choices are separated by vertical (OR) bars.
[**a**|**b**|**c**] [<a>|<b>|<c>]	Represent optional alternate keywords or variables in a command line description. Choose one command line argument enclosed within the braces, if desired. Choices are separated by vertical (OR) bars.

 Copyright 2001-2024 Thales Group

Support Contacts

If you encounter a problem while installing, registering, or operating this product, please refer to the documentation before contacting support. If you cannot resolve the issue, contact your supplier or Thales Customer Support. Thales Customer Support operates 24 hours a day, 7 days a week. Your level of access is governed by the support plan negotiated between Thales and your organization. Please consult this plan for details regarding your entitlements, including the hours when telephone support is available to you.

Customer Support Portal

The Customer Support Portal, at https://supportportal.thalesgroup.com, is where you can find solutions for most common problems and create and manage support cases. It offers a comprehensive, fully searchable database of support resources, including software and firmware downloads, release notes listing known problems and workarounds, a knowledge base, FAQs, product documentation, technical notes, and more.

> **NOTE** You require an account to access the Customer Support Portal. To create a new account, go to the portal and click on the **REGISTER** link.

Telephone

The support portal also lists telephone numbers for voice contact (Contact Us).

 Copyright 2001-2024 Thales Group

CHAPTER 1: Luna SDK Overview

This chapter provides an overview of the Luna Software Development Kit (SDK), a development platform you can use to integrate a Luna PCIe HSM 7 into your application or system. It contains the following topics:

> "Supported Cryptographic Algorithms" below

> "Application Programming Interface (API) Overview" below

> "Supported Integrations" on page 23

Supported Cryptographic Algorithms

The K7 Cryptographic engine supports cryptographic algorithms that include:

> RSA

> DSA

> Diffie-Hellman

> DES and triple DES

> MD2 and MD5

> SHA-1, SHA-224, SHA-256, SHA-384, SHA-512

> RC2, RC4 and RC5

> AES

> PBE

> ECC

> ECIES

> ARIA, SEED

Included with Luna Product Software Development Kit is a sample application – and the source code – to accelerate integration of Thales's cryptographic engine into your system.

Application Programming Interface (API) Overview

The major API provided with Luna Product Software Development Kit conforms to RSA Laboratories' Public-Key Cryptography Standards #11 (PKCS #11) v2.20, as described in "PKCS#11 Support" on page 25. A set of API services (called PKCS #11 Extensions) designed by Thales, augments the services provided by PKCS#11, as described in "Extensions to PKCS#11" on page 32. The extensions to each API enable optimum use of Luna hardware for commonly used calls and functions, where the unaugmented API would tend to use software, or to make generic, non-optimized use of available HSMs.

In addition, support is provided for Microsoft's cryptographic APIs (CAPI/CNG) (see "Microsoft Interfaces" on page 678 and Oracle's Java Security API (see "Java Interfaces" on page 651).

The API is a library – a DLL in Windows, a shared object in Solaris, AIX and Linux – called Chrystoki. Applications wanting to use token services must connect with Chrystoki.

> **NOTE** Luna HSM Client 10.1.0 and newer includes libraries for 64-bit operating systems only.

Table 1: Luna libraries by platform

Platform	Key name	Libraries
Windows	LibNT	C:\Program Files\SafeNet\LunaClient\cryptoki.dll
		C:\Program Files\SafeNet\LunaClient\cklog201.dll
		C:\Program Files\SafeNet\LunaClient\shim.dll
		C:\Program Files\SafeNet\LunaClient\LunaCSP\LunaCSP.dll
		C:\WINDOWS\system32\SafeNetKSP.dll
Solaris (32-bit)	LibUNIX	/opt/safenet/lunaclient/lib/libCryptoki2.so
		/opt/safenet/lunaclient/lib/libcklog2.so
		/opt/safenet/lunaclient/lib/libshim.so
Solaris (64-bit)	LibUNIX64	/opt/safenet/lunaclient/lib/libCryptoki2_64.so
		/opt/safenet/lunaclient/lib/libcklog2.so
		/opt/safenet/lunaclient/lib/libshim_64.so
Linux (32-bit)	LibUNIX	/usr/safenet/lunaclient/lib/libCryptoki2.so
		/usr/safenet/lunaclient/lib/libcklog2.so
		/usr/safenet/lunaclient/lib/libshim.so
Linux (64-bit)	LibUNIX64	/usr/safenet/lunaclient/lib/libCryptoki2_64.so
		/usr/safenet/lunaclient/lib/libcklog2.so
		/usr/safenet/lunaclient/lib/libshim_64.so

 Copyright 2001-2024 Thales Group

Platform	Key name	Libraries
AIX (32- and 64-bit)	LibAIX	/usr/safenet/lunaclient/lib/libCryptoki2.so
		/usr/safenet/lunaclient/lib/libCryptoki2_64.so
		/usr/safenet/lunaclient/lib/libcklog2.so
		/usr/safenet/lunaclient/lib/libshim.so

Sample Application

Included with Luna Product Software Development Kit is a sample application – and the source code – to accelerate integration of Thales's cryptographic engine into your system.

> **NOTE** To reduce development or adaptation time, you may re-distribute the salogin program to customers who use Luna PCIe HSM 7, in accordance with the terms of the End User License Agreement. However, you may not re-distribute the Luna Software Development Kit itself.

A Note About RSA Key Attributes 'p' and 'q'

When RSA keys are generated, 'p' and 'q' components are generated which, theoretically, could be of considerably different sizes.

Unwrapping

The Luna PCIe HSM 7 allows RSA private keys to be unwrapped onto the HSM where the lengths of the 'p' and 'q' components are unequal. Because the effective strength of an RSA key pair is determined by the length of the shorter component, choosing 'p' and 'q' to be of equal length provides the maximum strength from the generated key pair. If your application is designed to generate key pairs that will be unwrapped onto the HSM, care should be taken in choosing the lengths of the 'p' and 'q' components such that they differ by no more than 15%.

Generation

Where you are generating RSA private keys within the HSM, the HSM enforces that 'p' and 'q' be equal in size, to the byte level.

A Note About the Shim

The Client install includes a shim library to support PKCS#11 integration with various third-party products. You should have no need for this shim library in your development. If for some reason you determine that you need the shim, Chrystoki supports it.

Supported Integrations

With the exception of some generic items that (for example) might need to be set in Windows when installing CSP, KSP, or Java, we do not include a list of integrations in the main product documentation.

Instead, you can check with the https://supportportal.thalesgroup.com website for third-party applications that have been integrated and tested with Luna PCIe HSM 7s by our Integrations group. That group is constantly testing and updating third-party integrations and publishing notes and instructions to help you integrate our HSMs with your applications.

As a general rule, if a specific version of an application and a specific version of a Luna PCIe HSM 7 product are mentioned in an Integration document, then those items will definitely work together. A newer version of the Luna PCIe HSM 7 or its attendant software is most likely to work with the indicated application without problem. We take care, for several generations of a given HSM product, to not break working relationships, though eventually it might happen that very old versions of third-party software and systems can no longer be supported. One thing that can sometimes happen is that we update HSM firmware to include newer algorithms, and to exclude older algorithms or key sizes that no longer meet industry-accepted standards (like NIST, Common Criteria, etc.).

A newer version of a third-party software might, or might not work with Luna PCIe HSM 7s that were tested to work with a specific earlier version of the same software. This is because some vendors make changes in their products that require new adaptation or at least new configuration instructions. If this happens to you, Thales Customer Support or Sales Engineering is usually happy to work with you to find a solution - both to support you as one of our customers and to have a revised/new integration that can be added to our portfolio.

Check the website or contact Thales Customer Support for the latest list of third-party applications that are tested and supported with Luna PCIe HSM 7s.

Why Is an Integration Not Listed Here Or On the Website?

In many cases, third-party application vendors see a need to integrate their application with Thales Luna products. In those cases, the third-party company performs the integration and testing, and also provides the support for the integrated solution to their customers (including you). For integrations not listed by Thales, please contact the application vendor for current information.

Similarly some value-added resellers and custom/third-party integrators or consultants might have performed specific integrations of Luna PCIe HSM 7s for the benefit of their specific customers. If you have purchased services or product from such a supplier, you will need to contact them for support of such integrations.

Third-party-tested integrations are not listed here or on the Thales website library of integration documents because we have not verified them in our own labs. If you contact Thales Support regarding use of our product with an application that we have not integrated, you will be asked to contact the third party that performed the integration.

 Copyright 2001-2024 Thales Group

CHAPTER 2: PKCS#11 Support

This chapter describes the PKCS#11 support provided by the Luna SDK. It contains the following topics:

> "PKCS#11 Compliance" below

> "Using the PKCS#11 Sample" on page 29

PKCS#11 Compliance

This section shows the compliance of Luna Software Development Kit HSM products to the PKCS#11 standard, with reference to particular versions of the standard. The text of the standard is not reproduced here.

Supported PKCS#11 Services

The table below identifies which PKCS#11 services this version of Luna Software Development Kit supports. The table following lists other features of PKCS#11 and identifies the compliance of this version of the Luna Software Development Kit to these features.

Table 1: PKCS#11 function support

Category	Function	Supported on Luna partitions	Supported on Luna keyrings
General purpose functions	C_Initialize	Yes	Yes
	C_Finalize	Yes	Yes
	C_GetInfo	Yes	Yes
	C_GetFunctionList	Yes	Yes

Category	Function	Supported on Luna partitions	Supported on Luna keyrings
Slot and token management functions	C_GetSlotList	Yes	Yes
	C_GetSlotInfo	Yes	Yes
	C_GetTokenInfo	Yes	Yes
	C_WaitForSlotEvent	No	No
	C_GetMechanismList	Yes	Yes
	C_GetMechanismInfo	Yes	Yes
	C_InitToken	Yes	Yes
	C_InitPIN	Yes	Yes
	C_SetPIN	Yes	Yes
Session management functions	C_OpenSession	Yes	Yes
	C_CloseSession	Yes	Yes
	C_CloseAllSessions	Yes	Yes
	C_GetSessionInfo	Yes	Yes
	C_GetOperationState	Yes	No
	C_SetOperationState	Yes	No
	C_Login	Yes	Yes
	C_Logout	Yes	Yes

 Copyright 2001-2024 Thales Group

Category	Function	Supported on Luna partitions	Supported on Luna keyrings
Object management functions	C_CreateObject	Yes	Yes
	C_CopyObject	Yes	No
	C_DestroyObject	Yes	Yes
	C_GetObjectSize	Yes	Yes
	C_GetAttributeValue	Yes	Yes
	C_SetAttributeValue	Yes	Yes
	C_FindObjectsInit	Yes	Yes
	C_FindObjects	Yes	Yes
	C_FindObjectsFinal	Yes	Yes
Encryption functions	C_EncryptInit	Yes	Yes
	C_Encrypt	Yes	Yes
	C_EncryptUpdate	Yes	Yes
	C_EncryptFinal	Yes	Yes
Decryption functions	C_DecryptInit	Yes	Yes
	C_Decrypt	Yes	Yes
	C_DecryptUpdate	Yes	Yes
	C_DecryptFinal	Yes	Yes
Message digesting functions	C_DigestInit	Yes	Yes
	C_Digest	Yes	Yes
	C_DigestUpdate	Yes	Yes
	C_DigestKey	Yes	Yes
	C_DigestFinal	Yes	Yes

 Copyright 2001-2024 Thales Group

Category	Function	Supported on Luna partitions	Supported on Luna keyrings
Signing and MACing functions	C_SignInit	Yes	Yes
	C_Sign	Yes	Yes
	C_SignUpdate	Yes	Yes
	C_SignFinal	Yes	Yes
	C_SignRecoverInit	No	No
	C_SignRecover	No	No
Functions for verifying signatures and MACs	C_VerifyInit	Yes	Yes
	C_Verify	Yes	Yes
	C_VerifyUpdate	Yes	Yes
	C_VerifyFinal	Yes	Yes
	C_VerifyRecoverInit	No	No
	C_VerifyRecover	No	No
Dual-purpose cryptographic functions	C_DigestEncryptUpdate	No	No
	C_DecryptDigestUpdate	No	No
	C_SignEncryptUpdate	No	No
	C_DecryptVerifyUpdate	No	No
Key management functions	C_GenerateKey	Yes	Yes
	C_GenerateKeyPair	Yes	Yes
	C_WrapKey	Yes	Yes
	C_UnwrapKey*	Yes	Yes
	C_DeriveKey	Yes	Yes

 Copyright 2001-2024 Thales Group

Category	Function	Supported on Luna partitions	Supported on Luna keyrings
Random number generation functions	C_SeedRandom	Yes	No
	C_GenerateRandom	Yes	Yes
Parallel function management functions	C_GetFunctionStatus	No	No
	C_CancelFunction	No	No
Callback function		No	No

*C_UnwrapKey has support for the CKA_Unwrap_Template object. All mechanisms which perform the unwrap function support an unwrap template. Nested templates are not supported.

Table 2: PKCS#11 feature support

Feature	Supported?
Exclusive sessions	Yes
Parallel sessions	No

Key Check Values

The Luna HSM firmware calculates a checksum or key check value for each key object created by the HSM. This value or checksum length is fixed at 3 bytes, as defined by PKCS#11.

Additional Functions

Please note that certain additional functions have been implemented by Thales as extensions to the standard. These include aspects of object cloning, and are described in detail in "Luna Extensions to PKCS#11" on page 32.

Using the PKCS#11 Sample

The Luna SDK includes a simple "C" language cross platform source example, **p11Sample**, that demonstrates the following:

> How to dynamically load the Luna cryptoki library.

> How to obtain the function pointers to the exported PKCS11 standard functions and the Luna extension functions.

The sample demonstrates how to invoke some, but not all of the API functions.

The SfntLibPath Environment Variable

The sample depends on an environment variable created and exported prior to execution. This variable specifies the location of **cryptoki.dll** (Windows) or **libCryptoki2.so** on Linux/UNIX. The variable is called **SfntLibPath**. You are free to provide your own means for locating the library.

What p11Sample Does

The p11Sample program performs the following actions:

1. The sample first attempts to load the dynamic library in the function called **LoadP11Functions**. This calls **LoadLibrary** (Windows) or **dlopen** (Linux/UNIX).
2. The function then attempts to get a function pointer to the PKCS11 API **C_GetFunctionList** using **GetProcAddress** (Windows) or **dlsym** (Linux/UNIX).
3. Once the function pointer is obtained, use the API to obtain a pointer called **P11Functions** that points to the static CK_FUNCTION_LIST structure in the library. This structure holds pointers to all the other PKCS11 API functions supported by the library.

   At this point, if successful, PKCS11 APIs may be invoked like the following:

   ```
   P11Functions->C_Initialize(...);
   P11Functions->C_GetSlotList(...);
   P11Functions->C_OpenSession(...);
   P11Functions->C_Login(...);
   P11Functions->C_GenerateKey(...);
   P11Functions->C_Encrypt(...);
   :
   :
   etc
   ```
4. The sample next attempts to get a function pointer to the Luna extension **API CA_GetFunctionList** using **GetProcAddress** (Windows) or **dlsym** (Linux/UNIX).
5. Once the function pointer is obtained, use the API to obtain a pointer called **SfntFunctions** that points to the static CK_SFNT_CA_FUNCTION_LIST structure in the library. This structure holds pointers to some but not all of the other Luna extension API functions supported by the library.
6. At this point, if successful, Luna extension APIs may be invoked like the following:

   ```
   SfntFunctions->CA_GetHAState(...);
   :
   :
   etc.
   ```
7. A sample makefile is provided for 64-bit AIX

   You can easily port to another platform with minor changes.
8. To build: make -f Makefile.aix.64

NOTE Please note that this simple example loads the cryptoki library directly. If your application requires integration with cklog or ckshim, you will need to load the required library (see SDK General for naming on your platform) in lieu of cryptoki. cklog and ckshim will then use the Chrystoki configuration file to locate and load cryptoki. You also have the option of locating the cryptoki library by parsing the Chrystoki2 section of the Chrystoki config file. If you do this, then the initial library (cryptoki, cklog, or ckshim) can be changed by simply updating the configuration file.

 Copyright 2001-2024 Thales Group

CHAPTER 3: Extensions to PKCS#11

This chapter describes the Luna extensions to the PKCS#11 standard. It contains the following topics:

> "Luna Extensions to PKCS#11" below
> "HSM Configuration Settings" on page 43
> "Secure PIN Port Authentication" on page 44
> "MofN Secret Sharing (quorum or multi-person access control)" on page 59
> "Key Export Features" on page 60
> "Luna HSM Cloning API CPv1 - Extensions to PKCS #11" on page 63
> "Luna HSM Cloning API CPv3 - Extensions to PKCS #11" on page 65
> "Luna HSM Cloning API CPv4 Extensions to PKCS#11" on page 69
> "Secure External Scalable Key Storage Extensions" on page 83
> "Derivation of Symmetric Keys with 3DES_ECB" on page 85
> "PKCS#11 Extension HA Status Call" on page 85
> "PKCS#11 Extension HA Status Call - improved" on page 86
> "Counter Mode KDF Mechanisms" on page 112
> "BIP32 Mechanism Support and Implementation" on page 91
> "SLIP 10" on page 99
> "Derive Template" on page 110
> "3GPP Mechanisms for 5G Mobile Networks" on page 103
> "SM2/SM4 Mechanisms" on page 113
> "SHA-3 Mechanisms" on page 115

Luna Extensions to PKCS#11

The following table provides a list of the Luna PKCS#11 C-API extensions.

Firmware Dependencies

Some functions are firmware-dependent, as indicated. Where there is a firmware dependency, the specified firmware version applies to all minor revisions of the firmware. In the following table, if no firmware version/series is mentioned, then the extension applies to all. If a firmware version is mentioned, then the extension applies to that firmware series, but not to others.

Other APIs

These commands and functions can also be used as extensions to other Application Programming Interfaces (for example, OpenSSL).

Cryptoki Version Supported

The current release of Luna Toolkit provides the Chrystoki library supporting version 2.20 of the Cryptoki standard.

Luna Extensions

Extension	Description
CA_ActivateMofN	Activates a token that has the secret sharing feature enabled.
CA_AssignKey	See "CA_AssignKey" on page 127. Only applicable to Luna HSM Firmware 7.7.0 and newer.
CA_AuthorizeKey	See "CA_AuthorizeKey" on page 126. Only applicable to Luna HSM Firmware 7.7.0 and newer.
CA_Bip32ExportPublicKey	Retrieve public BIP32 key attributes and returned serialized format (base58 encoded).
CA_Bip32ImportPublicKey	Import BIP32 serialized format (base58 encoded) and create BIP32 public key object.
CA_CapabilityUpdate	Apply configuration update file as Security Officer only.
CA_CheckOperationState	Checks if the specified cryptographic operation (encrypt, decrypt, sign, verify,digest) is in progress or not in the given session.
CA_CloneAsSource	Refer to "Luna HSM Cloning API CPv1 - Extensions to PKCS #11" on page 63, "Luna HSM Cloning API CPv3 - Extensions to PKCS #11" on page 65, and "Luna HSM Cloning API CPv4 Extensions to PKCS#11" on page 69
CA_CloneAsTarget	Refer to "Luna HSM Cloning API CPv1 - Extensions to PKCS #11" on page 63, "Luna HSM Cloning API CPv3 - Extensions to PKCS #11" on page 65, and "Luna HSM Cloning API CPv4 Extensions to PKCS#11" on page 69
CA_CloneAsTargetInit	Refer to "Luna HSM Cloning API CPv1 - Extensions to PKCS #11" on page 63, "Luna HSM Cloning API CPv3 - Extensions to PKCS #11" on page 65, and "Luna HSM Cloning API CPv4 Extensions to PKCS#11" on page 69

Extension	Description
CA_CloneObject	Refer to "Luna HSM Cloning API CPv1 - Extensions to PKCS #11" on page 63, "Luna HSM Cloning API CPv3 - Extensions to PKCS #11" on page 65, and "Luna HSM Cloning API CPv4 Extensions to PKCS#11" on page 69
CA_ClonePrivateKey	Permits the secure transfer a private key (RSA) between a source token and a target token.
CA_CloseApplicationID	Deactivate an application identifier.
CA_CloseApplicationIDForContainer	Deactivate an application identifier for a container.
CA_ConfigureRemotePED	Configure the given slot to use the provided remote PED information (appliance slot only).
CA_CreateContainer	Create a partition for non-PPSO users.
CA_CreateContainerLoginChallenge	Create a challenge for a role on a partition.
CA_CreateLoginChallenge	Create a login challenge for the specified user.
CA_Deactivate	Deactivate a partition.
CA_DeleteContainer	Delete a partition.
CA_DeleteContainerWithHandle	Delete a partition.
CA_DeleteRemotePEDVector	Delete the Remote PED vector.
CA_DeriveKeyAndWrap	This is an optimization of C_DeriveKey with C_Wrap, merging the two functions into one (the in and out constraints are the same as for the individual functions). A further optimization is applied when mechanism CKM_ECDH1_DERIVE is used with CA_DeriveKeyAndWrap.
CA_DestroyMultipleObjects	Delete multiple objects.
CA_DismantleRemotePED	Inverse of CA_ConfigureRemotePED(). Delete remote PED configuration information.
CA_DuplicateMofN	Create duplicates (copies) of all MofN secret splits.
CA_EncodeECChar2Params	Encode EC curve parameters for user defined curves.

Extension	Description
CA_EncodeECParamsFromFile	Encode EC curve parameters for user defined curves.
CA_EncodeECPrimeParams	Encode EC curve parameters for user defined curves.
CA_Extract	Extract a SIM3 blob.
CA_FactoryReset	Factory Reset the HSM.
CA_FindAdminSlotForSlot	Get the Admin slot for the current slot.
CA_FirmwareRollback	Rollback firmware.
CA_GenerateCloneableMofN	Create a cloneable secret-splitting vector on a token.
CA_GenerateCloningKEV	Refer to "Luna HSM Cloning API CPv1 - Extensions to PKCS #11" on page 63, "Luna HSM Cloning API CPv3 - Extensions to PKCS #11" on page 65, and "Luna HSM Cloning API CPv4 Extensions to PKCS#11" on page 69
CA_GenerateMofN	Generate the secret information on a token.
CA_GenerateMofN_Common	Refer to the M of N document.
CA_Get	Get HSM parameters such as serial numbers, and certificates.
CA_GetApplicationID	Get an application's accessID.
CA_GetConfigurationElementDescription	Get capability / policy description and properties.
CA_GetContainerCapabilitySet	Get all partition capability values.
CA_GetContainerCapabilitySetting	Get a single partition capability value.
CA_GetContainerList	Get the list of all partitions on a slot.
CA_GetContainerName	Get the name of a specific partition.
CA_GetContainerPolicySet	Get all partition policy values.
CA_GetContainerPolicySetting	Get a single partition policy value.
CA_GetContainerStatus	Get partition status, which returns authentication status flags.

Extension	Description
CA_GetContainerStorageInformation	Get partition storage information such as size, usage, and number of objects.
CA_GetCurrentHAState()	Get HA status from the application perspective. Same functional behavior as CA_Get HAState, but uses parallel checks of members, avoids delays once a peer is found unreachable, and returns all member statuses within 3 seconds (*).
CA_GetDefaultHSMPolicyValue	Get the default value of a single HSM policy.
CA_GetDefaultPartitionPolicyValue	Get the default value of a single partition policy.
CA_GetFirmwareVersion	Get the vendor-specific firmware version of the Luna HSM.
CA_GetHAState	Get HA status from the application perspective.
CA_GetHSMCapabilitySet	Get all HSM capability values.
CA_GetHSMCapabilitySetting	Get a single HSM capability value.
CA_GetHSMPolicySet	Get all HSM policy values.
CA_GetHSMPolicySetting	Get a single HSM policy value.
CA_GetHSMStats	Get HSM usage stats such as operational counters and how busy the HSM is.
CA_GetHSMStorageInformation	Get HSM storage information such as storage and usage.
CA_GetMofNStatus	Retrieve the MofN structure of the specified token.
CA_GetNumberOfAllowedContainers	Get the number of allowed partitions depending on the partition license count.
CA_GetObjectHandle	Get the object handle for a given OUID.
CA_GetObjectUID	Get the OUID for a given object handle.
CA_GetPedId	Get the PED ID.
CA_GetRemotePEDVectorStatus	Get the status of the RPV, created or not.

 Copyright 2001-2024 Thales Group

Extension	Description
CA_GetRollbackFirmwareVersion	Get the available rollback version.
CA_GetServerInstanceBySlotID	Get the instance # in the chrystoki.conf (crystoki.ini) file for the appliance/server the specified slot maps to.
CA_GetSessionInfo	Gets the session info that includes vendor specific information such as authentication state and container handle.
CA_GetSlotIdForContainer	Return a slot for a given container handle.
CA_GetSlotIdForPhysicalSlot	Return a slot for a given physical slot.
CA_GetSlotListFromServerInstance	Get the list of slots for the specified appliance/server instance #, as defined in the chrystoki.conf (crystoki.ini) file.
CA_GetTime	Get the HSM time.
CA_GetTokenCapabilities	Get the capabilities for the specified partition.
CA_GetTokenCertificateInfo	Get the cloning certificate.
CA_GetTokenCertificates	Get all HSM certificates. Token Wrapping Certificates are used for cloning. [See * below table]
CA_GetTokenInsertionCount	Get the insertion or reset count of HSM in the given slot.
CA_GetTokenObjectHandle	Retrieves a partition's handle, if there is a partition security officer. Same as CA_GetObjectHandle.
CA_GetTokenObjectUID	Retrieves a partition's OUID, if there is a partition security office. Same as CA_GetObjectOUID.
CA_GetTokenPolicies	Get partition policies.
CA_GetTokenStatus	Get partition status.
CA_GetTokenStorageInformation	Get partition storage information.
CA_GetTunnelSlotNumber	Get the tunnel slot number for a given slot number.
CA_HAActivateMofN	See "High Availability Indirect Login" on page 44.
CA_HAAnswerLoginChallenge	See "High Availability Indirect Login" on page 44.

Extension	Description
CA_HAAnswerMofNChallenge	See "High Availability Indirect Login" on page 44.
CA_HAGetLoginChallenge	See "High Availability Indirect Login" on page 44.
CA_HAGetMasterPublic	See "High Availability Indirect Login" on page 44.
CA_HAInit	See "High Availability Indirect Login" on page 44.
CA_HALogin	See "CA_HALogin()" on page 49.
CA_IncrementFailedAuthCount	See "CA_IncrementFailedAuthCount" on page 127. Only applicable to Luna HSM Firmware 7.7.0 and newer.
CA_InitAudit	Initialize the Auditor role.
CA_InitializeRemotePEDVector	Create the Remote PED Vector.
CA_InitRolePIN	Initialize a role on the current slot.
CA_InitSlotRolePIN	Initialize a role on a different slot.
CA_InitToken	Same as CA_Init_token with PPT support.
CA_Insert	Insert a SIM3 blob.
CA_ListSecureTokenInit	Retrieve information from an SFF backup token.
CA_ListSecureTokenUpdate	Continue retrieving information from a backup SFF token.
CA_LogExportSecret	Export (backup) the audit log HMAC key.
CA_LogExternal	Log external message - pushes an application-provided message to the HSM and logs it via the audit log.
CA_LogGetConfig	Get the audit log configuration.
CA_LogGetStatus	Get the audit log status (audit role, logs needing export, HSM to PedClient communication status).
CA_LogImportSecret	Restore the audit log HMAC key.
CA_LogSetConfig	Modify the audit log configuration.

Extension	Description
CA_LogVerify	Verify the audit log record(s).
CA_LogVerifyFile	Verify the audit log record file.
CA_ManualKCV	Set the key cloning vector (KCV) (sets the domain).
CA_ModifyMofN	Modify the secret-splitting vector on a token.
CA_ModifyUsageCount	Modify key usage count (Crypto Officer).
CA_MTKResplit	Generate new MTK split, new purple key value.
CA_MTKRestore	Return MTK, provide purple key to recover from tamper.
CA_MTKSetStorage	Create purple key, enables STM/SRK.
CA_MTKZeroize	Erase the MTK, user invoked tamper. Puts HSM in to transport mode.
CA_OpenApplicationID	Activate an application identifier, independent of any open sessions.
CA_OpenApplicationIDForContainer	Same as CA_OpenApplicationID, but partition specific.
CA_OpenSession	Same as C_OpenSession, but lets you specify partition.
CA_OpenSessionWithAppID	Same as CA_OpenSession, but lets you specify an application ID (AppID)
CA_PerformSelfTest	Invoke a self test on HSM (RNG statistics, Cryptographic Algorithms).
CA_QueryLicense	Get License/CUF information.
CA_RandomizeApplicationID	Set an application accessID to a random value.
CA_ResetAuthorizationData	See "CA_ResetAuthorizationData" on page 126. Only applicable to Luna HSM Firmware 7.7.0 and newer.
CA_ResetDevice	Reset the HSM .
CA_ResetPIN	SO reset of a CO role PIN (if "SO can reset PIN" policy is on).

 Copyright 2001-2024 Thales Group

Extension	Description
CA_Restart	Clean up all sessions for a given slot.
CA_RestartForContainer	Clean up all sessions for a given partition.
CA_RetrieveLicenseList	Get a list of all Licenses/CUFs.
CA_RoleStateGet	Get the state of a role (initialized, activated, failed logins, challenge created, etc).
CA_SetApplicationID	Set the application's identifier.
CA_SetAuthorizationData	See "CA_SetAuthorizationData" on page 126. Only applicable to Luna HSM Firmware 7.7.0 and newer.
CA_SetCloningDomain	Set the domain string used during token initialization.
CA_SetContainerPolicies	Set multiple partition policies.
CA_SetContainerPolicy	Set single partition policy.
CA_SetContainerSize	Set container storage size.
CA_SetDestructiveHSMPolicies	Set multiple destructive HSM policies.
CA_SetDestructiveHSMPolicy	Set single destructive HSM policy.
CA_SetHSMPolicies	Set multiple HSM policies.
CA_SetHSMPolicy	Set single HSM policy.
CA_SetKCV	Set KCV (domain).
CA_SetLKCV	Set a legacy KCV (legacy domain).
CA_SetMofN	Set the security policy for the token to use the secret sharing feature.
CA_SetPedId	Set the PED ID for a specific slot.
CA_SetRDK	Set the RDK (role specific KCV) for the current role.
CA_SetTokenPolicies	Set partition policies for given slot (PPSO only)
CA_SetUserContainerName	Set the name the library should use for the user partition on non-PPSO partitions.

Extension	Description
CA_SIMMultiSign	SIM2, SKS, firmware 4.x, firmware 6.x. Sign multiple data blobs with multiple keys provided as SIM2 blobs.
CA_SMKRollover	Invoke once to move current SMK to RolloverSMK slot and create new PrimarySMK - allows insertion/decrypting of existing blobs with Rollover SMK and re-encryption/extraction with new Primary - then invoke again to end.
CA_SpRawRead	PED key migration - read PED key value from DataKey PED Key.
CA_SpRawWrite	PED key migration - store PED key value to iKey PED Key.
CA_STCClearCipherAlgorithm	Remove the specified Cipher Algorithm from use with STC for the specified slot.
CA_STCClearDigestAlgorithm	Remove the specified Digest Algorithm from use with STC for the specified slot.
CA_STCDeregister	Remove STC registration of a client from the specified slot.
CA_STCGetAdminPubKey	Get the public key for the Admin slot's STC identity RSA keypair.
CA_STCGetChannelID	Get the Secure Trusted Channel ID for the current slot.
CA_STCGetCipherAlgorithm	Get all the valid cipher suites allowed for the specified slot.
CA_STCGetCipherID	Get the ID for the cipher currently in use on active STC to this slot.
CA_STCGetCipherIDs	Get all cipher IDs valid for use with STC to the specified slot.
CA_STCGetCipherNameByID	Get the readable name string for the specified Cipher ID.
CA_STCGetClientInfo	Get the STC registration details (name, public key, active access) about the specified client on the specified slot.

Extension	Description
CA_STCGetClientsList	Get the list of all STC clients registered to the specified slot.
CA_STCGetCurrentKeyLife	Get the remaining lifetime (in operations) for the active negotiated STC session key.
CA_STCGetDigestAlgorithm	Get all the valid digest algorithms allowed for the specified slot.
CA_STCGetDigestID	Get the ID for the digest currently in use on active STC to this slot.
CA_STCGetDigestIDs	Get all digest IDs valid for use with STC to the specified slot.
CA_STCGetDigestNameByID	Get the readable name string for the specified Digest ID.
CA_STCGetKeyActivationTimeOut	Get the amount of time allowed between the initiation and completion of STC session negotiation.
CA_STCGetKeyLifeTime	Get the configured session key lifetime (in operations) for the specified slot.
CA_STCGetPartPubKey	Get the public key for the specified slot STC identity RSA keypair.
CA_STCGetPubKey	Get the specified slot's public key.
CA_STCGetSequenceWindowSize	Get the replay window size for the specified slot.
CA_STCGetState	Get the STC state of the specified slot.
CA_STCIsEnabled	Determine if STC is configured for the specified slot.
CA_STCRegister	Register a client for STC to the specified slot.
CA_STCSetCipherAlgorithm	Set a cipher algorithm as valid for use with STC on the specified slot.
CA_STCSetDigestAlgorithm	Set a digest algorithm as valid for use with STC on the specified slot.
CA_STCSetKeyActivationTimeOut	Set the amount of time allowed between the initiation and completion of STC session negotiations for the specified slot.

 Copyright 2001-2024 Thales Group

Extension	Description
CA_STCSetKeyLifeTime	Set how long a STC key can live before STC rekeying occurs.
CA_STCSetSequenceWindowSize	Set the replay window size for the specified slot.
CA_STMGetState	Get STM state (enabled or disabled).
CA_STMToggle	Enter, or recover from, Secure Transport Mode.
CA_TamperClear	Used by the SO to clear tamper status.
CA_TimeSync	Synchronize the HSM time with the host time.
CA_TokenDelete	SO can delete a partition (PPSO only).
CA_TokenZeroize	Zeroize a PPSO partition.
CA_ValidateContainerPolicySet	Validate partition policy settings prior to calling SetPolicies.
CA_ValidateHSMPolicySet	Validate HSM policy settings prior to calling SetPolicies.
CA_WaitForSlotEvent	For PCMCIA HSMs, extends C_WaitForSlotEvent and provides some history of events.
CA_Zeroize	Zeroize the HSM.

(* The 3 seconds is expected to be achievable for an HA group up to 32 members and is verified in supportive conditions, meaning in laboratory-like conditions, when not affected by appliance CPU, memory, network, or HSM bottlenecks that are outside the control of the cryptographic module and its host. The CA_GetCurrentHaState() function, along with CKDemo option 49, is available starting at Luna HSM Client version 10.7.0.)

HSM Configuration Settings

Luna PCIe HSM 7s implement configuration settings that can be used to modify the behavior of the HSM, or can be read to determine how the HSM will behave. There are multiple settings that may be manipulated. Other than the "allow non-FIPS algorithms", most customers have no need to either query or change HSM settings. If you believe that your application needs more control over the HSM, please contact Thales for guidance.

 Copyright 2001-2024 Thales Group

Secure PIN Port Authentication

Generally, an application collects an authentication code or PIN from a user and/or other source controlled by the host computer. With Thales's multifactor quorum-authenticated products (such as Luna PCIe HSM 7), the PIN must come from a device connected to the secure port of the physical interface (or connected via a secure Remote PED protocol connection). The Luna PED (PIN Entry Device) or Luna USB HSM touchscreen is used for secure entry of PINs.

A bit setting in the device's capabilities settings determines whether the HSM requires that PINs be entered through the secure port. If the appropriate configuration bit is set, PINs must be entered through the secure port.

If the device's configuration bit is off, the application must provide the PIN through the existing mechanism. Through setting the PIN parameters, the application tells the token where to look for PINs. A similar programming approach applies to define the key cloning domain identifier.

Applications wanting PINs to be collected via the secure port must pass a NULL pointer for the pPin parameter and a value of zero for the ulPinLen parameter in function calls with PIN parameters. This restriction applies everywhere PINs are used. The following functions are affected:

- C_InitToken
- C_InitIndirectToken
- C_InitPIN
- C_SetPIN
- CA_InitIndirectPIN
- C_Login
- CA_IndirectLogin

When domains are generated/collected through the secure port during a C_InitToken call, the application must pass a NULL pointer for the pbDomainString parameter and a value of zero for the ulDomainStringLen parameter in the CA_SetCloningDomain function.

High Availability Indirect Login

This section describes High Availability (HA) Indirect Login, a feature that allows you to create and maintain High Availability programmatically by using extensions to the PKCS#11 API. HA Indirect Login allows you to program your own complete HA environment, in full, or tie into (integrate with) suitable COTS High Availability solutions that provide a suitable Application Interface, using HA Indirect Login calls to handle common authentication among HSM partitions in groups.

If you are looking for information about client-mediated HA, which is the majority of HA implementations among Luna HSM customers, refer to High-Availability Groups.

Overview of High Availability Indirect Login

HA Indirect Login allows a secondary partition to be configured to cache the authentication state for a user, and for these cached credentials to be used to re-achieve an authenticated state using the HA Indirect Login exchange between the secondary partition and the primary partition. In pre-firmware-7.7.0 versions of HA Login (v1), this is achieved by caching the login credentials for a user.

HA Login can be viewed as the following two distinct steps:

> HA Login setup. For more information, refer to "High Availability Login Setup" below.

> An HA Login exchange. For more information, refer to "High Availability Login Exchange" below.

High Availability Login Setup

HA Login Setup requires the generation of an RSA key-pair to be used as the HA Login Public and Private keys. HA Login Setup is then done by transferring some of the HA Login Public/Private key material to a secondary partition and calling the CA_HAInit() API as an authenticated role on the secondary partition.

The key material that needs to be transferred varies based on the version of the HSM being used. It may be the HA Login Private Key, the HA Login Public Key or a PKC chain for the HA Login Private Key. The specific key material required is defined later in this section.

High Availability Login Exchange

The HA Login Exchanges is the specific exchange of information between a primary partition and a secondary partition used to achieve an authenticated state on the secondary partition.

The exchange of information is performed using a set of PKCS#11 extensions that are dedicated to the HA Login Protocol. The specific APIs and their calling sequence are defined later in this section.

High Availability Login HSM and Partition Policies

There are no HSM level policies related to the HA Indirect Login Protocol.

Each partition has a policy called ‘allow high availability recovery’. This policy must be enabled for any partition (primary or secondary) to take part in HA Login Setup or an HA Login Exchange.

> **NOTE** In order to implement High Availability Recovery, the primary and secondary tokens must exist on separate systems.

High Availability Login Public and Private Keys

The HA Login Public and Private Keys are a standard user RSA key-pair on the partition. As such, it is possible for any authorised role on the primary partition to make use of the HA Login Public and Private Keys in the context of HA Login Setup and/or an HA Login exchange.

For the HA Login Public and Private Keys to be valid for use with the HA Login Protocol, they must have all of their key-usage attributes set to false. The required attribute templates are defined in later sections. Any additional restrictions put on the HA Login Public and Private Keys are defined in the protocol version specific sections in this document.

Supported HSM Roles

The HSM supports many different roles which vary depending on which version of the product is being referenced and which types of partition is being discussed. The next two tables define which roles are supported on which versions and which roles support HA Login Setup.

Roles that can generate the HA Indirect Login public/private key-pair

		Roles

Firmware Versions		HSM SO	HSM Admin	Audit	PSO	Crypto Officer	Limited Crypto Officer	Crypto User
	< 6.22.0 (pre-PPSO)	Yes	N/A[2]	No	N/A[2]	Yes[1]	N/A[2]	No
	>= 6.22.0 < 7.7.0	Yes	Yes	No	No	Yes	N/A[2]	No
	>= 7.7.0	Yes	Yes	No	No	Yes	Yes	No

Roles that can be set up for HA Indirect Login

		Roles						
Firmware Versions		**HSM SO**	**HSM Admin**	**Audit**	**PSO**	**Crypto Officer**	**Limited Crypto Officer**	**Crypto User**
	< 6.22.0 (pre-PPSO)	Yes	N/A[2]	No	N/A[2]	Yes[1]	N/A[2]	Yes[1]
	>= 6.22.0 < 7.7.0	Yes	Yes	No	No	Yes	N/A[2]	No
	>= 7.7.0	Yes	Yes	No	No	Yes	Yes	Yes

[1] On pre-PPSO partitions (before Partition SO was introduced), the partition maintains only a single HA Login state. This means that the partition can be setup for HA Indirect Login for a single login only. For example, if the CO calls the CA_HAInit() API, then the HA Indirect Login state stores the role ID for the CO and the CO authentication state can be re-achieved using the HA Indirect Login Protocol. If the CU then decides to call the CA_HAInit() API, this overwrites the current HA Indirect Login state with the role ID for the CU. This means that the CO authentication state can no longer be re-achieved using the HA Indirect Login Protocol, an HA Indirect Login exchange with this partition will now achieve only the CU authentication state.

[2] This role does not exist on this version.

> **TIP** Where you have an HA Indirect Login setup, your HSM is made accessible by other HSMs. Adding a challenge secret to your role, that is unknown to other parties, does not prevent other parties from logging into your HSM. Rather it prevents other parties from using your particular role without that extra credential. To prevent other parties accessing your HSM, change the PIN.

High Availability Indirect Login Functions Prior to Luna HSM Firmware 7.7.0

This section provides the following information, relevant to the HA Indirect Login protocol prior to Luna HSM Firmware 7.7.0:

> "High Availability Indirect Login Protocol" on the next page

> "Initialization Functions" on page 48

> "Recovery Functions" on page 48

> "Login Key Attributes" on page 50

> "Control of HA Functionality" on page 50

High Availability Indirect Login Protocol

The version of the HA Indirect Login Protocol discussed here is used by all HSMs running firmware version 6.0.0 up to Luna HSM Firmware 7.4.0. This protocol has been in use for many years and has received the following two major changes:

> **Support for the HA Login Public Key**

Prior to firmware version 6.10.0, the firmware required that the HA Login Private Key be cloned to the secondary partition so that a role on the secondary partition can be initialized for HA Login. Firmware 6.10.0 was updated such that the HA Login Public key could be used to initialize a role on the secondary partition. This eliminated the need to first initialize the secondary partition with the same cloning domain as the primary so that the HA Login Private Key could be cloned. Now the HA Login Public key can be extracted and re-created on the secondary directly.

The public key based setup is typically used when the secondary should *only* be able to act as a secondary and should not be able to act as a primary. If both partitions should be able to act as the primary and secondary, then the HA Login Private Key based setup should be used.

> **Per-Partition SO**

Firmware version 6.22.0 introduced the Per-Partition SO (PPSO) feature. This feature introduced a new partition type (PPSO partition) that supports its own security officer role, the Partition Security Officer (PSO), as well as a greater level of role separation between the Crypto-Officer and Crypto-User.

The existing role behavior was maintained and was available through the use of a Legacy pre-PPSO Partition. Non-PPSO partitions were deprecated in 7.x HSMs, but are mentioned here for anyone seeking to migrate from older Luna HSMs.

On a Legacy pre-PPSO Partition, when HA Login is setup, the Crypto-Officer or the Crypto-User is required to execute the command to setup HA Login. The role ID of the role that issues the command is stored in the partition so that when the HA Login exchange is performed, the same level of authentication as the role that setup HA Login is restored. The Legacy per-PPSO Partition only maintains one set of state information for HA Login. This means that at any time, the Crypto-Officer or the Crypto-User can re-issue the HA Login setup command to override which role's authenticated state will be restored by an HA Login exchange.

On a PPSO partition, only the Crypto-Officer can be setup for HA Login.

Cryptographic Primitives

The pre-7.7.0 HA Login Protocol makes use of the following cryptographic primitives for the purposes of key wrapping and key transport:

> AES-256-ECB Encryption/Decryption

> RSA-PKCS v1.5 Encryption/Decryption

During HA Login Setup, a random AES 256-bit is wrapped using RSA-PKCS v1.5.

The HA Login Exchange is essentially key transport operation that is used to transport wrapped key material from the secondary to the primary, and then wrapped key material sent back to the secondary from the primary.

The key-transport is performed using RSA-PKCS v1.5, and it transports the random AES 256-bit which was wrapped using RSA-PKCS v1.5 during HA Login Setup.

The key material sent back to the secondary from the primary is wrapped using AES-256-ECB.

> **NOTE** The pre-7.7.0 protocol does not place any size restriction on the HA Login Private Key. If the HSM-level policy to allow non-FIPS algorithms is disabled, then the FIPS related key size restrictions are applied to the key generation routines. When using Luna HSM Firmware 7.7.0 (or newer) as primary, the user should ensure to use different RSA Key pair to setup a pre-7.7.0 HA Login and 7.7.0 HA login -- otherwise there is a minor risk of being non-compliant with FIPS rules.

Initialization Functions

Initialization of tokens in a high-availability environment involves three steps:

1. The generation of an RSA login key pair (the public key of the pair may be discarded),
2. Cloning of the private key member to the User (and optionally to the SO) spaces of all tokens within that environment and,
3. Calling the **CA_HAInit** function on all tokens within that environment, in the context of the session owned by the User or SO.

The first two steps are performed using ordinary key generate and cloning Cryptoki function calls. The **CA_HAInit** function is implemented as follows:

CA_HAInit()

```
CK_RV CK_ENTRY CA_HAInit(
CK_SESSION_HANDLE hSession, // Logged-in session of user
// who owns the Login key pair
CK_OBJECT_HANDLE hLoginPrivateKey // Handle to Login private key
);
```

Recovery Functions

The HA recovery mechanism requires the following commands and interface functions:

CA_HAGetMasterPublic()

Called on the primary token, CA_HAGetMasterPublic() retrieves the primary token's TWC (Token Wrapping Certificate) and returns it as a blob (octet string and length). The format of this function is as follows:

```
CK_RV CK_ENTRY CA_HAGetMasterPublic(
CK_SLOT_ID slotId, // Slot number of the primary
// token
CK_BYTE_PTR pCertificate, // pointer to buffer to hold
//TWC certificate
CK_ULONG_PTR pulCertificateLen // pointer to value to hold
//TWC certificate length
);
```

CA_HAGetLoginChallenge()

Called on the secondary token, **CA_HAGetLoginChallenge()** accepts the TWC blob and returns the secondary token's login challenge blob. The format of this command is as follows:

```
CK_RV CK_ENTRY CA_HAGetLoginChallenge(
CK_SESSION_HANDLE hSession, // Public session
CK_USER_TYPE userType, // User type - SO or USER
CK_BYTE_PTR pCertificate, // TWC certificate retrieved
```

```
// from primary
CK_ULONG ulCertificateLen, // TWC certificate length
CK_BYTE_PTR pChallengeBlob, // pointer to buffer to hold
// challenge blob
CK_ULONG_PTR pulChallengeBlobLen // pointer to value to hold
// challenge blob length
);
```

CA_HAAnswerLoginChallenge()

Called on the primary token, **CA_HAAnswerLoginChallenge()** accepts the login challenge blob and returns the encrypted SO or User PIN, as appropriate.

```
CK_RV CK_ENTRY CA_HAAnswerLoginChallenge(
CK_SESSION_HANDLE hSession, // Session of the Login Private
// key owner
CK_OBJECT_HANDLE hLoginPrivateKey, // object handle to login key
CK_BYTE_PTR pChallengeBlob, // pointer to buffer containing
// challenge blob
CK_ULONG ulChallengeBlobLen, // length of challenge blob
CK_BYTE_PTR pEncryptedPin, // pointer to buffer holding
// encrypted PIN
CK_ULONG_PTR pulEncryptedPinLen // pointer to value holding
// encrypted PIN length
);
```

CA_HALogin()

Called on the secondary token, **CA_HALogin()** accepts the encrypted PIN and logs the secondary token in. If the second-ary token requires MofN authentication, an MofN challenge blob is returned. If no MofN authentication is required, a zero-length blob is returned. The format of this function is as follows:

```
CK_RV CK_ENTRY CA_HALogin(
CK_SESSION_HANDLE hSession, // Same public session opened
// in CA_HAGetLoginChallenge,
//above
CK_BYTE_PTR pEncryptedPin, // pointer to buffer holding
// encrypted PIN
CK_ULONG ulEncryptedPinLen,  //  length of encrypted PIN
CK_BYTE_PTR pMofNBlob, // pointer to buffer to hold
// MofN blob
CK_ULONG_PTR pulMofNBlobLen // pointer to value to hold the
// length of MofN blob
);
```

If the call is successful, then the session now becomes a pri-vate session owned by the User or SO (as appropriate).

CA_AnswerMofNChallenge()

Called on the primary token, **CA_AnswerMofNChallenge()** accepts the MofN challenge blob and returns the primary token's masked MofN secret. The format of this function is as follows:

```
CK_RV CK_ENTRY CA_HAAnswerMofNChallenge(
CK_SESSION_HANDLE hSession, // Private session
CK_BYTE_PTR pMofNBlob, // passed in MofN blob
CK_ULONG ulMofNBlobLen, // length of MofN blob
CK_BYTE_PTR pMofNSecretBlob, // pointer to buffer to hold
// MofN secret blob
```

```
CK_ULONG_PTR pulMofNSecretBlobLen//pointer to value that holds
// the MofN secret blob len
);
```

CA_HAActivateMofN()

Called on the secondary token, **CA_HAActivateMofN()** accepts the masked MofN secret and performs MofN authentication. The resulting MofN secret is checked against the CRC stored in the MofN PARAM structure.

```
CK_RV CK_ENTRY CA_HAActivateMofN(
CK_SESSION_HANDLE hSession, // The now-private session from
// successful CA_HALogin call
CK_BYTE_PTR pMofNSecretBlob, // pointer to MofN secret
// blob that is passed in
CK_ULONG ulMofNSecretBlobLen // length of MofN secret blob
);
```

It is expected that the recovery functions will be executed in the proper sequence and as part of an atomic operation. Nonetheless, the recovery operation may be restarted at any time due to an error. Restarting the recovery operation resets the state condition of the secondary token, and any data that has been stored or generated on the token is discarded.

Login Key Attributes

The login keys must possess the following attributes to function properly in a HA recovery scenario:

```
// Object
CKA_CLASS = CKO_PRIVATE_KEY,
// StorageClass
CKA_TOKEN = True,
CKA_PRIVATE = True,
CKA_MODIFIABLE = False,
// Key
CKA_KEY_TYPE = CKK_RSA,
CKA_DERIVE = False,
CKA_LOCAL = True,
// Private
CKA_SENSITIVE = True,
CKA_DECRYPT = False,
CKA_SIGN = False,
CKA_SIGN_RECOVER = False,
CKA_UNWRAP = False,
CKA_EXTRACTABLE = False
```

Control of HA Functionality

Refer to for the mechanisms by which the SO can control availability of the HA functionality.

High Availability Indirect Login For HSM Firmware 7.7.0 and Newer

This section provides the following information, which is relevant to the HA Indirect Login protocol for Luna HSM Firmware 7.7.0 and newer:

> "Setting Up HA Indirect Login" on the next page

> "Performing HA Indirect Login Exchange" on page 54

> "HA Indirect Login Public and Private Key Templates" on page 54

> "Use Cases" on page 55

> "Partition Versions and HA Indirect Login Protocol Compatibility" on page 56

Setting Up HA Indirect Login

In this section, the abbreviations "V0" and "V1" refer to partition versions and infrastructure that come with Luna HSM Firmware 7.7.0 (or newer), including the new cloning regime and SKS in V1. The distinctions between the V0 and V1 partition structures, and their behavior, are the reason for the existence of this page.

Separately, HA Indirect Login has existed for a long time and has gone through several iterations, where

> HA Indirect Login version 1.0 was introduced in an earlier-hardware HSM 6.x and continues through 7.x up-to-but-not-including-7.7;

> HA Indirect Login version 1.1 applies to Luna HSM Firmware 7.7.0. This protocol support serves three purposes:

- It allows any HA Login state that was configured on these partitions pre-update to be used post-update. Customerd do not need to manually login and setup HA Login again. Customers that have written their own custom HA Login code might not not need to re-write their application if buffers used to transfer data from primary to secondary HSM during login process are large enough. They can continue to use the old HA Login APIs on these partitions.
- It allows customers to continue to operate in FIPS approved mode after they update their HSM without setting up version 2 HA Login. Clients are not forced into non-FIPS mode during firmware update. The version 1 HA Indirect Login protocol was never FIPS compliant, whereas the version 1.1 protocol is compliant. The version 1.1 protocol re-uses the version 1 setup data where that is available.
- It provides a gateway to migrate partitions to version 2 HA Indirect Login and then to partitions from V0 to V1 (policy 41) while maintaining HA Indirect Login functionality.

> HA Indirect Login version 2.0 is introduced with Luna HSM Firmware 7.7.0, to account for structural changes and modernization.

So you might be starting with version 2.0, or you might have migrated from older implementations and need to update.

To setup HA Indirect Login between primary partition and a secondary partition the following steps should be followed:

1. Login to the primary partition as the Crypto-Officer.
2. Generate the HA Indirect Login Public/Private key pair.
3. If a secondary should not act as primary, then request a PKC chain for the HA Login Private Key. Otherwise set up the HA Login private on secondary using cloning for V0 partition, or using SKS for V1 partition.
4. Log into the secondary partition as the role you wish to setup for HA Indirect Login.

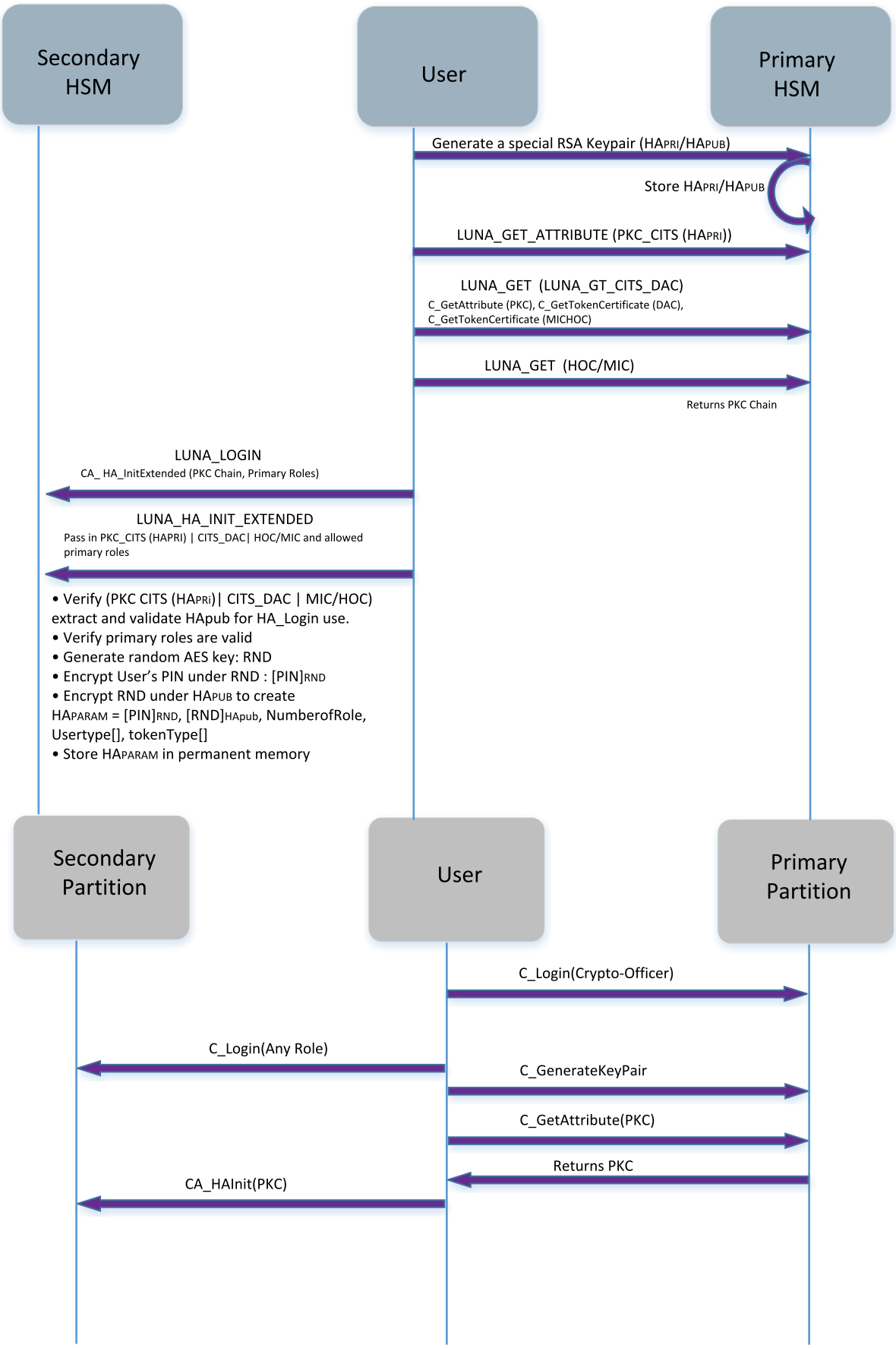
> **NOTE** As part of the Luna HSM Firmware 7.7.0 (or newer) Indirect Login version, all roles on the HSM can be setup for HA Login on the secondary partition.
>
> If Luna HSM Firmware 7.7.0 (or newer) setup has already been established, then first revoke current HA Login credential and allowed primary roles. All sessions opened on this partition, except the current one, will be implicitly closed.
>
> If HA Indirect Login version 1.1 setup has been done then no need to revoke. All sessions opened on this partition, except current one, will be implicitly closed.

5. If the secondary is not to be permitted to act as primary, then initialize the role on the secondary partition for HA Indirect Login by passing the PKC and allowed primary roles in to the CA_HAInitExtended() API; otherwise the handle of the private login key and allowed primary roles must be passed in. Primary roles, when logged in as primary, are allowed to log in on the secondary as the role currently logged in. It is possible to later update or change the allowed primary roles.
   a. Allowed roles are Security Officer, Administrator, Crypto Officer, Crypto User and Limited Crypto Officer.
   b. Partition Officer and Auditor do not have access to any key, and therefore are not able to log in using HA Private Key, and thus cannot be valid primary roles.
   c. If an allowed role is updated with different roles then any opened session is closed except current one.

Step 5 can be repeated to add additional partitions to the HA Indirect Login group. To act as a primary partition, a partition requires a copy of the HA Login Private Key.

- For V0 partitions this would require cloning the HA Login Private Key to any additional partitions that need to act as a primary partition.
- For V1 partitions it would require SKS extracting the HA Login Private Key from the first partition and inserting to any additional partitions that need to act as a primary partition.

The figure below shows the version 2 HA Indirect Login Setup using the new APIs.


Secondary HSM
User
Primary HSM
Generate a special RSA Keypair (HAPRI/HAPUB)
Store HAPRI/HAPUB
LUNA_GET_ATTRIBUTE (PKC_CITS (HAPRI))
LUNA_GET (LUNA_GT_CITS_DAC)
C_GetAttribute (PKC), C_GetTokenCertificate (DAC), C_GetTokenCertificate (MICHOC)
LUNA_GET (HOC/MIC)
Returns PKC Chain
LUNA_LOGIN
CA_ HA_InitExtended (PKC Chain, Primary Roles)
LUNA_HA_INIT_EXTENDED
Pass in PKC_CITS (HAPRI) | CITS_DAC| HOC/MIC and allowed primary roles
• Verify (PKC CITS (HAPRi)| CITS_DAC | MIC/HOC) extract and validate HApub for HA_Login use.
• Verify primary roles are valid
• Generate random AES key: RND
• Encrypt User's PIN under RND : [PIN]RND
• Encrypt RND under HAPUB to create HAPARAM = [PIN]RND, [RND]HApub, NumberofRole, Usertype[], tokenType[]
• Store HAPARAM in permanent memory
Secondary Partition
User
Primary Partition
C_Login(Crypto-Officer)
C_Login(Any Role)
C_GenerateKeyPair
C_GetAttribute(PKC)
Returns PKC
CA_HAInit(PKC)

Performing HA Indirect Login Exchange

To perform an HA Indirect Login exchange, the following steps must be performed:

1. Login to the primary partition as a role on the partition that has access to the HA Login Private Key and can act as a primary in an HA Login Exchange.
2. Call CA_HAGetMasterPublicData() on the primary, passing in the handle to the HA Login Private Key and retrieve the Master Public Data (public certificates required by the secondary).
3. Call CA_HAGetLoginChallenge() on the secondary partition, passing in the role ID that is to be authenticated, the Master Public Data from the primary partition (the TWC of the primary and a PKC chain for the HA Login Private Key) and receive the challenge from the secondary partition.
4. Call CA_HAAnswerLoginChallenge() on the primary passing in challenge received from the secondary partition along with the key handle of the HA Login Private Key and receive the challenge response.
5. Call CA_HALogin on the secondary and pass in the challenge response received from the primary partition. Upon success the user is logged in on secondary partition.

HA Indirect Login Public and Private Key Templates

The HSM requires that the HSM Login Private Key and HA Login Public Key have a very restricted set of attribute values to prevent the keys from being abused. Every time the HSM makes use of an HA Login Public/Private Key, it verifies that its attributes match the templates defined below.

HA Login Private Key:

```
static struct { AttributeId Id; ATTR_VAL ExpVal; UInt32 Size; } attrTbl[] =
{
   // Object
   { CKA_CLASS,               { CKO_PRIVATE_KEY },  sizeof( ClassAttribute ) },

   // StorageClass
   { CKA_TOKEN,               { true },             sizeof( Boolean ) },
   { CKA_PRIVATE,             { true },             sizeof( Boolean ) },
   { CKA_MODIFIABLE,          { false },            sizeof( Boolean ) },

   // Key
   { CKA_KEY_TYPE,            { CKK_RSA },          sizeof( KeyTypeAttribute ) },
   { CKA_DERIVE,              { false },            sizeof( Boolean ) },
   { CKA_LOCAL,               { true },             sizeof( Boolean ) },

   // Private
   { CKA_SENSITIVE,           { true },             sizeof( Boolean ) },
   { CKA_DECRYPT,             { false },            sizeof( Boolean ) },
   { CKA_SIGN,                { false },            sizeof( Boolean ) },
   { CKA_SIGN_RECOVER,        { false },            sizeof( Boolean ) },
   { CKA_UNWRAP,              { false },            sizeof( Boolean ) },
   { CKA_EXTRACTABLE,         { false },            sizeof( Boolean ) },
   { CKA_NEVER_EXTRACTABLE,   { true },             sizeof( Boolean ) }
};
```

HA Login Public Key:

```
static struct { AttributeId Id; ATTR_VAL ExpVal; UInt32 Size; } attrTbl[] =
{
   // Object
   { CKA_CLASS,               { CKO_PUBLIC_KEY },   sizeof( ClassAttribute ) },

   // StorageClass
```

```
   { CKA_TOKEN,                { true },             sizeof( Boolean ) },
   { CKA_PRIVATE,              { true },             sizeof( Boolean ) },
   { CKA_MODIFIABLE,           { false },            sizeof( Boolean ) },

   // Key
   { CKA_KEY_TYPE,             { CKK_RSA },          sizeof( KeyTypeAttribute ) },
   { CKA_DERIVE,               { false },            sizeof( Boolean ) },

   // Private
   { CKA_ENCRYPT,              { false },            sizeof( Boolean ) },
   { CKA_VERIFY,               { false },            sizeof( Boolean ) },
   { CKA_VERIFY_RECOVER,       { false },            sizeof( Boolean ) },
   { CKA_WRAP,                 { false },            sizeof( Boolean ) }
};
```

Use Cases

The HA Indirect Login feature is typically used in two ways, as follows.

Single Primary HSM with many Secondary HSMs

It is possible to setup a model where a single primary partition is able to HA Indirect Login to one or more secondary partitions. In this model, only the primary partition needs to be in possession of the HA Login Private Key.

This corresponds with the Crypto Command Center (CCC) use case, wherein one HSM acts as a root of trust and is able to HA Indirect Login to the HSM SO role on all other HSMs. This allows CCC to remotely administer the HSM.

Pool of HSMs that can act as the Primary

It is also possible to setup a pool of partitions such that they can all act as the primary in an HA Login exchange, to all other members in the pool. In this setup it is an application monitor the pool of partitions and, if any members goes down (for example, power loss, network outage), any other members that are still online can use the HA Login exchange to restore that member to an authenticated state without any user intervention. This is the original use case for the HA Indirect Login feature.

This model does not require the use of auto-activation in PED-authenticated HSMs, but the challenge secret must still be provided.

To set this up, each partition must have a copy of the HA Login Private Key and the desired role, and each partition must be initialized to use HA Indirect Login.

> **NOTE** The HA Login Private key must have the attribute CKA_NEVER_EXTRACTABLE set to TRUE which means that it must be transferred to the other HSMs using cloning if partition has V0 for Policy 41.
>
> SKS is used for this purpose in V1 partitions. All partitions MUST be in the same cloning domain and the SMK secret should have previously been cloned to them.
>
> Due to an incompatibility with the new cloning regime, when a partition is V0 (because you created it that way, or because a partition already existed when you updated to Luna HSM Firmware 7.7.0 or newer), a Client library contemporaneous with Luna HSM Firmware 7.7.0, or newer, is required if you wish to use the *older* HA Indirect Login APIs described elsewhere in this document.

> **NOTE** A newly created partition defaults to V0 and assumes the Luna use-case - just as with updated partitions, the purpose is to support older libraries and applications where needed.

Partition Versions and HA Indirect Login Protocol Compatibility

After firmware update to Luna HSM Firmware 7.7.0 (or newer), all pre-existing pre-7.7.0 partitions are updated with Partition policy 41 set to partition version zero (V0).

A V0 partition, as primary, supports HA Indirect Login protocol version 1, version 1.1, and version 2 (the most recent).
A V0 partition, as secondary, supports HA Indirect Login protocol version 1.1 and version 2.

The HA Indirect Login version 2 protocol support on V0 partition allows interested customers to use version 1.1 protocol to access their partition to setup V2 protocol. They can then use V2 protocol, and eventually change policy 41 to convert the partition to V1. This migration can be fully automated.

A V1 partition (policy 41) as primary supports all versions, but as secondary, supports only version 2 protocol, so prior to converting partitions from V0 to V1, you must perform setup of HA Indirect Login version 2 protocol. Otherwise, you will need to manually login to setup HA Indirect Login version 2 protocol.

API Compatibility for HA Indirect Login Setup

Primary	Secondary	Library	version1/version1.1 API For Setup	version2 API For Setup
Pre-Firmware 7.7	Firmware 7.7.0 Partition V0	Pre-7.7	Supported	APIs Not Available
Pre-Firmware 7.7	Firmware 7.7.0 Partition V0	7.7	Supported	Not Supported
Pre-Firmware 7.7	Firmware 7.7.0 Partition V1	Pre-7.7	Not Supported	APIs Not Available

Primary	Secondary	Library	version1/version1.1 API For Setup	version2 API For Setup
Pre-Firmware 7.7	Firmware 7.7.0 Partition V1	7.7	Not Supported	Not Supported
Firmware 7.7.0 V0	Firmware 7.7.0 Partition V0	Pre-7.7	Supported	APIs Not Available
Firmware 7.7.0 V0	Firmware 7.7.0 Partition V0	7.7	Supported	Supported
Firmware 7.7.0 V0	Firmware 7.7.0 Partition V1	Pre-7.7	Not Supported	APIs Not Available
Firmware 7.7.0 V0	Firmware 7.7.0 Partition V1	7.7	Not Supported	Supported
Firmware 7.7.0 V1	Firmware 7.7.0 Partition V0	Pre-7.7	Supported	APIs Not Available
Firmware 7.7.0 V1	Firmware 7.7.0 Partition V0	7.7	Supported	Supported
Firmware 7.7.0 V1	Firmware 7.7.0 Partition V1	Pre-7.7	Not Supported	APIs Not Available
Firmware 7.7.0 V1	Firmware 7.7.0 Partition V1	7.7	Not Supported	Supported

 Copyright 2001-2024 Thales Group

> **NOTE** On a Legacy pre-PPSO Partition (meaning all 5.x firmware HSM partitions as well as all 6.x up to firmware 6.22 and then some post-6.22 partitions, optionally - 7.x HSMs are PPSO-only), when HA Login is setup, the Crypto-Officer or the Crypto-User is required to execute the command to setup HA Login. The role ID of the role that issues the command is stored in the partition so that when the HA Login exchange is performed, the same level of authentication as the role that setup HA Login is restored. The Legacy per-PPSO Partition only maintains one set of state information for HA Login. This means that at any time, the Crypto-Officer or the Crypto-User can re-issue the HA Login setup command to override which role's authenticate state will be restored by an HA Login exchange.
>
> On a PPSO partition, the Crypto-Officer and Crypto-User have a greater level of separation and as such the PPSO partition maintains HA Login state for both the Crypto-Officer and Crypto-User. However due to a quirk of the PPSO feature, only the Crypto-Officer can be setup for HA Login.

API Compatibility for HA Indirect Login Exchange

Primary	Secondary	Library	version1/version1.1 API For HA Login Exchange	version2 API For HA Login Exchange
Firmware 7.7.0 V0	Pre-Firmware 7.7.0	Pre-Firmware 7.7.0	Not Supported	APIs Not Available
Firmware 7.7.0 V0	Pre-Firmware 7.7.0	Firmware 7.7.0 (or newer)	Supported	Not Supported
Firmware 7.7.0 V1	Pre-Firmware 7.7.0	Pre-Firmware 7.7.0	Not Supported	APIs Not Available
Firmware 7.7.0 V1	Pre-Firmware 7.7.0	Firmware 7.7.0 (or newer)	Supported	Not Supported
Pre-Firmware 7.7	Firmware 7.7.0 Partition V0	Pre-Firmware 7.7	Supported	APIs Not Available
Pre-Firmware 7.7	Firmware 7.7.0 Partition V0	Firmware 7.7	Supported	Not Supported
Pre-Firmware 7.7	Firmware 7.7.0 Partition V1	Pre-Firmware 7.7	Not Supported	APIs Not Available
Pre-Firmware 7.7	Firmware 7.7.0 Partition V1	Firmware 7.7	Not Supported	Not Supported

 Copyright 2001-2024 Thales Group

Primary	Secondary	Library	version1/version1.1 API For HA Login Exchange	version2 API For HA Login Exchange
Firmware 7.7.0 V0	Firmware 7.7.0 Partition V0	Pre-Firmware 7.7	Supported	APIs Not Available
Firmware 7.7V0	Firmware 7.7.0 Partition V0	Firmware 7.7	Supported	Supported
Firmware 7.7.0 V0	Firmware 7.7.0 Partition V1	Pre-Firmware 7.7	Not Supported	APIs Not Available
Firmware 7.7.0 V0	Firmware 7.7.0 Partition V1	Firmware 7.7	Not Supported	Supported
Firmware 7.7.0 V1	Firmware 7.7.0 Partition V0	Pre-Firmware 7.7	Supported	APIs Not Available
Firmware 7.7.0 V1	Firmware 7.7.0 Partition V0	Firmware 7.7	Supported	Supported
Firmware 7.7.0 V1	Firmware 7.7.0 Partition V1	Pre-Firmware 7.7	Not Supported	APIs Not Available
Firmware 7.7.0 V1	Firmware 7.7.0 Partition V1	Firmware 7.7	Not Supported	Supported

MofN Secret Sharing (quorum or multi-person access control)

Setting up

When a Luna HSM with multifactor quorum authentication is initialized, or a partition on such an HSM is initialized, the roles and domain secrets are mediated by the PIN Entry Device (PED). As each role or secret is brought into existence, PED keys are imprinted with the corresponding secret. This can occur in one of two ways:

> a fresh, unique secret is generated on the HSM, and is imprinted on an PED key, making the current HSM-or-partition completely independent and stand-alone, or the first of a new group

> if the option to reuse an existing PED key is selected, then no new secret is generated, and a secret created for another HSM or partition is read from an already-imprinted PED key and saved to the current HSM - this is how roles and domains can be shared among HSMs and partitions, and how exclusive groups of HSMs and partitions can be created for specific operational and security-regime purposes.

While an authentication or domain secret is being imprinted on PED keys, the HSM (via the Luna PED) offers the following options:

> have the current secret be complete on a single PED key to be held / controlled by one officer

> divide the secret into several splits/shards/components and specify how many of those components must later be brought together to reconstruct that secret

The Luna convention is that "N" is the total number of splits (up to sixteen, but a smaller number is generally recommended*) into which a secret is divided, and "M" is the necessary-and-sufficient number of splits that must be brought together (usually fewer than N) in order to recreate the full secret.

Using

The HSM prompts, by means of PED key input device (historicallyLuna PED), for the individual components of the MofN split secret that accesses the roles or the cloning/security domain associated with the HSM or partition currently being addressed.

The split values from individual PED keys, as they are passed to the HSM, are validated as being part of the correct type of secret (appropriate role [SO, CO, CU, Audit...], domain, RPV) and as being component members of the particular secret, and as not being an attempt to offer the same split portion more than once.

If the correct type** and number of the sub-components are received by the HSM to recreate the needed secret, the HSM permits access to the current partition/slot by that role, or permits cloning operations in-or-out of the partition for backup/restore, HA synchronization, etc.

* Thales recommends that you choose M as a reasonable quorum of officers that must be available every time you need to present the secret, and then choose N sufficiently larger to allow some co-officers to be unavailable [vacation, sickness, business travel] while still having enough split-holders to achieve quorum. The larger the number for M, the more time it takes to complete authentications, which might run up against timeouts and require you to adjust timeout values.

** When PED keys containing splits are offered, they must be be part of the specific secret that is needed here, and must not be a split (or a copy of a split) that was already presented during the current attempt. A lapse of any of those requirements results in the attempt being declined.

Key Export Features

The Luna PCIe HSM 7 with Key Export provides the feature(s) detailed in this section (see Configuring the Partition for Cloning or Export of Private/Secret Keys).

RSA Key Component Wrapping

The RSA Key Component Wrapping is a feature that allows an application to wrap any subset of attributes from an RSA private key with 3-DES. Access to the feature is through the PKCS #11 function C_WrapKey with the CKM_DES3_ECB mechanism. The wrapping key must be a CKK_DES2 or CKK_DES3 key with its CKA_WRAP attribute set to TRUE. The key to wrap must be an RSA private key with CKA_EXTRACTABLE set to TRUE and the FPV must have FPV_WRAPPING_TOKEN turned on.

The details of the wrapping format are specified with a format descriptor. The format descriptor is provided as the mechanism parameter to the CKM_DES3_ECB mechanism. This descriptor consists of a 32-bit format version, followed by a set of field element descriptors. Each field element descriptor consists of a 32-bit Field Type Identifier and optionally some additional data. The SafeNet firmware parses the set of field element descriptors and builds the custom layout of the RSA private key in an internal buffer. Once all field element descriptors are processed, the buffer is wrapped with 3-DES and passed out to the calling application. It is the responsibility of the calling application to ensure that the buffer is a multiple of 8 bytes.

The format descriptor version (the first 32-bit value in the format data) must always be set to zero.

The set of supported field element descriptor constants is as follows:

> #define KM_APPEND_STRING 0x00000000

> #define KM_APPEND_ATTRIBUTE 0x00000001

> #define KM_APPEND_REVERSED_ATTRIBUTE 0x00000002

> #define KM_APPEND_RFC1423_PADDING 0x00000010

> #define KM_APPEND_ZERO_PADDING 0x00000011

> #define KM_APPEND_ZERO_WORD_PADDING 0x00000012

> #define KM_APPEND_INV_XOR_CHECKSUM 0x00000020

> #define KM_DEFINE_IV_FOR_CBC 0x00000030

The meanings of the field element descriptors is as follows:

Field element descriptor	Description
KM_APPEND_STRING	Appends an arbitrary string of bytes to the custom layout buffer. The field type identifier is followed by a 32-bit length field defining the number of bytes to append. The length field is followed by the bytes to append. There is no restriction of the length of data that may be appended, as long as the total buffer length does not exceed 3072 bytes.
KM_APPEND_ATTRIBUTE	Appends an RSA private key component into the buffer in big endian representation. The field type identifier is followed by a 32-bit CK_ATTRIBUTE_TYPE value set to one of the following: CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2, CKA_EXPONENT_1, CKA_EXPONENT_2, or CKA_COEFFICIENT.. The key component is padded with leading zeros such that the length is equal to the modulus length in the case of the private exponent, or equal to half of the modulus length in the case of the other 5 components.
KM_APPEND_REVERSED_ATTRIBUTE	Appends an RSA private key component into the buffer in little endian representation. The field type identifier is followed by a 32-bit CK_ATTRIBUTE_TYPE value set to one of the following: CKA_PRIVATE_EXPONENT, CKA_PRIME_1, CKA_PRIME_2, CKA_EXPONENT_1, CKA_EXPONENT_2, or CKA_COEFFICIENT. The key component is padded with trailing zeros such that the length is equal to the modulus length in the case of the private exponent, or equal to half of the modulus length in the case of the other 5 components.
KM_APPEND_RFC1423_PADDING	Applies RFC 1423 padding to the buffer (appends 1 to 8 bytes with values equal to the number of bytes, such that the total buffer length becomes a multiple of 8). This would typically be the last formatting element in a set, but this is not enforced.

Field element descriptor	Description
KM_APPEND_ ZERO_ PADDING	Applies Zero padding to the buffer (appends 0 to 7 bytes with values equal to Zero, such that the total buffer length becomes a multiple of 8). This would typically be the last formatting element in a set, but this is not enforced.
KM_APPEND_ ZERO_WORD_ PADDING	Zero pads the buffer to the next 32-bit word boundary.
KM_APPEND_ INV_XOR_ CHECKSUM	Calculates and adds a checksum byte to the buffer. The checksum is calculated as the complement of the bytewise XOR of the buffer being built.
KM_DEFINE_ IV_FOR_CBC	Allows definition of an IV so that 3DES_CBC wrapping can be performed even though the functionality is invoked with the CKM_3DES_ECB mechanism. The field type identifier is followed by a 32-bit length field, which must be set to 8. The length is followed by exactly 8 bytes of data which are used as the IV for the wrapping operation.

Examples

To wrap just the private exponent of an RSA key in big endian representation, the parameters would appear as follows:

> **NOTE** Ensure that the packing alignment for your structures uses one (1) byte boundaries.

```
struct
{
UInt32 version = 0;
UInt32 elementType = KM_APPEND_ATTRIBUTE;
CK_ATTRIBUTE_TYPE attribute = CKA_PRIVATE_EXPONENT;
}
```

To wrap the set of RSA key components Prime1, Prime2, Coefficient, Exponent1, Exponent2 in little endian representation with a leading byte of 0x05 and ending with a CRC byte and then zero padding, the parameters would appear in a packed structure as follows:

```
struct
{
UInt32 version = 0;
UInt32 elementType1 = KM_APPEND_STRING;
UInt32 length = 1;
UInt8 byteValue = 5;
UInt32 elementType2 = KM_APPEND_REVERSED_ATTRIBUTE;
CK_ATTRIBUTE_TYPE attribute1 = CKA_PRIME_1;
UInt32 elementType3 = KM_APPEND_REVERSED_ATTRIBUTE;
CK_ATTRIBUTE_TYPE attribute2 = CKA_PRIME_2;
UInt32 elementType4 = KM_APPEND_REVERSED_ATTRIBUTE;
CK_ATTRIBUTE_TYPE attribute3 = CKA_COEFFICIENT;
UInt32 elementType5 = KM_APPEND_REVERSED_ATTRIBUTE;
CK_ATTRIBUTE_TYPE attribute4 = CKA_EXPONENT_1;
```

```
UInt32 elementType6 = KM_APPEND_REVERSED_ATTRIBUTE;
CK_ATTRIBUTE_TYPE attribute5 = CKA_EXPONENT_2;
UInt32 elementType7 = KM_APPEND_INV_XOR_CHECKSUM;
UInt32 elementType8 = KM_APPEND_ZERO_PADDING;
}
```

Luna HSM Cloning API CPv1 - Extensions to PKCS #11

This section provides details of the Thales Luna HSM Cloning API CPv1 for these scenarios

> where the source and target token (or slot) are visible on the same host,

> where the source and target tokens are visible only to separate host systems.

In the latter case, you might need to develop significant logic in order to initiate and synchronize the cloning process between the two host systems. The CPv1 APIs have been in use since before Luna HSM 4.x and continue to be used, where appropriate, alongside the newer CPv3 (since Luna HSM Firmware 7.7.0) and CPv4 (since Luna HSM Firmware 7.8.0).

Use the information in this section, about the low-level APIs, when it is not appropriate to use the high-level CA_MigrateKeys() API in your application.

Cloning on the Same Host System

When cloning HSM objects between two tokens that are visible to the same host system, then a session can be opened on each token (slot) and the following function used:

CK_RV CA_CloneObject(CK_SESSION_HANDLE hTargetSession,

CK_SESSION_HANDLE hSourceSession,

CK_ULONG ulObjectType,

CK_OBJECT_HANDLE hObjectHandle,

CK_OBJECT_HANDLE_PTR phClonedObject);

Where ulObjectType is CK_CRYPTOKI_ELEMENT, hObjectHandle is the handle of the object on the source token to be cloned, and phCloned Object will contain the object handle of the newly cloned object on the target token.

Cloning between Host Systems

When cloning between two HSMs that are only visible to separate host systems, additional logic must be created in order to synchronize the cloning process between the systems. How this is initiated or what protocol should be deployed is somewhat system dependent, and beyond the scope of this document. However, below is the sequence of PKCS#11 extension functions that need to be directed to the source and target HSMs (or slots) in order to complete the cloning operation:

Step1 – the Token Wrapping Certificate (TWC) is first obtained from the source token:

```
CK_RV CA_GetTokenCertificates( CK_SLOT_ID slotID,
      CK_ULONG ulCertType,
      CK_BYTE_PTR pCertificate,
      CK_ULONG_PTR pulCertificateLen );
```

The ulCertType parameter is defined in the cryptoki_v2.h header file as follows:

```
#define CKHSC_CERT_TYPE_TWC                 0x00000009
#define CKHSC_CERT_TYPE_TWC2                0x0000000A
#define CKHSC_CERT_TYPE_TWC3                0x0000000B
```

Use the TWC3 macro definition for all Luna firmware 6.x and firmware 4.8.5 and later. For firmware versions earlier than 4.8.5 use theTWC2 definition.

Step2 – obtain the KEV from the target token – this ensures that the tokens are part of the same cloning domain:

```
CK_RV CA_GenerateCloningKEV(CK_SESSION_HANDLE hSession,
      CK_BYTE_PTR pKEV,
      CK_ULONG_PTR pulKEVSize);
```

The TWC obtained in Step 1 must now be passed to the target host for the next step in the cloning operation.

Step 3 – Initialize the cloning operation on the target HSM, passing in the parameters obtained from the first two steps:

```
CK_RV CA_CloneAsTargetInit(CK_SESSION_HANDLE hSession,
      CK_BYTE_PTR pTWC,
      CK_ULONG ulTWCSize,
      CK_BYTE_PTR pKEV,
      CK_ULONG ulKEVSize,
      CK_BBOOL bReplicate,
      CK_BYTE_PTR pPart1,
      CK_ULONG_PTR pulPart1Size);
```

Where

> bReplicate boolean indicates whether this is a pure cloning operation, or it involves network replication. For the Luna PCIe HSM 7, this flag should be FALSE, whereas for cloning objects to/from the Luna Network HSM 7, this flag should be set to TRUE.

This function fills in the Part1 buffer and return the size of Part1 – these parameters must be returned to the source host system to be used in the next function call.

Step 4 – Clone the desired HSM object from the source token:

```
CK_RV CA_CloneAsSource( CK_SESSION_HANDLE hSession,
      CK_ULONG hType,
      CK_ULONG hHandle,
      CK_BYTE_PTR pPart1,
      CK_ULONG ulPart1Size,
      CK_BBOOL bReplicate,
      CK_BYTE_PTR pPart2,
      CK_ULONG_PTR pulPart2Size);
```

Where

> hType should always be set to CK_CRYPTOKI_ELEMENT for cloning standard PKCS#11 objects,

> hHandle points to the object to be cloned, and the bReplicate flag is defined as in the previous step.

Pass the Part2 buffer and size variable back to the target host system to be used in the next function call.

Step 5 – Clone the HSM object to the target token:

```
CK_RV CA_CloneAsTarget(CK_SESSION_HANDLE hSession,
      CK_BYTE_PTR pKEV,
      CK_ULONG ulKEVSize,
      CK_BYTE_PTR pPart2,
      CK_ULONG ulPart2Size,
      CK_ULONG hType,
      CK_ULONG hHandle,
      CK_BBOOL bReplicate,
      CK_OBJECT_HANDLE_PTR phClonedHandle);
```

Again,

- hType should be set to CK_CRYPTOKI_ELEMENT,
- hHandle is not required for standard crypto objects (set to 0), and
- bReplicate flag is set as defined in Step 3.

IMPORTANT CONSIDERATIONS

1. The cloning process must be completed in the context of a single session opened on the source token, and one opened on the target token. For example, you must use the same session handle on the target token for Steps 3 and 5 above.
2. In general, when creating buffers for the various data items passed back from the function, a 4k buffer should be sufficient (certificate, part1, part2) but it should be possible to set these pointers to NULL, and the corresponding size parameter to zero – the function then fills the size variable with the size of the data item. The appropriate memory can then be malloc'd and the function called again to retrieve the actual data.
3. Only one HSM object can be cloned for each operation.
4. All macro-definitions indicated in this section can be found in the cryptoki_v2.h header file.

Luna HSM Cloning API CPv3 - Extensions to PKCS #11

This section provides details of the Thales Luna HSM Cloning API for these scenarios

- where the source and target token (or slot) are visible on the same host,
- where the source and target tokens are visible only to separate host systems.

In the latter case, you might need to develop significant logic in order to initiate and synchronize the cloning process between the two host systems. The CPv3 APIs are based on Luna HSM Firmware 7.7.0 Cloning Protocol version 3.

Use the information in this section, about the low-level APIs, when it is not appropriate to use the high-level CA_MigrateKeys() API in your application.

Cloning on the Same Host System

When cloning HSM objects between two tokens that are visible to the same host system, then a session can be opened on each token (slot) and the following function used:

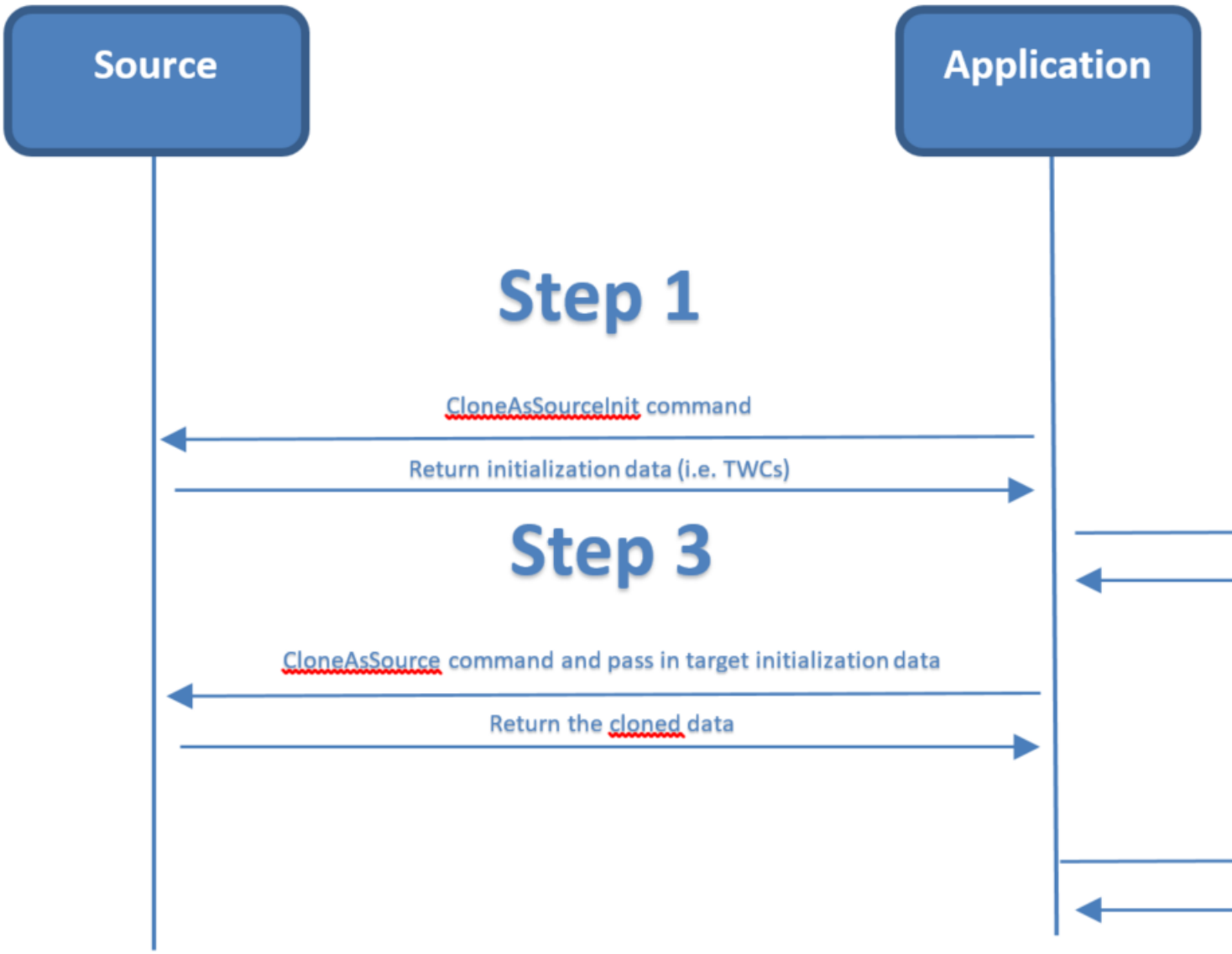
```
CK_RV CA_CloneObject( CK_SESSION_HANDLE    hTargetSession,
                      CK_SESSION_HANDLE    hSourceSession,
```

```
CK_ULONG             ulObjectType,
CK_OBJECT_HANDLE     hObjectHandle,
CK_OBJECT_HANDLE_PTR phClonedObject );
```

Where ulObjectType is CK_CRYPTOKI_ELEMENT, hObjectHandle is the handle of the object on the source token to be cloned, and phCloned Object will contain the object handle of the newly cloned object on the target token.

Cloning between Host Systems

When cloning between two HSMs that are visible only to separate host systems, additional logic must be created in order to synchronize the cloning process between the systems. How this is initiated or what protocol should be deployed is system dependent, and beyond the scope of this document. However, below is the sequence of PKCS#11 extension functions to direct to the source and target HSMs (or slots) in order to complete the cloning operation:



Step1 – Initialize the cloning operation on the source HSM, obtaining initialization data from the source HSM:

```
CK_RV CA_CloneAsSourceInit( CK_SESSION_HANDLE hSession,
                            CK_BYTE_PTR       pInParameter,
                            CK_ULONG          ulInParameterSize,
                            CK_BYTE_PTR       pOutParameter,
                            CK_ULONG_PTR      pulOutParameterSize,
                            CK_BBOOL          bReplicate);
```

Where

- hSession is the session handle to the source HSM,
- pInParameter is NULL,
- ulInParameterSize is 0,
- pOutParameter is the output of the initialization data (which is the cloning certificate, also known as TWC, in the current protocol), and
- pulOutParameterSize is the size of the initialization data
- bReplicate boolean indicates whether this is a pure cloning operation, or it involves network replication; for the Luna PCIe HSM 7, this flag should be FALSE, whereas for cloning objects to/from the Luna Network HSM 7, this flag should be set to TRUE.

Step 2 – Initialize the cloning operation on the target HSM, passing in the TWC data obtained from the first step:

```
CK_RV CA_CloneAsTargetInit( CK_SESSION_HANDLE hSession,
                            CK_BYTE_PTR       pTWC,
                            CK_ULONG          ulTWCSize,
                            CK_BYTE_PTR       pKEV,
                            CK_ULONG          ulKEVSize,
                            CK_BBOOL          bReplicate,
                            CK_BYTE_PTR       pPart1,
                            CK_ULONG_PTR      pulPart1Size);
```

Where

- hSession is the session handle to the target HSM,
- pTWC and ulTWCSize are the initialization data (i.e., pOutParameter and pulOutParameterSize) from the source HSM in step 1,
- the pKEV and ulKEVSize parameters are for backward compatibility with older HSMs, set to NULL and 0,
- the bReplicate flag is defined as in step 1.

This function fills in the Part1 buffer and return the size of Part1 – these parameters must be returned to the source host system to be used in the next function call.

Step 3 – Clone the desired HSM object from the source token:

```
CK_RV CA_CloneAsSource( CK_SESSION_HANDLE hSession,
                        CK_ULONG          hType,
                        CK_ULONG          hHandle,
```

```
                        CK_BYTE_PTR       pPart1,
                        CK_ULONG          ulPart1Size,
                        CK_BBOOL          bReplicate,
                        CK_BYTE_PTR       pPart2,
                        CK_ULONG_PTR      pulPart2Size);
```

Where

> hSession is the session handle to the source HSM,

> hType should always be set to CK_CRYPTOKI_ELEMENT for cloning standard PKCS#11 objects,

> hHandle points to the object to be cloned, and

> bReplicate flag is defined as in the previous step,

> pPart1 and ulPart1Size parameters are from the target HSM in step 2.

Pass the Part2 buffer and size variable back to the target host system to be used in the next function call.

Step 4 – Clone the HSM object to the target token:

```
CK_RV CA_CloneAsTarget( CK_SESSION_HANDLE    hSession,
                        CK_BYTE_PTR          pKEV,
                        CK_ULONG             ulKEVSize,
                        CK_BYTE_PTR          pPart2,
                        CK_ULONG             ulPart2Size,
                        CK_ULONG             hType,
                        CK_ULONG             hHandle,
                        CK_BBOOL             bReplicate,
                        CK_OBJECT_HANDLE_PTR phClonedHandle);
```

Again,

> pKEV and ulKEVSize parameters are for backward compatibility with older HSMs, set to NULL and 0,

> hType should be set to CK_CRYPTOKI_ELEMENT,

> hHandle is not required for standard crypto objects (set to 0), and

> bReplicate flag is set as defined in Step 1,

> phClonedHandle is returned as the cloned object handle.

IMPORTANT CONSIDERATIONS

1. The cloning process must be completed in the context of
   - a single session opened on the source token, and
   - one opened on the target token.

   For example, you must use the same session handle on the target token for Steps 2 and 4 above.
2. The crypto officer (CO) must have already logged in both the source and target HSMs.
3. In general, when creating buffers for the various data items passed back from the function, an 8KB buffer should be sufficient (certificate, part1, part2) but it should be possible to set these pointers to NULL, and the corresponding size parameter to zero – the function then fills the size variable with the size of the data item. The appropriate memory can then be malloc'd and the function called again to retrieve the actual data.
4. The whole cloning process must be repeated for each object.

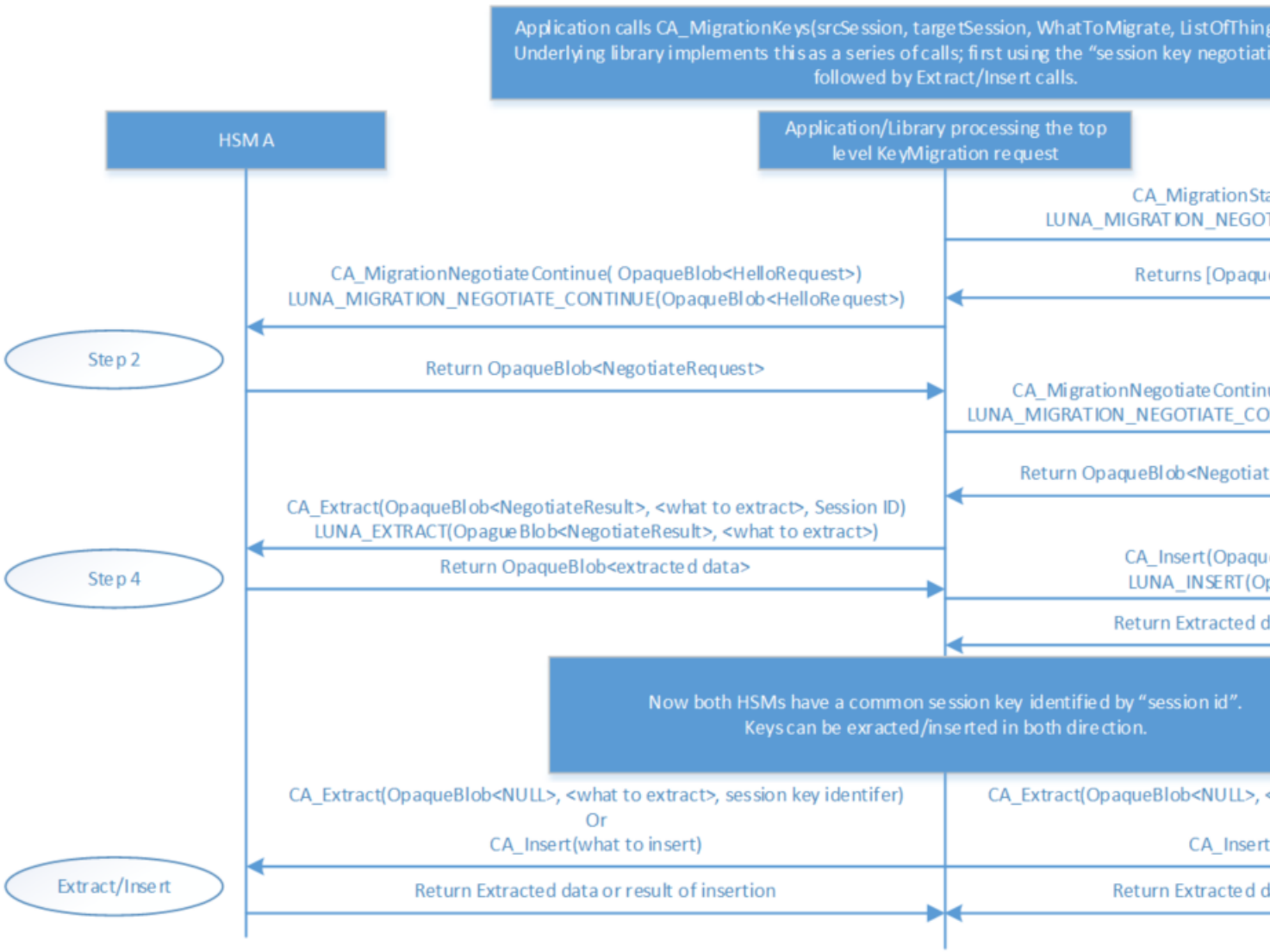
5. The source HSM and the target HSM must be in the same domain for the cloning operation to be successful.

Luna HSM Cloning API CPv4 Extensions to PKCS#11

Similar to earlier cloning protocols, CPv4 has two sets of APIs; a single top level API that performs the full key migration and a set of low level APIs that can break the protocol into two parts;

- session key establishment
  vs
- key extraction/insertion.

The low level APIs can also be used to execute the protocol on two partitions that are not directly accessible by a single client application.



Top Level API

The top level API CA_MigrateKeys() is defined below. This API clones one-or-more objects from a source session to a target session. The API can clone user objects (a.k.a. CryptokiObjects) or parameters like the SMK (a.k.a. ParamObjects). The API also supports a “flags” field to alter/change the behavior of the API when errors are encountered.

In addition to implementing CPv4, the top level API takes on the behavior that allows it to use existing key migration methods.

```
CK_RV CA_MigrateKeys(
        CK_SESSION_HANDLE       sourceSession;          // input
        CK_SESSION_HANDLE       targetSession;          // input
        CK_ULONG                migrationFlags;         // input
        CK_ULONG                numberOfObjects;        // input
        CK_MIGRATION_DATA_PTR   migrationData;          // input/output
);
```

The function parameters have the following meaning:

sourceSession	an authenticated session on the source partition.
targetSession	an authenticated session on the target partition.
migrationFlags	flags used to define the behavior the migration protocol.
numberOfObjects	the number of objects to migrate. Implicitly defines the size of the array pointed to by “migrationData”. This parameter cannot be 0.
migrationData	an array of CK_MIGRATION_DATA objects whose length is defined by “numberOfObjects”. This parameter cannot be NULL.

Migration Flags:

CKF_ CONTINUE_ON_ ERR (0x01)	If specified, the API continues attempting to clone objects if an individual object fails to clone. If the flag is not specified, the API fails after the first failure is encountered.

CKF_CONTINUE_ON_ERR (0x01) If specified, the API continues attempting to clone objects if an individual object fails to clone. If the flag is not specified, the API fails after the first failure is encountered.

```
typedef struct CK_OBJECT_MIGRATION_DATA (
        CK_ULONG          objectType;
        CK_OBJECT_HANDLE sourceHandle;
        CK_OBJECT_HANDLE targetHandle;
        CK_RV             rv
} CK_OBJECT_MIGRATION_DATA;
```

The fields of the CK_OBJECT_MIGRATION_DATA structure have the following meanings:

objectType	used to specify if the object is a CryptokiObject or a ParamObject.
sourceHandle	handle of the object to be cloned.

targetHandle	Handle of the object after it has been cloned to the target device.
rv	result of the clone operation for this specific object. This field is initialized to CKR_CLONE_NOT_ATTEMPTED for every object.

In all cases, if any object fails to clone, then the `rv` field for that object is populated with the specific error code for that failure. If the CKF_CONTINUE_ON_ERR is specified, the API continues to attempt to clone objects, otherwise the API stops attempting to clone objects. If the failure occurs in the context of cloning an individual object, the API returns CKR_OK. If an error is encountered in the core logic of CA_MigrateKeys, then the error code for that event is returned by the API, and for all objects that were not attempted to be cloned, their error code is left at CKR_CLONE_NOT_ATTEMPTED in the CK_OBJECT_MIGRATION_DATA structure.

Callers of this API should verify the `rv` field for each object to determine if the object was successfully cloned.

Low-Level APIs

This section defines the low-level APIs. The low-level APIs map to the internal APIs defined within the client library, that are used by the top level CA_MigrateKeys API.

If an application does not have access (is unable to open a session) to both the target and source HSM, then the low level APIs can be used and the application must propagate the parameters from one device to the other. Essentially, this mimics the exchange implemented by the CA_MigrateKeys API. The low level APIs should not be used under any other circumstances as their use increases the effort and complexity of maintaining backward compatibility.

Session key negotiation is intended to be an atomic operation. To avoid resource leaking, if the session key negotiation is not completed within 10 minutes the HSM cleans up resources associated with the negotiation. If no other CPv4 related calls have been made, CKR_SESSION_NEGOTIATION_EXPIRED is returned if an attempt is made to continue the negotiation after it has expired. If other CPv4 related calls have been made, the HSM might have already cleaned up the resources and CKR_SESSION_NEGOTIATION_NOT_STARTED is returned. Applications that are using the low-level APIs must be able to handle these error scenarios. The resource clean-up is described later.

Once established, a session expires after 1 hour. Any attempt to use an expired session results in CKR_SESSION_ID_EXPIRED. Applications using the low level API must be able to handle this error and are required to re-negotiate a new session.

Expired sessions are cleaned up by the HSM, as the HSM is not required to maintain expired sessions any longer than it needs for its own internal implementation reasons. Therefore, it is expected that if multiple threads/processes are using the same session, and are not synchronizing during expiry and re-negotiation scenarios, an attempt to use an expired session can result in CKR_SESSION_ID_INVALID as the HSM no longer knows about the session ID. Applications using the low level APIs must be able to handle this error scenario. As one session ID can be used/shared for an entire access, care should be taken so that each thread does not end up re-negotiating their own session ID as this would be extremely inefficient for the application and HSM.

CA_MigrationStartSessionNegotiation

This API starts a session key negotiation. The input parameter is currently not used and is defined for future-proofing.

```
CK_RV CA_MigrationStartSessionNegotiation (
  CK_SESSION_HANDLE     sessionHandle;  // input
```

```
  CK_ULONG              inputLength;    // input
  CK_BYTE_PTR           input;          // input
  CK_ULONG_PTR          step;           // output
  CK_ULONG_PTR          outputLength;   // intput/output
  CK_BYTE_PTR           output;         // output
)
```

The function parameters have the following meaning:

sessionHandle	an authenticated session on the partition on HSM B.
inputLength	the length of the buffer pointed to by “input”. With CPv4, this value must be 0. However the APIs and library support passing this value to the HSM. If this value is not zero, then a valid memory buffer must be pointed to by “intput”.
input	a memory buffer of size “inputLength”. With CPv4, this value must be NULL. However the APIs and library support passing this value to the HSM. If “inputLength” is not zero, then this pointer must point to a valid memory buffer.
step	a “step” identifier used by the HSM to identify the step of the protocol being returned by the specific call to this API. The value is used by the HSM to identify the content of the opaque blob referred to by “output”.
outputLength	defines the length of the memory buffer pointed to by “output”. This parameter cannot be NULL. If “output” is NULL, this parameter isupdated with the size of the memory buffer required.
output	a pointer to a memory buffer of size “outputLength”. This pointer can be set to NULL to request the length of the required buffer.

CA_MigrationContinueSessionNegotiation

This is called to continue the negotiation, however when it is first called on the second HSM, technically it starts the negotiation on the second HSM.

As the API is called from one HSM to the next, all of the output values are passed to the “other” HSM as input values.

The first call to CA_MigrationContinueSessionNegotiation invokes a session ID for the session being negotiation. All following calls to CA_MigrationContinueSessionNegotiation are required to pass in the same session ID.

When the negotiation is complete, status=done is returned. The content of the output values must be passed in to the other HSM as input to the first call to either CA_Extract or CA_Insert to complete the negotiation on the other HSM.

```
CK_RV MigrationContinueSessionNegotiation (
  CK_SESSION_HANDLE     sessionHandle;
  CK_ULONG              inputStep;
  CK_ULONG              inputLength;
  CK_BYTE_PTR           input;
  CK_ULONG_PTR          sessionIdInputLength;   // input
  CK_BYTE               sessionIdInput;         // input
  CK_ULONG_PTR          outputStep;             // output
  CK_ULONG_PTR          outputLength;           // input/output
  CK_BYTE_PTR           output;                 // output
  CK_ULONG              status;                 // output
```

```
  CK_ULONG_PTR          sessionIdOutputLength;  // output
  CK_BYTE               sessionIdOutput;        // output
)
```

The function parameters have the following meaning:

sessionHandle	an authenticated session on the partition on HSM A or HSM B depending on which step of the protocol is being implemented.
inputStep	the step identifier used by the HSM to identify the content of the “input” memory buffer.
inputLength	the length of the buffer pointed to by “input”. This value cannot be 0.
input	a memory buffer of size “inputLength”. This value cannot be NULL.
sessionIdInputLength	defines the length of the memory buffer pointed to by “sessionIdInput.
sessionIdInput	the Identifier for the session used to extract/insert key blobs. During a negotiation phase, the first time this API is called, this length+value pair can be NULL and zero. For all following calls to this API, the value returned via the sessionIdOutput and sessionIdOutputLength parameters should be passed in via this length+value pair.
outputStep	the step identifier used by the HSM to identify the content of the “output”.
outputLength	defines the length of the memory buffer pointed to by “output”. This parameter cannot be NULL. If “output” is NULL, this parameter is updated with the size of the memory buffer required.
output	a pointer to a memory buffer of size “outputLength”. This pointer can be set to NULL to request the length of the required buffer. tmp tmp
status	tmp
sessionIdOutputLength	defines the length of the memory buffer pointed to by “sessionIdOutput”. This parameter cannot be NULL. If “sessionIdOutput” is NULL, this parameter is updated with the size of the memory buffer required.
sessionIdOutput	the Identifier for the session used to extract/insert key blobs. If this parameter is not NULL, then this buffer receives the session identifier for the session being negotiated.

This API can return more than one piece of output data. Simplify the application and the API implementation, when querying the required buffer size, by providing a NULL pointer; all possible output fields must be queried at the same time.

CA_MigrationCloseSession

This API terminates a session. When it is called, the session key and all of its context/state is deleted. If the session key does not exist, no error is returned. This is because some implementations might proactively clean up sessions that have expired, so it is expected that by the time this API is called, the session might no longer exist. CKR_SESSION_ID_INVALID is returned.

```
CK_RV CA_MigrationSessionKeyDelete (
  CK_SESSION_HANDLE     sessionHandle;
  CK_ULONG_PTR          sessionIdLength;
  CK_BYTE               sessionId;
)
```

The function parameters have the following meaning:

sessionHandle	an authenticated session on the partition on HSM A or HSM B depending on which step of the protocol is being implemented.
sessionIdLength	the length of the session ID.
sessionId	the Identifier for the session to be closed.

CA_Extract

This API extracts objects or internal CSPs using the specified session id.

This is the same API used for SIM3 in the past. The API functionality is defined by a mechanism and a mechanism parameter which allows for “any” functionality to be defined on a per-mechanism basis. This makes it ideal for re-use for the CPv4 extract/insert operations and is consistent with the PKCS#11 API.

```
CK_RV CA_Extract (
  CK_SESSION_HANDLE     sessionHandle;          // input
  CK_MECHANISM          pMechanism;             // input/output
)
```

Mechanism and parameter structure.

```
#define CKM_CPV4_EXTRACT                        0x80000208
CK_CPV4_EXTRACT_PARAMS {
  CK_ULONG_PTR          sessionIdLength;        // input
  CK_BYTE               sessionId;              // input
  CK_ULONG              inputLength;            // input
  CK_BYTE_PTR           input;                  // input
  CK_ULONG              extractionFlags;        // input
  CK_ULONG              numberOfObjects         // input
  CK_ULONG_PTR          objectType;             // input
  CK_ULONG_PTR          objectHandle;           // input
  CK_RV_PTR             result;                 // output
  CK_ULONG_PTR          keyBlobLength;          // output
  CK_BYTE_PTR_PTR       keyBlob;                // output
}
```

The mechanism parameters have the following meanings:

sessionIdLength	the length of the session ID.
sessionId	the Identifier for the session to be used to extract the key blob(s).
inputLength	the length of data pointed by “input”
input	when executing step 4 in the API flow, “input” and “inputLength” must refer to a valid memory location with a non-zero size; specifically the output of the final call to CA_ MigrationContinueSessionNegotiation. All other calls to this API should be NULL and 0.

extractionFlags	flags used to define how errors are handled during extraction. The default value is 0, which is to return on the first error. The flag CKF_CONTINUE_ON_ERR (0x01) can be specified to indicate that the HSM should proceed trying to extract objects if any single object fails.
numberOfObjects	number of objects to be extracted
objectType	an array of object types to define the type of objects pointed to by the array of object handles. Possible values are CK_CRYPTOKI_ELEMENT and CK_PARAM_ELEMENT.
objectHandles	an array of object handles, defining the objects to be extracted.
result	an array of result codes defining the result of each object extraction. This field should be initialized to CKR_CLONE_NOT_ATTEMPTED for all objects.
keyBlobLength	an array of length fields that correspond to the array of memory buffers pointed by “keyBlobs”. This value and the value pointed to by each array cannot be NULL.
keyBlob	an array of the memory buffers to receive the extracted key blobs. This value cannot be NULL. If all of the array elements are NULL, then the required buffer size is returned in keyBlobLength array. Otherwise all values in the array must be non-NULL.

If during the first call to CA_Extract, the final step of session key negotiation fails, then a number of possible error codes might be returned, depending on specific implementation details. The following CPv4 specific error codes are expected:

> CKR_SESSION_NEGOTIATION_NO_PSK

> CKR_SESSION_NEGOTIATION_NO_ROOTS

> CKR_SESSION_NEGOTIATION_NO_ALGS

> CKR_SESSION_NEGOTIATION_EXPIRED

> CKR_SESSION_NEGOTIATION_NOT_STARTED

If a specified session ID does not exist, CKR_SESSION_ID_INVALID is returned.

If the specified session ID exists but is expired, CKR_SESSION_ID_EXPIRED is returned.

The “extractionFlags” field, reuses the migration flag CKF_CONTINUE_ON_ERR (0x01), which if it is specified, causes the API to continue attempting to extract objects if an individual object fails. If the flag is not specified, the API fails after the first failure is encountered.

If an error is encountered trying to extract an object, then that error is set in the result field that corresponds to that object. If the CKF_CONTINUE_ON_ERR is specified, then the API continues attempting to extract objects. Otherwise, the API stops and returns CKR_OK. If an error is encountered in the core logic of CA_Extract, than the error code for that event is returned by the API and, for all objects that were not attempted to be cloned, they have their error code set to CKR_CLONE_NOT_ATTEMPTED in the ”result” array.

Callers of this API should verify the `rv` field for each object to determine if the object was successfully extracted.

CA_Insert

This API inserts objects, or internal CPS, using the specified session id.

This is the same API that is used for SIM3. The API functionality is defined by a mechanism parameter that allows for “any” functionality to be defined.

```
CK_RV CA_Insert (
  CK_SESSION_HANDLE     sessionHandle;          // input
  CK_MECHANISM          pMechanism;             // input/output
)
```

Define Mech and param structure.

```
#define CKM_CPV4_INSERT                         0x80000209
CK_CPV4_INSERT_PARAMS {
  CK_ULONG_PTR          sessionIdLength;        // input
  CK_BYTE               sessionId;              // input
  CK_ULONG              insertionFlags;         // input
  CK_ULONG              numberOfObjects;        // input
  CK_ULONG_PTR          storageType;            // input
  CK_ULONG_PTR          objectType;             // input
  CK_ULONG_PTR          keyBlobLength;          // input
  CK_BYTE_PTR_PTR       keyBlob;                // input
  CK_RV_PTR             result;                 // output
  CK_ULONG_PTR          objectHandle;           // output
}
```

The mechanism parameters have the following meanings:

sessionIdLength	the length of the session ID.
sessionId	the Identifier for the session to be used to insert the key blob(s).
insertionFlags	flags used to define how errors are handled during extraction. The default value is 0, which is to return on the first error. The flag CKF_CONTINUE_ON_ERR (0x01) can be specified to indicate that the HSM should proceed trying to extract objects if any single object fails.
numberOfObjects	number of objects to be extracted
storageType	an array of storage type identifiers used to define how the inserted object should be inserted.
objectType	an array of object types to define the type of objects pointed to by the array of object handles. Possible values are CK_CRYPTOKI_ELEMENT and CK_PARAM_ELEMENT.
keyBlobLength	an array of length fields that correspond to the array of memory buffers pointed by “keyBlobs”. This value and the value pointed to by each array cannot be NULL.

keyBlob	an array of the memory buffers that contain key blob. This value and each array element cannot be NULL.
result	an array of result codes defining the result of each object insertion. This field should be initialized to CKR_CLONE_NOT_ ATTEMPTED for all objects.
objectHandle	an array of object handles, to receive the object handle for the inserted objects.

If during the first call to CA_Insert, the final step of session key negotiation fails, then a number of possible error codes might be returned depending on specific implementation details. The following CPV4 specific error codes are expected:

> CKR_SESSION_NEGOTIATION_NO_PSK

> CKR_SESSION_NEGOTIATION_NO_ROOTS

> CKR_SESSION_NEGOTIATION_NO_ALGS

> CKR_SESSION_NEGOTIATION_EXPIRED

> CKR_SESSION_NEGOTIATION_NOT_STARTED

The “insertionFlags” field, reuses the migration flag CKF_CONTINUE_ON_ERR (0x01), which if it is specified, the API continues attempting to insert objects if an individual object fails. If the flag is not specified, the API fails after the first failure is encountered.

If an error is encountered trying to insert an object, then that error is set in the result field that corresponds to that object. If the CKF_CONTINUE_ON_ERR is specified, then the API continues attempting to insert objects, otherwise the API stops and returns CKR_OK. If an error is encountered in the core logic of CA_Insert, then the error code for that event is returned by the API and, for all objects that were not attempted to be cloned, their error code is set to CKR_CLONE_NOT_ATTEMPTED in the ”result” array.

Callers of this API should verify the `rv` field for each object to determine if the object was successfully inserted.

New PKCS#11 Error Code Summary

This section provides a summary of all of the error codes introduced by the CPv4 APIs, and their intended meanings.

> CKR_SESSION_NEGOTIATION_NO_PSK

This error is returned when no common PSK (pre-shared key) can be found. It is returned when both devices provide PSK IDs and no matching ID can be found, as well as when one-or-both devices provide LEET and no cryptographic match can be found.

> CKR_SESSION_NEGOTIATION_NO_ROOTS

This error is returned when a device cannot find a known/supported Root.

> CKR_SESSION_NEGOTIATION_NO_ALGS

This error is returned when one device cannot find a supported algorithm suite in the provided algorithms suites in the relevant step of the protocol. This can also be returned if an unsupported algorithm suite is found in the NegotiationRequest.

- CKR_SESSION_NEGOTIATION_EXPIRED

  This error is returned when an attempt is made to continue session negotiation after the session negotiation state has expired.

- CKR_SESSION_NEGOTIATION_INVALID

  This error is returned when unexpected/invalid messages are encountered during negotiation. This can happen when an attempt is made to continue session negotiation, using a session UID that does not exist. This can also occur when an attempt to extract objects from HSM B is made before an object is inserted.

- CKR_SESSION_ID_INVALID

  This error is returned when an attempt is made to extract/insert key blobs with a session ID that is not known by the device.

- CKR_SESSION_ID_EXISTS

  This error is returned when an attempt is made to negotiate a session (the NegotiateRequest message) using a session UID that already exists in the device.

- CKR_SESSION_ID_EXPIRED

  This error is returned when an attempt is made to use extract/insert key blobs using a session ID that is expired. This error will rarely occur on Luna HSMs as Luna HSMs perform cleanup of expired sessions on every CPv4-related command.

- CKR_PROTOCOL_DISABLED

  This error is returned when requested protocol (CPv3 or CPv4) is disabled at the partition configuration level.

- CKR_SESSION_NEGOTIATION_NO_SESSION_DURATION

  This error is returned when no commonly supported session duration can be found during session negotiation.

- CKR_SESSION_NEGOTIATION_NO_KDF

  This error is returned when no commonly supported KDF can be found during session negotiation.

- CKR_SESSION_NEGOTIATION_NO_ENCODING

  This error is returned when no commonly supported object encoding can be found during session negotiation.

- CKR_SESSION_NEGOTIATION_NO_CHAIN_ATT

  This error is returned when the required certification chain attestation mechanisms cannot be found during session negotiation.

- CKR_SESSION_NEGOTIATION_NO_EPHERMAL_KEY

  This error is returned when the provided ephemeral key blob is invalid; invalid type, invalid curve or invalid ECC point.

- CKR_ATTESTATION_EXPIRED

  This error is returned when an expired attestation message is encountered.

- CKR_CLONE_NOT_ATTEMPTED

  If no attempt is made to clone an object, which could happen due to negotiation failure or other errors unrelated to the core CPv4 logic and implementation, this error is returned.

- CKR_TIME_NOT_INITIALIZED

The source and target HSMs must have their time within 3 seconds of each other, or CPv4 negotiation fails. If your HSMs' hosts are synchronized with a reliable time source, you can use HSM Policy 57 Allow Sync with Host Time to prevent drift.

When processing parameters, at the API level in the client or in the HSM, error codes defined for operations and features of prior releases might also be returned when invalid parameters are detected.

Co-existence with Existing PKCS#11 APIs

The pre-existing cloning protocols (CPv1 and CPv3) have two sets of APIs; the low level API and a top level API.

Top Level:	CA_ClonePrivateKey
	CA_CloneObject
Low Level:	CA_CloneAsSourceInit
	CA_CloneAsTargetInit
	CA_CloneAsSource
	CA_CloneAsTarget

In the pre-CPv4 client library, all calls to the top level APIs end up in a block of logic that determines which key migration method is supported by both devices (CPv1, CPv3, SKS), and uses the correct low level APIs to execute the key transfer.

CPv4 adheres to this model. The top level API for CPv4 also calls in to the same logic that determines which key migration method is supported by both devices (CPv1, CPv3, SKS, CPv4), and uses the correct low level APIs to execute the key transfer.

The pre-CPv4 low level APIs for cloning include logic to remap the calls to use SKS, in order to increase compatibility with old applications. The CPv4 low level APIs do not include any remapping as they are new APIs and new applications are encouraged to use the CPv4 APIs correctly, and strongly encouraged to use only the top level CPv4 API.

PSK APIs

This section defines the APIs and ICD commands that are defined to support the domain management functionality.

PKCS#11 Extension APIs

This section defines the PKCS#11 Extension APIs.

PWD vs Local and Remote PED

Similar to existing APIs like C_Login, CA_ResetPIN, etc, these APIs can receive a value (the KCV) as a typed string (like a password) or from a PED key.

When providing a PED key-based KCV, the KCV parameters defined below must be set to NULL and 0. If the KCV is being entered from a local PED, that is all that is required. If the KCV is to be entered via a Remote PED, then a Remote PED connection must be setup and a PED ID defined for the connection. Just like the other APIs

that may make use of a Remote PED, the APIs defined in this section will pick up and use whichever PED ID is defined. Setting up Remote PED connections and PEDs is covered elsewhere, in the management/administration sections of these docs.

CA_AddKCV

The CA_AddKCV API can be used by the partition's security officer to add additional domains to the partition. If the "Allow Extended Domain Management" policy is disabled, then any attempt to use this API to add more than one domain returns CKR_DOMAIN_MANAGMENT_NOT_ALLOWED. If the provided domain label already exists, this API returns CKR_DOMAIN_LABEL_ALREADY_EXISTS.

```
CK_RV CA_AddKCV (
  CK_SESSION_HANDLE     session;                // input
  CK_ULONG              ulKCVLength;            // input
  CK_BYTE_PTR           pKCV;                   // input
  CK_ULONG              ulLabelLength;          // input
  CK_BYTE_PTR           pLabel;                 // input
  CK_BBOOL              bMakePrimary            // input
);
```

The function parameters have the following meaning:

session	a session on the partition authenticated by the partition's security officer
ulKCVLength	the length of the KCV pointed to by pKCV. If the KCV is to be entered via a PED, then the length must be zero.
pKCV	a pointer to a byte array the contains the KCV value. If the KCV is to be entered via a PED, this pointer must be set to NULL.
ulLabelLength	the length of the label pointed to by pLabel. The label length cannot be 0.
pLabel	a pointer to a buffer that contains the label for the domain to be added. The label must be between 1 and 32 bytes in length and is NOT a NULL terminated string. This parameter cannot be NULL.
bMakePrimary	flag to indicate that the new domain should be the primary domain.

CA_ChangeKCVLabel

The CA_ChangeKCVLabel API can be used by the partition's security officer to change the label of a KCV. The primary use of this API is to add a label to a pre-existing KCV that does not already have a label. It can also be used to change an existing label of a KCV, which may be useful when merging/splitting domains and the same domain label has been used for different KCV values. If the provided domain label already exists, this API returns CKR_DOMAIN_LABEL_ALREADY_EXISTS. If the specified domain label does not exist, this API returns CKR_DOMAIN_LABEL_INVALID. To change a domain that does not have a label, the "old" label parameters must be set to 0 and NULL.

```
CK_RV CA_ChangeKCVLabel (
  CK_SESSION_HANDLE     session;                // input
  CK_ULONG              ulOldLabelLength;       // input
  CK_BYTE_PTR           pOldLabel;              // input
  CK_ULONG              ulNewLabelLength;       // input
```

```
  CK_BYTE_PTR           pNewLabel               // input
);
```

The function parameters have the following meaning:

session	a session on the partition authenticated by the partition's security officer
ulOldLabelLength	the length of the label pointed to by pOldLabel. If pOldLabel is NULL, then this value must be 0. Otherwise this value must be set to the length of the label pointed to by pOldLabel.
pOldLabel	a pointer to a buffer that contains the label for the domain to be re-labelled. If ulOldLabelLength is 0, then this value must be NULL. Otherwise this value must point to a buffer containing a label that is between 1 and 32-bytes in length.
ulNewLabelLength	the length of the label pointed to by pNewLabel. The label length cannot be 0.
pNewLabel	a pointer to a buffer that contains the new label for the domain. The label must be between 1 and 32 bytes in length and is NOT a NULL terminated string. This parameter cannot be NULL.

CA_DeleteKCV

The CA_DeleteKCV API can be used by the partition's security officer (PO) to delete domains on the partition. If the "Allow Extended Domain Management" policy is disabled, then any attempt to use this API returns CKR_DOMAIN_MANAGMENT_NOT_ALLOWED. If the specified domain label does not exist, this API returns CKR_DOMAIN_LABEL_INVALID. To delete a domain that does not have a label, the label parameters must be set to 0 and NULL.

```
CK_RV CA_DeleteKCV (
  CK_SESSION_HANDLE     session;                // input
  CK_ULONG              ulLabelLength;  // input
  CK_BYTE_PTR           pLabel                  // input
);
```

The function parameters have the following meaning:

session	a session on the partition authenticated by the partition's security officer
ulLabelLength	the length of the label pointed to by pLabel. If pLabel is NULL, then this parameter must be set to 0. Otherwise this parameter must be set to the length of the label pointed to by pLabel.
pLabel	a pointer to a buffer that contains the label for the domain to be deleted. If ulLabelLength is 0, then this parameter must be set to NULL. Otherwise this parameter must point to a buffer that containers the label.

CA_GetKCVLabels

The CA_GetKCVLabels API can be used by any role to retrieve the domain labels.

```
CK_RV CA_GetKCVLabels (
  CK_SESSION_HANDLE     session;                // input
  CK_ULONG_PTR          ulNumberOfLabels;       // input/output
  CK_ULONG_PTR          ulLabelLengths;         // input/output
  CK_BYTE_PTR*          pLabels                 // input/output
);
```

The function parameters have the following meaning:

session	a session on the partition authenticated by the partition's security officer
ulNumberOfLabels	a pointer to receive the number of labels. This parameter cannot be NULL. When requesting the number of labels, this parameter must be set to CK_ULONG value that is set to 0 and it will be populated with the number of labels. If a non-zero value is provided, then it must define the size of the ulLabelLengths and pLabels arrays. If the non-zero value provides is too small, then CKR_BUFFER_TOO_SMALL is returned and this parameter will be populated with the number of labels.
ulLabelLengths	a pointer to an array to receive the lengths of each label. When retrieving the number of labels, this parameter is ignored. Otherwise, it must be set to an array of ulNumberOfLabels CK_ULONG values. On output, the array will be populated with the length of each label.
pLabels	a pointer to an array of CK_BYTE_PTR. When retrieving the number of labels, this parameter is ignored. Otherwise, it must be set to an array of length ulNumberOfLabels, where each element of the array is at least 32 bytes in size. On output, each element of the array is populated with the domain label.

Error Codes

Here are the error codes introduced by the domain management APIs.

> CKR_DOMAIN_MANAGEMENT_NOT_ALLOWED

This error is returned when extended domain management features are attempted and the "Allow Extended Domain Management" partition policy is disabled. Specifically, this error is returned when adding more than one domain, deleting a domain or attempting to assign a domain that is of a different authentication type than the HSM (i.e. specifying a PED domain on a pwd-auth HSM).

> CKR_DOMAIN_LABEL_INVALID

This error is returned when the provided domain label does not match a domain that is currently assigned to the partition.

> CKR_DOMAIN_LABEL_ALREADY_EXISTS

This error is returned when the label provided for a new domain, or when changing the label of an existing domain, already exists. This includes trying to create a domain with no label as well as removing a domain's label when there is already a domain with no label.

> CKR_DOMAIN_MAX_REACHED

This error is returned when an attempt to add a domain is made, but the limit has already been reached. The current limit is 3.

> CKR_DOMAIN_NO_PRIMARY

This error is returned when the HSM is unable to locate the primary domain of a partition. This error is unexpected and should never happen in a production environment.

> CKR_DOMAIN_EXTRA_PRIMARY

This error is returned when the HSM finds more than one primary domain in a partition. This error is unexpected and should never happen in a production environment.

When processing parameters, at the API level in the client or in the HSM, the error codes that have previously been available for Luna might be returned when invalid parameters are detected (DATA_INVALID, and/or INVALID_POINTER when validating input parameters, USER_NOT_AUTHORIZED, etc.).

Secure External Scalable Key Storage Extensions

Two forms of authorization are supported: no authorization, and M-of-N passwords. Note that the password form of authorization does not cryptographically protect the key material, it consists of a comparison between the N encrypted values stored in the header versus M plain-text passwords specified upon insertion. The form of authorization data will continue to be identified using the following definitions.

> **NOTE** Individual SKS blobs are limited to 64KB in size. Large groups of keys, or larger data objects might need to be split across multiple blobs for extraction or insertion.

```
typedef CK_ULONG CKA_SIM_AUTH_FORM;
#define CKA_SIM_NO_AUTHORIZATION 0  // no authorization needed
#define CKA_SIM_PASSWORD         1  // plain-text passwords
```

When **0** is chosen for authForm, the `pulAuthSecretSizes` and `ppbAuthSecretList` PTRs are both NULL, so no passwords are required.

When **1** is chosen for authForm, then `pulAuthSecretSizes` is an array of string lengths for each plain-text string password in the array of strings pointed to by `ppbAuthSecretList`.

The existing extract and insert API functions remain unchanged..

CA_SIMExtract

```
CK_RV CK_ENTRY CA_SIMExtract( CK_SESSION_HANDLE     hSession,
                              CK_ULONG              ulHandleCount,
                              CK_OBJECT_HANDLE_PTR  pHandleList,
                              CK_ULONG              ulAuthSecretCount, // N value
                              CK_ULONG              ulAuthSubsetCount, // M value
                              CKA_SIM_AUTH_FORM     authForm,
                              CK_ULONG_PTR          pulAuthSecretSizes,
                              CK_BYTE_PTR           *ppbAuthSecretList,
```

```
                                CK_BBOOL              deleteAfterExtract,
                                CK_ULONG_PTR          pulBlobSize,
                                CK_BYTE_PTR           pBlob );
```

CA_SIMExtract takes a list of object handles, extracts them using the given blob (**b**inary **l**arge **ob**ject) authorization data for protection and returns the extracted set of objects as a single data blob. The objects are left on the partition or destroyed, based on the value of the `delete-after-extract` flag.

The `authDataCount` parameter defines the N value. The `subsetRequired` parameter defines the M value. The `authDataList` parameter should have N entries in it, if it is used.

An empty handle list attempts to extract all the keys on the partition, as long as the size limit is respected.

CA_SIMInsert

```
CK_RV CK_ENTRY CA_SIMInsert( CK_SESSION_HANDLE     hSession,
                             CK_ULONG              ulAuthSecretCount, // M value
                             CKA_SIM_AUTH_FORM     authForm,
                             CK_ULONG_PTR          pulAuthSecretSizes,
                             CK_BYTE_PTR           *ppbAuthSecretList,
                             CK_ULONG              ulBlobSize,
                             CK_BYTE_PTR           pBlob,
                             CK_ULONG_PTR          pulHandleCount,
                             CK_OBJECT_HANDLE_PTR  pHandleList );
```

CA_SIMInsert takes a previously extracted blob as input, validates the blob authorization data, inserts the objects contained in the blob into the HSM, and returns the list of handles assigned to the objects.

The `authDataCount` value should equal M, and the `authDataList` should have M elements in it.

CA_SIM_MultiSign

```
CK_RV CK_ENTRY CA_SIMMultiSign( CK_SESSION_HANDLE   hSession,
                                CK_MECHANISM_PTR    pMechanism,
                                CK_ULONG            ulAuthSecretCount, // M value
                                CKA_SIM_AUTH_FORM   authForm,
                                CK_ULONG_PTR        pulAuthSecretSizes,
                                CK_BYTE_PTR         *ppbAuthSecretList,
                                CK_ULONG            ulBlobSize,
                                CK_BYTE_PTR         pBlob,
                                CK_ULONG            ulInputDataCount,
                                CK_ULONG_PTR        pulInputDataLengths,
                                CK_BYTE_PTR         *ppbInputDataList,
                                CK_ULONG_PTR        pulSignatureLengths,
                                CK_BYTE_PTR         *ppbSignatureList );
```

CA_SIM_MultiSign takes a previously extracted blob as input, validates the authorization data, then uses the key material in the given key blob to sign the various pieces of data in the input data table, returning the signatures through the signature table. Note that the key exists on the HSM only during the processing of the command and does not persist afterward.

If the blob contains more than one key, the key in the blob that is suitable for the requested signature mechanism is used to sign the data. If there are multiple candidates, an error is returned.

The blob authorization data parameters are handled as for the SIMInsert function.

Per-key authorization data is not passed in to the HSM with this call to authorize the inserted key object. If the inserted key has per-key authorization attribute defined, this function is tied to access-based per-key authorization.

CA_SMKRollover

```
CK_RV CA_SMKRollover( CK_SESSION_HANDLE ulSessionNumber,
                      CK_ULONG ulValue
)
```

This operation moves the current primary SMK to the Rollover SMK location and creates a new primary SMK. It starts with value 1, and ends with value 0. The open time between start and end is intended for inserting any blobs that were encrypted with the old SMK and [re-]extracting them with the new SMK. If no such blobs exist, or have value, then issuing a SMKRollover start (1) followed immediately by a stop (0) just creates a new SMK.

Derivation of Symmetric Keys with 3DES_ECB

Luna supports derivation of symmetric keys by the encryption of "diversification data" with a base key. Access to the derivation functionality is through the PKCS #11 C_DeriveKey function with the CKM_DES3_ECB and CKM_DES_ECB mechanism. Diversification data is provided as the mechanism parameter. The derived key can be any type of symmetric key. The encrypted data forms the CKA_VALUE attribute of the derived key. A template provided as a parameter to the C_DeriveKey function defines all other attributes.

Rules for the derivation are as follows:

- The Base Key must be of type CKK_DES2 or CKK_DES3 when using CKM_DES3_ECB. It must be of type CKK_DES when using CKM_DES_ECB.
- The base key must have its CKA_DERIVE attribute set to TRUE.
- The template for the derived key must identify the key type (CKA_KEY_TYPE) and length (CKA_VALUE_LEN). The type and length must be compatible. The length can be omitted if the key type supports only one length. (E.g., If key type is CKK_DES2, the length must either be explicitly defined as 16, or be omitted to allow the value to default to 16). Other attributes in the template must be consistent with the security policy settings of the Luna PCIe HSM 7.
- The derivation mechanism must be set to CKM_DES3_ECB or CKM_DES_ECB, the mechanism parameter pointer must point to the diversification data, and the mechanism parameter length must be set to the diversification data length.
- The diversification data must be the same length as the key to be derived, with one exception. If the key to be derived is16 bytes, the base key is CKK_DES2 and the diversification data is only 8 bytes, then the data is encrypted twice - once with the base key and once with the base key with its halves reversed. Joining the two encrypted pieces forms the derived key.
- If the derived key is of type CKK_DES, CKK_DES2 or CKK_DES3, odd key parity is applied to the new key value immediately following the encryption of the diversification data. The encrypted data is taken as-is for the formation of all other types of symmetric keys.

PKCS#11 Extension HA Status Call

A Luna extension to the PKCS#11 standard allows query of the HA group state.

This is the HA Status call available in Luna HSM Client 10.6.0 and older. It operates sequentially, and was useful in more forgiving environments. For near-real-time status in demanding environments like 5G and other high-uptime applications see "PKCS#11 Extension HA Status Call - improved" on the next page instead.

Function Definition

```
CK_RV CK_ENTRY CA_GetHAState( CK_SLOT_ID slotId, CK_HA_STATE_PTR pState );
```

The structure definitions for a CK_HA_STATE_PTR and CK_HA_MEMBER are:

```
typedef struct CK_HA_MEMBER{
CK_ULONG memberSerial;
CK_RV memberStatus;
}CK_HA_MEMBER;


typedef struct CK_HA_STATUS{
CK_ULONG groupSerial;
CK_HA_MEMBER memberList[CK_HA_MAX_MEMBERS];
CK_USHORT listSize;
}CK_HA_STATUS;
```

See the JavaDocs included with the software for a description of the Java methods derived from this cryptoki function.

PKCS#11 Extension HA Status Call - improved

A Luna extension to the PKCS#11 standard allows query of the HA group state in near real-time.

Unlike the old, similar-named CA_GetHAState, this function **CA_GetCurrentHAState**

> performs all checks in parallel

> does not open a session with each member partition and then close it -- aside from the overhead of the old approach, it also took place behind the auto-recovery mechanism, which could result in orphan sessions, so the newer approach avoids that risk

> rather, this API

- first, checks the driver instance (there is one for every member), and only if that is valid does it go on (for that member) and
- second, checks socket status attached to the instance, and if that is OK, then
- third, uses a lightweight call (via existing session) to confirm that the member is reachable and usable; this call is sent only to those members that have the ability to handle it (determined by their passing the first two checks).

If any of those checks fails the check on that member stops trying and the failed status goes to the report that is displayed following the configured timeout. Timout defaults to 3 seconds for the check of all group members, but can be set as high as 60 seconds by the statusTimeout configuration option in the *HAConfiguration* section of the Chrystoki.conf / crystoki.ini file (see Configuration File Summary).

The default of 3 seconds assumes the environment is supportive, meaning an operational environment with good IT conditions (including network, CPU, RAM, storage), and reasonable levels of perturbations, within 20% of optimal conditions.

> **NOTE** This feature includes internal fail-safes to avoid race conditions, but invocation from an outside application *must be threadsafe*.

At the time of introduction (December 2023) this feature is implemented for C++.

Function Definition

```
CK_RV CK_ENTRY CA_GetCurrentHAState( CK_SLOT_ID slotId, CK_HA_STATE_PTR pState );
```

The structure definitions for a CK_HA_STATE_PTR and CK_HA_MEMBER are:

```
typedef struct CK_HA_MEMBER{
CK_ULONG memberSerial;
CK_RV memberStatus;
}CK_HA_MEMBER;


typedef struct CK_HA_STATUS{
CK_ULONG groupSerial;
CK_HA_MEMBER memberList[CK_HA_MAX_MEMBERS];
CK_USHORT listSize;
}CK_HA_STATUS;
```

ECIES Enhancement for HKDF

A new KDF definition is added with Luna HSM Firmware 7.8.7, complying with RFC 5869, to expand the utility of the prior ECIES implementation.

The implementation avoids conflict with existing Luna headers:

```
#define CKD_HKDF_SHA256  0x80000020


/** Mechanism parameter structure for ECIES */
typedef struct CK_ECIES_PARAMS
{
   /** Diffie-Hellman primitive used to derive the shared secret value */
    CK_EC_DH_PRIMITIVE dhPrimitive;  // CKDHP_STANDARD (private key and ephemeral public as
input

   /**   key derivation function used on the shared secret value */
    CK_EC_KDF_TYPE kdf;  // CKD_HKDF_SHA256

   /** the length in bytes of the key derivation shared data */
    CK_ULONG ulSharedDataLen1;  // sizeof HKDF Shared Data 1
   /** the key derivation padding data shared between the two parties */
    CK_BYTE_PTR pSharedData1; //HKDF optional


   /** the encryption scheme used to transform the input data */
    CK_EC_ENC_SCHEME encScheme;   // CKES_AES_CTR

   /** the bit length of the key to use for the encryption scheme */
    CK_ULONG ulEncKeyLenInBits;   // 256

   /** the MAC scheme used for MAC generation or validation */
    CK_EC_MAC_SCHEME macScheme;   // CKMS_HMAC_SHA256

   /** the bit length of the key to use for the MAC scheme */
    CK_ULONG ulMacKeyLenInBits;   // 256

   /** the bit length of the MAC scheme output */
    CK_ULONG ulMacLenInBits;   // 256
```

```
   /** the length in bytes of the MAC shared data */
    CK_ULONG ulSharedDataLen2;  // 0

   /** the MAC padding data shared between the two parties */
    CK_BYTE_PTR pSharedData2;  // NULL

} CK_ECIES_PARAMS;
```

Luna Proprietary Extension to CKM_ECIES

```
typedef struct CK_ECIES_PARAMS_EXT2
{
   /** Legacy ECIES  parameters*/
    CK_ECIES_PARAMS eciesParams;  // as above

    /** reference encryption scheme structure extension = AES_CTR params */
    CK_VOID_PTR pEncSchemeMechanismParameter;  // NULL for zero

    /** length encryption scheme structure extension*/
    CK_ULONG    ulEncSchemeMechanismParameterLen;

    /** Flags for ECIES KDF/ENC/DEC additional shared data (sharedData1)        * LSB:
0x0000XXXX UInt16 KDF encoding
     * 0 = no addition to shared data
     * 1 = shared data | ephemeral public key
     * 2 = shared data | compressed ephemeral public key
     * 3 = ephemeral public key            | shared data
     * 4 = compressed ephemeral public key | shared data
     * MSB: 0xXXXX0000 32 bits Encoding
     * 0x0001XXXX NULL ICV for AES_CTR ENC/DEC
     *  0x0002XXXX Prepend ephemeral key for KDF     */
    CK_ULONG    ulKDFSharedDataFlags;   // Must be 0x00030000 for Google Pay

} CK_ECIES_PARAMS_EXT2;

typedef struct CK_ECIES_PARAMS_EXT3
{
   /** Legacy proprietary ECIES_EXT2 parameters*/
    CK_ECIES_PARAMS_EXT2 eciesParams; // as above

    /** reference derivation scheme structure extension = Salt params */
    CK_VOID_PTR pSalt; // NULL for zero

    /** length derivation scheme structure extension*/
    CK_ULONG    ulSaltLen;


} CK_ECIES_PARAMS_EXT3
```

RULES FOR CK_ECIES_PARAMS_EXT3 USE WITH HKDF

The `ulKDFSharedDataFlags` field is a proprietary extension, it controls the behavior of the KDF stage. For certain payment use cases it is possible to define flags to selectively enable prepending the ephemeral key to the shared secret to allow using an IV set to 0 for AES_CTR encryption and to control the encoding of the Salt value which could be zero for some use cases.

The `pEncSchemeMechanismParameter` and `ulEncSchemeMechanismParameterLen` parameters are used in the symmetric encrypt/Decrypt stage defined by encScheme. This not related to the KDF operation (it comes after). So, adding HKDF support does not affect how `pEncSchemeMechanismParameter`, `ulEncSchemeMechanismParameterLen` and `encScheme` interact.

In the case where `encScheme` is CKES_AES_CTR, the `pEncSchemeMechanismParameter` is a pointer to an optional IV value for the AES CTR mode.

The IV pointed at by `pEncSchemeMechanismParameter` is unrestricted. It should be assumed to mean all zero IV if it is NULL; otherwise, it is handled by AES CTR enc/dec operation as previously implemented.

CK_ECIES_PARAMS_EXT AND CK_ECIES_PARAMS

The CK_ECIES_PARAMS_EXT structure is an older version of the CK_ECIES_PARAMS_EXT2. It lacks the ulKDFSharedDataFlags field.

The CK_ECIES_PARAMS struct is the parameter from PKCS#11. It lacks the pEncSchemeMechanismParameter, ulEncSchemeMechanismParameterLen and ulKDFSharedDataFlags fields.

The latest host library (in client UC 10.7.1 onward) accepts CK_ECIES_PARAMS CK_ECIES_PARAMS_EXT and CK_ECIES_PARAMS_EXT2 convert the information into CK_ECIES_PARAMS_EXT3 by setting the undefined fields to zero/NULL.

Some payment scenarios might require the use of CK_ECIES_PARAMS_EXT2 extended fields.

So, the caller may use either CK_ECIES_PARAMS_EXT2 or CK_ECIES_PARAMS_EXT3 when doing some payment-related decrypts/encrypts.

However older Library implementations (preceding the HKDF feature; earlier than UC 10.7.1) will not work because the library would not recognize the HKDF kdf type (even if the firmware has that feature). Also, only firmware at version 7.8.7 onward can recognize the HKDF type either, so you will need both firmware 7.8.7 onward and client UC 10.7.1 onward, with the newer library, to perform some payment operations (depending on the requirements of your scheme).

DECRYPTION

C_DECRYPTINIT

```
C_DecryptInit function have the following parameters.
CK_SESSION_HANDLE hSession,     The session handle
CK_MECHANISM_PTR pMechanism,    A pointer to the ECIES_PARAM_EXT2 structure
CK_OBJECT_HANDLE hKey ,                 The ECC Private key handle
```

C_DECRYPT

```
C_Decrypt function have the following parameters.
CK_SESSION_HANDLE hSession,     The session handle
CK_BYTE_PTR pEncryptedData,     Ephemeral Public Key | Encrypted Data | Signature
CK_ULONG ulEncryptedDataLen,    The length of the encryptedData
CK_BYTE_PTR pData,              The decrypted message
CK_ULONG_PTR pulDataLen         The decrypted message length
```

ENCRYPTION

The encryption mechanism generates the ephemeral key pair based on the provided ECC public key and associated curve. At the end of the operation the ephemeral public key is returned along with the encrypted data and the ephemeral private key is deleted.

C_ENCRYPTINIT

```
C_EncryptInit function have the following parameters.
CK_SESSION_HANDLE hSession,     The session handle
CK_MECHANISM_PTR pMechanism,    A pointer to the ECIES_PARAM_EXT2 structure
CK_OBJECT_HANDLE hKey ,                 The ECC Public key handle
```

C_ENCRYPT

This process generates an ephemeral ECDSA key pair based on the given Curve OID. Then encryption and mac key are computed and used to encrypt and sign the message.

```
CK_SESSION_HANDLE HSESSION,     THE SESSION HANDLE
CK_BYTE_PTR pData,              Data
CK_ULONG uldDataLen,            The length of the data
CK_BYTE_PTR pEncData,           The ephemeral public key | the encrypted message | MAC
CK_ULONG_PTR pulEncDataLen      The encrypted message length
```

TOOLS

CKDEMO

CKD_HKDF_SHA256 is supported for key derivation of ECIES encryption/decryption mechanism.

Sample Payments Use case:

(98) Options: Turn on option 12- ECIES Parameters.

- Create an ECC key pair

(45) Sample generate key

1. [15] ECDSA
2. [19] X9_62_prime256v1(P-256)+
3. Set up template Sensitive Private Encrypt/Decrypt set to 1.

(40) Encrypt File

1. [51] ECIES and file to encrypt.
2. ECDH Primitive [] ECDH1 with COFACTOR (Standard)
3. KDF Type [35] CKD_HKDF_SHA256_KDF
4. Enter shared data file containing “Google” in ascii.
5. Add Shared data for HKDF [1] prepend ephemeral public key: 1
6. Enter salt data (enter “none” for no data): none
7. Encryption/decryption scheme: [6] = CKES_AES_CTR
   a. Enter Number of counter bits for AES CTR mode : 0
   b. AES_CTR_IV Derived?

[0] Not Derived

8. Select the AES key size in bits options [3] = 256-bits
9. Available signature/verification schema [5] CKMS_HMAC_SHA256
   a. Enter Mac key length in bits (0…1024) 256.
   b. Enter Mac length in bits (0…1024) 256
10. Enter shared data file for the HMAC (enter “none” for no data): none
11. Enter key to use. You should use your public key to send message encrypted: The generated ECDSA public key handle.

The encrypted data have been stored in file ENCRYPT.BIN.

The ENCRYPT.BIN file can be decrypted in the same way using the option (41) Decrypt file using your generated private key.

MULTITOKEN AND FMULTITOKEN

These ECIES Encryption Decryption mechanisms have been added to MultiToken and fmultiToken at client version UC 10.7.1 onward.

Command line – multitoken or fmultitoken

```
-s 1 -mode eciesaes256hkdfsha256hmacsha256 -curve 19 -esch = 1 -psw password -f -t 20
```

or/and

```
-s 1 -mode eciesaes256hkdfsha256hmacsha256 -curve 19 -esch = 0 -psw password -f -t 20
```

BIP32 Mechanism Support and Implementation

This section describes the BIP32 functions, key attributes, error codes, and mechanisms supported for BIP32 with the HSM.

> **NOTE** This feature requires minimum Luna HSM Firmware 7.3.0 and Luna HSM Client 7.3.0.

Curve Support

Only curve secp256k1 is supported. The BIP32 derivation mechanisms fail with CKR_TEMPLATE_INCONSISTENT if you attempt to specify a curve with CKA_ECDSA_PARAMS.

Key Type and Form

The key type CKK_BIP32 is used to distinguish keys that can be used for BIP32 from all the existing ECDSA keys. Existing ECDSA keys cannot be used with any of the BIP32 mechanisms because they lack a chain code. The serialization format when importing, exporting, wrapping and unwrapping keys is also different from ECDSA keys. All mechanisms supported by ECDSA keys are supported for BIP32 keys.

Extended Keys and Hardened Keys

BIP32 includes hardened and non-hardened (normal) child keys. Each has a 32-bit index. Child keys are considered hardened if the most significant bit of their index is set. This bit is defined as CKF_BIP32_HARDENED. This allows 2^31 hardened keys and 2^31 non-hardened keys per parent.

Hardened private keys create a firewall through which multi-level key derivation compromises cannot happen. For normal (non-hardened) keys one can derive child public keys of a given parent key without knowing any private key. So if an attacker gets a normal parent chain code and parent public key, he can brute-force all chain codes deriving from it. If the attacker also obtains a child, grandchild, or further-descended private key, he can use the chain code to generate all of the extended private keys descending from that private key. The formula for creating hardened keys makes it impossible to create child public keys without knowing the parent private key.

Key Derivation

Two new mechanisms are added to support all the key derivations in BIP32.

CKM_BIP32_MASTER_DERIVE

This mechanism derives the master key pair from a seed. The input key must have the type `CKK_GENERIC_SECRET` (size between 128 and 512 bits). This mechanism is unique in that it derives two keys from one. This requires us to accept two templates as input, and to output the two derived key handles. In order to avoid confusion, the three last arguments of `C_DeriveKey()(pTemplate`, `ulAttributeCount` and `phKey`) must be null or zero. `CKR_ARGUMENTS_BAD` is returned if any of those parameters is non-NULL. The templates and returned handles are instead passed in through the mechanism parameters, which are clearly labeled public and private. Choose to not generate the public or private key by leaving those parameters as zero or null.

```
typedef struct CK_BIP32_MASTER_DERIVE_PARAMS {
      CK_ATTRIBUTE_PTR pPublicKeyTemplate;
      CK_ULONG ulPublicKeyAttributeCount;
      CK_ATTRIBUTE_PTR pPrivateKeyTemplate;
      CK_ULONG ulPrivateKeyAttributeCount;
      CK_OBJECT_HANDLE hPublicKey; // output parameter
      CK_OBJECT_HANDLE hPrivateKey; // output parameter
} CK_BIP32_MASTER_DERIVE_PARAMS;
```

See "Code Samples" on page 96 for a code example.

CKM_BIP32_CHILD_DERIVE

This mechanism derives child keys from a parent key. The mechanism can generate both the private and public part of the key pair, and can accept a BIP32 public or private key as input. An error is returned if a public to private derivation is attempted. Like the master key derivation, the templates and key handle outputs are passed through the mechanism parameters. Choose to not generate the public or private key by leaving those parameters as zero or null.

The BIP32 and BIP44 specifications recommend wallet structures and use cases. The specifications provide a good reference for deciding how a key tree should be organized and if a particular key should be hardened or not. Follow the specifications to avoid potential security holes.

This mechanism can be used to generate keys that are several levels deep in the key hierarchy. The path of the key is specified with `pulPath` and `ulPathLen`. The path is an array of 32-bit unsigned integers (key indices).

The highest bit (0x80000000) is used to indicate a hardened key. So a path value for a normal key must be <= 0x7FFFFFFF (decimal 2147483647). For a hardened key, 0x80000000 <= path value <= 0xFFFFFFFF. The path is relative to the input key. For example, if the path is [5, 1, 4] and the path of the input key is m/0 then the resulting path is m/0/5/1/4.

The max number of levels (path length) is 255 for BIP32 in the HSM firmware. This is expected to be generally adequate.

```
typedef struct CK_BIP32_CHILD_DERIVE_PARAMS {
      CK_ATTRIBUTE_PTR pPublicKeyTemplate;
      CK_ULONG ulPublicKeyAttributeCount;
      CK_ATTRIBUTE_PTR pPrivateKeyTemplate;
      CK_ULONG ulPrivateKeyAttributeCount;
      CK_ULONG_PTR pulPath;
      CK_ULONG ulPathLen;
      CK_OBJECT_HANDLE hPublicKey; // output parameter
      CK_OBJECT_HANDLE hPrivateKey; // output parameter
      CK_ULONG ulPathErrorIndex; // output parameter
} CK_BIP32_CHILD_DERIVE_PARAMS;
```

See "Code Samples" on page 96 for a code example.

Error Codes

These mechanisms can fail in ways not applicable to other mechanisms.

CKR_BIP32_CHILD_INDEX_INVALID: This error is returned on the rare occurrence (1 / 2^127) that a child derivation returns an all-zero private key, a private key bigger than or equal to the curve order parameter `n`, or a point at infinity. This error signifies that the child key index cannot be used to derive keys. Choose a different index and try the derivation again. The problematic child index is indicated by `ulPathErrorIndex`.

PCKS#11 does not have fixed width integers. This error can also be returned on platforms where `CK_ULONG` is bigger than 32 bits and a child index is bigger than 2^32 – 1.

CKR_BIP32_INVALID_HARDENED_DERIVATION: This error is returned from an attempt to derive a hardened key from a public key. The BIP32 specification does not support such a derivation.

CKR_BIP32_MASTER_SEED_LEN_INVALID: The BIP32 specification recommends deriving the master key from a seed that is between 128 and 512 bits long. This error is returned if the seed length is outside of that range.

CKR_BIP32_MASTER_SEED_INVALID: This error is returned on the rare occurrence (1 / 2^127) that the master derivation returns an all zero private key, a private key bigger than or equal to the curve order parameter `n`, or a point at infinity. This error signifies that the master seed cannot be used for BIP32. Generate a new master seed and retry the derivation.

CKR_BIP32_INVALID_KEY_PATH_LEN: This error is returned when `ulPathLen` is 0 or greater than 255. The BIP44 standard only requires paths of length 5 so this limit should be acceptable for all customers.

Key Attributes

The following attributes will exist on all keys created with one of the above derivation mechanisms.

CKA_BIP32_CHAIN_CODE: The chain code is essential for BIP32 keys and is used to derive future keys. The public and private key share this value. Read only.

CKA_BIP32_VERSION_BYTES: Version bytes are used to further identify BIP32 keys. The version bytes help determine if a key is used on the main bitcoin network or the test network. This attribute defaults to `CKG_BIP32_VERSION_MAINNET_PUB/PRIV` if it was not specified at key creation time. You can set this value to `CKG_BIP32_VERSION_TESTNET_PUB/PRIV` if applicable.

CKA_BIP32_CHILD_INDEX: The child index stores which index was used to derive this key. An index with the `CKF_BIP32_HARDENED` bit set is considered a hardened child. The child index is 0 for the master key. The public and private key share this value. Read only.

CKA_BIP32_CHILD_DEPTH: The depth of the child key in the key tree. The master key has a depth of 0. The public and private key share this value. Read only.

CKA_BIP32_ID: The unique identifier for the key. This value is derived from the HASH160 of the compressed public key. The first 32 bits of this value is known as the fingerprint. (`CKA_ID` is not used for this purpose because it is writable by the user.) The public and private key share this value. Read only.

> **NOTE** No attribute is included for the parent ID because it should not be required. The anticipated use-case is to derive a key, use it and then delete it. In general, there should not be a need to discover how keys are organized based on the fingerprints or IDs. The parent fingerprint is available in case there is need to rediscover a key tree, but the wallet software must deal with any collisions. The BIP32 designers considered the parent ID not sufficiently important to include in serialized keys; therefore we exclude it as well.

CKA_BIP32_FINGERPRINT and **CKA_BIP32_PARENT_FINGERPRINT**:

The fingerprints for the key and parent key are the first 32 bits of the BIP32 key identifier. These can be used to identify keys but the wallet software must handle any collisions. For identifying keys, it is better to use **CKA_BIP32_ID** because it is long enough that collisions should not be an issue. The public and private key share this value. The master key has a parent fingerprint of 0. Read only.

Public Key Import/Export

To support importing existing BIP32 keys, we support their serialization format. For public keys, we will have functions in our library to facilitate importing and exporting.

```
CK_RV CA_Bip32ImportPubKey(
      CK_SESSION_HANDLE hSession,
      CK_ATTRIBUTE_PTR pTemplate,
      CK_ULONG ulCount,
      const CK_CHAR_PTR pKey,          //in BIP32 serialization format
      CK_OBJECT_HANDLE_PTR phObject
);
CK_RV CA_Bip32ExportPubKey(
      CK_SESSION_HANDLE hSession,
      CK_OBJECT_HANDLE hObject,
      CK_CHAR_PTR pKey,                   //in BIP32 serialization format
      CK_ULONG_PTR pulKeyLen       //on input contains max. buffer size, returns
                                   // actual size
);
```

Importing is done with `CA_Bip32ImportPubKey()`. The function is similar to `C_CreateObject()` but it takes an additional parameter for the serialized public key. The template passed in should contain all the desired non-BIP32 attributes like `CKA_TOKEN`, `CKA_PRIVATE`, `CKA_DERIVE`, etc. The function decodes the public key to get all the BIP32 attributes. Both sets of attributes are then used to create the public key on the HSM.

> **NOTE** When importing a serialized extended public key, implementations must verify whether the X coordinate in the public key data corresponds to a point on the curve. If not, the extended public key is invalid.

Exporting is done with `CA_Bip32ExportPubKey()`. The specified object is extracted from the HSM and encoded in the BIP32 format. The result is a NULL-terminated string and is placed in the `pKey` parameter. The length of `pKey` has a maximum of 112 characters. This constant is defined as `CKG_BIP32_MAX_SERIALIZED_LEN`. It's possible that not all characters are needed to serialize the key. Any unused characters are set to 0.

See "Code Samples" on the next page for code examples.

BIP32 Serialization Format

Extended public and private keys are serialized as follows:

- 4 byte: version bytes (mainnet: 0x0488B21E public, 0x0488ADE4 private; testnet: 0x043587CF public, 0x04358394 private)
- 1 byte: depth: 0x00 for master nodes, 0x01 for level-1 derived keys, ...
- 4 bytes: the fingerprint of the parent's key (0x00000000 if master key)
- 4 bytes: child number (index) – 32-bit unsigned integer with most significant byte first (0x00000000 if master key)
- 32 bytes: the chain code
- 33 bytes: the public key or private key data

This 78 byte structure is encoded like other Bitcoin data in Base58, by first adding 32 checksum bits (derived from the double SHA-256 checksum), and then converting to the Base58 representation. This results in a Base58-encoded string of up to `CKG_BIP32_MAX_SERIALIZED_LEN` characters. Because of the choice of the version bytes, the Base58 representation will start with "xprv" or "xpub" on mainnet, "tprv" or "tpub" on testnet.

Private Key Import/Export

Private keys can be imported and exported with existing PKCS#11 functions. They can be imported and exported only if the HSM uses the key wrap model instead of cloning. Import a key by calling `C_Encrypt*()` on the serialized key followed by `C_UnwrapKey()`. Exporting keys by calling `C_WrapKey()` followed by `C_Decrypt*()`. Use `C_WrapKey()` and `C_UnwrapKey()` to store keys off the HSM, or to move them between HSMs.

See "Code Samples" on the next page for code examples.

Key Backup and Cloning

Backups and cloning of BIP32 keys are supported only between version 7.x Luna HSMs. Further, cloning of BIP32 keys is supported only in firmware versions that have BIP32 support. BIP32 keys cannot be cloned to older firmware versions made before BIP32 support was added.

Non-FIPS Algorithm

The BIP32 mechanisms are available only if non-FIPS algorithms are allowed.

Host Tools

Multitoken and Ckdemo support BIP32.

Code Samples

Deriving the master key pair

We highly recommend setting `CKA_PRIVATE` on the master public and private keys to TRUE to prevent the chain code from being seen by unauthorized users. The master key should be used only for derivations so it is the only operation allowed. The version bytes default to 0x0488B21E/0x0488ADE4 for the public/private keys if the attribute is missing in the template. Those are the values specified in BIP32 for keys on the main bitcoin network.

```
CK_ATTRIBUTE pubTemplate[] =
{
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
   {CKA_DERIVE,            &bTrue,       sizeof(bTrue)},
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
};
CK_ATTRIBUTE privTemplate[] =
{
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
   {CKA_SENSITIVE,         &bTrue,       sizeof(bTrue)},
   {CKA_DERIVE,            &bTrue,       sizeof(bTrue)},
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
};

CK_BIP32_MASTER_DERIVE_PARAMS mechParams;
mechParams.pPublicKeyTemplate = pubTemplate;
mechParams.ulPublicKeyAttributeCount = ARRAY_SIZE(pubTemplate);
mechParams.pPrivateKeyTemplate = privTemplate;
mechParams.ulPrivateKeyAttributeCount = ARRAY_SIZE(privTemplate);
CK_MECHANISM mechanism = {CKM_BIP32_MASTER_DERIVE, &mechParams, sizeof(mechParams)};

CK_RV rv = C_DeriveKey(hSession, &mechanism, hSeedKey, NULL, 0, NULL);
// fail if rv != CKR_OK

CK_OBJECT_HANDLE pubKey = mechanism.mechParams->hPublicKey;
CK_OBJECT_HANDLE privKey = mechanism.mechParams->hPrivateKey;
```

The new key handles will be stored in `pubKey` and `privKey` if the derivation was successful.

Deriving a child leaf key

We highly recommend setting `CKA_PRIVATE` on the child public and private keys to TRUE to prevent the chain code from being seen by unauthorized users. A child leaf key (the bottom key in the tree) should not be used for derivation, and is meant for signing, verifying, encrypting and decrypting. Parent child keys need the derive attribute turned on. The version bytes default to 0x0488B21E/0x0488ADE4 for the public/private keys if the attribute is missing. Those are the values specified in BIP32 for keys on the main bitcoin network.

```
CK_ATTRIBUTE pubTemplate[] =
{
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
```

```
   {CKA_ENCRYPT,           &bTrue,       sizeof(bTrue)},
   {CKA_VERIFY,            &bTrue,       sizeof(bTrue)},
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
};
CK_ATTRIBUTE privTemplate[] =
{
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
   {CKA_SENSITIVE,         &bTrue,       sizeof(bTrue)},
   {CKA_SIGN,              &bTrue,       sizeof(bTrue)},
   {CKA_DECRYPT,           &bTrue,       sizeof(bTrue)},
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
};

CK_ULONG path[] = {
       CKF_BIP32_HARDENED | CKG_BIP44_PURPOSE,
       CKF_BIP32_HARDENED | CKG_BIP44_COIN_TYPE_BTC,
       CKF_BIP32_HARDENED | 1,
       CKG_BIP32_EXTERNAL_CHAIN,
       0
};

CK_BIP32_MASTER_DERIVE_PARAMS mechParams;
mechParams.pPublicKeyTemplate = pubTemplate;
mechParams.ulPublicKeyAttributeCount = ARRAY_SIZE(pubTemplate);
mechParams.pPrivateKeyTemplate = privTemplate;
mechParams.ulPrivateKeyAttributeCount = ARRAY_SIZE(privTemplate);
mechParams.pulPath = path;
mechParams.ulPathLen = ARRAY_SIZE(path);
CK_MECHANISM mechanism = {CKM_BIP32_CHILD_DERIVE, &mechParams, sizeof(mechParams)};

CK_RV rv = C_DeriveKey(hSession, &mechanism, hMasterPrivKey, NULL, 0, NULL);
// fail if rv != CKR_OK

CK_OBJECT_HANDLE pubKey = mechanism.mechParams->hPublicKey;
CK_OBJECT_HANDLE privKey = mechanism.mechParams->hPrivateKey;
```

The new key handles are stored in `pubKey` and `privKey` if the derivation was successful. The path generates a key pair that follows the BIP44 convention and can be used to receive BTC.

Importing a public extended key

```
CK_ATTRIBUTE template[] =
{
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
   {CKA_DERIVE,            &bTrue,       sizeof(bTrue)},
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
};

CK_CHAR_PTR encodedKey = “xpub661MyMwAqRbcFtXgS5…”; //BIP32 serialization format
CK_OBJECT_HANDLE pubKey;

CK_RV rv = CA_Bip32ImportKey(hSession, template, ARRAY_SIZE(template), encodedKey, &pubKey);
```

The handle for the newly create key is stored in `pubKey` if the import was successful.

Exporting a public extended key

```
CK_CHAR encodedKey[CKG_BIP32_MAX_SERIALIZED_LEN+1];
CK_ULONG ulEncodedKeySize = sizeof(encodedKey);

CK_RV rv = CA_Bip32ExportPubKey(hSession, hObject, encodedKey, &ulEncodedKeySize );
```

The encoded key is stored in `encodedKey` (BIP32 serialization format) if there were no errors.

Importing a private extended key

```
CK_ATTRIBUTE template[] =
{
   {CKA_CLASS              &keyClass,    sizeof(keyClass)},
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_KEY_TYPE           &keyType,     sizeof(keyType)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
   {CKA_DERIVE,            &bTrue,       sizeof(bTrue)},
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
   {CKA_SENSITIVE          &bTrue,       sizeof(bTrue)},
};

CK_CHAR_PTR encodedKey = "xprv9s21ZrQH143K3QTDL4LXw2F…";
CK_MECHANISM mechanism = {CKM_AES_KWP, NULL, 0};
CK_BYTE wrappedKey[256];
CK_ULONG wrappedKeyLen = sizeof(wrappedKey);
CK_OBJECT_HANDLE hUnwrappedKey;

CK_RV rv = C_EncryptInit(hSession, &mechanism, hWrappingKey);
// fail if rv != CKR_OK

rv = C_Encrypt(hSession, encodedKey, sizeof(encodedKey), wrappedKey, &wrappedKeyLen);
// fail if rv != CKR_OK

rv = C_UnwrapKey(hSession, &mechanism, hWrappingKey, wrappedKey, wrappedKeyLen, template,
ARRAY_SIZE(template), &hUnwrappedKey);
```

After unwrapping, the encoded key's BIP32 serialization format is decoded (the template key type is checked for BIP32). The handle of the unwrapped key is stored in `hUnwrappedKey` if there were no errors.

Exporting a private extended key

```
CK_MECHANISM mechanism = {CKM_AES_KWP, NULL, 0};
CK_BYTE key[256];
CK_ULONG keyLen = sizeof(key);

CK_RV rv = C_WrapKey(hSession, &mechanism, hWrappingKey, hKeyToWrap, key, &keyLen);
// fail if rv != CKR_OK

rv = C_DecryptInit(hSession, &mechanism, hWrappingKey);
// fail if rv != CKR_OK

rv = C_Decrypt(hSession, key, keyLen, key, &keyLen);
// fail if rv != CKR_OK

key[keyLen] = 0 // The key isn't NULL terminated after C_Decrypt().
```

C_WrapKey() must convert the BIP32 key to the BIP32 serialization format before wrapping.

The serialized key is stored in `key` if there were no errors.

PKCS#11 Definitions

```
#define CKK_BIP32 (CKK_VENDOR_DEFINED  | 0x14)
#define CKM_BIP32_MASTER_DERIVE (CKM_VENDOR_DEFINED | 0xE00)
#define CKM_BIP32_CHILD_DERIVE (CKM_VENDOR_DEFINED |  0xE01)
#define CKR_BIP32_CHILD_INDEX_INVALID (CKR_VENDOR_DEFINED |         0x83)
#define CKR_BIP32_INVALID_HARDENED_DERIVATION (CKR_VENDOR_DEFINED | 0x84)
#define CKR_BIP32_MASTER_SEED_LEN_INVALID (CKR_VENDOR_DEFINED |     0x85)
#define CKR_BIP32_MASTER_SEED_INVALID (CKR_VENDOR_DEFINED |         0x86)
#define CKR_BIP32_INVALID_KEY_PATH_LEN (CKR_VENDOR_DEFINED |        0x87)
#define CKA_BIP32_CHAIN_CODE (CKA_VENDOR_DEFINED |         0x1100)
#define CKA_BIP32_VERSION_BYTES (CKA_VENDOR_DEFINED |      0x1101)
#define CKA_BIP32_CHILD_INDEX (CKA_VENDOR_DEFINED |        0x1102)
#define CKA_BIP32_CHILD_DEPTH (CKA_VENDOR_DEFINED |        0x1103)
#define CKA_BIP32_ID (CKA_VENDOR_DEFINED |                 0x1104)
#define CKA_BIP32_FINGERPRINT (CKA_VENDOR_DEFINED |        0x1105)
#define CKA_BIP32_PARENT_FINGERPRINT (CKA_VENDOR_DEFINED | 0x1106)
#define CKG_BIP32_VERSION_MAINNET_PUB   (0x0488B21E)
#define CKG_BIP32_VERSION_MAINNET_PRIV  (0x0488ADE4)
#define CKG_BIP32_VERSION_TESTNET_PUB   (0x043587CF)
#define CKG_BIP32_VERSION_TESTNET_PRIV  (0x04358394)
#define CKG_BIP44_PURPOSE               (0x0000002C)
#define CKG_BIP44_COIN_TYPE_BTC         (0x00000000)
#define CKG_BIP44_COIN_TYPE_BTC_TESTNET (0x00000001)
#define CKG_BIP32_EXTERNAL_CHAIN        (0x00000000)
#define CKG_BIP32_INTERNAL_CHAIN        (0x00000001)
#define CKG_BIP32_MAX_SERIALIZED_LEN    (112)
#define CKF_BIP32_HARDENED              (0x80000000)
#define CKF_BIP32_MAX_PATH_LEN          (255)
```

SLIP 10

Of the SatoshiLabs Improvement Proposals (SLIPs), SLIP10 extends the Bitcoin Improvement Proposal 32 (BIP32). SLIP10 support is introduced with Luna HSM firmware version 7.8.7, and also requires Luna HSM Client version 10.7.1 for the matching library updates.

SLIP10 supports curves Ed25519 and secp256k1, as well as NIST P-256 curve. All operations appear as BIP32 operations and one of the 3 SLIP-10 curves must be explicitly specified.

SLIP10 BIP32 Master Key derivation and Child Key derivation

CKM_BIP32_MASTER_DERIVE and CKM_BIP32_CHILD_DERIVE – need to explicitly specify the curve to be used, if curve not specified then BIP32 derivation is performed.

- key type CKK_BIP32
- attributes CKA_BIP32_…
- error codes CKR_BIP32_…
- defines CKG_BIP32_... and CKF_BIP32_...
- public key import/export via CA_Bip32ImportPubKey() and CA_Bip32ExportPubKey()
- when attempted against an HA slot, requires that group members be running BIP32 firmware (7.3.0 onward).

Caveats

The following circumstances are worth noting.

> Cloning SLIP10 ED25519 keys to older firmware (versions before 7.8.7) will not work. That includes Backup.

> The BIP32 and SLIP10 mechanisms are available only if non-FIPS algorithms are allowed (see Enable non-FIPS algorithms)

> A SLIP10 CKK_BIP32 key for curve secp256k1 stored on the HSM cannot be differentiated from a BIP32 key. For SLIP10 curves NIST P-256 and Ed25519, the curve parameters attribute can be used to indicate it is a SLIP10 key. Therefore, a user could pick a SLIP10 secp256k1 key for BIP32 operations.
The concern is that BIP32 key derivations are sometimes invalid causing the tree to stop at that point, while *SLIP10 will **retry** with a new value* and keep going. Thus, some levels on a SLIP10 tree cannot be reached if using BIP32 derivations. But the chance of an invalid BIP32 key is lower than 1 in 2^{127}.

> When importing/exporting a SLIP10 key (uses serialization format to store extra BIP32 attributes) the serialization data does not indicate if SLIP10 or BIP32, nor which curve was used. The user must specify the curve parameters in the key template when importing.

Some of the above issues are reduced/eliminated if you keep track of how the keys were derived (add label?) and only use appropriate keys for BIP32 or SLIP10 operations.

Curve Support

Curve secp256k1 is supported for BIP32. If you attempt to specify a curve with CKA_ECDSA_PARAMS in the derive key templates, the BIP32 derivation mechanisms fail with CKR_TEMPLATE_INCONSISTENT.

For SLIP-10 the ability to specify the curve parameters with the BIP32 mechanisms is allowed. Specify the attribute CKA_ECDSA_PARAMS when using the BIP32 derive mechanisms. If the attribute is not specified, then a BIP32 secp256k1 key is derived (same as pre-7.8.7 LUNA firmware). If this attribute is present in the incoming derive template, then the firmware checks if it specifies one of the three supported curves and if so, it performs SLIP-10 operations (if not then an error is thrown).

Hardened Keys

The specification allows both extended child keys and hardened child keys. For SLIP10 Ed25519 only hardened key generation from private parent key to private child key is supported. For secp256k1 and NIST P-256 curves both hardened and non-hardened keys are allowed.

Key Derivation

This section shows the differences between sample code for BIP32 and SLIP-10. Basically, for SLIP-10 the attribute CKA_ECDSA_PARAMS must be specified in the key templates.

Deriving the master key pair

We strongly recommended to set CKA_PRIVATE on the master public and private keys to TRUE to prevent the chain code from being seen by unauthorized users. The master key should be used only for derivations so it is the only operation allowed. The version bytes default to 0x0488B21E/0x0488ADE4 for the public/private keys if the attribute is missing in the template. Those are the values specified in BIP32 for keys on the main bitcoin network. The desired SLIP-10 curve must be provided in the key templates.

```
unsigned char ecParams[] =
{0x06,0x05,0x2B,0x81,0x04,0x00,0x0A};                          /* secp256k1 */
```

```
//OR    {0x06,0x08,0x2A,0x86,0x48,0xCE,0x3D,0x03,0x01,0x07};    /* X9_62_prime256v1 */
//OR    {0x06,0x09,0x2B,0x06,0x01,0x04,0x01,0xDA,0x47,0x0F,0x01};       /* Ed25519 */
CK_ATTRIBUTE pubTemplate[] =
{
{CKA_TOKEN,             &bToken,      sizeof(bToken)},
{CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
{CKA_DERIVE,            &bTrue,       sizeof(bTrue)},
{CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
{CKA_LABEL,             pbLabel,      strlen(pbLabel)},
{CKA_ECDSA_PARAMS,      &ecParams,    sizeof(ecParams)},
};
CK_ATTRIBUTE privTemplate[] =
{
{CKA_TOKEN,             &bToken,      sizeof(bToken)},
{CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
{CKA_SENSITIVE,         &bTrue,       sizeof(bTrue)},
{CKA_DERIVE,            &bTrue,       sizeof(bTrue)},
{CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
{CKA_LABEL,             pbLabel,      strlen(pbLabel)},
{CKA_ECDSA_PARAMS,      &ecParams,    sizeof(ecParams)},
};
CK_BIP32_MASTER_DERIVE_PARAMS mechParams;
mechParams.pPublicKeyTemplate = pubTemplate;
mechParams.ulPublicKeyAttributeCount = ARRAY_SIZE(pubTemplate);
mechParams.pPrivateKeyTemplate = privTemplate;
mechParams.ulPrivateKeyAttributeCount = ARRAY_SIZE(privTemplate);
CK_MECHANISM mechanism = {CKM_BIP32_MASTER_DERIVE, &mechParams, sizeof(mechParams)};
CK_RV rv = C_DeriveKey(hSession, &mechanism, hSeedKey, NULL, 0, NULL);
// fail if rv != CKR_OK
CK_OBJECT_HANDLE pubKey = mechanism.mechParams->hPublicKey;
CK_OBJECT_HANDLE privKey = mechanism.mechParams->hPrivateKey;
```

The new key handles will be stored in `pubKey` and `privKey` if the derivation was successful.

Deriving a child leaf key

It's highly recommended to set CKA_PRIVATE on the child public and private keys to TRUE to prevent the chain code from being seen by unauthorized users. A child leaf key (the bottom key in the tree) should not be used for derivation and is meant for signing, verifying, encrypting and decrypting. Parent child keys need the derive attribute turned on. The version bytes default to 0x0488B21E/0x0488ADE4 for the public/private keys if the attribute is missing. Those are the values specified in BIP32 for keys on the main bitcoin network. The desired SLIP-10 curve must be given in the key templates.

See previous section for "ecParams[]" definition.

```
CK_ATTRIBUTE pubTemplate[] =
{
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
   {CKA_ENCRYPT,           &bTrue,       sizeof(bTrue)},
   {CKA_VERIFY,            &bTrue,       sizeof(bTrue)},
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
   {CKA_ECDSA_PARAMS,      &ecParams,    sizeof(ecParams)},
};
CK_ATTRIBUTE privTemplate[] =
{
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
   {CKA_SENSITIVE,         &bTrue,       sizeof(bTrue)},
   {CKA_SIGN,              &bTrue,       sizeof(bTrue)},
   {CKA_DECRYPT,           &bTrue,       sizeof(bTrue)},
```

```
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
   {CKA_ECDSA_PARAMS,      &ecParams,    sizeof(ecParams)},
};
CK_ULONG path[] = {
      CKF_BIP32_HARDENED | CKG_BIP44_PURPOSE,
      CKF_BIP32_HARDENED | CKG_BIP44_COIN_TYPE_BTC,
      CKF_BIP32_HARDENED | 1,
      CKG_BIP32_EXTERNAL_CHAIN,
      0
};
CK_BIP32_MASTER_DERIVE_PARAMS mechParams;
mechParams.pPublicKeyTemplate = pubTemplate;
mechParams.ulPublicKeyAttributeCount = ARRAY_SIZE(pubTemplate);
mechParams.pPrivateKeyTemplate = privTemplate;
mechParams.ulPrivateKeyAttributeCount = ARRAY_SIZE(privTemplate);
mechParams.pulPath = path;
mechParams.ulPathLen = ARRAY_SIZE(path);
CK_MECHANISM mechanism = {CKM_BIP32_CHILD_DERIVE, &mechParams, sizeof(mechParams)};
CK_RV rv = C_DeriveKey(hSession, &mechanism, hMasterPrivKey, NULL, 0, NULL);
// fail if rv != CKR_OK
CK_OBJECT_HANDLE pubKey = mechanism.mechParams->hPublicKey;
CK_OBJECT_HANDLE privKey = mechanism.mechParams->hPrivateKey;
```

The new key handles is stored in `pubKey` and `privKey` if the derivation was successful. The path generates a key pair that follows the BIP44 convention and can be used to receive BTC.

Importing a public extended key (for UC 10.7.1 onwards)

```
CK_ATTRIBUTE template[] =
{
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
   {CKA_DERIVE,            &bTrue,       sizeof(bTrue)},
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
   {CKA_ECDSA_PARAMS,      &ecParams,    sizeof(ecParams)},
};
CK_CHAR_PTR encodedKey = “xpub661MyMwAqRbcFtXgS5…”; //BIP32 serialization format
CK_OBJECT_HANDLE pubKey;
CK_RV rv = CA_Bip32ImportKey(hSession, template, ARRAY_SIZE(template), encodedKey, &pubKey);
```

The handle for the newly create key is stored in `pubKey` if the import was successful.

Exporting a public extended key

(Same as BIP32)

```
CK_MECHANISM mechanism = {CKM_AES_KWP, NULL, 0};
CK_BYTE key[256];
CK_ULONG keyLen = sizeof(key);
CK_RV rv = C_WrapKey(hSession, &mechanism, hWrappingKey, hKeyToWrap, key, &keyLen);
// fail if rv != CKR_OK
rv = C_DecryptInit(hSession, &mechanism, hWrappingKey);
// fail if rv != CKR_OK
rv = C_Decrypt(hSession, key, keyLen, key, &keyLen);
// fail if rv != CKR_OK
key[keyLen] = 0 // The key isn’t NULL terminated after C_Decrypt().
```

C_WrapKey() must convert the BIP32 key to the BIP32 serialization format before wrapping.

The serialized key is stored in `key` if there were no errors.

Importing a private extended key

```
CK_ATTRIBUTE template[] =
{
   {CKA_CLASS              &keyClass,    sizeof(keyClass)},
   {CKA_TOKEN,             &bToken,      sizeof(bToken)},
   {CKA_KEY_TYPE           &keyType,     sizeof(keyType)},
   {CKA_PRIVATE,           &bTrue,       sizeof(bTrue)},
   {CKA_DERIVE,            &bTrue,       sizeof(bTrue)},
   {CKA_MODIFIABLE,        &bTrue,       sizeof(bTrue)},
   {CKA_LABEL,             pbLabel,      strlen(pbLabel)},
   {CKA_SENSITIVE          &bTrue,       sizeof(bTrue)},
   {CKA_ECDSA_PARAMS,      &ecParams,    sizeof(ecParams)},
};
CK_CHAR_PTR encodedKey = "xprv9s21ZrQH143K3QTDL4LXw2F…";
CK_MECHANISM mechanism = {CKM_AES_KWP, NULL, 0};
CK_BYTE wrappedKey[256];
CK_ULONG wrappedKeyLen = sizeof(wrappedKey);
CK_OBJECT_HANDLE hUnwrappedKey;
CK_RV rv = C_EncryptInit(hSession, &mechanism, hWrappingKey);
// fail if rv != CKR_OK
rv = C_Encrypt(hSession, encodedKey, sizeof(encodedKey), wrappedKey, &wrappedKeyLen);
// fail if rv != CKR_OK
rv = C_UnwrapKey(hSession, &mechanism, hWrappingKey, wrappedKey, wrappedKeyLen, template,
ARRAY_SIZE(template), &hUnwrappedKey);
```

After unwrapping the encoded key its BIP32 serialization format is decoded (the template key type is checked for BIP32). The handle of the unwrapped key is stored in `hUnwrappedKey` if there were no errors.

Exporting a private extended key

(Same as BIP32)

```
CK_MECHANISM mechanism = {CKM_AES_KWP, NULL, 0};
CK_BYTE key[256];
CK_ULONG keyLen = sizeof(key);
CK_RV rv = C_WrapKey(hSession, &mechanism, hWrappingKey, hKeyToWrap, key, &keyLen);
// fail if rv != CKR_OK
rv = C_DecryptInit(hSession, &mechanism, hWrappingKey);
// fail if rv != CKR_OK
rv = C_Decrypt(hSession, key, keyLen, key, &keyLen);
// fail if rv != CKR_OK
key[keyLen] = 0 // The key isn't NULL terminated after C_Decrypt().
```

C_WrapKey() must convert the BIP32 key to the BIP32 serialization format before wrapping.

The serialized key is stored in `key` if there were no errors.

3GPP Mechanisms for 5G Mobile Networks

This section describes the C-based PKCS#11 interface to the 3GPP functions in the HSM firmware. The mechanisms described below are also represented in JCPROV.

> **NOTE** This feature requires minimum Luna HSM Firmware 7.4.2 (Luna HSM Firmware 7.7.0 for Luna PCIe HSM 7) and Luna HSM Client 10.2.0 (or a patched Luna HSM Client 7.4.0). You require a backup HSM with minimum Luna Backup HSM 7 Firmware 7.7.1 to back up these objects.

> "MILENAGE" on the next page

> "TUAK" on page 106
> "Comp128" on page 107
> "Storage Key (SK)" on page 107
> See also "Luna Key Translation" on page 107.

MILENAGE

Authentication

As with all the 3GPP crypto operations, the C_SignInit/C_Sign function calls are made.

C_SignInit

```
(
    CK_SESSION_HANDLE hSession,                 // the session's handle
    CK_MECHANISM_PTR  pMechanism,       // the signature mechanism
                      mechanism                 // mechanism type   CKM_MILENAGE
                      pParameter        // pointer to the milenage mechanism CK_MILENAGE_SIGN_PARAMS
                      ulParameterLen    // length (sizeof) in bytes of the parameter

    CK_OBJECT_HANDLE  hKey              // AES storage key used to encrypt/decrypt Ki and OP (optional)
);
```

C_Sign

```
(
    CK_SESSION_HANDLE hSession,         // the session's handle
    CK_BYTE_PTR       pData,            // should be NULL for CKM_MILENAGE (see CKM_MILENAGE_RESYNC)
    CK_ULONG          ulDataLen,        // should be set to zero for CKM_MILENAGE
    CK_BYTE_PTR       pSignature,       // gets the signature - see OUTPUT response string below
    CK_ULONG_PTR      pulSignatureLen   // gets signature length
);
```

Mechanism: CKM_MILENAGE

Parameter Structure:

```
typedef struct CK_MILENAGE_SIGN_PARAMS {
    CK_ULONG          ulMilenageFlags;
    CK_ULONG          ulEncKiLen;
    CK_BYTE_PTR       pEncKi;
    CK_ULONG          ulEncOPcLen;
    CK_BYTE_PTR       pEncOPc;          // Encrypted or plain - see flags
    CK_OBJECT_HANDLE  hSecondaryKey;    // optional OP object handle - see flags
    CK_OBJECT_HANDLE  hRCKey;           // optional R and C params - see flags
    CK_BYTE           sqn[6];
    CK_BYTE           amf[2];
} CK_MILENAGE_SIGN_PARAMS;
typedef CK_MILENAGE_SIGN_PARAMS CK_PTR CK_MILENAGE_SIGN_PARAMS_PTR;
```

The ulMilenageFlags can consist of one or more of the following:

```
#define LUNA_5G_OPC                 0x00000001      // OPC is provided rather than OP
#define LUNA_5G_ENCRYPTED_OP        0x00000002      // OP or OPC is encrypted by Storage Key
#define LUNA_5G_OP_OBJECT           0x00000004      // OP or OPC is an object in HSM partition
#define LUNA_5G_USE_TLV             0x00000008      // Use the Tag/Len/Val format in response
```

```
#define LUNA_5G_USER_DEFINED_RC     0x00000010      // User defined R and C constants
```

> **NOTE** If the **LUNA_5G_OP_OBJECT** flag is specified, then the hSecondaryKey parameter should contain the handle of the generic secret object which holds the OP string. As well, if the **LUNA_5G_USER_DEFINED_RC** flag is set, then set the hRCKey to point to the RC generic secret object.

Although the R and C values are hard-coded (per the 3GPP specification) the Luna implementation allows the user to define his own constants. It is first necessary to create a generic secret object on the HSM where the CKA_VALUE attribute contains a concatenation of the R and C fields as follows:

C1[16 bytes], C2[16 bytes], C3[16 bytes], C4[16 bytes], C5[16 bytes], R1[1 byte], R2[1 byte], R3[1 byte], R4[1 byte], R5[1 byte]

> **TIP** Use a hex editor to create the binary RC file. Use the API or the **ckdemo** utility to encrypt the file with the SK, and subsequently unwrap it onto the HSM as an 85 byte generic secret object.

Output: The signature (output) produced from the above function call will contain the following data (note the exact function as per the 3GPP spec is show in parentheses):

| RANDOM | XRES(f2) | CK(f3) | IK(f4) | AUTN | where AUTN = | SQN xor AK(f5) | AMF | MAC-A(f1) |

Although by default the data is returned as one concatenated binary string, the user may have the data returned in TLV (Tag/Length/Value) format by setting the “LUNA_5G_USE_TLV” flag. The output will appear as follows:

| Tag (UINT8) | Length (UINT8) | Value | Tag (UINT8) | Length (UINT8) | Value | …

The tag values are defined as follows:

```
#define LUNA_5G_TAG_RANDOM          0x00000001      // Random data generated in HSM
#define LUNA_5G_TAG_RES             0x00000002      // Response string
#define LUNA_5G_TAG_CK              0x00000003      // Confidentiality Key
#define LUNA_5G_TAG_IK              0x00000004      // Integrity Key
#define LUNA_5G_TAG_SQN_XOR_AK      0x00000005      // Sequence # xor'd with Anonymity Key (AK)
#define LUNA_5G_TAG_AMF             0x00000006      // Authentication Management Field
#define LUNA_5G_TAG_MAC             0x00000007      // MAC-A for authentication
#define LUNA_5G_TAG_SEQUENCE        0x00000008      // Sequence # in resynch operation
```

Resynchronization

C_SignInit and C_Sign function calls same as Authentication.

Mechanism: **CKM_MILENAGE_RESYNC**

Parameter Structure: same as for Authentication.

The API allows the USIM to resynchronize with a new SQN number in the event that a message was lost. The SQN is not sent in the clear however; rather it is XOR’ed with the AK (f5*). As well the MAC-S (f1*) is also sent to verify that this request is coming from a legitimate subscriber. This requires that the USIM sends the random data and an AUTS (Auth string) to the AUC which contains the following:

AUTS = | SQN xor AK | MAC-S |

The user should format the pData (data to sign in the C_Sign call) as follows:

pData = | RAND | AUTS |

This mechanism uses the same mechanism parameter structure as for authentication, as the Ki, OP and AMF are required to generate the sequence number. (The sqn[] field in the param structure will be ignored.) The LUNA_5G_OP_OBJECT, LUNA_5G_USE_TLV, and LUNA_5G_USER_DEFINED_RC are valid flags.

Output: The HSM will first generate the new SQN number but will check the MAC-S signature to ensure the legitimacy of the request. If the MAC-S signature passed in is not equal to the signature calculated by the HSM, the HSM will return the error CKR_SIGNATURE_INVALID. If the return code is CKR_OK, the signature will contain the new 6 byte SQN number.

AUTS Generation (Testing Only)

The HSM supports generating the AUTS string for the purpose of testing the Resynchronization operation.

C_SignInit and C_Sign function calls same as Authentication.

Mechanism: **CKM_MILENAGE_AUTS**

Parameter Structure: same as for Authentication. The user should pass in the random string in the pData field of the C_Sign function call.

Output: The signature returned from the HSM will be a concatenation of the Random string plus the AUTS as follows (no TLV):

| RAND | SQN xor AK | MAC-S |

This string may be fed directly into the pData field of the C_Sign call when doing the RESYNC operation.

TUAK

Authentication

TUAK is similar to MILENAGE in respect to the output data however the calculated components are largely based upon the Keccak (SHA-3) hashing scheme. In addition, TUAK allows some variability in the lengths of the output components. The desired lengths can be specified in input parameter structure as follows:

```
typedef struct CK_TUAK_SIGN_PARAMS {
    CK_ULONG             ulTuakFlags;
    CK_ULONG             ulEncKiLen;
    CK_BYTE_PTR          pEncKi;
    CK_ULONG             ulEncTOPcLen;
    CK_BYTE_PTR          pEncTOPc;       // Encrypted or plain - see flags
    CK_ULONG             ulIterations;   // # of Keccak iterations
    CK_OBJECT_HANDLE     hSecondaryKey;  // optional OP key handle
    CK_ULONG             ulResLen;       // length of expected response (XRES)
    CK_ULONG             ulMacALen;      // length of MAC
    CK_ULONG             ulCkLen;        // length of crypto key CK
    CK_ULONG             ulIkLen;        // length of identity key IK
    CK_BYTE              sqn[6];
    CK_BYTE              amf[2];
} CK_TUAK_SIGN_PARAMS;

typedef CK_TUAK_SIGN_PARAMS CK_PTR CK_TUAK_SIGN_PARAMS_PTR;
```

The ulTuakFlags are the same as the MILENAGE flags with the exception of the LUNA_5G_USER_DEFINED_RC flag which is not applicable to TUAK.

Output: same as MILENAGE authentication output.

Resynchronization

Mechanism: **CKM_TUAK_RESYNC**

Parameter Structure: same as for authentication, although some fields are not be required (key length fields, for example).

Format of the input data, and the resulting output are the same as MILENAGE.

AUTS Generation (Testing Only)

As with MILENAGE, this function supports generation of the AUTS string for subsequent testing of the RESYNC operation.

Comp128

Authentication

The Comp128 algorithms supports device authentication for the legacy GSM 2/2.5 mobile networks.

Mechanism: CKM_COMP128

Parameter Structure:

```
typedef struct CK_COMP128_SIGN_PARAMS {
    CK_ULONG            ulVersion;              // Version of Comp128
    CK_ULONG            ulEncKiLen;
    CK_BYTE_PTR         pEncKi;
} CK_COMP128_SIGN_PARAMS;

typedef CK_COMP128_SIGN_PARAMS CK_PTR CK_COMP128_SIGN_PARAMS_PTR;
```

This API supports the 3 versions of the COMP128 algorithm. This is passed in via the ulVersion field.

As with the other mechanisms, it is expected that the Ki will be encrypted under the Storage Key (SK) – the handle for this key is passed in as the signing key in C_SignInit function call.

Output: The signature (output) produced from the above function call will contain the following data:

| RANDOM | SRES | Kc |

> **NOTE** Unlike MILENAGE and TUAK, the output is only in binary form, and the TLV formatting is not supported at this time. Only a single triplet output is supported via this API.

Storage Key (SK)

It is assumed that under the security constraints imposed by the HSM, the subscribers key (Ki) will always be encrypted by the HSM resident Storage Key (SK). The SK is an AES key and can be any size although a 256 bit (32 byte) value is recommended. Currently, the encryption/decryption mechanism used is the NIST approved CKM_AES_KWP (PKCS#11 definition) and where the default IV (per NIST SP800-38F) is used. The Ki (and optionally the OP) must be encrypted using this mechanism for later use in the authentication and resynchronization operations.

Luna Key Translation

On this page:

> "Mechanism Description" below
> "Data size" below
> "Summary" below
> "Notes" on the next page
> "Tooling" on page 110

Mechanism Description

CKM_KEY_TRANSLATE

Key Translation function - allows to securely import subscriber authentication keys into a 5G authentication platform (UDM).
This is a Proprietary Luna mechanism.

This mechanism receives a cryptogram from the client and re-encrypts it using a different key and/or mechanism. The mechanism returns the resulting cryptogram to the client.

The mechanism is used with the C_WrapKey command with the following parameters:

CK_SESSION_HANDLE hSession	current session
CK_MECHANISM_PTR pMechanism	Mechanism parameter is a pointer to CK_MECHANISM_PARAMS
CK_OBJECT_HANDLE hWrappingKey	handle of output wrapping key
CK_OBJECT_HANDLE hKey	In this case, it will be always set to CK_INVALID_HANDLE
CK_BYTE_PTR pWrappedKey	address to where new cryptogram is stored – (length prediction supported)
CK_ULONG_PTR pulWrappedKeyLen	address where to store output buffer size and actual/predicted output length

Data size

The maximum allowed data size for this mechanism is 8000 bytes

Summary

See "CKM_KEY_TRANSLATE" on page 339.

CK_MECHANISM_PARAM for KEY_TRANSLATE mechanism is structured as follow

```
typedef struct CK_MECHANISM{
      CK_MECHANISM_TYPE mechanism; /* CKM_KEY_TRANSLATE*/
      CK_VOID_PTR pParameter; /* pointer to CK_KEY_TRANSLATE_PARAMs */
      CK_ULONG ulParameterLen;
} CK_MECHANISM;
```

CK_ KEY_TRANSLATE_PARAMS is a structure that provides the parameters to the CKM_KEY_TRANSLATE mechanism. The structure is defined as follows:

```
typedef struct CK_KEY_TRANSLATE_PARAMS {
      CK_FLAGS  flags;
      CK_MECHANISM wrapMech;
      CK_MECHANISM unWrapMech;
      CK_BYTE_PTR pData;
      CK_ULONG ulDataLen;
      CK_OBJECT_HANDLE hUnwrapKey; /* input unwrapped handle (hA4key)*/
} CK_ KEY_TRANSLATE_PARAMS;
```

Notes

The *flags* field of the mechanism parameter is reserved for future use and must be set to zero.

The *wrapMech* parameter must be a valid key wrapping mechanism for the key type of *hWrappingKey*. The key *hWrappingKey* must have the *CKA_WRAP* attribute set true.

The *pData* is the wrapped key data with a length of *ulDataLen* to translate using the wrap mechanism

The *unWrapMech* parameter must be a valid key unwrapping mechanism for the key type of *hUnwrapKey*. The key *hUnwrapKey* must have the *CKA_UNWRAP* attribute set true. Any *CKA_UNWRAP_TEMPLATE* attribute on the *hUnwrapKey* is ignored.

If a key is wrapped/unwrapped with a mechanism that does not support content padding – such as CKA_AES_ECB then the implication is that the key content is a multiple of block size.

Partition Policy settings that control key import/export such as:

> Enable private key cloning,

> Enable private key wrapping,

> Enable private key unwrapping,

> Enable private key masking,

> Enable secret key cloning,

> Enable secret key wrapping,

> Enable secret key unwrapping,

> Enable secret key masking,

> Enable private key unmasking,

> Enable secret key unmasking

...have no effect on the CKM_KEY_TRANSLATE.

These partition policies affect which values for the *wrapMech* or *unWrapMech* can be used:

> Enable non-FIPS algorithms,

> Enable RSA PKCS mechanism,

> Enable CBC-PAD (un)wrap keys of any size

Handling of wrapping and padding

Wrapping mechanisms that do not encode a padding length automatically append zeros to the key value, if the length of the key being wrapped is not a multiple of the wrapping algorithm block size.
For example: CKM_AES_ECB has a block size of 16 and, if it is used to wrap a 24 byte key, then 8 zeros are appended to the key before wrapping.

When unwrapping a key that has such padding applied the extra zeros are NOT stripped from the key value, Therefore the key value that is wrapped includes the exrtra zeros.

Constants

#define LUNA_MECH_KEY_TRANSLATE	0x80000E10

Tooling

CKDemo
See item 69 Translate Key in KEY Menu Functions.

JCPROV:
JCPROV API supports the mechanism. A program “KeyTranslate” added to the JCPROV Samples.

Luna HSM Client 10.5.1 and newer has fixed input and output mechanisms (DES3_CBC and AES_KWP respectively). These mechanisms were chosen as they represent the initial use case.

multitoken:
The multitoken tool has an option to test performance of the KeyTranslate mechanism.

Luna HSM Client 10.5.1 and newer has fixed input and output mechanisms (DES3_CBC and AES_KWP respectively). These mechanisms were chosen as they represent the initial use case.

Derive Template

The CKA_DERIVE_TEMPLATE attribute is an optional extension to the C_DeriveKey function. This attribute points to an array template which provides additional security by restricting important attributes in the resulting derived key. This derive template, along with the user-supplied application template (called pTemplate in the PKCS#11 specification), determine the attributes of the derived key.

To invoke a derive template, the base key must have the CKA_DERIVE_TEMPLATE attribute set, pointing to a user-supplied derive template. When you specify this attribute on the base key and then attempt to derive a key, the derive operation adds the attributes of the application template to the attributes in the derive template. If there are any mismatches between attribute values specified in the two templates, the derive operation fails. Otherwise, the operation succeeds, producing a derived key with the combined attributes of the two templates.

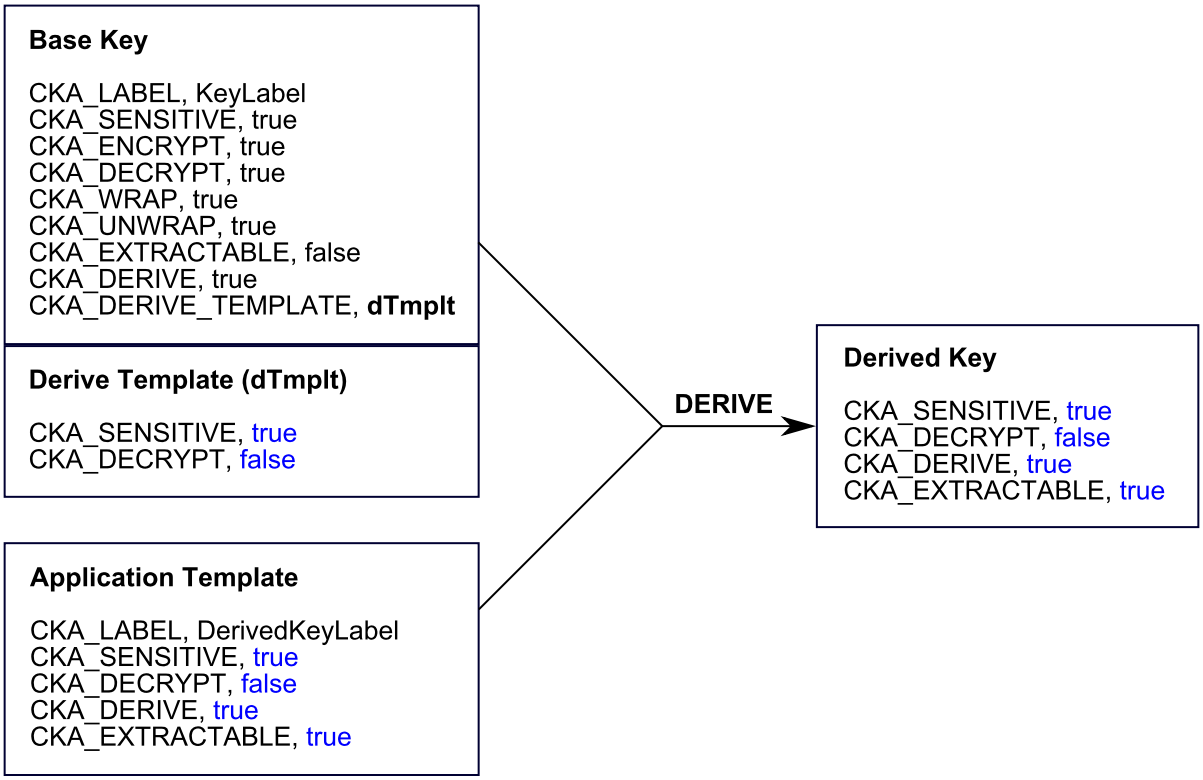
Any and all attributes which are valid for application template of a particular mechanism are also valid for the derive template. For security, the most effective attributes to restrict are those which might allow the derived key to be misused or expose secret information. Broadly these include but are not limited to encryption/decryption capabilities, extractability, the CKA_SENSITIVE attribute and the CKA_MODIFIABLE attribute. All mechanisms which support key derivation also support derive templates.

Examples

The following examples show a successful derivation with a derive template, and a failed derivation.

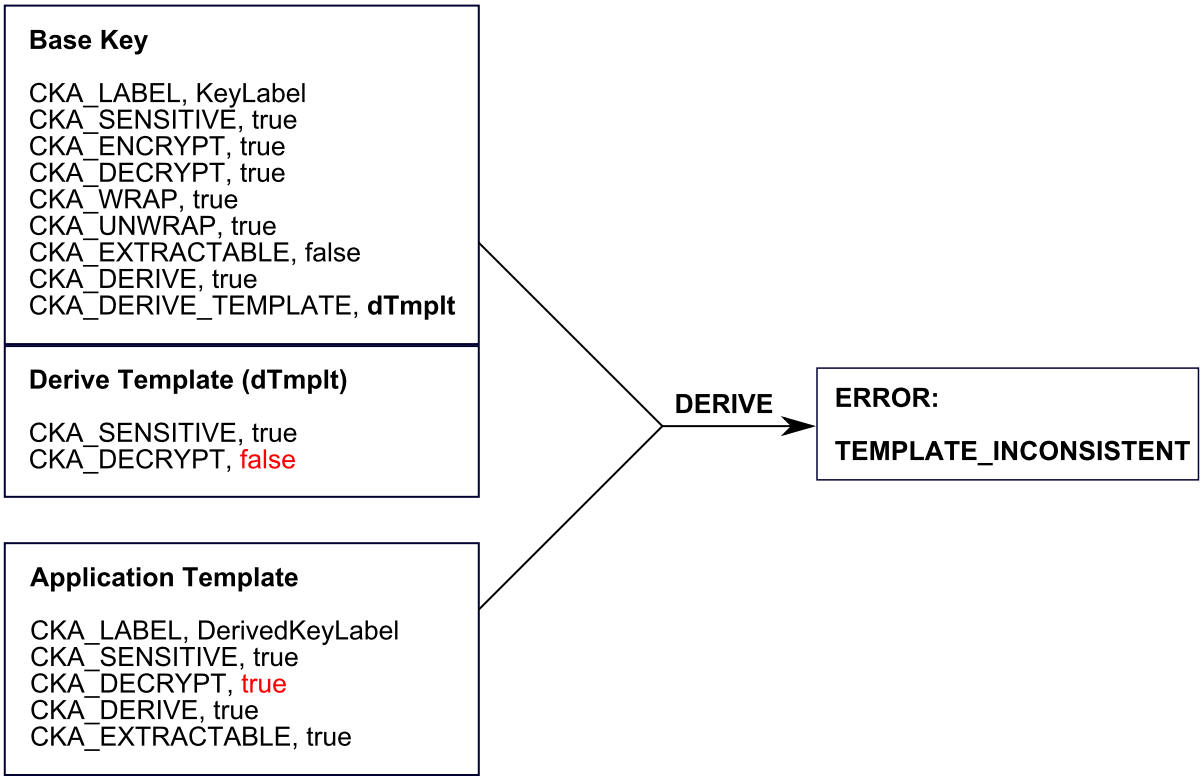
Successful Derivation

Here, the base key has the CKA_DERIVE_TEMPLATE attribute pointing to the derive template dTmplt. There are no conflicts between dTmplt and the application template. The application template's extra attributes are added to dTmplt's attributes, and the derivation operation produces a derived key containing the attributes in the two templates.



Failed Derivation

Here, the base key has the CKA_DERIVE_TEMPLATE attribute pointing to the derive template dTmplt. Notice that dTmplt has the CKA_DECRYPT attribute set to false, where the application template has the CKA_DECRYPT attribute set to true. This conflict causes the derivation operation to fail with the error TEMPLATE_INCONSISTENT.



Counter Mode KDF Mechanisms

The Luna PCIe HSM 7s support the following two vendor defined mechanisms. They can be used to perform Counter Mode KDF (key derivation functions) using various CMAC algorithms (DES3, AES, ARIA, SEED) as the PRF (pseudo-random function). See NIST SP 800-108.

```
#define CKM_NIST_PRF_KDF                        (CKM_VENDOR_DEFINED + 0xA02)
#define CKM_PRF_KDF                             (CKM_VENDOR_DEFINED + 0xA03)

/* Parameter and values used with CKM_PRF_KDF and * CKM_NIST_PRF_KDF. */

typedef CK_ULONG CK_KDF_PRF_TYPE;
typedef CK_ULONG CK_KDF_PRF_ENCODING_SCHEME;

/** PRF KDF types */
#define CK_NIST_PRF_KDF_DES3_CMAC      0x00000001
#define CK_NIST_PRF_KDF_AES_CMAC       0x00000002
#define CK_PRF_KDF_ARIA_CMAC           0x00000003
#define CK_PRF_KDF_SEED_CMAC           0x00000004

#define LUNA_PRF_KDF_ENCODING_SCHEME_1     0x00000000
#define LUNA_PRF_KDF_ENCODING_SCHEME_2     0x00000001

typedef struct CK_KDF_PRF_PARAMS {
   CK_KDF_PRF_TYPE            prfType;
   CK_BYTE_PTR                pLabel;
   CK_ULONG                   ulLabelLen;
   CK_BYTE_PTR                pContext;
   CK_ULONG                   ulContextLen;
```

```
   CK_ULONG                    ulCounter;
   CK_KDF_PRF_ENCODING_SCHEME ulEncodingScheme;
} CK_PRF_KDF_PARAMS;

typedef CK_PRF_KDF_PARAMS CK_PTR CK_KDF_PRF_PARAMS_PTR;
```

SM2/SM4 Mechanisms

This section describes the C-based PKCS#11 interface to the SM2/SM4 functions in the HSM firmware. Although PKCS#11 constitutes the core client-side API to the HSM, it is expected that other API layers (like Java) will be required in order to support the customers' application environment. These APIs are yet to be defined.

> **NOTE** This feature requires minimum Luna HSM Firmware 7.4.2 (Luna HSM Firmware 7.7.0 for Luna PCIe HSM 7) and Luna HSM Client 10.2.0 (or a patched Luna HSM Client 7.4.0). You require a backup HSM with minimum Luna Backup HSM 7 Firmware 7.7.1 to back up these objects.

> "SM2" below

> "SM4" on the next page

SM2

Generate Key Pair

Generate a keypair of type CKK_SM2.

```
CK_BYTE sm2p256v1[] =  { 0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, 0x82, 0x2D };

CK_RV C_GenerateKeyPair(
CK_SESSION_HANDLE hSession,
CK_MECHANISM_PTR pMechanism,           // CKM_SM2_KEY_PAIR_GEN
CK_ATTRIBUTE_PTR pPublicKeyTemplate,   // eg CKA_EC_PARAMS with curve sm2p256v1 or any other
curve
CK_ULONG ulPublicKeyAttributeCount,
CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
CK_ULONG ulPrivateKeyAttributeCount,
CK_OBJECT_HANDLE_PTR phPublicKey,
CK_OBJECT_HANDLE_PTR phPrivateKey
);
```

Sign

Use the C_Sign* family of functions.

```
typedef struct CK_SM2DSA_PARAMS {
   CK_MECHANISM_TYPE zhashAlg;         // eg CKM_SM3
   CK_ULONG ulUserIdLen;
   CK_VOID_PTR pUserId;
} CK_SM2DSA_PARAMS;

CK_RV C_SignInit(
CK_SESSION_HANDLE hSession,
```

```
CK_MECHANISM_PTR pMechanism,           // eg CKM_SM3_SM2DSA with CK_SM2DSA_PARAMS
CK_OBJECT_HANDLE hKey
);

CK_RV C_Sign(
CK_SESSION_HANDLE hSession,
CK_BYTE_PTR pData,
CK_ULONG ulDataLen,
CK_BYTE_PTR pSignature,
CK_ULONG_PTR pulSignatureLen
);
```

Also supported:

- C_SignUpdate, C_SignFinal for multi-part operations
- C_VerifyInit, C_Verify for verifying (single-part operations)
- C_VerifyUpdate, C_VerifyFinal for verifying (multi-part operations)

Available mechanisms for signing include:

- **CKM_SM2DSA**
- **CKM_SM3_SM2DSA**
- **CKM_SHA1_SM2DSA**
- **CKM_SHA224_SM2DSA**
- **CKM_SHA256_SM2DSA**
- **CKM_SHA384_SM2DSA**
- **CKM_SHA512_SM2DSA**

Available mechanisms for field **zHashAlg** include:

- **CKM_SM3**
- **CKM_SHA1**
- **CKM_SHA224**
- **CKM_SHA256**
- **CKM_SHA384**
- **CKM_SHA512**

SM4

Generate Key

Generate a secret key of type CKK_SM4.

```
CK_RV C_GenerateKey(
CK_SESSION_HANDLE hSession
CK_MECHANISM_PTR pMechanism,           // CKM_SM4_KEY_GEN
CK_ATTRIBUTE_PTR pTemplate,
CK_ULONG ulCount,
CK_OBJECT_HANDLE_PTR phKey
);
```

Encrypt

Use the C_Encrypt* family of functions.

```
CK_RV C_EncryptInit(
CK_SESSION_HANDLE hSession,
CK_MECHANISM_PTR pMechanism,          // eg CKM_SM4_CBC_PAD with InitializationVector [16 bytes]
CK_OBJECT_HANDLE hKey
);

CK_RV C_Encrypt(
CK_SESSION_HANDLE hSession,
CK_BYTE_PTR pData,
CK_ULONG ulDataLen,
CK_BYTE_PTR pEncryptedData,
CK_ULONG_PTR pulEncryptedDataLen
);
```

Also supported:

- C_EncryptUpdate, C_EncryptFinal for multi-part operations
- C_DecryptInit, C_Decrypt for decrypting (single-part operations)
- C_DecryptUpdate, C_DecryptFinal for decrypting (multi-part operations)

Available mechanisms for encryption include:

- **CKM_SM4_ECB**
- **CKM_SM4_CBC**
- **CKM_SM4_CBC_PAD**

SHA-3 Mechanisms

This section describes the PKCS#11 interface to the **SHA-3** mechanisms in the HSM firmware for the digest bit lengths of **224**, **256**, **384** and **512**.

> **NOTE** This feature requires minimum Luna HSM Firmware 7.4.2 (Luna HSM Firmware 7.7.0 for Luna PCIe HSM 7) and Luna HSM Client 10.2.0 (or a patched Luna HSM Client 7.4.0). You require a backup HSM with minimum Luna Backup HSM 7 Firmware 7.7.1 to back up these objects.

- "Digest Mechanisms" on the next page
- "HMAC Mechanisms" on the next page
- "Signature/Verification Mechanisms" on page 117
- "Encrypt/Decrypt Mechanisms" on page 118
- "Digest Key Derive Mechanisms" on page 118
- "Key Derivation Function (KDF) Mechanisms" on page 119

Digest Mechanisms

Mechanism Type	Description
SHA-3	The following mechanisms for performing a **SHA-3** hash have been added: > **CKM_SHA3_224** > **CKM_SHA3_256** > **CKM_SHA3_384** > **CKM_SHA3_512**

SHAKE	The following mechanisms for performing a **SHAKE XOF** have been added:
	> **CKM_SHAKE_128**
	> **CKM_SHAKE_256**
	These mechanisms require a **CK_SHAKE_PARAMS** mechanism parameters structure defined as follows:
	`typedef struct CK_SHAKE_PARAMS {`
	`  CK_ULONG ulOutputLen;`
	`} CK_SHAKE_PARAMS`
	The output length of the digest can be specified using the **ulOutputLen** field with a maximum value of **2048**.

KECCAK	There are variants of the **SHA-3** mechanisms that are included for compatibility with implementations that preceded the publication of **FIPS PUB 202** where there is a difference in a single padding byte. These mechanisms are: > **CKM_KECCAK_224** > **CKM_KECCAK_256** > **CKM_KECCAK_384** > **CKM_KECCAK_512**

HMAC Mechanisms

The following mechanisms for performing an **HMAC** with **SHA-3** have been added:

> **CKM_SHA3_224_HMAC**

> **CKM_SHA3_224_HMAC_GENERAL**

> **CKM_SHA3_256_HMAC**

> **CKM_SHA3_256_HMAC_GENERAL**

> **CKM_SHA3_384_HMAC**

> **CKM_SHA3_384_HMAC_GENERAL**

> **CKM_SHA3_512_HMAC**

> **CKM_SHA3_512_HMAC_GENERAL**

 Copyright 2001-2024 Thales Group

Signature/Verification Mechanisms

Mechanism Type	Description
RSA PKCS	The following mechanisms for performing a **RSA PKCS #1 v1.5** signature/verification with a **SHA-3** digest have been added: > **CKM_SHA3_224_RSA_PKCS** > **CKM_SHA3_256_RSA_PKCS** > **CKM_SHA3_384_RSA_PKCS** > **CKM_SHA3_512_RSA_PKCS**
RSA PSS	The following mechanisms for performing a **RSA** signature/verification with **PSS** encoding with a **SHA-3** digest have been added: > **CKM_SHA3_224_RSA_PKCS_PSS** > **CKM_SHA3_256_RSA_PKCS_PSS** > **CKM_SHA3_384_RSA_PKCS_PSS** > **CKM_SHA3_512_RSA_PKCS_PSS** The following **MGF1** constants have been defined with corresponding support: > **CKG_MGF1_SHA3_224** > **CKG_MGF1_SHA3_256** > **CKG_MGF1_SHA3_384** > **CKG_MGF1_SHA3_512** These values can be specified via the **mgf** field of the CK_RSA_PKCS_PSS_PARAMS mechanism parameters. The **hashAlg** field of the CK_RSA_PKCS_PSS_PARAMS mechanism parameters can be given the new values of **CKM_SHA3_224**, **CKM_SHA3_256**, **CKM_SHA3_384** or **CKM_SHA3_512**.
DSA	The following mechanisms for performing a **DSA** signature/verification with a **SHA-3** digest have been added: > **CKM_DSA_SHA3_224** > **CKM_DSA_SHA3_256** > **CKM_DSA_SHA3_384** > **CKM_DSA_SHA3_512**
ECDSA	The following mechanisms for performing an **ECDSA** signature/verification with a **SHA-3** digest have been added: > **CKM_ECDSA_SHA3_224** > **CKM_ECDSA_SHA3_256** > **CKM_ECDSA_SHA3_384** > **CKM_ECDSA_SHA3_512**

 Copyright 2001-2024 Thales Group

Mechanism Type	Description
EDDSA	The following mechanisms for performing an **EDDSA** signature/verification with a **SHA-3** digest have been added: > **CKM_SHA3_224_EDDSA** > **CKM_SHA3_256_EDDSA** > **CKM_SHA3_384_EDDSA** > **CKM_SHA3_512_EDDSA**

Encrypt/Decrypt Mechanisms

CKM_RSA_PKCS_OAEP
For the **CKM_RSA_PKCS_OAEP** mechanism, the following values can be specified for the **mgf** field of the **CK_RSA_PKCS_OAEP_PARAMS** mechanism parameters:

> **CKG_MGF1_SHA3_224**

> **CKG_MGF1_SHA3_256**

> **CKG_MGF1_SHA3_384**

> **CKG_MGF1_SHA3_512**

For the **hashAlg** field of the **CK_RSA_PKCS_OAEP_PARAMS** mechanism parameters, the following hash algorithms can be specified:

> **CKM_SHA3_224**

> **CKM_SHA3_256**

> **CKM_SHA3_384**

> **CKM_SHA3_512**

Digest Key Derive Mechanisms

Mechanism Type	Description
SHA-3	The following mechanisms can be used to derive a key using **SHA-3**: > **CKM_SHA3_224_KEY_DERIVE** > **CKM_SHA3_256_KEY_DERIVE** > **CKM_SHA3_384_KEY_DERIVE** > **CKM_SHA3_512_KEY_DERIVE**
SHAKE	The following mechanisms can be used to derive a key using **SHAKE**: > **CKM_SHAKE_128_KEY_DERIVE** > **CKM_SHAKE_256_KEY_DERIVE**

 Copyright 2001-2024 Thales Group

Key Derivation Function (KDF) Mechanisms

Mechanism Type	Description
CKM_X9_42_DH_DERIVE **CKM_ECDH1_DERIVE**	The following values can be specified for the **kdf** field of the **CK_X9_42_DH1_DERIVE_PARAMS** and **CK_ECDH1_DERIVE_PARAMS** mechanism parameters to make use of the **SHA-3** variants: > **CKD_SHA3_224_KDF** > **CKD_SHA3_256_KDF** > **CKD_SHA3_384_KDF** > **CKD_SHA3_512_KDF** > **CKD_SHA3_224_NIST_KDF** > **CKD_SHA3_256_NIST_KDF** > **CKD_SHA3_384_NIST_KDF** > **CKD_SHA3_512_NIST_KDF** > **CKD_SHA3_224_SES_KDF** > **CKD_SHA3_256_SES_KDF** > **CKD_SHA3_384_SES_KDF** > **CKD_SHA3_512_SES_KDF**
CKM_PRF_KDF	The following values can be specified for the **prfType** field of the **CK_PRF_KDF_PARAMS** mechanism parameters to make use of the **SHA-3** variants: > **CK_NIST_PRF_KDF_HMAC_SHA3_224** > **CK_NIST_PRF_KDF_HMAC_SHA3_256** > **CK_NIST_PRF_KDF_HMAC_SHA3_384** > **CK_NIST_PRF_KDF_HMAC_SHA3_512**

 Copyright 2001-2024 Thales Group

CHAPTER 4: Per-Key Authorization API

Design

Changes and additions to the Luna HSM firmware and libraries for the Per Key Authorization (PKA) feature were introduced to implement the feature in compliance with eIDAS standards, and to ensure co-existence of the new feature with existing behavior and use-cases.

The Luna use-case

This use-case covers traditional Luna customers who want the latest firmware and host software, but do *not* need the new per-key authorization and sole control functionality, and do not want to make changes to their existing applications, and yet want to be able to maintain the HSM in the Common Criteria mode of operations.

Subject to the installation of the new Client Cryptoki library and the use of regular APIs without per-key auth parameters, the existing APIs will continue to work as usual, while the new PKCS#11 APIs, attributes, and roles, related to the per-key authorization and sole-control functionality, are dealt with within the library and remain invisible to the user.

In the Luna use-case, we have

> the Crypto Officer role (CO) for creating/modifying/administering key material in an application partition

> the Crypto User role (CU) with read/use capability

The eIDAS use-case

This use-case covers all the scenarios where client applications need to take advantage of the per-key authorization and sole control (assigned keys) capabilities. It includes all eIDAS scenarios such as remote server signing/sealing with external/internal SAM, local server signing/sealing (no SAM), long-lived signing keys, single-use signing keys, etc.

In the eIDAS use-case,

> the Crypto Officer role (the CO) does much the same as in the Luna use-case, and maps to the eIDAS "*Administrator*" role

> a new Limited Crypto Officer role (the LCO), in line with the principle of least privilege, and mapping to the eIDAS "*User*" role, allows signing applications to provide key access to users, thus reserving the CO role for administrative purposes only

The CU role has insufficient capability and is not employed for the eIDAS use case.

Applications need the new firmware, new host software, and use new P11 APIs, attributes, and roles related to the per-key authorization and sole control.

Customer applications and tools explicitly use the new features:

> Applications specify the Authorization Data and Assigned Flag attributes in templates and may later change the attributes via new P11 APIs.

> Applications use new PKCS#11 API to explicitly authorize each individual key before it can be used in crypto operations.

> Applications also deal with the re-authorization logic and the authorization failure handling.

The following sections detail the changes implemented for this feature.

New Assigned Key Attribute

The Protection Profile defines two types of secret keys (PP defines “secret key” as either a symmetric or a private key), **assigned** and **unassigned** (general). Assigned keys have tighter controls around them, their security attributes are non-modifiable, and they are also non-exportable. Both key types, however, require per-key authorization.

We support the following use-cases:

> CO (Crypto Officer a.k.a. the eIDAS Administrator) generates the key unassigned, then assigns it later.

> CO (Administrator) generates the key assigned.

> Limited-CO (the eIDAS User) generates the key assigned.

> Limited-CO (User) generates the key unassigned (which might or might not be assigned by the Administrator later).

To differentiate between assigned and unassigned keys, we introduce an attribute, CKA_ASSIGNED. This attribute is initialized and modified through regular templates and commands. CKA_ASSIGNED defaults to FALSE, and as such, it does not need to be present in the templates for the Luna use-case, since we exclusively want to use unassigned keys in that use-case.

Getting Assigned Keys

There are two ways to get assigned keys:

> Direct generation of assigned keys

This is accomplished by generating keys with the CKA_ASSIGNED attribute set to TRUE. The keys must also have the following attributes:

- CKA_EXTRACTABLE set to FALSE.
- CKA_MODIFIABLE set to FALSE.
- CKA_AUTH_DATA set to some authorization data. For more information about this attribute, see "New Authorization Data Attribute" on the next page.

> Assignment of previously created keys

This is accomplished by performing an unassigned-to-assigned transition as the eIDAS Administrator (the CO role) and changing the CKA_ASSIGNED attribute from FALSE to TRUE by using the CA_AssignKey command (see "CA_AssignKey" on page 127). The keys must also have the following attributes:

- CKA_ALWAYS_SENSITIVE set to TRUE.
- CKA_NEVER_EXTRACTABLE set to TRUE.
- CKA_AUTH_DATA set to some authorization data. For more information about this attribute, see "New Authorization Data Attribute" on the next page.

A template for key assignment must have the following attributes:

> CKA_MODIFIABLE set to FALSE.

> CKA_ASSIGNED set to TRUE.

New Authorization Data Attribute

The authorization data is a new sensitive attribute (CKA_AUTH_DATA) on each symmetric/private key object. It is initialized through the regular means of passing in attribute values in templates during key generation. As in other sensitive attributes, it is possible to query the value. It is presented as a byte array password, therefore no multifactor quorum options are possible. This attribute is available in the eIDAS and Luna use-cases (V1 Partitions), but not in the non-PP-compatible legacy use-case (V0).

Unassigned keys use the authorization data simply to verify access rights. The initial authorization data is hashed, along with a random pepper value (stored encrypted in the partition structure with the PSK) and the hash is stored in the key object metadata. Authorization operation involves hashing of the incoming authorization data and comparing the result to the reference one in the key object.

For assigned keys, the use of the authorization data is more complex, and is detailed in the following section.

Initializing the Authorization Data

The auth data is initialized through regular means of passing in the CKA_AUTH_DATA attribute value in templates during key generation, key derivation and key unwrapping.

Modifying the Authorization Data

For authorization data modification, the Protection Profile introduces the following conditions:

> In the case of *assigned* keys, the auth data can be “modified only when modification operation includes successful validation of current (pre-modification) authorisation data”

> In the case of *unassigned* keys, the auth data can be “modified only when modification operation includes successful validation of current (pre-modification) authorisation data, that is by anybody who already has that auth data, or by an Administrator”

To this end, the following command modifies the authorization data when in possession of current auth data:

```
CA_SetAuthorizationData(
           CK_SESSION_HANDLE hSession,          // the session's handle
           CK_OBJECT_HANDLE  hObject,           // the object's handle
           CK_UTF8CHAR_PTR   pOldAuthData,      // the user's old/current auth data
           CK_ULONG          ulOldAuthDataLen   // the length of the old/current auth data
           CK_UTF8CHAR_PTR   pNewAuthData,      // the user's new auth data
           CK_ULONG          ulNewAuthDataLen   // the length of the new auth data
         )
```

This command will be available to all the roles without explicit requirement to have authorized first with CA_AuthorizeKey(), since the call itself will take in the current authorization data as a parameter.

On assigned keys, this command will decrypt and re-encrypt the KUSK with the new authorization data.

Resetting the Authorization Data

For authorization data modification, the Protection Profile introduces the following conditions:

> In the case of assigned keys, it can be “modified only when modification operation includes successful validation of current (pre-modification) authorisation data”

> In the case of unassigned keys, it can be “modified only when modification operation includes successful validation of current (pre-modification) authorisation data, or by an Administrator”

To this end, the following command at the PKCS#11 API level allows the CO (Administrator) to reset the authorization data on unassigned keys without having to present the current auth data:

```
CA_ResetAuthorizationData(
          CK_SESSION_HANDLE hSession,       // the session's handle
          CK_OBJECT_HANDLE  hObject,        // the object's handle
          CK_UTF8CHAR_PTR   pAuthData,      // the user's auth data
          CK_ULONG          ulAuthDataLen   // the length of the auth data
        )
```

This command is available only to the CO role, and only for unassigned keys.

This command also resets the authorization failure count (CKA_FAILED_KEY_AUTH_COUNT) for a locked-out key and unlocks it.

Authorizing/Rescinding Authorization on a Key *per Session*

To preserve compatibility with the PKCS#11 API, and rather than require the input of per-key authorization data on every single “keyed” operation such as sign, verify, encrypt, decrypt, derive, etc., the following command explicitly authorizes a key (assigned or unassigned) by key handle in a given session. Once authorized, the key remains authorized within that session only (meaning that the authorization is not inherited by the other sessions under that access), until authorization is explicitly rescinded. In line with PP 419 221-5, FIA_UAU.6/KeyAuth Re-authenticating, other re-authorization conditions are not supported (time/count-based, or otherwise).

```
CA_AuthorizeKey(
          CK_SESSION_HANDLE hSession,       // the session's handle
          CK_OBJECT_HANDLE  hObject,        // the object's handle
          CK_UTF8CHAR_PTR   pAuthData,      // the user's auth data
          CK_ULONG          ulAuthDataLen   // the length of the auth data
        )
```

The command above can be used only in an already authenticated session for any role, and attempts to authorize the given key. If successful it marks and stores the key's OUID in the session as authorized. Further requests to use that key within that session will simply verify that the key is still marked as authorized, and will proceed with the operation (provided that the key usage attributes are respected) without having to re-authorize.

Only one authorized key at a time is supported per session. The command can be used to overwrite the existing authorization in a session with authorization for a different key.

If the application wants to authorize a second key concurrently, a second session will be required.

The same command will also allow explicit authorization rescinding in that given session alone, if the auth data length is 0 (object handle still must be set to the correct one). Other rescinding methods include but are not limited to:

> Pull the power plug

> Decommission

> Delete the partition

> Close the session

> Logout

> Change auth data on the key (this will revoke any existing authorizations for that key in any sessions/accesses)

> Delete the key

Authorizing/Rescinding Authorization on a Key *per Access*

Authorization data can be directly put into the access, but only to access unassigned keys.

This is achieved, using the same API call as above, when hObject is set to CK_INVALID_HANDLE.

The provided authorization data is not immediately used as a result of that command, since the command does not specify a key at that point. Instead, the hash of the authorization data is kept in the access and is available to all of the sessions sharing the same access.

The authorization in this case is a two-step process:

1. Authorization data only is sent in
2. A “keyed” command is sent in, and the authorization data stored in the access is validated against the individual key.

If, in a given session, authorization already exists for an individual key at the session level, that is checked first, if the OUID matches the key in the command.

Note that the overall authorization process remains per individual key, and each key is individually authorized for use.

As with the authorization in the previously described session, the same command also allows explicit authorization rescinding if the auth data length is 0 (object handle is set to CK_INVALID_HANDLE). Other rescinding methods include but are not limited to:

> Pull the power plug

> Decommission

> Delete the partition

> Logout

> Change auth data on the key (this will revoke any existing authorizations for that key in any sessions/accesses)

> Delete the key

Per-Key Authorization Failure Handling

Access to keys is blocked after a set amount of authorization failures in accordance with the authentication failure handling requirements of the PP, and the CO role has the ability to unblock blocked keys.

The authorization failure count is stored as an attribute (CKA_FAILED_KEY_AUTH_COUNT) directly in the object. Another new-to-firmware 7.7.0 attribute, CKA_KEY_STATUS, contains the max allowed failed key auth attempt count, along with the current status of the key - locked or not.

Once the max failed attempt count is reached, the key is marked as locked. The CO can then reset the count to 0 (through C_SetAttributeValue()) in the attribute to unblock the access to the key.

Template Handling in the Cryptoki Library

For the Luna use-case (partition is V0 and SKS and PKA are not available), the library masks the artifacts of per-key authorization.

The C_Login() call for CO/CU is overridden to include the additional action:

1. A normal C_Login() call with all the provided parameters, followed by
2. A CA_AuthorizeKey() call with the string “Luna” passed in as authorization data to the access

The following calls that create a secret/private key are overridden, and add a CKA_AUTH_DATA attribute with the value “Luna” is added to the input template:

> C_GenerateKey()

> C_GenerateKeyPair()

> C_UnwrapKey()

> C_DeriveKey()

These modifications in the library, along with the two-step per-key authorization functionality, allow operation through the regular Luna API with no modifications required to existing applications, while still providing the ability to make use of the extended per-key auth API should the customer choose to do so.

V0 vs V1 Partitions

To support legacy client installations, the V0 and V1 partition types were introduced in Luna HSM Firmware 7.7.0 and newer, where

> the V0 partition preserves compatibility with the “legacy” Luna use-case, or keys always in hardware (whether a partition is V0 due to firmware update, or V0 due to default choice at partition creation), while

> the V1 partition supports Per Key Authorization, Scalable Key Storage, and other eIDAS-use-case functions that cannot be compatible with the previous scheme. These are indicated and selected by a new policy: “Partition Version”, which are set to 0 (the state of any pre-existing partitions after upgrade from pre-version-7.7.0 firmware, or the default setting for any newly created partitions) regular, non-legacy partitions will be version 1), and will control both internal legacy functionality (such as old HA login and old STC support), as well as govern toggling of certain other policy bits (such as per-key auth and SKS policies).

V0 partitions are not compliant with the Protection Profile, as the key objects in these partitions do not have authorization data attribute at all. This configuration is intended for customers who cannot upgrade their host-side library.

If a customer decides to move on to a PP compatible library, they can change the partition policy bit to V1. The firmware will assign the default auth data that is in the access to all of the key objects in the partition (please see section "Import via V0 to V1 Partition Conversion" on page 130 for further considerations) . If using the:

> Luna use-case, there is nothing further to do, the objects already have the correct auth data assigned to be used

> eIDAS use-case, you can follow-up the conversion operation by a separate call to set the per-key auth data to the desired value.

Cryptoki API

This section highlights the changes at the Cryptoki API level.

CA_AuthorizeKey

```
CA_AuthorizeKey(
          CK_SESSION_HANDLE hSession,       // the session's handle
          CK_OBJECT_HANDLE  hObject,        // the object's handle
          CK_UTF8CHAR_PTR   pAuthData,      // the user's auth data
          CK_ULONG          ulAuthDataLen   // the length of the auth data
        )
```

This is a new command, as of firmware 7.7.0, to explicitly authorize a key (assigned or unassigned) by key handle in a given session. It can be used only in an already-authenticated session for any role.

CA_SetAuthorizationData

```
CA_SetAuthorizationData(
          CK_SESSION_HANDLE hSession,          // the session's handle
          CK_OBJECT_HANDLE  hObject,           // the object's handle
          CK_UTF8CHAR_PTR   pOldAuthData,      // the user's old/current auth data
          CK_ULONG          ulOldAuthDataLen   // the length of the old/current auth data
          CK_UTF8CHAR_PTR   pNewAuthData,      // the user's new auth data
          CK_ULONG          ulNewAuthDataLen   // the length of the new auth data
        )
```

This command modifies the authorization data for a key, and is available to all the roles without explicit requirement to have been authorized first with CA_AuthorizeKey(), since the call itself takes in the current authorization data as a parameter.

Old (current) auth data is an optional parameter. If not provided, this data is filled in by the library to the “Luna” value to accommodate the case of keys imported through the migration scenarios in section (which will have their auth data set initially from the access, hence “Luna” as well).

This case appears to the end-user as though they are setting the per-key auth of an imported key for the first time.

The following return codes are added, that can be returned by this command:

> CKR_AUTH_DATA_TOO_LARGE

> CKR_AUTH_DATA_TOO_SMALL

CA_ResetAuthorizationData

```
CA_ResetAuthorizationData(
          CK_SESSION_HANDLE hSession,       // the session's handle
          CK_OBJECT_HANDLE  hObject,        // the object's handle
          CK_UTF8CHAR_PTR   pAuthData,      // the user's auth data
          CK_ULONG          ulAuthDataLen   // the length of the auth data
        )
```

This command resets the authorization data for a key, and is available to the CO role only, and only for the unassigned keys.

This command also resets the authorization failure count (CKA_FAILED_KEY_AUTH_COUNT) for a locked out key and unlocks it.

Two new return codes can be returned by this command:

> CKR_AUTH_DATA_TOO_LARGE

> CKR_AUTH_DATA_TOO_SMALL

CA_AssignKey

```
CA_AssignKey(
          CK_SESSION_HANDLE hSession,       // the session's handle
          CK_OBJECT_HANDLE  hObject         // the object's handle
        )
```

This command flags a key as assigned by setting its CKA_ASSIGNED attribute to 1, and is available to the CO role only, and only for the unassigned keys.

The key has to satisfy the following conditions:

> It must have CKA_AUTH_DATA

> It must have CKA_EXTRACTABLE = false

> It must have CKA_SENSITIVE = true

> It must have CKA_MODIFIABLE = false

These new return codes can be returned by this command:

> CKR_ASSIGNED_KEY_REQUIRES_AUTH_DATA

> CKR_ROLE_CANNOT_MAKE_KEYS_ASSIGNED

> CKR_INVALID_ASSIGNED_ATTRIBUTE_TRANSITION

> CKR_ASSIGNED_KEY_FAILED_ATTRIBUTE_DEPENDENCIES

CA_IncrementFailedAuthCount

```
CA_IncrementFailedAuthCount(
          CK_SESSION_HANDLE hSession,       // the session's handle
          CK_OBJECT_HANDLE  hObject         // the object's handle
        )
```

This command increments the CKA_FAILED_KEY_AUTH_COUNT for a key.

It is intended to be used to keep members of an HA group in sync.

CKA_AUTH_DATA

This is a new (as of Luna HSM Firmware 7.7.0) sensitive attribute on each symmetric/private key object, and holds the hashed authorization data. It is initialized through the regular means of passing in attribute values in templates during key generation, derivation, or unwrapping.

It can be modified only through the use of CA_SetAuthorizationData() (section "CA_SetAuthorizationData" on the previous page), and CA_ResetAuthorizationData() (section "CA_ResetAuthorizationData" on the previous page), and not through the regular C_SetAttributeValue() API, as the the current value of the auth data must be provided at the time of the modification. Having previously authenticated for per-key auth is not enough.

This is akin to the case of a user password modification/reset. Even when the user is logged in, the current password must be re-specified at the time of the password modification, with the exception of the administrator being able to reset the password.

CKA_ASSIGNED

This is an attribute (as of Luna HSM Firmware 7.7.0) on each symmetric/private key object, and indicates whether the key is an assigned key or a general key. The default value is false. It can be modified from false to true, but not the other way around.

The following are the new error codes that can be returned by an attempt to modify this attribute through C_SetAttribute() command:

- CKR_ASSIGNED_KEY_REQUIRES_AUTH_DATA
- CKR_ROLE_CANNOT_MAKE_KEYS_ASSIGNED
- CKR_INVALID_ASSIGNED_ATTRIBUTE_TRANSITION
- CKR_ASSIGNED_KEY_FAILED_ATTRIBUTE_DEPENDENCIES

CKA_KEY_STATUS

This is an attribute (as of Luna HSM Firmware 7.7.0) on each symmetric/private key object, and holds the key lock status flags and the failed per-key auth count limit. It is present only along with CKA_AUTH_DATA, otherwise never there, and never allowed.

```
typedef struct OH_KEY_STATUS_S {
            UInt8   flags1;
            UInt8   failedAuthLimt;
            UInt8   reserved1;
            UInt8   reserved2;
         } OH_KEY_STATUS;
```

Flags:

- CK_KEY_STATUS_F_AUTH_DATA_SET
- CK_KEY_STATUS_F_LOCKED_DUE_TO_FAILED_AUTH
- CK_KEY_STATUS_F_LOCKED_DUE_TO_DATE
- CK_KEY_STATUS_F_LOCKED_DUE_TO_DES3_BLOCK_COUNTER
- CK_KEY_STATUS_F_LOCKED_DUE_TO_USAGE_COUNTER

CKA_FAILED_KEY_AUTH_COUNT

This is an attribute (as of Luna HSM Firmware 7.7.0) on each symmetric/private key object, and holds the number for the failed per-key auth attempts. It will be present only along with CKA_AUTH_DATA, otherwise never there, and never allowed. The CO role can modify this attribute to lock/unlock the key through C_SetAttributeValue(). It is also reset upon auth data resetting by the CO through CA_ResetAuthorizationData().

Library/Tool Considerations

Tool Changes

LunaCM/LunaSH

LunaCM changes are highlighted below, LunaSH gets similar changes.

LunaCM, for Luna HSM Firmware 7.7.0 and newer, supports an optional flag on partition creation that can be specified to tell the HSM that the partition should be a V0 partition (through partition version) to support Luna use-case and customers with legacy applications using older libraries, or V1 Partition, supporting the eIDAS use-case (SKS, PKA, etc.).

Ckdemo

Ckdemo has added support for CKA_AUTH_DATA and CKA_ASSIGNED attributes in all required commands for:

1. Key generation/derivation/unwrap
2. Attribute setting/modification

New commands are added, to:

> Authorize keys
> Assign general keys
> Set/Reset per-key auth data for general keys
> Increment failed auth count

The following is accomplished through existing commands:

> Query key status (locked/active) – Display Key
> Unlock locked keys – Set Attribute (failed auth count)

High Availability (HA)

Mixed modes, such as the following, are not supported:

> Pre-FW7.7.0 and Luna HSM Firmware 7.7.0 partitions
> V0 and V1 partitions

HA Group Migration

HA group migration from pre-FW7.7.0 will be done in two steps:

1. Updating the group to Luna HSM Firmware 7.7.0 (starting with non-primary members), which results in all partitions being flagged as V0 Partitions.
2. If desired, converting these V0 partitions to V1 partitions that are PP 419-221.5 compliant

To update an HA group to Luna HSM Firmware 7.7.0, all the non-primary partitions must be updated to Luna HSM Firmware 7.7.0 *first*, to ensure that the key objects from the pre-FW7.7.0 primary can still move to the non-primaries through key cloning. Then the primary can be updated to Luna HSM Firmware 7.7.0. At the conclusion of this step, all of the existing partitions on the upgraded HSMs are flagged as V0 Partition.

In the optional next step to convert these partitions to PP 419-221.5 compliant, V1 partitions, once again, all the non-primary partitions must be converted first, and only then the primary partition can be converted.

Migration Scenarios for Per-Key Auth

All of these migration scenarios imply key objects without per-key auth data being imported into a Luna HSM Firmware 7.7.0 partition that enforces per-key authentication.

To that end, we have to consider all paths from legacy devices such as FW4, FW6, and FW7, as well as legacy partitions on the FW7 that are converted to non-legacy.

There are several migration methods to be considered that are all detailed in the following sections:

1. Cloning

2. Pre-firmware-7.7.0 SKS (a.k.a. SIM)
3. Unwrapping
4. V0 to V1 Partition Conversion

Import via Cloning

When a key object is being imported via cloning from a legacy HSM/partition, the flattened object attributes within the incoming blob will not have a CKA_AUTH_DATA attribute present. Instead, the f/w will initialize the CKA_AUTH_DATA of the imported object to the value from the access.

> In the Luna use-case where the per-key auth is not visible to the end user, this is all that is needed.

> In the eIDAS use-case, the import via cloning should be followed by an explicit call to CA_SetAuthorizationData() to reset the per-key auth of the imported object to the desired value. Please note that imported keys cannot be assigned until their per-key auth is set.

Import via Legacy SKS

When a key object is being imported via legacy SKS from a legacy HSM/partition, the flattened object attributes within the incoming blob do not have a CKA_AUTH_DATA attribute present. Instead, the firmware initializes the CKA_AUTH_DATA of the imported object to the value from the access.

> In the Luna use-case where the per-key auth is not visible to the end user, this is all that is needed.

> In the eIDAS use-case, the import via legacy SKS should be followed by an explicit call to CA_SetAuthorizationData() to reset the per-key auth of the imported object to the desired value. Please note that imported keys cannot be assigned until their per-key auth is set.

Import via Unwrapping

When a key object is being imported via unwrapping, the unwrap template will have a CKA_AUTH_DATA attribute present:

> In the eIDAS use-case, the users will set the CKA_AUTH_DATA in the template explicitly.

> In the Luna use-case, the library fills the CKA_AUTH_DATA in the template (please refer to section "Template Handling in the Cryptoki Library" on page 124).

Import via V0 to V1 Partition Conversion

Existing keys will not have auth-data. Consider partition conversion as import (from outside into an eIDAS compliant environment), allowing for the setting of initial per-key auth values by the Administrator.

When the PSO or the PCO flip the partition version bit to convert the V0 partition to V1, the access auth-data (which was set upon login) is applied to all objects. The setting of this per-key auth is tracked (in case it was interrupted by a power failure or otherwise h/w reset), and the partition is mostly unusable until all the objects in the partition have been successfully converted.

Only in the eIDAS use-case (unlikely, as V0 or pre-firmware 7.7.0 devices/partitions would not have been involved in an eIDAS scenario), the conversion of the partition should be followed by an explicit call to CA_SetAuthorizationData() to reset the per-key auth of each key to the desired value. Please note that the keys on the converted partition cannot be assigned until their per-key auth is set.

Summary of New PKA Commands and Capabilities

This following table lists all of the new commands, HSM policies and APIs that are added by Luna HSM Firmware 7.7.0. It also provides a cross-reference to the section of the document with more details for the new item.

New Item	Description	Cross-reference
partition create -version	LunaCM command enhancement	"LunaCM/LunaSH" on page 128
CKA_AUTH_DATA	New object attribute	"New Authorization Data Attribute" on page 122 "CKA_AUTH_DATA" on page 127
CKA_ASSIGNED	New object attribute	"CKA_ASSIGNED" on page 127
CKA_KEY_STATUS	New object attribute	"CKA_KEY_STATUS" on page 128
CKA_FAILED_KEY_AUTH_COUNT	New object attribute	"CKA_FAILED_KEY_AUTH_COUNT" on page 128
CA_AuthorizeKey(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hObject, CK_UTF8CHAR_PTR pAuthData, CK_ULONG ulAuthDataLen)	New Cryptoki API	"CA_AuthorizeKey" on page 126
CA_SetAuthorizationData(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hObject, CK_UTF8CHAR_PTR pOldAuthData, CK_ULONG ulOldAuthDataLen CK_UTF8CHAR_PTR pNewAuthData, CK_ULONG ulNewAuthDataLen)	New Cryptoki API	"CA_SetAuthorizationData" on page 126
CA_ResetAuthorizationData(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hObject, CK_UTF8CHAR_PTR pAuthData, CK_ULONG ulAuthDataLen)	New Cryptoki API	"CA_ResetAuthorizationData" on page 126

New Item	Description	Cross-reference
CA_AssignKey(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hObject)	New Cryptoki API	"CA_AssignKey" on page 127
CA_IncrementFailedAuthCOunt(CK_SESSION_HANDLE hSession, CK_OBJECT_HANDLE hObject)	New Cryptoki API	"CA_IncrementFailedAuthCount" on page 127
CKR_ASSIGNED_KEY_REQUIRES_AUTH_DATA	New return code	"CKA_ASSIGNED" on page 127
CKR_ROLE_CANNOT_MAKE_KEYS_ASSIGNED	New return code	"CKA_ASSIGNED" on page 127
CKR_INVALID_ASSIGNED_ATTRIBUTE_TRANSITION	New return code	"CKA_ASSIGNED" on page 127
CKR_ASSIGNED_KEY_FAILED_ATTRIBUTE_DEPENDENCIES	New return code	"CKA_ASSIGNED" on page 127
CKR_AUTH_DATA_TOO_LARGE	New return code	"CA_SetAuthorizationData" on page 126 "CA_ResetAuthorizationData" on page 126
CKR_AUTH_DATA_TOO_SMALL	New return code	"CA_SetAuthorizationData" on page 126 "CA_ResetAuthorizationData" on page 126
CK_KEY_STATUS_F_AUTH_DATA_SET	New flag	"CKA_KEY_STATUS" on page 128
CK_KEY_STATUS_F_LOCKED_DUE_TO_FAILED_AUTH	New flag	"CKA_KEY_STATUS" on page 128
CK_KEY_STATUS_F_LOCKED_DUE_TO_DATE	New flag	"CKA_KEY_STATUS" on page 128
CK_KEY_STATUS_F_LOCKED_DUE_TO_DES3_BLOCK_COUNTER	New flag	"CKA_KEY_STATUS" on page 128
CK_KEY_STATUS_F_LOCKED_DUE_TO_USAGE_COUNTER	New flag	"CKA_KEY_STATUS" on page 128

V0 PARTITIONS

Version zero (V0) partitions are designed to support older clients. They can be created in two ways: through firmware update (all existing partitions will be marked as legacy) and directly during partition creation. They will be marked with a new policy: “Partition Version”, which will be set to 0. The regular, non-legacy partitions will be version 1.

Firmware Update

1. During firmware update from a pre-7.7.0 version, existing partitions are always converted to V0 partitions
2. Zeroizes old STC keys as Luna HSM Firmware 7.7.0 (and newer) will not support it at all.
3. Partition v0 to v1 transition is non-destructive (both across firmware update and through normal toggling):

Existing keys will not have auth-data. The first time the CO logs in, the access auth-data is set and can be applied to all objects. A partition on an HSM updated to firmware version 7.7.0 (or newer) becomes usable when a one-time CO login is performed.

The firmware update process, in this case, is considered equivalent to “import” because the existing keys are being imported in to the ecosystem of the new firmware.

For purposes of STC: to avoid breaking the clients’ access to the partition, old STC must be disabled before firmware update, then, post-update a partition updated to V0 allows new STC setup. So the appropriate order would be to update the client first, then setup new STC, then set the partition version to V1 if desired.

Neither V0 nor V1 partitions will support old STC

Partition Creation

1. An optional flag can be specified to tell the HSM that the partition should be Partition Version v0 (supported by LUSH and LunaCM)
   Normally, in the case of legacy client support, we’re talking about appliance, and partitions are created through Lush, which will be updated anyway.
2. Old clients (or old client apps + new client library) are unable to provide this, so partitions created with the old client will be v1.
   As mentioned above, this should only apply to PCI case, and we don’t have to support legacy client installations there.
3. If a partition is pre-created, and it is later determined that a partition version v0 is required, HSM SO has to delete the partition and re-create a partition v0
4. Identified by “Partition Version policy”.
   a. Partition Version 0 == new default or pre-existing upgraded to Luna HSM Firmware 7.7.0 (or newer), Partition Version 1 == eIDAS (SKS and PKA)
   b. Policy is always destructive for V1 to V0 transition (eIDAS use-case to Luna use-case), and cannot be changed by customer via PPT (Partition Policy Template)
   c. V0 to V1 (Luna use-case to eIDAS use-case), non-destructive by default, can be set to be destructive.

5. Other default partition policy settings that are forced on the legacy partition (either directly upon creation or through firmware update)
   a. SKS capability/policy = 1/0 (even in the case of insertion, since pre-firmware-7.7.0 does not have any SKS support)
   b. Cloning capability/policy = 1/1
   c. Per-key-auth capability/policy = 1/0
   d. If CKE (cloning-key-export), then remains as CKE with same SIM/Cloning settings as above.
6. For V0 partition:
   a. Does not allow partition policy changes that would require new clients
      i. Cannot turn SKS On (both masking and unmasking private/secret keys)
      ii. Cannot turn Per-key-auth On
   b. Allows user objects to be cloned off using CPV3
   c. HA Login
      i. v1.1 is supported on Secondary HSM
      ii. v1.0 is not supported on Secondary HSM
      iii. v1.0, v1.1 and v2.0 are all always supported on Primary HSMs
7. For V1 partition:
   a. Per-key-auth is turned On by default, but can be turned Off for performance
   b. Does not allow user objects to be cloned off
   c. HA Login
      i. v1.1 is not supported on Secondary HSM
      ii. v1.0 is not supported on Secondary HSM
      iii. Only v2.0 is supported on Secondary HSM
      iv. v1.0, v1.1 and v2.0 are all always supported on Primary HSMs

Converting from V0 to V1 (changing the policy)

1. Non-destructive to the partition by default (can be changed)
2. Leaves Cloning On (but V1 does not allow cloning out, only in for migration)
3. Turn SKS On
4. Turn Per-key-auth On

Converting from V1 to V0 (changing the policy)

1. Destructive to the partition (and can’t be changed)
   a. Zeroize everything except for STC keys: SMKs, etc.
2. Turn SKS Off
3. Turn Per-key-auth Off

G5 Backup HSM and Cloning Protocol

1. V0 and V1 partitions are not distinguishable, they all have CPV3 and same migration rules.
2. When cloning out of CPV3 HSM, new things like DES3 usage counters are included
   i. FW7.7.0 brings updated features for FIPS compliance, so adjustment is needed to allow their backup -- FW7.7.0 update for G5 Backup HSM allows those counters to be stored in its header
3. Support for creating backup partitions with old client during **partition archive** command
   a. Backup partitions default to a “set on first use” value for CPVx and FM-enabled
   b. Then when used, (for example, CloneAsTargetInit) the CPVx (cloning protocol version) and FM-enabled values are set
   c. All backup partitions are created in this way (as “set on first use”). Partition attributes are visible in LunaCM.
4. Some new things like new key types (ed25519 etc) cannot be added to Luna Backup HSM G5. That still requires moving to blob-based-backup; consider Luna Backup HSM 7.

eIDAS partitions

1. New partitions can be V1 by selecting the option at partition creation time.
2. SKS is ON
3. Per-key-auth is ON
4. No key cloning (outbound)
   a. When high level CA_CloneObject() call is issued, the client library chooses cloning or SKS based on what is available at the source and target HSMs,
5. For full backward compatibility for old client applications that use the step-by-step cloning APIs, the solution is to employ cloning CPV3 APIs in the firmware to achieve key transport over SKS blobs:
   a. CloneAsTargetInit always performs the regular TargetInit operation, because it does not know
      i. if the source is V0 or not, or
      ii. if the requested operation is a command to clone the SMK, or
      iii. if the requested operation is will be an SKS extract/insert that needs to pose as cloning
      iv. this adds overhead for one part of the cloning exchange, an RSA operation, which can slow applications that don’t use new SKS APIs.
   b. CloneAsTarget, from V1 to V1
      i. This can receive SKS blob (for key objects) or cloning blob (only for SMK)
      ii. Both blobs start with SIM: [length|SIM-MECH|SMK ID], cloning: [length|legacy bit + cloning version|zSizeField]
      iii. New mechanism values are used to prevent collision with cloning version

Limited Crypto Officer (LCO) role

As the name suggests, this role is between the CO and CU. It is a subset of the CO role – that is, everything LCO can do, CO can do as well but not the other way around. The relationship between LCO and CU is somewhat more complicated. For the most part, LCO is a superset of CU; however, there are some nuances with cloning – LCO does not support cloning in any way, shape, or form while CU can clone public objects. This is why

- the CO role is common both for the traditional use-cases of the Luna HSM and for the eIDAS use-case
- the CU role is retained for traditional Luna use-cases without a real place in the eIDAS use-case (which uses SKS in place of cloning)
- the LCO role is suited to the eIDAS use-case

Below are the details:

1. LCO is created by CO.
2. LCO supports password authentication and multifactor quorum authentication. For multifactor quorum authentication:
   a. A new PED key can be separate from CO and CU keys;
   b. MofN is supported.
3. LCO is supported by HA_Login for primary and secondary nodes.
4. CO can reset LCO's primary credentials (password or PED key), for example lunacm **role resetpw**
   a. This can be done regardless of the Enable SO reset of a partition PIN HSM policy 15 (contrast this to PSO resetting CO's primary credentials).
5. CO can create LCO's challenge (multifactor quorum authentication)
   a. LCO is subject to activation / auto-activation (partition policies 22 and 23);
   b. LCO can be deactivated (e.g., lunacm **role deactivate**) by any role or even w/o a session.
6. CO can reset LCO's challenge (multifactor quorum authentication) e.g. lunacm **role createChallenge**
   a. This can be done regardless of the Enable SO reset of a partition PIN HSM policy 15 (contrast this to PSO resetting CO's challenge).
7. LCO, and only LCO, can change its own primary and/or secondary credentials (password or PED key and/or challenge secret); for example **role changepw**.
8. LCO is subject to the Enable forcing user PIN change HSM policy 21 with respect to either primary or secondary credentials (password or PED key and/or challenge secret).
9. LCO is subject to failed logins logic:
   a. The Max failed user logins allowed partition policy 20;
   b. Upon reaching the limit, LCO locks out; CO and CU remain operational;
   c. The Partition CO can unlock the Partition Limited Crypto Officer role by resetting its credentials;
   d. The Max failed challenge responses >partition policy 15;
   e. HA_Login is governed by the same logic (for primary credentials; that is, challenge excluding).
10. LCO is subject to the following partition policies:
    a. Minimum PIN length (25),

   b. Maximum PIN length (26).

11. Pre-firmware 7.7.0 partitions or V0 partitions:
   a. Luna HSM Client 10.2.0 or older: LCO role is not exposed (not visible through any tools).
   b. Luna HSM Client 10.3.0 or newer: LCO role is visible but any login attempt will always fail.
12. LCO can generate keys (assigned or unassigned)
   a. LCO cannot assign keys (but can generate them assigned).
13. LCO can delete keys:
   a. Unlike CO, LCO has to provide per-key authorization data,
   b. LCO needs to be able to delete keys to support the “single-use signing keys” scenario where a user generates a key, signs with that key, and deletes the key.
14. LCO can create and destroy private objects.
15. LCO can copy keys
   a. Like CO, LCO has to provide per-key authorization data
16. LCO can copy private objects
17. LCO can modify keys
   a. Like CO, LCO has to provide per-key authorization data for unassigned keys.
18. LCO can modify private objects.
19. LCO has the same rules with respect to Usage Counters as CU
   a. Can increment the counter but, unlike CO, cannot change/set the limit.
20. LCO can generate domain parameters
21. LCO can authorize PKA keys
   a. Same as CO, CU, SO, and AD.
22. LCO can set a new per-key authorization data (the old one must be provided)
   a. Same as CO, CU, SO, and AD.
23. Unlike SO and CO, LCO cannot reset (i.e. w/o providing the old auth data) the per-key authorization data
   a. SO and CO can do it for unassigned keys. Nobody can for assigned.
24. Unlike SO and CO, LCO cannot unblock blocked (due to per-key auth failures) PKA keys.
25. LCO can wrap/unwrap:
   a. PKA behaviour for wrap: has to provide the per-key auth data for both the wrapping and the wrapped keys.
   b. PKA behaviour for unwrap: has to provide the per-key auth data for unwrapping key and specify the per-key auth data for the unwrapped key in the template.
   c. For the unwrapped key, if auth data is not in the template, the library will insert the access level auth data.
26. LCO can derive keys:
   a. PKA behaviour: has to provide the per-key auth data for the key used for derivation and specify the per-key auth data for the key being derived in the template.
   b. For the derived key, if auth data is not in the template, the library will insert the access level auth data.

27. LCO can derive-and-wrap
    a. PKA behaviour: as per the wrap/unwrap and derive points above.
28. LCO can SIM extract/insert in all scenarios:
    a. Including SKS key migration (old SKS: SIM Insert; no SIM Extract),
    b. Including new SKS (SIM Extract and SIM Insert).
29. LCO cannot clone/replicate in any scenario:
    a. CPV1 user key migration – no.
    b. CPV3 user key migration from legacy partitions – no.
    c. SMK (SKS secret) cloning (including G5 backup/restore) – no.
        i. Among other things, this means that LCO is not self-sufficient to perform HA – CO is required to clone SMK(s).
    d. Note that CU can do some limited cloning but LCO cannot.
30. Unlike CO, LCO cannot perform SMK rollover.

 Copyright 2001-2024 Thales Group

CHAPTER 5: Supported Mechanisms

This chapter provides information about the supported PKCS#11 standard mechanisms and the Thales-proprietary mechanisms supported in all released versions of the Luna HSM firmware. Refer to the page that applies to your installed firmware version.

The individual mechanism pages provide additional information about using each mechanism. You can also compare the attributes/requirements of each mechanism across different firmware versions.

> Firmware 7.8.7 Mechanisms
> Firmware 7.8.4 Mechanisms
> Firmware 7.8.1 Mechanisms
> Firmware 7.8.0 Mechanisms
> Firmware 7.7.2 Mechanisms
> Firmware 7.7.1-20 Mechanisms
> Firmware 7.7.0 Mechanisms
> Firmware 7.4.0 Mechanisms
> Firmware 7.3.3 Mechanisms
> Firmware 7.3.0 Mechanisms
> Firmware 7.2.0 Mechanisms
> Firmware 7.1.0 Mechanisms
> Firmware 7.0.3 Mechanisms
> Firmware 7.0.2 Mechanisms
> Firmware 7.0.1 Mechanisms

If you are using RSA keys in FIPS mode, refer to "RSA Mechanism Remap for FIPS Compliance" below for information about remapping your mechanisms to maintain FIPS compliance. This applies to Luna HSM Client 7.4.0 and older.

> **NOTE** Starting with Luna HSM Firmware 7.7.0, the tables linked above include a column that specifies any operations that are restricted when the HSM is in FIPS mode.

RSA Mechanism Remap for FIPS Compliance

Under FIPS 186-3/4, the only RSA methods permitted for generating keys are 186-3 with primes and 186-3 with aux primes. This means that RSA PKCS and X9.31 key generation is no longer approved for operation in a FIPS-compliant HSM.

Supported Mechanisms	FIPS-mode Allowed Mechanisms
PKCS, X9.31, 186-3 with primes, 186-3 with aux primes	186-3 with primes, 186-3 with aux primes

Luna HSM Client allows you to automatically remap calls to these old, less-secure mechanisms, to new mechanisms that are FIPS-approved. This remapping can allow you to operate the HSM securely without having to rewrite your applications. With this feature enabled, the following remapping is applied:

> Calls for PKCS key generation using "CKM_RSA_PKCS_KEY_PAIR_GEN" on page 384 are remapped to "CKM_RSA_FIPS_186_3_PRIME_KEY_PAIR_GEN" on page 379, which uses 186-3 Prime key generation.

> Calls for X9.31 key generation using "CKM_RSA_X9_31_KEY_PAIR_GEN" on page 391 are remapped to "CKM_RSA_FIPS_186_3_AUX_PRIME_KEY_PAIR_GEN" on page 378, which uses 186-3 Aux Prime key generation

Effects of Remapping in FIPS Mode

When the Luna HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**) or the application partition is in FIPS mode (**partition policy 43:** Allow non-FIPS algorithms set to **0**) and RSA remapping is enabled:

> "CKM_RSA_PKCS_KEY_PAIR_GEN" on page 384 appears in the **C_GetMechanismList** output.

> **C_GetMechanismInfo** for "CKM_RSA_PKCS_KEY_PAIR_GEN" on page 384 returns the default information from the client library.

> "CKM_RSA_X9_31_KEY_PAIR_GEN" on page 391 appears in the **C_GetMechanismList** output.

> **C_GetMechanismInfo** for "CKM_RSA_X9_31_KEY_PAIR_GEN" on page 391 returns the default information from the client library.

Applying the Mechanism Remapping

Mechanism remapping has been enabled automatically in recent versions of the Luna HSM Client. Refer to the following table for older version requirements.

Luna HSM Firmware	Luna HSM Client	To apply RSA mechanism remapping
Luna HSM Firmware 7.7.1-20 or newer	Luna HSM Client 10.4.0 or newer	If you have FIPS mode set on individual partitions (**partition policy 43:** Allow non-FIPS algorithms set to **0**), remapping is automatic; the **RSAKeyGenMechRemap** configuration setting is ignored.
	Luna HSM Client 10.1.0 or newer	If you have FIPS mode set on the entire HSM (**HSM policy 12:** Allow non-FIPS algorithms set to **0**), remapping is automatic; the **RSAKeyGenMechRemap** configuration setting is ignored. **NOTE** Remapping on individual partitions requires Luna HSM Client 10.4.0 or newer.

Luna HSM Firmware	Luna HSM Client	To apply RSA mechanism remapping
Luna HSM Firmware 7.7.0 or older	Luna HSM Client 10.1.0 or newer	Remapping is automatic; the **RSAKeyGenMechRemap** configuration setting is ignored.

	Luna HSM Client 7.4.0 or older	Remapping must be enabled using the **RSAKeyGenMechRemap** setting in the Luna HSM Client configuration file (see Configuration File Summary):
		> **Linux:**
		`Misc = {`
		`  RSAKeyGenMechRemap=1;`
		`}`
		> **Windows:**
		`[Misc]`
		`RSAKeyGenMechRemap=1`

CKM_AES_CBC

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	CBC
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to wrap objects.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128

 Copyright 2001-2024 Thales Group

Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	CBC
Flags	Extractable

> **TIP** The CKM_AES_CBC mechanism can wrap/unwrap other *symmetric* keys.
> Wrap/unwrap of *asymmetric* keys (RSA, ECC, etc.) is supported only by mechanisms that can process any byte length - thus CBC_PAD, GCM, or KWP. However, of those three, only KWP is permitted by FIPS mode.

 Copyright 2001-2024 Thales Group

CKM_AES_CBC_ENCRYPT_DATA

Summary

FIPS approved?	Yes
Supported functions	Derive
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	0
Digest size	0
Key types	AES
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_AES_CBC_PAD

Usage Notes

Although this convention allows control of the padding on the input buffer it is highly recommended to use CKM_AES_KWP instead.

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	CBC_PAD
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to wrap objects.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None

 Copyright 2001-2024 Thales Group

Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_AES_CBC_PAD_IPSEC

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	CBC_PAD_IPSEC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_AES_CFB8

Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	1
Key types	AES
Algorithms	AES
Modes	CFB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_AES_CFB128

Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	16
Key types	AES
Algorithms	AES
Modes	CFB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_AES_CMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	MAC
Flags	Extractable | CMAC

 Copyright 2001-2024 Thales Group

CKM_AES_CMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	MAC
Flags	Extractable | CMAC

 Copyright 2001-2024 Thales Group

CKM_AES_CTR

> **NOTE** Luna HSM Firmware 7.7.2 and newer improves support of 3GPP TS 33.501 with ICB derivation. Compressed ephemeral public keys are accepted as input to the decrypt operation.

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	CTR
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to wrap objects.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None

Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	CTR
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_AES_ECB

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	ECB
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to wrap objects.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128

Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_AES_ECB_ENCRYPT_DATA

Summary

FIPS approved?	Yes
Supported functions	Derive
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	0
Digest size	0
Key types	AES
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_AES_GCM

GCM is the Galois/Counter Mode of operation of the AES algorithm for symmetric key encryption.

Usage Notes

Initialization Vector

If the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**), the initialization vector (IV) is generated in the HSM and returned to the PKCS#11 function call. The buffer must be large enough to store the GMAC tag plus the generated IV (which has a length of 16 bytes). When both encrypting/decrypting on a Luna HSM, do not provide the tag bits themselves, just the AAD and IV (for decryption only).

Using the GCM method, encrypted data is the same size as the clear data. The IV generated by the HSM is concatenated at the end -- to determine the IV, find the last 16 bytes of the encrypted data.

If the HSM is **not** in FIPS mode, you must specify an IV. Random IV is supported and recommended for GCM and GMAC. If you are not using random IV, then the most efficient IV value length is 12 bytes; any other size reduces performance and requires more work (per NIST SP-800-38D).

The internal IV is a randomly generated 16-byte IV.

Performance

For authentication, it is possible to use CKM_AES_GCM mechanism, when passing no data to encrypt (for strict compliance with NIST specification), and performance in that mode is better than in previous Luna releases.

The correlation is not exact but, as a general rule for a given mechanism, invocation by PKCS#11 API always provides the best performance, JSP performance is close but lower due to Java architecture, and JCPROV performance is somewhat lower still than PKCS#11 and JSP performance levels.

JCPROV

AES-GMAC and AES-GCM are supported in JCPROV. Use CK_AES_GCM_PARAMS.java to define the GMAC operation. Implementation is the same as for PKCS#11.

Java Provider (JSP)

Both GMC and GMAC are supported. "GmacAesDemo.java" provides a sample for using GMAC with Java.

Java Parameter Specification class LunaGmacParameterSpec.java defines default values recommended by the NIST specification.

Accumulating Data

Our GMAC and GCM are single part operations, so even if they are called using multi-part API, we accumulate the data (up to a maximum) and return data only on the “final” operation. That is the meaning of "Accumulating" in the table, below.

Firmware 7.7.2 and Newer Summary

FIPS approved?	Yes

Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	GCM
Flags	Extractable | Accumulating

Firmware 7.7.1 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES

 Copyright 2001-2024 Thales Group

Algorithms	AES
Modes	GCM
Flags	Extractable | Accumulating

 Copyright 2001-2024 Thales Group

CKM_AES_GMAC

GCM is the Galois/Counter Mode of operation of the AES algorithm for symmetric key encryption.

GMAC is a variant of GCM for sign/verify operation. If GCM input is confined to data that will not be encrypted, then GMAC is purely an authentication mode on the input data. The Luna PCIe HSM 7 GMAC implementation, formerly invoked only via PKCS#11 interface, can now be accessed via JCPROV and via our Java Provider.

The GMAC implementation follows NIST SP-800-38D. It supports AES symmetric key sizes of 128, 192, and 256 bits. The output is `[ ciphertext | tag | IV ]`.

Usage Notes

Initialization Vector

If the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**), the initialization vector (IV) is generated in the HSM and returned to the PKCS#11 function call. The buffer must be large enough to store the GMAC tag plus the generated IV (which has a length of 16 bytes). When both encrypting/decrypting on a Luna HSM, do not provide the tag bits themselves, just the AAD and IV (for decryption only).

Using the GCM method, encrypted data is the same size as the clear data. The IV generated by the HSM is concatenated at the end -- to determine the IV, find the last 16 bytes of the encrypted data.

If the HSM is **not** in FIPS mode, you must specify an IV. Random IV is supported and recommended for GCM and GMAC. If you are not using random IV, then the most efficient IV value length is 12 bytes; any other size reduces performance and requires more work (per NIST SP-800-38D).

The internal IV is a randomly generated 16-byte IV.

Performance

For authentication, it is possible to use CKM_AES_GCM mechanism, when passing no data to encrypt (for strict compliance with NIST specification), and performance in that mode is better than in previous Luna releases.

The correlation is not exact but, as a general rule for a given mechanism, invocation by PKCS#11 API always provides the best performance, JSP performance is close but lower due to Java architecture, and JCPROV performance is somewhat lower still than PKCS#11 and JSP performance levels.

JCPROV

AES-GMAC and AES-GCM are supported in JCPROV. Use CK_AES_GCM_PARAMS.java to define the GMAC operation. Implementation is the same as for PKCS#11.

Java Provider (JSP)

Both GMC and GMAC are supported. "GmacAesDemo.java" provides a sample for using GMAC with Java.

Java Parameter Specification class LunaGmacParameterSpec.java defines default values recommended by the NIST specification.

Accumulating Data

Our GMAC and GCM are single part operations, so even if they are called using multi-part API, we accumulate the data (up to a maximum) and return data only on the “final” operation. That is the meaning of "Accumulating" in the table, below.

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	GCM
Flags	Extractable | Accumulating

 Copyright 2001-2024 Thales Group

CKM_AES_KEY_GEN

Summary

FIPS approved?	Yes
Supported functions	Generate Key
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	0
Digest size	0
Key types	AES
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_AES_KW

NIST Special Publication 800-38F describes cryptographic methods that are approved for “key wrapping,” that is, the protection of the confidentiality and integrity of cryptographic keys. In addition to describing existing methods, that publication specifies two new, deterministic authenticated-encryption modes of operation of the Advanced Encryption Standard (AES) algorithm: the AES Key Wrap (KW) mode and the AES Key Wrap With Padding (KWP) mode. Luna PCIe HSM 7 implements the AES Key Wrap (KW) mode at this time, which SP800-38F recommends as more secure than CKM_AES_CBC. This mechanism meets the requirements specified in RFC 3394 for key wrapping.

Data size

The maximum allowed data size for this mechanism is 64KB (64 * 1024).

NOTE

> NIST Special Publication 800-38F recommends this method as more secure than CKM_AES_CBC. This mechanism meets the requirements specified in RFC 3394 for key wrapping.

> For key generation or unwrap operations, the length of the new/unwrapped key must be specified with CKA_VALUE_LEN.

> This mechanism can be used to wrap/unwrap symmetric keys only. To wrap/unwrap private/asymmetric keys, use a mechanism that can wrap data of any length, such as "CKM_AES_CBC_PAD" on page 145, "CKM_AES_GCM" on page 157, or "CKM_AES_KWP" on page 165.

Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	8
Digest size	0

Key types	AES
Algorithms	AES
Modes	KEYWRAP
Flags	Extractable | Accumulating

 Copyright 2001-2024 Thales Group

CKM_AES_KWP

RFC 5649 specifies a padding convention for use with the AES Key Wrap algorithm specified in RFC 3394. This convention eliminates the requirement that the length of the key is to be wrapped by a multiple of 64 bits, allowing a key of any practical length to be wrapped.

This convention should always be used instead of CKM_AES_CBC when wrapping a key from the HSM.

Data size

The maximum allowed data size for this mechanism is 64KB (64 * 1024).

Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	8
Digest size	0
Key types	AES
Algorithms	AES
Modes	KEYWRAP_PAD
Flags	Extractable | Accumulating

 Copyright 2001-2024 Thales Group

CKM_AES_MAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.7 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	MAC
Flags	Extractable

Firmware 7.7.0-7.8.4 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign

Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	MAC
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to sign data.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES

Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_AES_MAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.7 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	MAC
Flags	Extractable

Firmware 7.7.0-7.8.4 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign

Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	MAC
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to sign data.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES

Modes	MAC
Flags	Extractable

CKM_AES_OFB

Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	OFB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_AES_XTS

Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	XTS
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_CBC

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	CBC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_CBC_ENCRYPT_DATA

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	0
Digest size	0
Key types	ARIA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_ARIA_CBC_PAD

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_CFB8

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	1
Key types	ARIA
Algorithms	ARIA
Modes	CFB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_CFB128

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	16
Key types	ARIA
Algorithms	ARIA
Modes	CFB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_CMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	MAC
Flags	Extractable | CMAC

 Copyright 2001-2024 Thales Group

CKM_ARIA_CMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	MAC
Flags	Extractable | CMAC

 Copyright 2001-2024 Thales Group

CKM_ARIA_CTR

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	CTR
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_ECB

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_ECB_ENCRYPT_DATA

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	0
Digest size	0
Key types	ARIA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_ARIA_KEY_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	0
Digest size	0
Key types	ARIA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_ARIA_L_CBC

Summary

FIPS approved?	No
Supported functions	Decrypt | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	CBC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_L_CBC_PAD

Summary

FIPS approved?	No
Supported functions	Decrypt | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_L_ECB

Summary

FIPS approved?	No
Supported functions	Decrypt | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_L_MAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_L_MAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_MAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_MAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ARIA_OFB

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	ARIA
Algorithms	ARIA
Modes	OFB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_BIP32_CHILD_DERIVE

This mechanism is used to derive child keys from a parent key, and can generate both the private and public part of the key pair, accepting a BIP32 public or private key as input.

Cloning (or backup) of BIP32 keys can be performed only between Luna HSMs containing firmware versions that support BIP32.

See "BIP32 Mechanism Support and Implementation" on page 91.

Firmware 7.3.0 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	0
Digest size	0
Key types	BIP32
Algorithms	None
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_BIP32_MASTER_DERIVE

This mechanism is used to derive the master key pair from a seed. The input key must have the type `CKK_GENERIC_SECRET` (size between 128 and 512 bits).

Only curve `secp256k1` is supported. Key type `CKK_BIP32` is introduced; existing ECDSA keys cannot be used with the BIP32 mechanisms. All mechanisms supported by ECDSA keys are supported for BIP32 keys.

Cloning (or backup) of BIP32 keys can be performed only between Luna HSMs containing firmware versions that support BIP32.

See "BIP32 Mechanism Support and Implementation" on page 91.

Firmware 7.3.0 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	512
Block size	0
Digest size	0
Key types	GENERIC_SECRET
Algorithms	None
Modes	None
Flags	Extractable

CKM_CAST3_CBC

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	CAST3
Algorithms	CAST3
Modes	CBC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_CAST3_CBC_PAD

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	CAST3
Algorithms	CAST3
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_CAST3_ECB

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	CAST3
Algorithms	CAST3
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_CAST3_KEY_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	0
Digest size	0
Key types	CAST3
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_CAST3_MAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	CAST3
Algorithms	CAST3
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_CAST3_MAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	CAST3
Algorithms	CAST3
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_CAST5_CBC

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	8
Digest size	0
Key types	CAST5
Algorithms	CAST5
Modes	CBC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_CAST5_CBC_PAD

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	8
Digest size	0
Key types	CAST5
Algorithms	CAST5
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_CAST5_ECB

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	8
Digest size	0
Key types	CAST5
Algorithms	CAST5
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_CAST5_KEY_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	0
Digest size	0
Key types	CAST5
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_CAST5_MAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	8
Digest size	0
Key types	CAST5
Algorithms	CAST5
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_CAST5_MAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	8
Digest size	0
Key types	CAST5
Algorithms	CAST5
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_COMP128

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	ECB
Flags	Allow zero-length input

 Copyright 2001-2024 Thales Group

CKM_DES_CBC

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	DES
Algorithms	DES
Modes	CBC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES_CBC_ENCRYPT_DATA

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DES_CBC_PAD

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	DES
Algorithms	DES
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES_CFB8

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	Cannot encrypt
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	1
Key types	DES3
Algorithms	DES3
Modes	CFB
Flags	Extractable

> **NOTE** In this firmware version, "Functions restricted from FIPS use" has changed for this mechanism, to comply with FIPS 140-3 requirements.

Firmware 7.8.1 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

 Copyright 2001-2024 Thales Group

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	1
Key types	DES3
Algorithms	DES3
Modes	CFB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES_CFB64

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	Cannot encrypt
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	8
Key types	DES3
Algorithms	DES3
Modes	CFB
Flags	Extractable

> **NOTE** In this firmware version, "Functions restricted from FIPS use" has changed for this mechanism, to comply with FIPS 140-3 requirements.

Firmware 7.8.1 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

 Copyright 2001-2024 Thales Group

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	8
Key types	DES3
Algorithms	DES3
Modes	CFB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES_ECB

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	DES
Algorithms	DES
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES_ECB_ENCRYPT_DATA

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DES_KEY_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	0
Digest size	0
Key types	DES
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DES_MAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	DES
Algorithms	DES
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES_MAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	8
Digest size	0
Key types	DES
Algorithms	DES
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES_OFB64

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	Cannot encrypt
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	OFB
Flags	Extractable

> **NOTE** In this firmware version, "Functions restricted from FIPS use" has changed for this mechanism, to comply with FIPS 140-3 requirements.

Firmware 7.8.1 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

 Copyright 2001-2024 Thales Group

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	OFB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES2_DUKPT_DATA

The CKM_DES2_DUKPT family of key derive mechanisms create keys used to protect EFTPOS terminal sessions. The mechanisms implement the algorithm for server side DUKPT derivation as defined by ANSI X9.24 part 1.

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

Usage

This mechanism has the following attributes:

- Only CKK_DES2 keys can be derived. The mechanism will force the CKA_KEY_TYPE attribute of the derived object to equal CKK_DES2. If the template does specify a CKA_KEY_TYPE attribute then it must be CKK_DES2.
- The mechanism takes a CK_KEY_DERIVATION_STRING_DATA structure as a parameter.
- The pData field of the parameter must point to a 10 byte array which holds the 80 bit Key Sequence Number (KSN).
- This mechanism contributes the CKA_CLASS and CKA_KEY_TYPE and CKA_VALUE to the resulting object.

 Copyright 2001-2024 Thales Group

The DUKPT MAC and DATA versions will default to the appropriate usage mechanism as described in the following table:

Mechanism	CKA_SIGN	CKA_VERIFY	CKA_DECRYPT	CKA_ENCRYPT
CKM_DES2_DUKPT_MAC	True	True		
CKM_DES2_DUKPT_MAC_RESP	True			
CKM_DES2_DUKPT_DATA			True	True
CKM_DES2_DUKPT_DATA_RESP				True

Example

```
#define CKM_DES2_DUKPT_PIN              (CKM_VENDOR_DEFINED + 0x611)
#define CKM_DES2_DUKPT_MAC             (CKM_VENDOR_DEFINED + 0x612)
#define CKM_DES2_DUKPT_MAC_RESP        (CKM_VENDOR_DEFINED + 0x613)
#define CKM_DES2_DUKPT_DATA            (CKM_VENDOR_DEFINED + 0x614)
#define CKM_DES2_DUKPT_DATA_RESP       (CKM_VENDOR_DEFINED + 0x615)

CK_OBJECT_HANDLE hBDKey; // handle of CKK_DES2 or CKK_DES2 Base Derive Key
CK_OBJECT_HANDLE hMKey;   // handle of CKK_DES2 MAC session Key
CK_MECHANISM svMech = { CKM_DES3_X919_MAC , NULL, 0};

CK_KEY_DERIVATION_STRING_DATA param;
CK_MECHANISM kdMech = { CKM_DES2_DUKPT_MAC , NULL, 0};
CK_CHAR ksn[10];
CK_CHAR inp[any length];
CK_CHAR mac[4];
CK_SIZE len;

// Derive MAC verify session key
param.pData=ksn;
param.ulLen = 10;

kdMech.mechanism = CKM_DES2_DUKPT_MAC;
kdMech.pParameter = &param;
kdMech.ulParameterLen = sizeof parram;

C_DeriveKey(hSes, &kdMech, hBDKey , NULL, 0, &hMKey);

// Single part verify operation

C_VerifyInit(hSes, &svMech, hMKey);
len = sizeof mac;
C_Verify(hSes, inp, sizeof inp, mac, len);

// clean up

C_DestroyObject(hSes, hMKey);

// Test vectors
```

CKM_DES2_DUKPT_DATA_RESP

The CKM_DES2_DUKPT family of key derive mechanisms create keys used to protect EFTPOS terminal sessions. The mechanisms implement the algorithm for server side DUKPT derivation as defined by ANSI X9.24 part 1.

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

Usage

This mechanism has the following attributes:

- Only CKK_DES2 keys can be derived. The mechanism will force the CKA_KEY_TYPE attribute of the derived object to equal CKK_DES2. If the template does specify a CKA_KEY_TYPE attribute then it must be CKK_DES2.
- The mechanism takes a CK_KEY_DERIVATION_STRING_DATA structure as a parameter.
- The pData field of the parameter must point to a 10 byte array which holds the 80 bit Key Sequence Number (KSN).
- This mechanism contributes the CKA_CLASS and CKA_KEY_TYPE and CKA_VALUE to the resulting object.

 Copyright 2001-2024 Thales Group

The DUKPT MAC and DATA versions will default to the appropriate usage mechanism as described in the following table:

Mechanism	CKA_SIGN	CKA_VERIFY	CKA_DECRYPT	CKA_ENCRYPT
CKM_DES2_DUKPT_MAC	True	True		
CKM_DES2_DUKPT_MAC_RESP	True			
CKM_DES2_DUKPT_DATA			True	True
CKM_DES2_DUKPT_DATA_RESP				True

Example

```
#define CKM_DES2_DUKPT_PIN              (CKM_VENDOR_DEFINED + 0x611)
#define CKM_DES2_DUKPT_MAC             (CKM_VENDOR_DEFINED + 0x612)
#define CKM_DES2_DUKPT_MAC_RESP        (CKM_VENDOR_DEFINED + 0x613)
#define CKM_DES2_DUKPT_DATA            (CKM_VENDOR_DEFINED + 0x614)
#define CKM_DES2_DUKPT_DATA_RESP       (CKM_VENDOR_DEFINED + 0x615)

CK_OBJECT_HANDLE hBDKey; // handle of CKK_DES2 or CKK_DES2 Base Derive Key
CK_OBJECT_HANDLE hMKey;   // handle of CKK_DES2 MAC session Key
CK_MECHANISM svMech = { CKM_DES3_X919_MAC , NULL, 0};

CK_KEY_DERIVATION_STRING_DATA param;
CK_MECHANISM kdMech = { CKM_DES2_DUKPT_MAC , NULL, 0};
CK_CHAR ksn[10];
CK_CHAR inp[any length];
CK_CHAR mac[4];
CK_SIZE len;

// Derive MAC verify session key
param.pData=ksn;
param.ulLen = 10;

kdMech.mechanism = CKM_DES2_DUKPT_MAC;
kdMech.pParameter = &param;
kdMech.ulParameterLen = sizeof parram;

C_DeriveKey(hSes, &kdMech, hBDKey , NULL, 0, &hMKey);

// Single part verify operation

C_VerifyInit(hSes, &svMech, hMKey);
len = sizeof mac;
C_Verify(hSes, inp, sizeof inp, mac, len);

// clean up

C_DestroyObject(hSes, hMKey);

// Test vectors
```

CKM_DES2_DUKPT_IPEK

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DES2_DUKPT_MAC

The CKM_DES2_DUKPT family of key derive mechanisms create keys used to protect EFTPOS terminal sessions. The mechanisms implement the algorithm for server side DUKPT derivation as defined by ANSI X9.24 part 1.

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

Usage

This mechanism has the following attributes:

- Only CKK_DES2 keys can be derived. The mechanism will force the CKA_KEY_TYPE attribute of the derived object to equal CKK_DES2. If the template does specify a CKA_KEY_TYPE attribute then it must be CKK_DES2.
- The mechanism takes a CK_KEY_DERIVATION_STRING_DATA structure as a parameter.
- The pData field of the parameter must point to a 10 byte array which holds the 80 bit Key Sequence Number (KSN).
- This mechanism contributes the CKA_CLASS and CKA_KEY_TYPE and CKA_VALUE to the resulting object.

 Copyright 2001-2024 Thales Group

The DUKPT MAC and DATA versions will default to the appropriate usage mechanism as described in the following table:

Mechanism	CKA_SIGN	CKA_VERIFY	CKA_DECRYPT	CKA_ENCRYPT
CKM_DES2_DUKPT_MAC	True	True		
CKM_DES2_DUKPT_MAC_RESP	True			
CKM_DES2_DUKPT_DATA			True	True
CKM_DES2_DUKPT_DATA_RESP				True

Example

```
#define CKM_DES2_DUKPT_PIN              (CKM_VENDOR_DEFINED + 0x611)
#define CKM_DES2_DUKPT_MAC             (CKM_VENDOR_DEFINED + 0x612)
#define CKM_DES2_DUKPT_MAC_RESP        (CKM_VENDOR_DEFINED + 0x613)
#define CKM_DES2_DUKPT_DATA            (CKM_VENDOR_DEFINED + 0x614)
#define CKM_DES2_DUKPT_DATA_RESP       (CKM_VENDOR_DEFINED + 0x615)

CK_OBJECT_HANDLE hBDKey; // handle of CKK_DES2 or CKK_DES2 Base Derive Key
CK_OBJECT_HANDLE hMKey;   // handle of CKK_DES2 MAC session Key
CK_MECHANISM svMech = { CKM_DES3_X919_MAC , NULL, 0};

CK_KEY_DERIVATION_STRING_DATA param;
CK_MECHANISM kdMech = { CKM_DES2_DUKPT_MAC , NULL, 0};
CK_CHAR ksn[10];
CK_CHAR inp[any length];
CK_CHAR mac[4];
CK_SIZE len;

// Derive MAC verify session key
param.pData=ksn;
param.ulLen = 10;

kdMech.mechanism = CKM_DES2_DUKPT_MAC;
kdMech.pParameter = &param;
kdMech.ulParameterLen = sizeof parram;

C_DeriveKey(hSes, &kdMech, hBDKey , NULL, 0, &hMKey);

// Single part verify operation

C_VerifyInit(hSes, &svMech, hMKey);
len = sizeof mac;
C_Verify(hSes, inp, sizeof inp, mac, len);

// clean up

C_DestroyObject(hSes, hMKey);

// Test vectors
```

CKM_DES2_DUKPT_MAC_RESP

The CKM_DES2_DUKPT family of key derive mechanisms create keys used to protect EFTPOS terminal sessions. The mechanisms implement the algorithm for server side DUKPT derivation as defined by ANSI X9.24 part 1.

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

Usage

This mechanism has the following attributes:

- Only CKK_DES2 keys can be derived. The mechanism will force the CKA_KEY_TYPE attribute of the derived object to equal CKK_DES2. If the template does specify a CKA_KEY_TYPE attribute then it must be CKK_DES2.
- The mechanism takes a CK_KEY_DERIVATION_STRING_DATA structure as a parameter.
- The pData field of the parameter must point to a 10 byte array which holds the 80 bit Key Sequence Number (KSN).
- This mechanism contributes the CKA_CLASS and CKA_KEY_TYPE and CKA_VALUE to the resulting object.

 Copyright 2001-2024 Thales Group

The DUKPT MAC and DATA versions will default to the appropriate usage mechanism as described in the following table:

Mechanism	CKA_SIGN	CKA_VERIFY	CKA_DECRYPT	CKA_ENCRYPT
CKM_DES2_DUKPT_MAC	True	True		
CKM_DES2_DUKPT_MAC_RESP	True			
CKM_DES2_DUKPT_DATA			True	True
CKM_DES2_DUKPT_DATA_RESP				True

Example

```
#define CKM_DES2_DUKPT_PIN              (CKM_VENDOR_DEFINED + 0x611)
#define CKM_DES2_DUKPT_MAC             (CKM_VENDOR_DEFINED + 0x612)
#define CKM_DES2_DUKPT_MAC_RESP        (CKM_VENDOR_DEFINED + 0x613)
#define CKM_DES2_DUKPT_DATA            (CKM_VENDOR_DEFINED + 0x614)
#define CKM_DES2_DUKPT_DATA_RESP       (CKM_VENDOR_DEFINED + 0x615)

CK_OBJECT_HANDLE hBDKey; // handle of CKK_DES2 or CKK_DES2 Base Derive Key
CK_OBJECT_HANDLE hMKey;   // handle of CKK_DES2 MAC session Key
CK_MECHANISM svMech = { CKM_DES3_X919_MAC , NULL, 0};

CK_KEY_DERIVATION_STRING_DATA param;
CK_MECHANISM kdMech = { CKM_DES2_DUKPT_MAC , NULL, 0};
CK_CHAR ksn[10];
CK_CHAR inp[any length];
CK_CHAR mac[4];
CK_SIZE len;

// Derive MAC verify session key
param.pData=ksn;
param.ulLen = 10;

kdMech.mechanism = CKM_DES2_DUKPT_MAC;
kdMech.pParameter = &param;
kdMech.ulParameterLen = sizeof parram;

C_DeriveKey(hSes, &kdMech, hBDKey , NULL, 0, &hMKey);

// Single part verify operation

C_VerifyInit(hSes, &svMech, hMKey);
len = sizeof mac;
C_Verify(hSes, inp, sizeof inp, mac, len);

// clean up

C_DestroyObject(hSes, hMKey);

// Test vectors
```

CKM_DES2_DUKPT_PIN

The CKM_DES2_DUKPT family of key derive mechanisms create keys used to protect EFTPOS terminal sessions. The mechanisms implement the algorithm for server side DUKPT derivation as defined by ANSI X9.24 part 1.

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

Usage

This mechanism has the following attributes:

- Only CKK_DES2 keys can be derived. The mechanism will force the CKA_KEY_TYPE attribute of the derived object to equal CKK_DES2. If the template does specify a CKA_KEY_TYPE attribute then it must be CKK_DES2.
- The mechanism takes a CK_KEY_DERIVATION_STRING_DATA structure as a parameter.
- The pData field of the parameter must point to a 10 byte array which holds the 80 bit Key Sequence Number (KSN).
- This mechanism contributes the CKA_CLASS and CKA_KEY_TYPE and CKA_VALUE to the resulting object.

 Copyright 2001-2024 Thales Group

The DUKPT MAC and DATA versions will default to the appropriate usage mechanism as described in the following table:

Mechanism	CKA_SIGN	CKA_VERIFY	CKA_DECRYPT	CKA_ENCRYPT
CKM_DES2_DUKPT_MAC	True	True		
CKM_DES2_DUKPT_MAC_RESP	True			
CKM_DES2_DUKPT_DATA			True	True
CKM_DES2_DUKPT_DATA_RESP				True

Example

```
#define CKM_DES2_DUKPT_PIN              (CKM_VENDOR_DEFINED + 0x611)
#define CKM_DES2_DUKPT_MAC             (CKM_VENDOR_DEFINED + 0x612)
#define CKM_DES2_DUKPT_MAC_RESP        (CKM_VENDOR_DEFINED + 0x613)
#define CKM_DES2_DUKPT_DATA            (CKM_VENDOR_DEFINED + 0x614)
#define CKM_DES2_DUKPT_DATA_RESP       (CKM_VENDOR_DEFINED + 0x615)

CK_OBJECT_HANDLE hBDKey; // handle of CKK_DES2 or CKK_DES2 Base Derive Key
CK_OBJECT_HANDLE hMKey;   // handle of CKK_DES2 MAC session Key
CK_MECHANISM svMech = { CKM_DES3_X919_MAC , NULL, 0};

CK_KEY_DERIVATION_STRING_DATA param;
CK_MECHANISM kdMech = { CKM_DES2_DUKPT_MAC , NULL, 0};
CK_CHAR ksn[10];
CK_CHAR inp[any length];
CK_CHAR mac[4];
CK_SIZE len;

// Derive MAC verify session key
param.pData=ksn;
param.ulLen = 10;

kdMech.mechanism = CKM_DES2_DUKPT_MAC;
kdMech.pParameter = &param;
kdMech.ulParameterLen = sizeof parram;

C_DeriveKey(hSes, &kdMech, hBDKey , NULL, 0, &hMKey);

// Single part verify operation

C_VerifyInit(hSes, &svMech, hMKey);
len = sizeof mac;
C_Verify(hSes, inp, sizeof inp, mac, len);

// clean up

C_DestroyObject(hSes, hMKey);

// Test vectors
```

CKM_DES2_KEY_GEN

Summary

FIPS approved?	Yes
Supported functions	Generate Key
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	128
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	0
Digest size	0
Key types	DES2
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DES3_CBC

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap | Cannot encrypt
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CBC
Flags	Extractable

> **NOTE** In this firmware version, "Functions restricted from FIPS use" has changed for this mechanism, to comply with FIPS 140-3 requirements.

Firmware 7.7.0-7.8.1 Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

 Copyright 2001-2024 Thales Group

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CBC
Flags	Extractable

> **NOTE** Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.
>
> The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.
>
> The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.
>
> The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to wrap objects.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CBC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES3_CBC_ENCRYPT_DATA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

NOTE In this firmware version, this mechanism is not approved for FIPS 140-3.

Firmware 7.8.1 and Older Summary

FIPS approved?	Yes
Supported functions	Derive
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128

 Copyright 2001-2024 Thales Group

Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

> **NOTE** Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.
>
> The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.
>
> The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.
>
> The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

 Copyright 2001-2024 Thales Group

CKM_DES3_CBC_PAD

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap | Cannot encrypt
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CBC_PAD
Flags	Extractable

> **NOTE** In this firmware version, "Functions restricted from FIPS use" has changed for this mechanism, to comply with FIPS 140-3 requirements.

Firmware 7.7.0-7.8.1 Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

 Copyright 2001-2024 Thales Group

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CBC_PAD
Flags	Extractable

> **NOTE** Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.
>
> The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.
>
> The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.
>
> The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to wrap objects.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES3_CBC_PAD_IPSEC

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CBC_PAD_IPSEC
Flags	Extractable

> **NOTE** Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.
>
> The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.
>
> The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.
>
> The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

CKM_DES3_CMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable | CMAC

> **NOTE** In this firmware version, "Functions restricted from FIPS use" has changed for this mechanism, to comply with FIPS 140-3 requirements.

Firmware 7.8.1 and Older Summary

FIPS approved?	Yes

Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable | CMAC

NOTE Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.

The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.

The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.

The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

CKM_DES3_CMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable | CMAC

> **NOTE** In this firmware version, "Functions restricted from FIPS use" has changed for this mechanism, to comply with FIPS 140-3 requirements.

Firmware 7.8.1 and Older Summary

FIPS approved?	Yes

Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable | CMAC

NOTE Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.

The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.

The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.

The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

CKM_DES3_CTR

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap | Cannot encrypt
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CTR
Flags	Extractable

> **NOTE** In this firmware version, "Functions restricted from FIPS use" has changed for this mechanism, to comply with FIPS 140-3 requirements.

Firmware 7.7.0-7.8.1 Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

 Copyright 2001-2024 Thales Group

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CTR
Flags	Extractable

> **NOTE** Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.
>
> The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.
>
> The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.
>
> The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to wrap objects.

Firmware 7.2.0-7.4.2 Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CTR
Flags	Extractable

Firmware 7.1.0 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	CTR
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES3_ECB

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap | Cannot encrypt
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	ECB
Flags	Extractable

> **NOTE** In this firmware version, "Functions restricted from FIPS use" has changed for this mechanism, to comply with FIPS 140-3 requirements.

Firmware 7.7.0-7.8.1 Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	ECB
Flags	Extractable

> **NOTE** Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.
>
> The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.
>
> The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.
>
> The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to wrap objects.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192

Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES3_ECB_ENCRYPT_DATA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

NOTE In this firmware version, this mechanism is not approved for FIPS 140-3.

Firmware 7.8.1 and Older Summary

FIPS approved?	Yes
Supported functions	Derive
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128

Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

> **NOTE** Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.
>
> The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.
>
> The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.
>
> The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

CKM_DES3_KEY_GEN

Summary

FIPS approved?	Yes
Supported functions	Generate Key
Functions restricted from FIPS use	None
Minimum key length (bits)	192
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	0
Digest size	0
Key types	DES3
Algorithms	None
Modes	None
Flags	None

NOTE Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.

The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.

The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.

The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

CKM_DES3_MAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.7 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

> **NOTE** Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.
>
> The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.
>
> The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.
>
> The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

Firmware 7.7.0-7.8.4 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable

CKM_DES3_MAC is no longer supported for MAC generation when 'HSM Policy (12) Allow Non-FIPS Algorithms' is off.

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to sign data.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES3_MAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.7 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

> **NOTE** Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.
>
> The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.
>
> The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.
>
> The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

Firmware 7.7.0-7.8.4 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to sign data.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	192
Minimum legacy key length for FIPS use (bits)	128
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DES3_X919_MAC

The CKM_DES3_X919_MAC is a signature generation and verification mechanism, as defined ANSI X9.19-1996 Financial Institution Retail Message Authentication annex 1 Cipher Block Chaining Procedure.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	8
Digest size	0
Key types	DES3
Algorithms	DES3
Modes	MAC
Flags	Extractable

NOTE Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^16 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.

The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.

The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.

The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

Usage

The CKM_DES3_X919_MAC mechanism is used with the **C_VerifyInit** and **C_SignInit** functions. It has the following attributes:

- Only CKK_DES2 and CKK_DES3 keys are supported.
- The mechanism takes no parameter.
- Multi-part operation is supported.
- The total input data length must be at least one byte.
- The length of result is half the size of the DES block (i.e. 4 bytes).

Example

```
#define CKM_DES3_X919_MAC (CKM_VENDOR_DEFINED + 0x150)

CK_OBJECT_HANDLE hKey; // handle of CKK_DES2 or CKK_DES3 key
CK_MECHANISM mech = { CKM_DES3_X919_MAC , NULL, 0};
CK_CHAR inp[any length];
CK_CHAR mac[4];
CK_SIZE len;

// Single-part operation

C_SignInit(hSes, &mech, hKey);
len = sizeof mac;
C_Sign(hSes, inp, sizeof inp, mac, &len);

// Multi-part operation

C_SignInit(hSes, &mech, hKey);
C_SignUpdate(hSes, inp, sizeof inp/2);
C_SignUpdate(hSes, inp+ (sizeof inp)/2, sizeof inp/2);
len = sizeof mac;
C_SignFinal(hSes, mac, &len);

// Test vectors

static const UInt8 retailKey[16] =
{
   0x58, 0x91, 0x25, 0x86, 0x3D, 0x46, 0x10, 0x7F,
   0x46, 0x3E, 0x52, 0x3B, 0xF7, 0x46, 0x9D, 0x52
};

static const UInt8 retailInputAscii[19] =
{
   't','h','e',' ','q','u','i','c','k',' ','b','r','o','w','n',' ','f','o','x'
};

static const UInt8 retailMACAscii[4] =
{
   0x55, 0xA7, 0xBF, 0xBA
};

static const UInt8 retailInputEBCDIC[19] =
{
```

```
   // "the quick brown fox" in EBCDIC
   0xA3, 0x88, 0x85, 0x40, 0x98, 0xA4, 0x89, 0x83,
   0x92, 0x40, 0x82, 0x99, 0x96, 0xA6, 0x95, 0x40,
   0x86, 0x96, 0xA7
};

static const UInt8 retailMACEBCDIC[4] =
{
   0x60, 0xAE, 0x2C, 0xD7
};
```

 Copyright 2001-2024 Thales Group

CKM_DH_PKCS_DERIVE

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	0
Digest size	0
Key types	DH
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DH_PKCS_KEY_PAIR_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key Pair
Functions restricted from FIPS use	N/A
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	0
Digest size	0
Key types	DH
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DH_PKCS_PARAMETER_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	0
Digest size	0
Key types	DH
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DSA

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072
Block size	0
Digest size	0
Key types	DSA
Algorithms	DSA
Modes	None
Flags	None

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072

 Copyright 2001-2024 Thales Group

Block size	0
Digest size	0
Key types	DSA
Algorithms	DSA
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DSA_KEY_PAIR_GEN

Firmware 7.8.7 and Newer Summary

FIPS approved?	No
Supported functions	Generate Key Pair
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	3072
Block size	0
Digest size	0
Key types	DSA
Algorithms	None
Modes	None
Flags	None

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Generate Key Pair
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072

 Copyright 2001-2024 Thales Group

Block size	0
Digest size	0
Key types	DSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DSA_PARAMETER_GEN

Firmware 7.8.7 and Newer Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	3072
Block size	0
Digest size	0
Key types	DSA
Algorithms	None
Modes	None
Flags	None

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Generate Key
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072

 Copyright 2001-2024 Thales Group

Block size	0
Digest size	0
Key types	DSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_DSA_SHA1

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072
Block size	64
Digest size	20
Key types	DSA
Algorithms	SHA
Modes	None
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to sign data.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048

Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072
Block size	64
Digest size	20
Key types	DSA
Algorithms	SHA
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DSA_SHA224

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072
Block size	64
Digest size	28
Key types	DSA
Algorithms	SHA224
Modes	None
Flags	Extractable

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	28
Key types	DSA
Algorithms	SHA224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_DSA_SHA256

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072
Block size	64
Digest size	32
Key types	DSA
Algorithms	SHA256
Modes	None
Flags	Extractable

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	32
Key types	DSA
Algorithms	SHA256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_EC_EDWARDS_KEY_PAIR_GEN

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Generate Key Pair
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	0
Digest size	0
Key types	EDDSA
Algorithms	None
Modes	None
Flags	None

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Generate Key Pair
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	0
Digest size	0
Key types	EDDSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_EC_KEY_PAIR_GEN

Summary

FIPS approved?	Yes
Supported functions	Generate Key Pair
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	ECDSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS

Summary

FIPS approved?	Yes
Supported functions	Generate Key Pair
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	ECDSA
Algorithms	None
Modes	None
Flags	Extra bits

 Copyright 2001-2024 Thales Group

CKM_EC_MONTGOMERY_KEY_PAIR_GEN

Generate keys over Montgomery curves. Keys generated with this mechanism are of type CKK_EC_MONTGOMERY. They can be used with the existing CKM_ECDH1_DERIVE mechanism. Given that the ECDH mechanism is the same, and relies on "point multiply" on the given curve, no Montgomery-specific mechanism is provided at this time. Allowed curve is "Curve25519".

Firmware 7.8.4 and Newer Summary

FIPS approved?	Yes
Supported functions	Generate Key Pair
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	256
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	448
Block size	0
Digest size	0
Key types	EC_MONT
Algorithms	None
Modes	None
Flags	None

Firmware 7.1.0-7.8.1 Summary

FIPS approved?	Yes
Supported functions	Generate Key Pair
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	256

 Copyright 2001-2024 Thales Group

Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	0
Digest size	0
Key types	EC_MONT
Algorithms	None
Modes	None
Flags	None

Firmware 7.0.3 and Older Summary

FIPS approved?	No
Supported functions	Generate Key Pair
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	0
Digest size	0
Key types	EC_MONT
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_ECDH1_COFACTOR_DERIVE

Firmware 7.3.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Derive
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	ECDSA | BIP32
Algorithms	None
Modes	None
Flags	None

Firmware 7.2.0 and Older Summary

FIPS approved?	Yes
Supported functions	Derive
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0

 Copyright 2001-2024 Thales Group

Digest size	0
Key types	ECDSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_ECDH1_DERIVE

Elliptic Curve Diffie-Hellman is an anonymous key-agreement protocol. CKM_ECDH1_DERIVE is the derive function for that protocol.

> **NOTE** To enhance performance, we have created a proprietary call CA_DeriveKeyAndWrap, which is an optimization of C_DeriveKey with C_Wrap, merging the two functions into one (the in and out constraints are the same as for the individual functions). A further optimization is applied when mechanism CKM_ECDH1_DERIVE is used with CA_DeriveKeyAndWrap.
>
> If CA_DeriveKeyAndWrap is called with other mechanisms, those would not be optimized.

Firmware 7.3.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Derive
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	ECDSA | EC_MONT | BIP32
Algorithms	None
Modes	None
Flags	None

Firmware 7.2.0 and Older Summary

FIPS approved?	Yes
Supported functions	Derive

 Copyright 2001-2024 Thales Group

Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	ECDSA | EC_MONT
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_ECDSA

Firmware 7.3.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	ECDSA | BIP32
Algorithms	ECDSA
Modes	None
Flags	None

Firmware 7.2.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0

Digest size	0
Key types	ECDSA
Algorithms	ECDSA
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_ECDSA_GBCS_SHA256

Firmware 7.3.0 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	64
Digest size	32
Key types	ECDSA | BIP32
Algorithms	SHA256
Modes	None
Flags	Extractable

Firmware 7.2.0 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	64

Digest size	32
Key types	ECDSA
Algorithms	SHA256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECDSA_SHA1

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	64
Digest size	20
Key types	ECDSA | BIP32
Algorithms	SHA
Modes	None
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to sign data.

Firmware 7.3.0-7.4.2 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224

Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	64
Digest size	20
Key types	ECDSA | BIP32
Algorithms	SHA
Modes	None
Flags	Extractable

Firmware 7.2.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	64
Digest size	20
Key types	ECDSA
Algorithms	SHA
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECDSA_SHA3_512

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	72
Digest size	64
Key types	ECDSA | BIP32
Algorithms	SHA3_512
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	72

 Copyright 2001-2024 Thales Group

Digest size	64
Key types	ECDSA | BIP32
Algorithms	SHA3_512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECDSA_SHA3_256

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	136
Digest size	32
Key types	ECDSA | BIP32
Algorithms	SHA3_256
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	136

Digest size	32
Key types	ECDSA | BIP32
Algorithms	SHA3_256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECDSA_SHA3_384

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	104
Digest size	48
Key types	ECDSA | BIP32
Algorithms	SHA3_384
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	104

Digest size	48
Key types	ECDSA | BIP32
Algorithms	SHA3_384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECDSA_SHA3_224

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	144
Digest size	28
Key types	ECDSA | BIP32
Algorithms	SHA3_224
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	144

Digest size	28
Key types	ECDSA | BIP32
Algorithms	SHA3_224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECDSA_SHA224

Firmware 7.3.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	64
Digest size	28
Key types	ECDSA | BIP32
Algorithms	SHA224
Modes	None
Flags	Extractable

Firmware 7.2.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	64

Digest size	28
Key types	ECDSA
Algorithms	SHA224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECDSA_SHA256

Firmware 7.3.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	64
Digest size	32
Key types	ECDSA | BIP32
Algorithms	SHA256
Modes	None
Flags	Extractable

Firmware 7.2.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	64

 Copyright 2001-2024 Thales Group

Digest size	32
Key types	ECDSA
Algorithms	SHA256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECDSA_SHA384

Firmware 7.3.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	128
Digest size	48
Key types	ECDSA | BIP32
Algorithms	SHA384
Modes	None
Flags	Extractable

Firmware 7.2.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	128

 Copyright 2001-2024 Thales Group

Digest size	48
Key types	ECDSA
Algorithms	SHA384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECDSA_SHA512

Firmware 7.3.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	128
Digest size	64
Key types	ECDSA | BIP32
Algorithms	SHA512
Modes	None
Flags	Extractable

Firmware 7.2.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	128

 Copyright 2001-2024 Thales Group

Digest size	64
Key types	ECDSA
Algorithms	SHA512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_ECIES

ECIES, or Elliptic Curve Integrated Encryption Scheme, is a public-key encryption scheme that combines

> **TIP** Luna HSM Firmware 7.7.2 and newer adds the derivation of the Initial Counter block (ICB) for ECIES AES-CTR encryption scheme to support the 5G 3GPP TS 33.501 standard, for processing of SUbscription Concealed Identifier (SUCI) de-concealment requests.
>
> **Decrypt operations** with curve ed25519 are accelerated with Luna HSM Firmware 7.7.2 and newer - optimum performance is achieved with 10 program threads for standalone Luna HSMs, while the best gain for HSMs in an HA group is around 20 threads, with smaller improvements observed up to 50 threads.

See also "ECIES general" on page 604 and "ECIES for 5G" on page 607.

Firmware 7.3.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	None
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	ECDSA | EC_MONT | BIP32
Algorithms	None
Modes	None
Flags	Accumulating

Firmware 7.2.0 and Older Summary

FIPS approved?	Yes

Supported functions	Encrypt | Decrypt
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	224
Minimum legacy key length for FIPS use (bits)	160
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	ECDSA | EC_MONT
Algorithms	None
Modes	None
Flags	Accumulating

NOTE This is a single part operation, so even if it is called using multi-part API, we accumulate the data (up to a maximum) and return data only on the “final” operation. That is the meaning of "Accumulating" in the tables, above.

 Copyright 2001-2024 Thales Group

CKM_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	0
Digest size	0
Key types	EDDSA
Algorithms	SHA512
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	0
Digest size	0
Key types	EDDSA
Algorithms	SHA512
Modes	None
Flags	Extractable

This mechanism makes use of keys generated by "CKM_EC_EDWARDS_KEY_PAIR_GEN" on page 280 (using keys generated over Edwards curves) for EDDSA signing. The keys used by this mechanism are of type CKK_EC_EDWARDS. For Luna HSM, the EDDSA algorithm is compliant with “PureEDDSA” as defined in RFC 8032 and “EdDSA for more curves, July 2015”.

Mechanism Parameters

Mechanism parameters are optional; not using the parameters selects the PureEdDSA algorithm **ed25519**. Setting the prehashed flag (phFlag) to TRUE will select the prehashed **ed25519ph** curve variant.

```
typedef struct CK_EDDSA_PARAMS
{
     CK_BBOOL     phFlag;
     CK_ULONG     ulContextDataLen;
     CK_BYTE_PTR  pContextData;
}

CK_EDDSA_PARAMS;

CK_EDDSA_PARAMS eddsaParams;
     eddsaParams.phFlag = CK_TRUE; // Set prehashed flag to true for Ed25519ph. Setting it to
false or not using mechanism parameters does Ed25519.
     eddsaParams.ulContextDataLen = 0; // Context length must be 0
     eddsaParams.pContextData = NULL; // Context must be NULL

CK_MECHANISM mechanism;
     mechanism.mechanism = CKM_EDDSA;
     mechanism.pParameter = &eddsaParams;
     mechanism.ulParameterLen = sizeof(eddsaParams);

C_SignInit(hSession, &mechanism, hKey); // or C_VerifyInit
// followed by C_Sign, C_SignUpdate/C_SignFinal or verify equivalents.
```

OIDs and Algorithm Identifiers for 25519 Keys

New OIDs and algorithm identifiers are as follows. Curve identifiers, including the plaintext curve names, must be ASN.1-encoded.

Edwards 25519 (sign/verify)

Curve Identifier (CKA_ECDSA_PARAMS):

> “edwards25519” (RFC7748)
> “Ed25519” (RFC8410)
> 1.3.6.1.4.1.11591.15.1 (https://www.alvestrand.no/objectid/1.3.6.1.4.1.11591.15.1.html)

Key OIDs (wrap/unwrap):

> 1.3.101.100 (https://tools.ietf.org/html/draft-josefsson-pkix-eddsa-04)
> 1.3.101.112 (RFC8410)

Curve 25519 (ECDH)

Curve Identifier (CKA_ECDSA_PARAMS):

> “curve25519” (RFC7748)
> “X25519” (RFC8410)
> 1.3.6.1.4.1.3029.1.5.1 (http://oidref.com/1.3.6.1.4.1.3029.1.5.1)

Key OIDs (wrap/unwrap):

> 1.3.6.1.4.1.11591.7 (https://tools.ietf.org/html/draft-josefsson-pkix-newcurves-00)
> 1.3.101.110 (RFC8410)

 Copyright 2001-2024 Thales Group

CKM_EDDSA_NACL

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	0
Digest size	0
Key types	EDDSA
Algorithms	SHA512
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	0
Digest size	0
Key types	EDDSA
Algorithms	SHA512
Modes	None
Flags	Extractable

Use EDDSA keys in Networking and Cryptography Library ("salt") sign/verify operations.

 Copyright 2001-2024 Thales Group

CKM_GENERIC_SECRET_KEY_GEN

Summary

FIPS approved?	Yes
Supported functions	Generate Key
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_HAS160

Summary

FIPS approved?	No
Supported functions	Digest
Functions restricted from FIPS use	N/A
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	64
Digest size	20
Key types	None
Algorithms	HAS160
Modes	None
Flags	Extractable | Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_HAS160

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	64
Digest size	20
Key types	KCDSA
Algorithms	HAS160
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_HAS160_NO_PAD

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	64
Digest size	20
Key types	KCDSA
Algorithms	HAS160
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_KEY_PAIR_GEN

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Generate Key Pair
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	0
Digest size	0
Key types	KCDSA
Algorithms	None
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_PARAMETER_GEN

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	0
Digest size	0
Key types	KCDSA
Algorithms	None
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA1

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	64
Digest size	20
Key types	KCDSA
Algorithms	SHA
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA1_NO_PAD

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	64
Digest size	20
Key types	KCDSA
Algorithms	SHA
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA224

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	64
Digest size	28
Key types	KCDSA
Algorithms	SHA224
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA224_NO_PAD

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	64
Digest size	28
Key types	KCDSA
Algorithms	SHA224
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA256

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	64
Digest size	32
Key types	KCDSA
Algorithms	SHA256
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA256_NO_PAD

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	64
Digest size	32
Key types	KCDSA
Algorithms	SHA256
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA384

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	128
Digest size	48
Key types	KCDSA
Algorithms	SHA384
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA384_NO_PAD

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	128
Digest size	48
Key types	KCDSA
Algorithms	SHA384
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA512

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	128
Digest size	64
Key types	KCDSA
Algorithms	SHA512
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KCDSA_SHA512_NO_PAD

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	128
Digest size	64
Key types	KCDSA
Algorithms	SHA512
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_KECCAK_224

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Digest
Functions restricted from FIPS use	N/A
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	144
Digest size	28
Key types	None
Algorithms	KECCAK_224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_KECCAK_256

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Digest
Functions restricted from FIPS use	N/A
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	136
Digest size	32
Key types	None
Algorithms	KECCAK_256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_KECCAK_384

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Digest
Functions restricted from FIPS use	N/A
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	104
Digest size	48
Key types	None
Algorithms	KECCAK_384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_KECCAK_512

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Digest
Functions restricted from FIPS use	N/A
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	72
Digest size	64
Key types	None
Algorithms	KECCAK_512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_KEY_TRANSLATE

This is a proprietary Luna mechanism, added to Luna HSM Firmware 7.8.1 and newer. This mechanism is used with the C_WrapKey command. The maximum allowed data size for this mechanism is 8000 bytes. See "Luna Key Translation" on page 107 for description and usage details.

Firmware 7.8.1 and Newer Summary

FIPS approved?	Yes
Supported functions	Wrap
Functions restricted from FIPS use	None
Minimum key length (bits)	112
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

NOTE This mechanism cannot be used with the CA_DeriveKeyAndWrap command.

CKM_KEY_WRAP_SET_OAEP

Summary

FIPS approved?	No
Supported functions	Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_MD2

Summary

FIPS approved?	No
Supported functions	Digest
Functions restricted from FIPS use	N/A
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	16
Digest size	16
Key types	None
Algorithms	MD2
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_MD2_KEY_DERIVATION

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	16
Digest size	16
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_MD5_HMAC

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	16
Key types	Symmetric
Algorithms	MD5
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A

 Copyright 2001-2024 Thales Group

Maximum key length (bits)	4096
Block size	64
Digest size	16
Key types	Symmetric
Algorithms	MD5
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_MD5_HMAC_GENERAL

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	16
Key types	Symmetric
Algorithms	MD5
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A

 Copyright 2001-2024 Thales Group

Maximum key length (bits)	4096
Block size	64
Digest size	16
Key types	Symmetric
Algorithms	MD5
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_MD5_KEY_DERIVATION

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	16
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_MILENAGE

> **NOTE** This mechanism can be used with HA with Luna HSM Client 10.4.0 and newer.

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	ECB
Flags	Allow zero-length input

 Copyright 2001-2024 Thales Group

CKM_MILENAGE_AUTS

> **NOTE** This mechanism can be used with HA with Luna HSM Client 10.4.0 and newer.

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	ECB
Flags	None

 Copyright 2001-2024 Thales Group

CKM_MILENAGE_RESYNC

> **NOTE** This mechanism can be used with HA with Luna HSM Client 10.4.0 and newer.

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	ECB
Flags	None

 Copyright 2001-2024 Thales Group

CKM_NIST_PRF_KDF

Summary

FIPS approved?	Yes
Supported functions	Derive
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

Usage

The CKM_NIST_PRF_KDF mechanism only supports counter mode. CKM_NIST_PRF_KDF is always allowed, whether **HSM policy 12: Allow Non-FIPS algorithms** is on or off. This mechanism can be used with the following mechanisms as the pseudorandom function:

- AES_CMAC
- DES3_CMAC
- HMAC_SHA1
- HMAC_SHA224
- HMAC_SHA256
- HMAC_SHA384
- HMAC_SHA512

NIST SP 800-108 allows for some variation on what/how information is encoded and describes some fields as optional. To accommodate this, there are multiple encoding schemes you can specify, with variations on what information is included and what order the fields are arranged in. All counters and lengths are represented in big endian format. The following schemes are available:

> LUNA_PRF_KDF_ENCODING_SCHEME_1: the **Counter** (4 bytes), **Context**, **Separator byte**, **Label**, and **Length** (4 bytes) fields are included.

> LUNA_PRF_KDF_ENCODING_SCHEME_2: the **Counter** (4 bytes), **Context** and **Label** fields are included.

> LUNA_PRF_KDF_ENCODING_SCHEME_3: the **Counter** (4 bytes), **Label**, **Separator byte**, **Context**, and **Length** (4 bytes) fields are included.

> LUNA_PRF_KDF_ENCODING_SCHEME_4: the **Counter** (4 bytes), **Label** and **Context** fields are included.

> LUNA_PRF_KDF_ENCODING_SCHEME_SCP03: the **Label**, **Separator byte**, **Length** (2 bytes), **Counter**, and **Context** fields are included.

> LUNA_PRF_KDF_ENCODING_SCHEME_HID_KD: the **Counter**, **Label**, **Separator byte**, **Context**, and **Length** (2 bytes) fields are included.

Example

```
/* Parameter and values used with CKM_PRF_KDF and CKM_NIST_PRF_KDF. */
typedef CK_ULONG CK_KDF_PRF_TYPE;
typedef CK_ULONG CK_KDF_PRF_ENCODING_SCHEME;
/** PRF KDF schemes */
#define CK_NIST_PRF_KDF_DES3_CMAC      0x00000001
#define CK_NIST_PRF_KDF_AES_CMAC       0x00000002
#define CK_PRF_KDF_ARIA_CMAC           0x00000003
#define CK_PRF_KDF_SEED_CMAC           0x00000004
#define CK_NIST_PRF_KDF_HMAC_SHA1      0x00000005
#define CK_NIST_PRF_KDF_HMAC_SHA224    0x00000006
#define CK_NIST_PRF_KDF_HMAC_SHA256    0x00000007
#define CK_NIST_PRF_KDF_HMAC_SHA384    0x00000008
#define CK_NIST_PRF_KDF_HMAC_SHA512    0x00000009
#define CK_PRF_KDF_HMAC_RIPEMD160      0x0000000A
#define LUNA_PRF_KDF_ENCODING_SCHEME_1     0x00000000 // Counter (4 bytes) || Context || 0x00
|| Label || Length
#define LUNA_PRF_KDF_ENCODING_SCHEME_2     0x00000001 // Counter (4 bytes) || Context || Label
#define LUNA_PRF_KDF_ENCODING_SCHEME_3     0x00000002 // Counter (4 bytes) || Label || 0x00 ||
Context || Length
#define LUNA_PRF_KDF_ENCODING_SCHEME_4     0x00000003 // Counter (4 bytes) || Label || Context
#define LUNA_PRF_KDF_ENCODING_SCHEME_SCP03 0x00000004 // Label || 0x00 || Length (2 bytes) ||
Counter (1 byte) || Context
#define LUNA_PRF_KDF_ENCODING_SCHEME_HID_KD 0x00000005 // Counter (1 byte) || Label || 0x00 ||
Context || Length (2 bytes)
typedef struct CK_KDF_PRF_PARAMS {
CK_KDF_PRF_TYPE            prfType;
CK_BYTE_PTR                pLabel;
CK_ULONG                   ulLabelLen;
CK_BYTE_PTR                pContext;
CK_ULONG                   ulContextLen;
CK_ULONG                   ulCounter;
CK_KDF_PRF_ENCODING_SCHEME ulEncodingScheme;
} CK_PRF_KDF_PARAMS;
typedef CK_PRF_KDF_PARAMS CK_PTR CK_KDF_PRF_PARAMS_PTR;
```

CKM_PBE_MD2_DES_CBC

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	64
Block size	16
Digest size	16
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_PBE_SHA1_CAST5_CBC

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	64
Digest size	20
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_PBE_SHA1_DES2_EDE_CBC

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	64
Digest size	20
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_PBE_SHA1_DES3_EDE_CBC

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	192
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	192
Block size	64
Digest size	20
Key types	None
Algorithms	None
Modes	None
Flags	None

NOTE Using Luna HSM Firmware 7.7.0 and newer, 3DES keys have a usage counter attribute (CKA_BYTES_REMAINING) that limits each key instance to encrypting a maximum of 2^{16} 8-byte blocks of data when the HSM is in FIPS mode (**HSM policy 12:** Allow non-FIPS algorithms set to **0**). When the counter runs out, that key can *no longer* be used for encryption, wrapping, deriving, or signing, but can still be used for decrypting, unwrapping, and verifying pre-existing objects.

The CKA_BYTES_REMAINING attribute is available when **HSM policy 12:** Allow non-FIPS algorithms is set to **0**, but cannot be viewed if the policy is set to **1**.

The attribute is preserved through backup/restore using a Luna Backup HSM 7; restoring the key restores the counter's setting at the time of backup.

The attribute is not preserved through backup/restore using a Luna Backup HSM G5; restoring the key resets the counter to the maximum.

CKM_PBE_SHA1_RC2_40_CBC

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	40
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	40
Block size	64
Digest size	20
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_PBE_SHA1_RC2_128_CBC

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	64
Digest size	20
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_PBE_SHA1_RC4_40

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	40
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	40
Block size	64
Digest size	20
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_PBE_SHA1_RC4_128

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	64
Digest size	20
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_PKCS5_PBKD2

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_PRF_KDF

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

Usage

The CKM_NIST_PRF mechanism only supports counter mode. CKM_PRF_KDF is only available with **HSM policy 12: Allow Non-FIPS algorithms** turned on. This mechanism can be used with the following mechanisms as the pseudorandom function:

> ARIA_CMAC

> HMAC_RIPEMD160

> SEED_CMAC

NIST SP 800-108 allows for some variation on what/how information is encoded and describes some fields as optional. To accommodate this, there are multiple encoding schemes you can specify, with variations on what information is included and what order the fields are arranged in. All counters and lengths are represented in big endian format. The following schemes are available:

> LUNA_PRF_KDF_ENCODING_SCHEME_1: the **Counter** (4 bytes), **Context**, **Separator byte**, **Label**, and **Length** (4 bytes) fields are included.

> LUNA_PRF_KDF_ENCODING_SCHEME_2: the **Counter** (4 bytes), **Context** and **Label** fields are included.

- LUNA_PRF_KDF_ENCODING_SCHEME_3: the **Counter** (4 bytes), **Label**, **Separator byte**, **Context**, and **Length** (4 bytes) fields are included.
- LUNA_PRF_KDF_ENCODING_SCHEME_4: the **Counter** (4 bytes), **Label** and **Context** fields are included.
- LUNA_PRF_KDF_ENCODING_SCHEME_SCP03: the **Label**, **Separator byte**, **Length** (2 bytes), **Counter**, and **Context** fields are included.
- LUNA_PRF_KDF_ENCODING_SCHEME_HID_KD: the **Counter**, **Label**, **Separator byte**, **Context**, and **Length** (2 bytes) fields are included.

Example

```
/* Parameter and values used with CKM_PRF_KDF and CKM_NIST_PRF_KDF. */
typedef CK_ULONG CK_KDF_PRF_TYPE;
typedef CK_ULONG CK_KDF_PRF_ENCODING_SCHEME;
/** PRF KDF schemes */
#define CK_NIST_PRF_KDF_DES3_CMAC      0x00000001
#define CK_NIST_PRF_KDF_AES_CMAC       0x00000002
#define CK_PRF_KDF_ARIA_CMAC           0x00000003
#define CK_PRF_KDF_SEED_CMAC           0x00000004
#define CK_NIST_PRF_KDF_HMAC_SHA1      0x00000005
#define CK_NIST_PRF_KDF_HMAC_SHA224    0x00000006
#define CK_NIST_PRF_KDF_HMAC_SHA256    0x00000007
#define CK_NIST_PRF_KDF_HMAC_SHA384    0x00000008
#define CK_NIST_PRF_KDF_HMAC_SHA512    0x00000009
#define CK_PRF_KDF_HMAC_RIPEMD160      0x0000000A
#define LUNA_PRF_KDF_ENCODING_SCHEME_1     0x00000000 // Counter (4 bytes) || Context || 0x00
|| Label || Length
#define LUNA_PRF_KDF_ENCODING_SCHEME_2     0x00000001 // Counter (4 bytes) || Context || Label
#define LUNA_PRF_KDF_ENCODING_SCHEME_3     0x00000002 // Counter (4 bytes) || Label || 0x00 ||
Context || Length
#define LUNA_PRF_KDF_ENCODING_SCHEME_4     0x00000003 // Counter (4 bytes) || Label || Context
#define LUNA_PRF_KDF_ENCODING_SCHEME_SCP03 0x00000004 // Label || 0x00 || Length (2 bytes) ||
Counter (1 byte) || Context
#define LUNA_PRF_KDF_ENCODING_SCHEME_HID_KD 0x00000005 // Counter (1 byte) || Label || 0x00 ||
Context || Length (2 bytes)
typedef struct CK_KDF_PRF_PARAMS {
CK_KDF_PRF_TYPE            prfType;
CK_BYTE_PTR                pLabel;
CK_ULONG                   ulLabelLen;
CK_BYTE_PTR                pContext;
CK_ULONG                   ulContextLen;
CK_ULONG                   ulCounter;
CK_KDF_PRF_ENCODING_SCHEME ulEncodingScheme;
} CK_PRF_KDF_PARAMS;
typedef CK_PRF_KDF_PARAMS CK_PTR CK_KDF_PRF_PARAMS_PTR;
```

CKM_RC2_CBC

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	1024
Block size	8
Digest size	0
Key types	RC2
Algorithms	RC2
Modes	CBC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC2_CBC_PAD

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	1024
Block size	8
Digest size	0
Key types	RC2
Algorithms	RC2
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC2_ECB

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	1024
Block size	8
Digest size	0
Key types	RC2
Algorithms	RC2
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC2_KEY_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	1024
Block size	0
Digest size	0
Key types	RC2
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_RC2_MAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	1024
Block size	8
Digest size	0
Key types	RC2
Algorithms	RC2
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC2_MAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	1024
Block size	8
Digest size	0
Key types	RC2
Algorithms	RC2
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC4

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	0
Digest size	0
Key types	RC4
Algorithms	RC4
Modes	STREAM
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC4_KEY_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2048
Block size	0
Digest size	0
Key types	RC4
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_RC5_CBC

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2040
Block size	8
Digest size	0
Key types	RC5
Algorithms	RC5
Modes	CBC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC5_CBC_PAD

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2040
Block size	8
Digest size	0
Key types	RC5
Algorithms	RC5
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC5_ECB

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2040
Block size	8
Digest size	0
Key types	RC5
Algorithms	RC5
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC5_KEY_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2040
Block size	0
Digest size	0
Key types	RC5
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_RC5_MAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2040
Block size	8
Digest size	0
Key types	RC5
Algorithms	RC5
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RC5_MAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	64
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	2040
Block size	8
Digest size	0
Key types	RC5
Algorithms	RC5
Modes	MAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_RSA_FIPS_186_3_AUX_PRIME_KEY_PAIR_GEN

Summary

FIPS approved?	Yes
Supported functions	Generate Key Pair
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_RSA_FIPS_186_3_PRIME_KEY_PAIR_GEN

Summary

FIPS approved?	Yes
Supported functions	Generate Key Pair
Functions restricted from FIPS use	None
Minimum key length (bits)	2048
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_RSA_PKCS

> **NOTE** This mechanism name and RSASSA-PKCS1-v1_5 are referring to the same underlying RSA signature scheme.

Firmware 7.8.4 and Newer Summary

> **NOTE** Using Luna HSM Firmware 7.8.4 and newer, this mechanism is restricted from all wrap/unwrap/encrypt/decrypt operations in FIPS mode. No exceptions are made for decrypt/unwrap operations using larger key sizes. This limited legacy use was permitted under FIPS 140-2; it is no longer approved under FIPS 140-3.

FIPS approved?	Yes
Supported functions	Sign | Verify | Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap | Cannot decrypt | Cannot unwrap | Cannot encrypt
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

Firmware 7.7.2-7.8.1 Summary

> **NOTE** Under **Functions restricted from FIPS use**, "Cannot legacy decrypt and "Cannot legacy unwrap" means that these operations are restricted with smaller keys (1024-bits, the previous minimum key size for FIPS use), but keys that meet the minimum FIPS size requirement (2048 bits) can still be used for decrypt and unwrap operations.

FIPS approved?	Yes
Supported functions	Sign | Verify | Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap | Cannot legacy decrypt | Cannot legacy unwrap | Cannot encrypt
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

Firmware 7.7.0-7.7.1 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify | Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot wrap
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048

Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to wrap objects.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify | Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

> **NOTE** When the HSM is in FIPS mode, this mechanism cannot be used to sign data using less than 224 bits.
>
> This algorithm must be combined with a FIPS-approved hash algorithm to be FIPS compliant.

CKM_RSA_PKCS_KEY_PAIR_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key Pair
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_RSA_PKCS_OAEP

The RSA PKCS OAEP mechanism can now use a supplied hashing mechanism. Previously RSA OAEP would always use SHA1 and returned an error if another was attempted.

With current firmware, PKCS#11 API and ckdemo now accept a new mechanism.

Allowed mechanisms are:

> CKM_SHA1
> CKM_SHA224
> CKM_SHA256
> CKM_SHA384
> CKM_SHA512
> 0 (use the firmware's default engine, which is currently SHA1)

In ckdemo menu option 98 has a new value 17 - OAEP Hash Params, which can be set to use either default (CKM_SHA1) or selectable. When it is set to selectable the user is prompted for a hash mechanism when using the OAEP mechanism.

Firmware 7.7.2 and Newer Summary

> **NOTE** Under **Functions restricted from FIPS use**, "Cannot legacy decrypt and "Cannot legacy unwrap" means that these operations are restricted with smaller keys (1024-bits, the previous minimum key size for FIPS use), but keys that meet the minimum FIPS size requirement (2048 bits) can still be used for decrypt and unwrap operations.

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	Cannot legacy decrypt | Cannot legacy unwrap
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA

Algorithms	None
Modes	None
Flags	None

Firmware 7.7.1 and Older Summary

FIPS approved?	Yes
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_RSA_PKCS_PSS

> **NOTE** This mechanism name and RSASSA-PSS are referring to the same underlying RSA signature scheme.

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	PSS

 Copyright 2001-2024 Thales Group

CKM_RSA_X_509

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_RSA_X9_31

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	Extractable | X9.31

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192

 Copyright 2001-2024 Thales Group

Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	Extractable | X9.31

 Copyright 2001-2024 Thales Group

CKM_RSA_X9_31_KEY_PAIR_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key Pair
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	X9.31

 Copyright 2001-2024 Thales Group

CKM_RSA_X9_31_NON_FIPS

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	0
Digest size	0
Key types	RSA
Algorithms	None
Modes	None
Flags	Extractable | X9.31 | Non-FIPS X9.31

 Copyright 2001-2024 Thales Group

CKM_SEED_CBC

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SEED
Algorithms	SEED
Modes	CBC
Flags	Extractable | Korean

 Copyright 2001-2024 Thales Group

CKM_SEED_CBC_PAD

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SEED
Algorithms	SEED
Modes	CBC_PAD
Flags	Extractable | Korean

 Copyright 2001-2024 Thales Group

CKM_SEED_CMAC

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SEED
Algorithms	SEED
Modes	MAC
Flags	Extractable | Korean | CMAC

 Copyright 2001-2024 Thales Group

CKM_SEED_CMAC_GENERAL

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SEED
Algorithms	SEED
Modes	MAC
Flags	Extractable | Korean | CMAC

 Copyright 2001-2024 Thales Group

CKM_SEED_CTR

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SEED
Algorithms	SEED
Modes	CTR
Flags	Extractable | Korean

 Copyright 2001-2024 Thales Group

CKM_SEED_ECB

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SEED
Algorithms	SEED
Modes	ECB
Flags	Extractable | Korean

 Copyright 2001-2024 Thales Group

CKM_SEED_KEY_GEN

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	0
Digest size	0
Key types	SEED
Algorithms	None
Modes	None
Flags	Korean

 Copyright 2001-2024 Thales Group

CKM_SEED_MAC

> **NOTE** The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SEED
Algorithms	SEED
Modes	MAC
Flags	Extractable | Korean

 Copyright 2001-2024 Thales Group

CKM_SEED_MAC_GENERAL

NOTE The SEED and KCDSA mechanisms are available on your HSM if Korean Algorithms are enabled.

TIP Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.

The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SEED
Algorithms	SEED
Modes	MAC
Flags	Extractable | Korean

 Copyright 2001-2024 Thales Group

CKM_SHA_1

Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	64
Digest size	20
Key types	None
Algorithms	SHA
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA_1_HMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	20
Key types	Symmetric
Algorithms	SHA
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	20
Key types	Symmetric
Algorithms	SHA
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA_1_HMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	20
Key types	Symmetric
Algorithms	SHA
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	20
Key types	Symmetric
Algorithms	SHA
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA1_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	64
Digest size	20
Key types	EDDSA
Algorithms	SHA
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	20
Key types	EDDSA
Algorithms	SHA
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA1_EDDSA_NACL

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	64
Digest size	20
Key types	EDDSA
Algorithms	SHA
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	20
Key types	EDDSA
Algorithms	SHA
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA1_KEY_DERIVATION

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	20
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHA1_RSA_PKCS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	20
Key types	RSA
Algorithms	SHA
Modes	None
Flags	Extractable

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to sign data.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048

Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	20
Key types	RSA
Algorithms	SHA
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA1_RSA_PKCS_PSS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	20
Key types	RSA
Algorithms	SHA
Modes	None
Flags	Extractable | PSS

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to sign data.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048

Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	20
Key types	RSA
Algorithms	SHA
Modes	None
Flags	Extractable | PSS

 Copyright 2001-2024 Thales Group

CKM_SHA1_RSA_X9_31

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	20
Key types	RSA
Algorithms	SHA
Modes	None
Flags	Extractable | X9.31

> **NOTE** To comply with FIPS SP800-131a Rev2 published in March 2019, when the HSM is in FIPS mode, this mechanism is not allowed to sign data.

Firmware 7.4.2 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048

Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	20
Key types	RSA
Algorithms	SHA
Modes	None
Flags	Extractable | X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA1_RSA_X9_31_NON_FIPS

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	64
Digest size	20
Key types	RSA
Algorithms	SHA
Modes	None
Flags	Extractable | X9.31 | Non-FIPS X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA1_SM2DSA

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	64
Digest size	20
Key types	SM2
Algorithms	SHA
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_224

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	144
Digest size	28
Key types	None
Algorithms	SHA3_224
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Digest
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	144

Digest size	28
Key types	None
Algorithms	SHA3_224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_224_DSA

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072
Block size	144
Digest size	28
Key types	DSA
Algorithms	SHA3_224
Modes	None
Flags	Extractable

Firmware 7.7.0-7.8.4 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072

 Copyright 2001-2024 Thales Group

Block size	144
Digest size	28
Key types	DSA
Algorithms	SHA3_224
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	3072
Block size	144
Digest size	28
Key types	DSA
Algorithms	SHA3_224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_224_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	144
Digest size	28
Key types	EDDSA
Algorithms	SHA3_224
Modes	None
Flags	Extractable

Firmware 7.4.2-7.8.1 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	144
Digest size	28
Key types	EDDSA
Algorithms	SHA3_224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_224_HMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	144
Digest size	28
Key types	Symmetric
Algorithms	SHA3_224
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	144
Digest size	28
Key types	Symmetric
Algorithms	SHA3_224
Modes	HMAC
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	144
Digest size	28
Key types	Symmetric
Algorithms	SHA3_224
Modes	HMAC
Flags	Extractable

CKM_SHA3_224_HMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	144
Digest size	28
Key types	Symmetric
Algorithms	SHA3_224
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	144
Digest size	28
Key types	Symmetric
Algorithms	SHA3_224
Modes	HMAC
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	144
Digest size	28
Key types	Symmetric
Algorithms	SHA3_224
Modes	HMAC
Flags	Extractable

CKM_SHA3_224_KEY_DERIVE

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	144
Digest size	28
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHA3_224_RSA_PKCS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	144
Digest size	28
Key types	RSA
Algorithms	SHA3_224
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	144

Digest size	28
Key types	RSA
Algorithms	SHA3_224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_224_RSA_PKCS_PSS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	144
Digest size	28
Key types	RSA
Algorithms	SHA3_224
Modes	None
Flags	Extractable | PSS

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	144

 Copyright 2001-2024 Thales Group

Digest size	28
Key types	RSA
Algorithms	SHA3_224
Modes	None
Flags	Extractable | PSS

 Copyright 2001-2024 Thales Group

CKM_SHA3_256

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	136
Digest size	32
Key types	None
Algorithms	SHA3_256
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Digest
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	136

Digest size	32
Key types	None
Algorithms	SHA3_256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_256_DSA

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072
Block size	136
Digest size	32
Key types	DSA
Algorithms	SHA3_256
Modes	None
Flags	Extractable

Firmware 7.7.0-7.8.4 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072

Block size	136
Digest size	32
Key types	DSA
Algorithms	SHA3_256
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	3072
Block size	136
Digest size	32
Key types	DSA
Algorithms	SHA3_256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_256_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	136
Digest size	32
Key types	EDDSA
Algorithms	SHA3_256
Modes	None
Flags	Extractable

Firmware 7.4.2-7.8.1 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	136
Digest size	32
Key types	EDDSA
Algorithms	SHA3_256
Modes	None
Flags	Extractable

CKM_SHA3_256_HMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	136
Digest size	32
Key types	Symmetric
Algorithms	SHA3_256
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	136
Digest size	32
Key types	Symmetric
Algorithms	SHA3_256
Modes	HMAC
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	136
Digest size	32
Key types	Symmetric
Algorithms	SHA3_256
Modes	HMAC
Flags	Extractable

CKM_SHA3_256_HMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	136
Digest size	32
Key types	Symmetric
Algorithms	SHA3_256
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	136
Digest size	32
Key types	Symmetric
Algorithms	SHA3_256
Modes	HMAC
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	136
Digest size	32
Key types	Symmetric
Algorithms	SHA3_256
Modes	HMAC
Flags	Extractable

CKM_SHA3_256_KEY_DERIVE

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	136
Digest size	32
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHA3_256_RSA_PKCS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	136
Digest size	32
Key types	RSA
Algorithms	SHA3_256
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	136

Digest size	32
Key types	RSA
Algorithms	SHA3_256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_256_RSA_PKCS_PSS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	136
Digest size	32
Key types	RSA
Algorithms	SHA3_256
Modes	None
Flags	Extractable | PSS

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	136

Digest size	32
Key types	RSA
Algorithms	SHA3_256
Modes	None
Flags	Extractable | PSS

 Copyright 2001-2024 Thales Group

CKM_SHA3_384

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	104
Digest size	48
Key types	None
Algorithms	SHA3_384
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Digest
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	104

Digest size	48
Key types	None
Algorithms	SHA3_384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_384_DSA

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072
Block size	104
Digest size	48
Key types	DSA
Algorithms	SHA3_384
Modes	None
Flags	Extractable

Firmware 7.7.0-7.8.4 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072

Block size	104
Digest size	48
Key types	DSA
Algorithms	SHA3_384
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	3072
Block size	104
Digest size	48
Key types	DSA
Algorithms	SHA3_384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_384_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	104
Digest size	48
Key types	EDDSA
Algorithms	SHA3_384
Modes	None
Flags	Extractable

Firmware 7.4.2-7.8.1 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	104
Digest size	48
Key types	EDDSA
Algorithms	SHA3_384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_384_HMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	104
Digest size	48
Key types	Symmetric
Algorithms	SHA3_384
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	104
Digest size	48
Key types	Symmetric
Algorithms	SHA3_384
Modes	HMAC
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	104
Digest size	48
Key types	Symmetric
Algorithms	SHA3_384
Modes	HMAC
Flags	Extractable

CKM_SHA3_384_HMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	104
Digest size	48
Key types	Symmetric
Algorithms	SHA3_384
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	104
Digest size	48
Key types	Symmetric
Algorithms	SHA3_384
Modes	HMAC
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	104
Digest size	48
Key types	Symmetric
Algorithms	SHA3_384
Modes	HMAC
Flags	Extractable

CKM_SHA3_384_KEY_DERIVE

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	104
Digest size	48
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHA3_384_RSA_PKCS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	104
Digest size	48
Key types	RSA
Algorithms	SHA3_384
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	104

Digest size	48
Key types	RSA
Algorithms	SHA3_384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_384_RSA_PKCS_PSS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	104
Digest size	48
Key types	RSA
Algorithms	SHA3_384
Modes	None
Flags	Extractable | PSS

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	104

 Copyright 2001-2024 Thales Group

Digest size	48
Key types	RSA
Algorithms	SHA3_384
Modes	None
Flags	Extractable | PSS

 Copyright 2001-2024 Thales Group

CKM_SHA3_512

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	72
Digest size	64
Key types	None
Algorithms	SHA3_512
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Digest
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	72

Digest size	64
Key types	None
Algorithms	SHA3_512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_512_DSA

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072
Block size	72
Digest size	64
Key types	DSA
Algorithms	SHA3_512
Modes	None
Flags	Extractable

Firmware 7.7.0-7.8.4 Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	3072

 Copyright 2001-2024 Thales Group

Block size	72
Digest size	64
Key types	DSA
Algorithms	SHA3_512
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	3072
Block size	72
Digest size	64
Key types	DSA
Algorithms	SHA3_512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_512_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	72
Digest size	64
Key types	EDDSA
Algorithms	SHA3_512
Modes	None
Flags	Extractable

Firmware 7.4.2-7.8.1 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	72
Digest size	64
Key types	EDDSA
Algorithms	SHA3_512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_512_HMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	72
Digest size	64
Key types	Symmetric
Algorithms	SHA3_512
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	72
Digest size	64
Key types	Symmetric
Algorithms	SHA3_512
Modes	HMAC
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	72
Digest size	64
Key types	Symmetric
Algorithms	SHA3_512
Modes	HMAC
Flags	Extractable

CKM_SHA3_512_HMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	72
Digest size	64
Key types	Symmetric
Algorithms	SHA3_512
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	72
Digest size	64
Key types	Symmetric
Algorithms	SHA3_512
Modes	HMAC
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	72
Digest size	64
Key types	Symmetric
Algorithms	SHA3_512
Modes	HMAC
Flags	Extractable

CKM_SHA3_512_KEY_DERIVE

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	72
Digest size	64
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHA3_512_RSA_PKCS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	72
Digest size	64
Key types	RSA
Algorithms	SHA3_512
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	72

Digest size	64
Key types	RSA
Algorithms	SHA3_512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA3_512_RSA_PKCS_PSS

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	72
Digest size	64
Key types	RSA
Algorithms	SHA3_512
Modes	None
Flags	Extractable | PSS

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Sign | Verify
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	72

 Copyright 2001-2024 Thales Group

Digest size	64
Key types	RSA
Algorithms	SHA3_512
Modes	None
Flags	Extractable | PSS

 Copyright 2001-2024 Thales Group

CKM_SHA224

Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	64
Digest size	28
Key types	None
Algorithms	SHA224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA224_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	64
Digest size	28
Key types	EDDSA
Algorithms	SHA224
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	28
Key types	EDDSA
Algorithms	SHA224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA224_EDDSA_NACL

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	64
Digest size	28
Key types	EDDSA
Algorithms	SHA224
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	28
Key types	EDDSA
Algorithms	SHA224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA224_HMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	28
Key types	Symmetric
Algorithms	SHA224
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	28
Key types	Symmetric
Algorithms	SHA224
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA224_HMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	28
Key types	Symmetric
Algorithms	SHA224
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	28
Key types	Symmetric
Algorithms	SHA224
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA224_KEY_DERIVATION

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	28
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHA224_RSA_PKCS

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	28
Key types	RSA
Algorithms	SHA224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA224_RSA_PKCS_PSS

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	28
Key types	RSA
Algorithms	SHA224
Modes	None
Flags	Extractable | PSS

 Copyright 2001-2024 Thales Group

CKM_SHA224_RSA_X9_31

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	28
Key types	RSA
Algorithms	SHA224
Modes	None
Flags	Extractable | X9.31

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	28
Key types	RSA
Algorithms	SHA224
Modes	None
Flags	Extractable | X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA224_RSA_X9_31_NON_FIPS

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	64
Digest size	28
Key types	RSA
Algorithms	SHA224
Modes	None
Flags	Extractable | X9.31 | Non-FIPS X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA224_SM2DSA

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	64
Digest size	28
Key types	SM2
Algorithms	SHA224
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA256

Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	64
Digest size	32
Key types	None
Algorithms	SHA256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA256_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	64
Digest size	32
Key types	EDDSA
Algorithms	SHA256
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	32
Key types	EDDSA
Algorithms	SHA256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA256_EDDSA_NACL

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	64
Digest size	32
Key types	EDDSA
Algorithms	SHA256
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	32
Key types	EDDSA
Algorithms	SHA256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA256_HMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	SHA256
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	SHA256
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA256_HMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	SHA256
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	SHA256
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA256_KEY_DERIVATION

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHA256_RSA_PKCS

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	32
Key types	RSA
Algorithms	SHA256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA256_RSA_PKCS_PSS

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	32
Key types	RSA
Algorithms	SHA256
Modes	None
Flags	Extractable | PSS

 Copyright 2001-2024 Thales Group

CKM_SHA256_RSA_X9_31

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	64
Digest size	32
Key types	RSA
Algorithms	SHA256
Modes	None
Flags	Extractable | X9.31

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192

 Copyright 2001-2024 Thales Group

Block size	64
Digest size	32
Key types	RSA
Algorithms	SHA256
Modes	None
Flags	Extractable | X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA256_RSA_X9_31_NON_FIPS

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	64
Digest size	32
Key types	RSA
Algorithms	SHA256
Modes	None
Flags	Extractable | X9.31 | Non-FIPS X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA256_SM2DSA

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	64
Digest size	32
Key types	SM2
Algorithms	SHA256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA384

Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	128
Digest size	48
Key types	None
Algorithms	SHA384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA384_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	128
Digest size	48
Key types	EDDSA
Algorithms	SHA384
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

Block size	128
Digest size	48
Key types	EDDSA
Algorithms	SHA384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA384_EDDSA_NACL

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	128
Digest size	48
Key types	EDDSA
Algorithms	SHA384
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	128
Digest size	48
Key types	EDDSA
Algorithms	SHA384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA384_HMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	128
Digest size	48
Key types	Symmetric
Algorithms	SHA384
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	128
Digest size	48
Key types	Symmetric
Algorithms	SHA384
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA384_HMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	128
Digest size	48
Key types	Symmetric
Algorithms	SHA384
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	128
Digest size	48
Key types	Symmetric
Algorithms	SHA384
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA384_KEY_DERIVATION

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	128
Digest size	48
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHA384_RSA_PKCS

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	128
Digest size	48
Key types	RSA
Algorithms	SHA384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA384_RSA_PKCS_PSS

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	512
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	128
Digest size	48
Key types	RSA
Algorithms	SHA384
Modes	None
Flags	Extractable | PSS

 Copyright 2001-2024 Thales Group

CKM_SHA384_RSA_X9_31

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	128
Digest size	48
Key types	RSA
Algorithms	SHA384
Modes	None
Flags	Extractable | X9.31

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192

 Copyright 2001-2024 Thales Group

Block size	128
Digest size	48
Key types	RSA
Algorithms	SHA384
Modes	None
Flags	Extractable | X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA384_RSA_X9_31_NON_FIPS

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	128
Digest size	48
Key types	RSA
Algorithms	SHA384
Modes	None
Flags	Extractable | X9.31 | Non-FIPS X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA384_SM2DSA

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	128
Digest size	48
Key types	SM2
Algorithms	SHA384
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA512

Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	128
Digest size	64
Key types	None
Algorithms	SHA512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA512_EDDSA

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	128
Digest size	64
Key types	EDDSA
Algorithms	SHA512
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	128
Digest size	64
Key types	EDDSA
Algorithms	SHA512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA512_EDDSA_NACL

Firmware 7.8.4 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	456
Block size	128
Digest size	64
Key types	EDDSA
Algorithms	SHA512
Modes	None
Flags	Extractable

Firmware 7.8.1 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256

 Copyright 2001-2024 Thales Group

Block size	128
Digest size	64
Key types	EDDSA
Algorithms	SHA512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA512_HMAC

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	128
Digest size	64
Key types	Symmetric
Algorithms	SHA512
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	128
Digest size	64
Key types	Symmetric
Algorithms	SHA512
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA512_HMAC_GENERAL

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	128
Digest size	64
Key types	Symmetric
Algorithms	SHA512
Modes	HMAC
Flags	Extractable | Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify

 Copyright 2001-2024 Thales Group

Functions restricted from FIPS use	None
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	112
Minimum legacy key length for FIPS use (bits)	80
Maximum key length (bits)	4096
Block size	128
Digest size	64
Key types	Symmetric
Algorithms	SHA512
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA512_KEY_DERIVATION

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	128
Digest size	64
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHA512_RSA_PKCS

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	256
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	128
Digest size	64
Key types	RSA
Algorithms	SHA512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHA512_RSA_PKCS_PSS

Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	128
Digest size	64
Key types	RSA
Algorithms	SHA512
Modes	None
Flags	Extractable | PSS

 Copyright 2001-2024 Thales Group

CKM_SHA512_RSA_X9_31

Firmware 7.8.7 and Newer Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	Cannot sign
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192
Block size	128
Digest size	64
Key types	RSA
Algorithms	SHA512
Modes	None
Flags	Extractable | X9.31

Firmware 7.8.4 and Older Summary

FIPS approved?	Yes
Supported functions	Sign | Verify
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	1024
Maximum key length (bits)	8192

 Copyright 2001-2024 Thales Group

Block size	128
Digest size	64
Key types	RSA
Algorithms	SHA512
Modes	None
Flags	Extractable | X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA512_RSA_X9_31_NON_FIPS

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	8192
Block size	128
Digest size	64
Key types	RSA
Algorithms	SHA512
Modes	None
Flags	Extractable | X9.31 | Non-FIPS X9.31

 Copyright 2001-2024 Thales Group

CKM_SHA512_SM2DSA

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	128
Digest size	64
Key types	SM2
Algorithms	SHA512
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHAKE_128

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	168
Digest size	0
Key types	None
Algorithms	SHAKE_128
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Digest
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	168

Digest size	0
Key types	None
Algorithms	SHAKE_128
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHAKE_128_KEY_DERIVE

Firmware 7.7.0 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	168
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Derive
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	168

Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SHAKE_256

Firmware 7.7.0 and Newer Summary

FIPS approved?	Yes
Supported functions	Digest
Functions restricted from FIPS use	None
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	0
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	136
Digest size	0
Key types	None
Algorithms	SHAKE_256
Modes	None
Flags	Extractable

Firmware 7.4.2 Summary

FIPS approved?	No
Supported functions	Digest
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	136

Digest size	0
Key types	None
Algorithms	SHAKE_256
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SHAKE_256_KEY_DERIVE

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	136
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SM2_KEY_PAIR_GEN

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Generate Key Pair
Functions restricted from FIPS use	N/A
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	SM2
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SM2DSA

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	0
Digest size	0
Key types	SM2
Algorithms	SM2
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SM3

SM3 is a hash function published by the Chinese Commercial Cryptography Administration Office for the use of electronic authentication service system. The design of SM3 builds upon the design of the SHA-2 hash function, but introduces additional strengthening features. For Luna PCIe HSM 7s, the available mechanisms are CKM_SM3, the hash function, and CKM_SM3_KEY_DERIVATION, and CKM_HMAC_SM3.

Summary

FIPS approved?	No
Supported functions	Digest
Functions restricted from FIPS use	N/A
Minimum key length (bits)	0
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	0
Block size	64
Digest size	32
Key types	None
Algorithms	SM3
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SM3_HMAC

SM3 is a hash function published by the Chinese Commercial Cryptography Administration Office for the use of electronic authentication service system. The design of SM3 builds upon the design of the SHA-2 hash function, but introduces additional strengthening features. For Luna PCIe HSM 7s, the available mechanisms are CKM_SM3, the hash function, and CKM_SM3_KEY_DERIVATION, and CKM_SM3_HMAC.

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	SM3
Modes	HMAC
Flags	Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	SM3
Modes	HMAC
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SM3_HMAC_GENERAL

SM3 is a hash function published by the Chinese Commercial Cryptography Administration Office for the use of electronic authentication service system. The design of SM3 builds upon the design of the SHA-2 hash function, but introduces additional strengthening features. For Luna PCIe HSM 7s, the available mechanisms are CKM_SM3, the hash function, and CKM_SM3_KEY_DERIVATION, and CKM_SM3_HMAC.

> **TIP** Some mechanisms in this collection have both a "general" variant and a similarly named variant without "general" in the name. Per the PKCS#11 specification the _GENERAL variant of mechanism accepts a mechanism parameter that is used to define the length of the signature that is returned. The length can typically be any value between 1 and the length of the underlying HASH algorithm.
>
> The variants without _GENERAL do not accept any mechanism parameters and always return a fixed length signature; where the length is defined by the underlying HASH algorithm.

Firmware 7.8.1 and Newer Summary

Luna HSM Firmware 7.8.1 and newer supports zero-byte input to HMAC functions.

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	SM3
Modes	HMAC
Flags	Allow zero-length input

Firmware 7.8.0 and Older Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	SM3
Modes	HMAC
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SM3_KEY_DERIVATION

SM3 is a hash function published by the Chinese Commercial Cryptography Administration Office for the use of electronic authentication service system. The design of SM3 builds upon the design of the SHA-2 hash function, but introduces additional strengthening features. For Luna PCIe HSM 7s, the available mechanisms are CKM_SM3, the hash function, and CKM_SM3_KEY_DERIVATION, and CKM_HMAC_SM3.

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	64
Digest size	32
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SM3_SM2DSA

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	105
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	571
Block size	64
Digest size	32
Key types	SM2
Algorithms	SM3
Modes	None
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SM4_CBC

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SM4
Algorithms	SM4
Modes	CBC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SM4_CBC_PAD

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SM4
Algorithms	SM4
Modes	CBC_PAD
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SM4_ECB

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt | Wrap | Unwrap
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	16
Digest size	0
Key types	SM4
Algorithms	SM4
Modes	ECB
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SM4_KEY_GEN

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	0
Digest size	0
Key types	SM4
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SSL3_KEY_AND_MAC_DERIVE

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	384
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	384
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SSL3_MASTER_KEY_DERIVE

Summary

FIPS approved?	No
Supported functions	Derive
Functions restricted from FIPS use	N/A
Minimum key length (bits)	384
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	384
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SSL3_MD5_MAC

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	128
Block size	64
Digest size	16
Key types	Symmetric
Algorithms	MD5
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_SSL3_PRE_MASTER_KEY_GEN

Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	384
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	384
Block size	0
Digest size	0
Key types	None
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_SSL3_SHA1_MAC

Summary

FIPS approved?	No
Supported functions	Sign | Verify
Functions restricted from FIPS use	N/A
Minimum key length (bits)	160
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	160
Block size	64
Digest size	20
Key types	Symmetric
Algorithms	SHA
Modes	HMAC
Flags	Extractable

 Copyright 2001-2024 Thales Group

CKM_TUAK

> **NOTE** This mechanism can be used with HA with Luna HSM Client 10.4.0 and newer.

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	ECB
Flags	Allow zero-length input

 Copyright 2001-2024 Thales Group

CKM_TUAK_AUTS

> **NOTE** This mechanism can be used with HA with Luna HSM Client 10.4.0 and newer.

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	ECB
Flags	None

 Copyright 2001-2024 Thales Group

CKM_TUAK_RESYNC

> **NOTE** This mechanism can be used with HA with Luna HSM Client 10.4.0 and newer.

Firmware 7.4.2 and Newer Summary

FIPS approved?	No
Supported functions	Sign
Functions restricted from FIPS use	N/A
Minimum key length (bits)	128
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	256
Block size	16
Digest size	0
Key types	AES
Algorithms	AES
Modes	ECB
Flags	None

 Copyright 2001-2024 Thales Group

CKM_X9_42_DH_DERIVE

Summary

FIPS approved?	Yes
Supported functions	Derive
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	X9_42_DH
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_X9_42_DH_HYBRID_DERIVE

Summary

FIPS approved?	Yes
Supported functions	Derive
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	X9_42_DH
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_X9_42_DH_KEY_PAIR_GEN

Summary

FIPS approved?	Yes
Supported functions	Generate Key Pair
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	X9_42_DH
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_X9_42_DH_PARAMETER_GEN

Firmware 7.8.0 Summary

FIPS approved?	No
Supported functions	Generate Key
Functions restricted from FIPS use	N/A
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	X9_42_DH
Algorithms	None
Modes	None
Flags	None

Firmware 7.7.2 and Older Summary

FIPS approved?	Yes
Supported functions	Generate Key
Functions restricted from FIPS use	None
Minimum key length (bits)	1024
Minimum key length for FIPS use (bits)	2048
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096

Block size	0
Digest size	0
Key types	X9_42_DH
Algorithms	None
Modes	None
Flags	None

 Copyright 2001-2024 Thales Group

CKM_XOR_BASE_AND_DATA_W_KDF

Firmware 7.4.0 and Older Summary

FIPS approved?	No
Supported functions	Encrypt | Decrypt
Minimum key length (bits)	8
Minimum key length for FIPS use (bits)	N/A
Minimum legacy key length for FIPS use (bits)	N/A
Maximum key length (bits)	4096
Block size	0
Digest size	0
Key types	Symmetric
Algorithms	None
Modes	OFB
Flags	None

 Copyright 2001-2024 Thales Group

CHAPTER 6: Using the Luna SDK

This chapter describes how to use the SDK to develop applications that exercise the HSM. It contains the following topics:

> "Libraries and Applications" below
> "Object handles and labels" on page 587
> "Application IDs" on page 582
> "Named Curves and User-Defined Parameters" on page 588
> "ECDH with Key Derive Function" on page 599
> "ECIES general" on page 604
> "ECIES for 5G" on page 607
> "Supported ECC Curves" on page 595
> "Capability and Policy Configuration Control Using the Luna API" on page 610
> "Connection Timeout" on page 613

Libraries and Applications

This section explains how to make the Chrystoki library available to the other components of the Luna Software Development Kit.

An application has no knowledge of which library is to be loaded nor does the application know the library's location. For these reasons, a special scheme must be employed to tell the application, while it is running, where to find the library. The next paragraphs describe how applications connect to Chrystoki.

Luna SDK Applications General Information

All applications provided in Luna PCIe HSM 7 Software Development Kit have been compiled with a component called CkBridge, which uses a configuration file to find the library it is intended to load. Ckbridge first uses the environment variable "ChrystokiConfigurationPath" to locate the corresponding configuration file. If this environment variable is not set, it attempts to locate the configuration file in a default location depending on the product platform (/etc on Unix, and c:\Program Files\SafeNet\LunaClient on Windows).

Configuration files differ from one platform to the next - refer to the appropriate sub-section for the operating system and syntax applicable to your development platform.

Windows

In Windows, an initialization file called **crystoki.ini** specifies which library is to be loaded. The syntax of this file is standard to Windows.

The following example shows proper configuration files for Windows:

```
[Chrystoki2]
LibNT=C:\Program Files\SafeNet\LunaClient\cryptoki.dll
[LunaSA Client]
SSLConfigFile=C:\Program Files\SafeNet\LunaClient\openssl.cnf
ReceiveTimeout=20000
NetClient=1
ServerCAFile=C:\Program Files\SafeNet\LunaClient\cert\server\CAFile.pem
ClientCertFile=C:\Program Files\SafeNet\LunaClient\cert\client\ClientNameCert.pem
ClientPrivKeyFile=C:\Program Files\SafeNet\LunaClient\cert\client\ClientNameKey.pem
[Luna]
DefaultTimeOut=500000
PEDTimeout1=100000
PEDTimeout2=200000
PEDTimeout3=10000
[CardReader]
RemoteCommand=1
```

> **CAUTION!** Never insert TAB characters into the crystoki.ini (Windows) or chrystoki.conf (UNIX) file.

UNIX

In UNIX, a configuration file called "Chrystoki.conf" is used to guide CkBridge in finding the appropriate library.

The configuration file is a regular text file with a special format. It is made up of a number of sections, each section containing one or multiple entries. The following example shows a typical UNIX configuration file:

```
Chrystoki2 =
{
LibUNIX=/usr/lib/libCryptoki2.so;
}
Luna = {
DefaultTimeOut=500000;
PEDTimeout1=100000;
PEDTimeout2=200000;
PEDTimeout3=10000;
KeypairGenTimeOut=2700000;
CloningCommandTimeOut=300000;
}
CardReader = {
RemoteCommand=1;
}
LunaSA Client = {
NetClient = 1;
ServerCAFile = /usr/safenet/lunaclient/cert/server/CAFile.pem;
ClientCertFile = /usr/safenet/lunaclient/cert/client/ClientNameCert.pem;
ClientPrivKeyFile = /usr/safenet/lunaclient/cert/client/ClientNameKey.pem;
SSLConfigFile = /usr/safenet/lunaclient/bin/openssl.cnf;
ReceiveTimeout = 20000;
}
```

The shared object "libcrystoki2.so" is a library supporting version 2.2.0 of the PKCS#11 standard.

> **CAUTION!** Never insert TAB characters into the crystoki.ini (Windows) or crystoki.conf (UNIX) file.

Compiler Tools

Tools used for Luna development are platform specific tools/development environments, where applicable (e.g., Visual C++ on Windows, or Workshop on Solaris). Current version information is provided in the Customer Release Notes.

> **NOTE** Contact Thales for information about the availability of newer versions of compilers.

Using CKlog

Luna Software Development Kit provides a facility which can record all interactions between an application and the PKCS#11-compliant library. It allows a developer to debug an application by viewing what the library receives.

This tool is known as the Cryptoki Logging Facility or cklog. Cklog is a shim library that an application accesses when seeking our PKCS#11 library. When cklog receives a call it does not service the request. Instead, it logs the call to a file and passes the request to the originally intended library. Configuration consists of redirecting the LibNT (Windows) or LibUNIX (Linux/UNIX) library locations, and setting some additional configuration options as summarized after the examples, below.

To configure CkLog:

Perform these steps:

1. Direct the application to use the cklog library instead of the regular Chrystoki library. Do this by modifying the configuration file to instruct CkBridge to load the Cklog library.

   Windows

   ```
   [Chrystoki2]
   LibNT=c:\Program Files\SafeNet\LunaClient\cklog201.dll
   ```

   Linux/UNIX

   ```
   Chrystoki2 =
   {
   LibUNIX=/usr/lib/libcklog2.so;
   ```

2. Instruct the cklog library where to access the regular cryptoki library.

   Windows

   ```
   [CkLog2]
   LibNT=c:\Program Files\SafeNet\LunaClient\cryptoki.dll
   ```

   Linux/UNIX

   ```
   CkLog2 =
   {
   LibUNIX=/usr/lib/libCryptoki2.so;
   }
   ```

3. Add appropriate entries to the CkLog2 section for the desired level of operation. See the examples and explanations of entries, below.

Windows Example

The following example shows a typical initialization file under Windows where cklog is in use:

```
[Chrystoki2]
LibNT=c:\Program Files\SafeNet\LunaClient\cklog201.dll
[CkLog2]
LibNT=c:\Program Files\SafeNet\LunaClient\cryptoki.dll
Enabled=1
File=c:\Program Files\SafeNet\LunaClient\cklog2.txt
Error=c:\Program Files\SafeNet\LunaClient\error2.txt
FileSize=100
NewFormat=1
LoggingMask=ALL_FUNC
```

UNIX Example

The following example shows a typical configuration file under UNIX where cklog is in use:

```
Chrystoki2 =
{
LibUNIX=/usr/lib/libcklog2.so;
}
CkLog2 =
{
LibUNIX=/usr/lib/libCryptoki2.so;
Enabled=1;
File=/tmp/cklog.txt;
FileSize=100
Error=/tmp/error.txt;
NewFormat=1;
LoggingMask=ALL_FUNC;
}
```

Here are descriptions of entries that might be applicable:

> LibNT - references to a Cryptoki library for Windows.

> LibUNIX - references to a Cryptoki library for UNIX.

> Enabled - 0 or 1. Allows turning the logging facility off or on.

> File - references the file to which the requests should be logged.

> FileSize - in MB is the size that triggers log-file rotation - when the file reaches the indicated value, it is renamed to indicate the current date and time (like cklog.txt-<YYMMDD-hhmmss>), and a new cklog.txt file begins accumulating log entries.

> Error - references a file where the logging facility can record fatal errors.

> NewFormat - affects the log output format. Possible values:

- **0**: standard log output format
- **1** (default): compact output format preferred by Thales Customer Support

  ```
  2023-06-27 15:38:48 07724--424540352:FINIInitialize CKR_OK(10ms) {}
  ```

- **2**: compact output format, including the amount of real time that Luna HSM Client took to process the request (`operation latency`, in ms). This option requires Luna HSM Client 10.6.0 or newer.

  ```
  2023-06-27 15:38:48 07724--424540352:FINIInitialize CKR_OK(10ms) (operation latency:
  40ms) {}
  ```

Selective Logging

When logging is turned on, all functions are logged, by default. If you wish to restrict logging to particular functions of interest only, you can edit the “LoggingMask=” parameter in the crystoki.ini [Windows] or Chrystoki.conf [UNIX] file to include flags for the desired logging.

LoggingMask= Flags

Here is the list of possible flags for cklog:

Flag	Description
GEN_FUNC	General functions
SLOT_TOKEN_FUNC	Slot/Token related functions
SESSION_FUNC	Session related functions
OBJ_MNGMNT_FUNC	Object Management functions
ENC_DEC_FUNC	Encrypt/Decrypt related functions
DIGEST_FUNC	Digest Related functions
SIGN_VERIFY_FUNC	Signing/Verifying related functions
KEY_MNGMNT_FUNC	Key Management related functions
MISC_FUNC	Misc functions
CHRYSALIS_FUNC	Luna Extensions functions
ALL_FUNC	All functions logged;

You can mix and match any or all of the flags, using the “|” operator. For example, the following:

LoggingMask=GEN_FUNC | SLOT_TOKEN_FUNC | ENC_DEC_FUNC | SIGN_VERIFY_FUNC;

would be valid.

> **NOTE** You can use the flags in any order. Using the ALL_FUNC flag overrides any other flag. If you have the “LoggingMask=” parameter, with NO flags set, then nothing is logged. If logging capability is enabled (cklog), but there is no “LoggingMask=” line, then default behavior prevails and everything is logged.

Application IDs

Within Chrystoki, each application has an application ID that is generated when the application starts, but which can also be specified in the configuration file. An application ID was historically a 64-bit integer, normally specified in two 32-bit parts. As of Luna HSM Firmware 7.7.0, application IDs are 128 bits. When an application

invokes **C_Initialize**, the Chrystoki library automatically generates a default application ID. The default value is based on the application's process ID, so different applications will always have different application IDs. The application ID is also associated with each session created by the application.

Compatibility Old with New

To address the issue of compatibility between combinations of older and newer clients and firmware, the following table summarizes the behavior.

	Old Client / New Firmware	**New Client / Old Firmware**
AppID	> The client will send its 8 byte App ID to the firmware. > The client will not include the App ID in session based commands. > The firmware will reject open session commands on partitions of HSMs where f/w >= 7.7.0, with CKR_LEGACY_CLIENT. The client needs to be updated to access such partitions.	> The client will truncate its 16 byte App ID to 8 bytes and set the high bit in the truncated App ID. > The client will not include the App ID in session based commands.
API	> The client will have access only to the old API	> The client has access to the old and new APIs. > The old API is still available to support old applications. > The old API will set 8 bytes of the App ID to the user value and will set the other 8 bytes to 0. > The new API is preferred and the old API is deprecated.

 Copyright 2001-2024 Thales Group

	Old Client / New Firmware	New Client / Old Firmware
Chrystoki.conf / crystoki.ini	> The client will have access only to the existing App ID fields: “AppIdMinor” and “AppIdMajor”. > The new “AppId” field will be ignored if it’s present.	> The client can use either the new or the old App ID fields. > The old “AppIdMinor” and “AppIdMajor” will work as they did before. > The “AppId” field is given priority over “AppIdMajor” and “AppIdMinor”.

Shared Login State and Application IDs

PKCS#11 specifies that sessions within an application (a single address space and all threads that execute within it) share a login state, meaning that if one session is logged in, all are logged in. If one logs out, all are logged out. Thus, if process A spawns multiple threads, and all of those threads open sessions on Token #1, then all of those sessions share a login state. If process B also has sessions open on Token #1, they are independent from the sessions of process A. The login state of process B sessions does not affect process A sessions, unless they conflict with one another (e.g. process A logs in as USER when process B is already logged in as SO).

Within Chrystoki and Luna tokens, login states are shared by sessions with the same application ID. This means that sessions within an application share a login state, but sessions across separate applications do not. However, Chrystoki does provides functionality allowing applications to alter their application IDs, so that separate applications can share a login state.

> **CAUTION!** The ability to share login states through the use of application IDs is a legacy feature. It can eliminate the need for repeated Luna PED authentication across multiple applications, but this is not ideal for security reasons. To avoid these risks, it is recommended to use auto-activation in conjunction with a challenge password instead (see Activation on Multifactor Quorum-Authenticated Partitions).

To change application IDs manually using the LunaCM **appid** command, see appid.

Why Share Login State Between Applications?

For most applications, this is unnecessary. If an application consists of a single perpetual process, unshared session states are sufficient. If the application supports multiple, separately-validated processes, unshared session states are also sufficient. Generally, applications that validate (login) separately are more secure.

It is only necessary to share login state between processes if the following conditions are true:

> the application designer wants to require only one login action by the user

> the application consists of multiple processes, each with their own sessions

Login State Sharing Overview

The simplest form of the Chrystoki state-sharing functionality is the **CA_SetApplicationID** function. This function should be invoked after **C_Initialize**, but before any sessions are opened. Two separate applications can use this function to set their application IDs to the same value, and thus allow them to share login states between their sessions.

Alternatively, set the **AppIdMajor** and **AppIdMinor** fields in the Misc section of the Chrystoki configuration file. This causes default application IDs for all applications to be generated from these fields, rather than from each application's process ID. When these fields are set, all applications on a host system will share login state between their sessions, unless individual applications use the **CA_SetApplicationID** function.

Example

A sample configuration file (**crystoki.ini** for Windows) using explicit application IDs is duplicated here:

```
[Chrystoki2]
LibNT=D:\Program Files\SafeNet\LunaClient\cryptoki.dl
[Luna]
DefaultTimeOut=500000
PEDTimeout1=100000
PEDTimeout2=200000
[CardReader]
RemoteCommand=1
[Misc]
AppIdMajor=2
AppIdMinor=4
```

Problems may still arise. When all sessions of a particular application ID are closed, that application ID reverts to a dormant state. When another session for that application ID is created, the application ID is recreated, but always in the logged-out state, regardless of the state it was in when it went dormant.

For example, consider an application where a parent process sets its application ID, opens a session, logs the session in, then closes the session and terminates. Several child processes then set their application IDs, open sessions and try to use them. However, since the application ID went dormant when the parent process closed its session, the child processes find their sessions logged out. The logged-in state of the parent process's session was lost when it closed its session.

The **CA_OpenApplicationID** function can ensure that the login state of an application ID is maintained, even when no sessions belonging to that application ID exist. When **CA_OpenApplicationID** is invoked, the application ID is tagged so that it never goes dormant, even if no open sessions exist.

> **NOTE** Running **CA_OpenApplication_ID** does not set the application ID for the current process. You must first explicitly run **CA_SetApplicationID** to do this.

Login State Sharing Functions

Use the following functions to configure and manage login state sharing:

> **NOTE** The following login state sharing functions cannot be used with STC-enabled slots.

CA_SetApplicationID

```
CK_RV CK_ENTRY CA_SetApplicationID(
        CK_ULONG ulHigh,
        CK_ULONG ulLow
```

```
);
```

The **CA_SetApplicationID** function allows an application to set its own application ID, rather than letting the application ID be generated automatically from the application's process ID. **CA_SetApplicationID** should be invoked after **C_Initialize**, but before any session manipulation functions are invoked. If **CA_SetApplicationID** is invoked after sessions have been opened, results will be unpredictable.

CA_SetApplicationID always returns CKR_OK.

CA_OpenApplicationID

```
CK_RV CK_ENTRY CA_OpenApplicationID(
        CK_SLOT_ID slotID,
        CK_ULONG ulHigh,
        CK_ULONG ulLow
);
```

The **CA_OpenApplicationID** function forces a given application ID on a given token to remain active, even when all sessions belonging to the application ID have been closed. Normally, an application ID on a token goes dormant when the last session that belongs to the application ID is closed. When an application ID goes dormant, login state is lost, so when a new session is created within the application ID, it starts in the logged-out state. However, if **CA_OpenApplicationID** is used, the application ID is prevented from going dormant, so login state is maintained even when all sessions for an application ID are closed.

> **NOTE** Running **CA_OpenApplication_ID** does not set the application ID for the current process. You must first explicitly run **CA_SetApplicationID** to do this.

CA_OpenApplicationID can return CKR_SLOT_ID_INVALID or CKR_TOKEN_NOT_PRESENT.

CA_CloseApplicationID

```
CK_RV CK_ENTRY CA_CloseApplicationID(
        CK_SLOT_ID slotID,
        CK_ULONG ulHigh,
        CK_ULONG ulLow
);
```

The **CA_CloseApplicationID** function removes the property of an application ID that prevents it from going dormant. **CA_CloseApplicationID** also closes any open sessions owned by the given application ID. Thus, when **CA_CloseApplicationID** returns, all open sessions owned by the given application ID have been closed and the application ID has gone dormant.

CA_CloseApplicationID can return CKR_SLOT_ID_INVALID or CKR_TOKEN_NOT_PRESENT.

Application ID Examples

The following code fragments show how two separate applications might share a single application ID:

```
app 1:        app 2:
C_Initialize()
CA_SetApplicationID(3,4)
C_OpenSession()
C_Login()
              C_Initialize()
              CA_SetApplicationID(3,4)
              C_OpenSession()
              C_GetSessionInfo()
              // Session info shows session
```

```
              // already logged in.
              <perform work, no login
              necessary>

C_Logout()
              C_GetSessionInfo()
              // Session info shows session
              // logged out.

C_CloseSession()
              C_CloseSession()
C_Finalize()
              C_Finalize()
```

The following code fragments show how one process might login for others:

Setup app:

```
C_Initialize()
CA_SetApplicationID(7,9)
CA_OpenApplicationID(slot,7,9)
C_OpenSession(slot)
C_Login()
C_CloseSession()
```

Spawn many child applications:

```
C_Finalize()
```

Terminate each child app:

```
              C_Initialize()
              CA_SetApplicationID(7,9)
              C_OpenSession(slot)
              <perform work, no login necessary>
```

Takedown app:

Terminate child applications:

```
              C_CloseSession()
              C_Finalize()
C_Initialize()
CA_CloseApplicationID(slot,7,9)
C_Finalize()
```

Object handles and labels

Objects in an HSM application partition are assigned unique handles when a session is initialized. A handle generally persists for the duration of the session, or until an object is deleted from a session.

From the PKCS#11 standard:

"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."

Where this association might not be applicable is when multiple Luna application partitions are gathered in an HA group. The same object on two different partitions can have different object handles. The HA functionality of the Luna HSM Client maps member partitions and their in-common contained objects in a virtual partition that is the exclusive application access point for the group. See High-Availability Groups. We recommend that, when deploying an HA group, you set hagroup haonly, which hides HA-group member partitions, and displays only the virtual partition in the slot list. Addressing member partitions directly, to make changes, can disrupt HA function. The ephemeral list of objects used by the virtual partition is recreated with each new session, and updated during the session. Within a session, that list is a cache of those objects requested during the session, making for faster look-up of objects that are used repeatedly.

Labels can be applied to objects to assist in identifying them across separate sessions and across multiple partitions; a label persists for the life of the object, and is therefore a more reliable method of identifying objects in those situations.

Named Curves and User-Defined Parameters

Luna PCIe HSM 7 is a PKCS#11-oriented device. The HSM firmware statically defines NIST named curve OIDs and curve parameters by default. You can also define other non-NIST curve OIDS and parameters such as Brainpool. Luna PCIe HSM 7 can decode and use the ecParameters structure for key generation, signing, and verification.

Curve Validation Limitations

The HSM can validate the curve parameters, but domain parameter validation guarantees only the consistency/sanity of the parameters and the most basic, well-known security properties. The HSM has no way of judging the quality of a user-specified curve.

It is therefore important that you perform Known Answer Tests to verify the operation of the HSM for any new Domain Parameter.set. To maintain NIST-FIPS compatibility the feature is selectively enabled with the feature being disabled by default. Therefore the Administrator must ‘opt-in’ by actively choosing to enable the appropriate HSM policy. Among other effects, this causes the HSM to display a shell/console message to the effect that the HSM is not operating in FIPS mode.

Storing Domain Parameters

Under PKCS#11 v2.20, Domain Parameters are stored in object attribute CKA_EC_PARAMS. The value of this parameter is the DER encoding of an ANSI X9.62 Parameters value.

```
Parameters ::= CHOICE {
ecParameters ECParameters,
namedCurve CURVES.&id({CurveNames}),
implicitlyCA NULL
}
```

Because PKCS#11 states that the implicitlyCA is not supported by cryptoki, therefore the CKA_EC_PARAMS attribute must contain the encoding of an ecParameters or a namedCurve. Cryptoki holds ECC key pairs in separate Private and Public key objects. Each object has its own CKA_EC_PARAMS attribute which must be provided when the object is created and cannot be subsequently changed.

Cryptoki also supports CKO_DOMAIN_PARAMETERS objects. These store Domain Parameters but perform no cryptographic operations. A Domain Parameters object is really only for storage. To generate a key pair, you must extract the attributes from the Domain Parameters object and insert them in the CKA_EC_PARAMS attribute of the Public key template. Cryptoki can create new ECC Public and Private key objects and Domain Parameters objects in the following ways:

> Objects can be directly stored using the C_CreateObject command.

> Public and private key objects can be generated internally with the C_GenerateKeyPair command and the CKM_EC_KEY_PAIR_GEN mechanism.

> Objects can be imported in encrypted form using C_UnwrapKey command.

Using Domain Parameters

ECC keys may be used for Signature Generation and Verification with the CKM_ECDSA and CKM_ECDSA_SHA1 mechanism and Encryption and Decryption with the CKM_ECIES mechanism. These three mechanism are enhanced so that they fetch the Domain Parameters from the CKA_EC_PARAMS attribute for both ecParameters and namedCurve choice and not just namedCurve choice.

User Friendly Encoder

Using ECC with Cryptoki to create or generate ECC keys requires that the CKA_EC_PARAMS attribute be specified. This is a DER encoded binary array. Usually in public documents describing ECC curves the Domain Parameters are specified as a series of printable strings so the programmer faces the problem of converting these to the correct format for Cryptoki use.

The cryptoki library is extended to support functions called CA_EncodeECPrimeParams and CA_EncodeECChar2Params which allow an application to specify the parameter details of a new curve. These functions implement DER encoders to build the CKA_EC_PARAMS attribute from large integer presentations of the Domain Parameter values.

Refer to "Sample Domain Parameter Files" on page 592.

Application Interfaces

CA_EncodeECPrimeParams

```
#include "cryptoki.h"

CK_RV CA_ EncodeECPrimeParams (
            CK_BYTE_PTR DerECParams,CK_ULONG_PTR DerECParams Len
            CK_BYTE_PTR prime,CK_USHORT primelen,
            CK_BYTE_PTR a,CK_USHORT alen,
            CK_BYTE_PTR b,CK_USHORT blen,
            CK_BYTE_PTR seed,CK_USHORT seedlen,
            CK_BYTE_PTR x,CK_USHORT xlen,
            CK_BYTE_PTR y,CK_USHORT ylen,
            CK_BYTE_PTR order,CK_USHORT orderlen,
            CK_BYTE_PTR cofactor,CK_USHORT cofactorlen,
);

Do DER enc of ECC Domain Parameters Prime
```

Parameters

DerECParams	Resultant Encoding (length prediction supported if NULL).
DerECParamsLen	Buffer len/Length of resultant encoding
prime	Prime modulus
primelen	Prime modulus len
a	Elliptic Curve coefficient a
alen	Elliptic Curve coefficient a length
b	Elliptic Curve coefficient b
blen	Elliptic Curve coefficient b length
seed	Seed (optional may be NULL)
seedlen	Seed length
x	Elliptic Curve point X coord
xlen	Elliptic Curve point X coord length
y	Elliptic Curve point Y coord
ylen	Elliptic Curve point Y coord length
order	Order n of the Base Point
orderlen	Order n of the Base Point length
cofactor	The integer h = #E(Fq)/n (optional)
cofactorlen	h length
Return	Status of operation. CKR_OK if ok.

CA_EncodeECChar2Params

```
#include "cryptoki.h"
CK_RV CA_EncodeECChar2Params(
            CK_BYTE_PTR DerECParams,   CK_ULONG_PTR DerECParams Len
            CK_USHORT m,
            CK_USHORT k1,
            CK_USHORT k2,
            CK_USHORT k3,
            CK_BYTE_PTR a,CK_USHORT alen,
            CK_BYTE_PTR b,CK_USHORT blen,
```

```
            CK_BYTE_PTR seed,CK_USHORT seedlen,
            CK_BYTE_PTR x,CK_USHORT xlen,
            CK_BYTE_PTR y,CK_USHORT ylen,
            CK_BYTE_PTR order,CK_USHORT orderlen,
            CK_BYTE_PTR cofactor,CK_USHORT cofactorlen,
            );
```

Do DER enc of ECC Domain Parameters 2^M

Parameters

DerECParams	Resultant Encoding (length prediction supported if NULL).
DerECParamsLen	Buffer len/Length of resultant encoding
M	Degree of field
k1	parameter in trinomial or pentanomial basis polynomial
k2	parameter in pentanomial basis polynomial
k3	parameter in pentanomial basis polynomial
a	Elliptic Curve coefficient a
alen	Elliptic Curve coefficient a length
b	Elliptic Curve coefficient b
blen	Elliptic Curve coefficient b length
seed	Seed (optional may be NULL)
seedlen	Seed length
x	Elliptic Curve point X coord
xlen	Elliptic Curve point X coord length
y	Elliptic Curve point Y coord
ylen	Elliptic Curve point Y coord length
order	Order n of the Base Point
orderlen	Order n of the Base Point length
cofactor	The integer h = #E(Fq)/n (optional)

cofactorlen	h length
Return	Status of operation. CKR_OK if ok.

Sample Domain Parameter Files

The following examples show some sample domain parameter files.

prime192v1

```
#
#This file describes the domain parameters of an EC curve
#
#File contains lines of text. All lines not of the form key=value are ignored.
#All values must be Hexidecimal numbers except m, k, k1, k2 and k3 which are decimal.
#Lines starting with '#' are comments.
#
#Keys recognised for fieldID values are -
#p          - only if the Curve is based on a prime field
#m          - only if the curve is based on a 2^M field
#k1, k2, k3 - these three only if 2^M field
#
#You should have these combinations of fieldID values -
#p          - if Curve is based on a prime field
#m,k1,k2,k3 - if curve is based on 2^M
#
#These are the values common to prime fields and polynomial fields.
#a          - field element A
#b          - field element B
#s          - this one is optional
#x          - field element Xg of the point G
#y          - field element Yg of the point G
#q          - order n of the point G
#h          - (optional) cofactor h
#
#
# Curve name prime192v1
p    = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF
a    = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC
b    = 64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1
s    = 3045AE6FC8422F64ED579528D38120EAE12196D5
x    = 188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012
y    = 07192B95FFC8DA78631011ED6B24CDD573F977A11E794811
q    = FFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831
h    = 1
```

C2tnB191v1

```
#
#This file describes the domain parameters of an EC curve
#
#File contains lines of text. All lines not of the form key=value are ignored.
#All values must be Hexidecimal numbers except m, k, k1, k2 and k3 which are decimal.
#Lines starting with '#' are comments.
#
#Keys recognised for fieldID values are -
```

```
#p          - only if the Curve is based on a prime field
#m          - only if the curve is based on a 2^M field
#k1, k2, k3 - these three only if 2^M field
#
#You should have these combinations of fieldID values -
#p          - if Curve is based on a prime field
#m,k1,k2,k3 - if curve is based on 2^M
#
#
#These are the values common to prime fields and polynomial fields.
#a          - field element A
#b          - field element B
#s          - this one is optional
#x          - field element Xg of the point G
#y          - field element Yg of the point G
#q          - order n of the point G
#h          - (optional) cofactor h
#
#
# Curve name C2tnB191v1
m     = 191
k1    = 9
k2    = 0
k3    = 0
a     = 2866537B676752636A68F56554E12640276B649EF7526267
b     = 2E45EF571F00786F67B0081B9495A3D95462F5DE0AA185EC
x     = 36B3DAF8A23206F9C4F299D7B21A9C369137F2C84AE1AA0D
y     = 765BE73433B3F95E332932E70EA245CA2418EA0EF98018FB
q     = 40000000000000000000000004A20E90C39067C893BBB9A5
```

brainpoolP160r1

```
#
#This file describes the domain parameters of an EC curve
#
#File contains lines of text. All lines not of the form key=value are ignored.
#All values must be Hexidecimal numbers except m, k, k1, k2 and k3 which are decimal.
#Lines starting with '#' are comments.
#
#Keys recognised for fieldID values are -
#p          - only if the Curve is based on a prime field
#m          - only if the curve is based on a 2^M field
#k1, k2, k3 - these three only if 2^M field
#
#You should have these combinations of fieldID values -
#p          - if Curve is based on a prime field
#m,k1,k2,k3 - if curve is based on 2^M
#
#These are the values common to prime fields and polynomial fields.
#a          - field element A
#b          - field element B
#s          - this one is optional
#x          - field element Xg of the point G
#y          - field element Yg of the point G
#q          - order n of the point G
#h          - (optional) cofactor h
#
#
```

```
# Curve name brainpoolP160r1

p      = E95E4A5F737059DC60DFC7AD95B3D8139515620F
a      = 340E7BE2A280EB74E2BE61BADA745D97E8F7C300
b      = 1E589A8595423412134FAA2DBDEC95C8D8675E58
x      = BED5AF16EA3F6A4F62938C4631EB5AF7BDBCDBC3
y      = 1667CB477A1A8EC338F94741669C976316DA6321
q      = E95E4A5F737059DC60DF5991D45029409E60FC09
h      = 1
```

brainpoolP512r1

```
#
#This file describes the domain parameters of an EC curve
#
#File contains lines of text. All lines not of the form key=value are ignored.
#All values must be Hexidecimal numbers except m, k, k1, k2 and k3 which are decimal.
#Lines starting with '#' are comments.
#
#Keys recognised for fieldID values are -
#p          - only if the Curve is based on a prime field
#m          - only if the curve is based on a 2^M field
#k1, k2, k3 - these three only if 2^M field
#
#You should have these combinations of fieldID values -
#p          - if Curve is based on a prime field
#m,k1,k2,k3 - if curve is based on 2^M
#
#These are the values common to prime fields and polynomial fields.
#a          - field element A
#b          - field element B
#s          - this one is optional
#x          - field element Xg of the point G
#y          - field element Yg of the point G
#q          - order n of the point G
#h          - (optional) cofactor h
#
#
# Curve name brainpoolP512r1


p=AADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639CCA703308717D4D9B009BC66842AECDA12AE6A38
0E62881FF2F2D82C68528AA6056583A48F3

a=7830A3318B603B89E2327145AC234CC594CBDD8D3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1C98B9AC
8B57F1117A72BF2C7B9E7C1AC4D77FC94CA

b=3DF91610A83441CAEA9863BC2DED5D5AA8253AA10A2EF1C98B9AC8B57F1117A72BF2C7B9E7C1AC4D77FC94CADC083
E67984050B75EBAE5DD2809BD638016F723

x=81AEE4BDD82ED9645A21322E9C4C6A9385ED9F70B5D916C1B43B62EEF4D0098EFF3B1F78E2D0D48D50D1687B93B97
D5F7C6D5047406A5E688B352209BCB9F822

y=7DDE385D566332ECC0EABFA9CF7822FDF209F70024A57B1AA000C55B881F8111B2DCDE494A5F485E5BCA4BD88A276
3AED1CA2B2FA8F0540678CD1E0F3AD80892
```

```
q=AADD9DB8DBE9C48B3FD4E6AE33C9FC07CB308DB3B3C9D20ED6639CCA70330870553E5C414CA92619418661197FAC1
0471DB1D381085DDADDB58796829CA90069
h      = 1
```

Bad Parameter File

```
#
#This file describes the domain parameters of an EC curve
#
#File contains lines of text. All lines not of the form key=value are ignored.
#All values must be Hexidecimal numbers except m, k, k1, k2 and k3 which are decimal
#Lines starting with '#' are comments.
#
#Keys recognised for fieldID values are -
#p          - only if the Curve is based on a prime field
#m          - only if the curve is based on a 2^M field
#k1, k2, k3 - these three only if 2^M field
#
#You should have these combinations of fieldID values -
#p          - if Curve is based on a prime field
#m,k1,k2,k3 - if curve is based on 2^M
#
#These are the values common to prime fields and polynomial fields.
#a          - field element A
#b          - field element B
#s          - this one is optional
#x          - field element Xg of the point G
#y          - field element Yg of the point G
#q          - order n of the point G
#h          - (optional) cofactor h
#
# Curve name prime192vx

p    = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF
a    = FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC
b    = 64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B13
s    = 34545567685743523457
x    = 188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012
y    = 07192B95FFC8DA78631011ED6B24CDD573F977A11E794811
q    = FFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831
h    = 12323435765786
```

Supported ECC Curves

The following table lists all supported Elliptic Curve Cryptography (ECC) curves and their Object Identifiers (OID, expressed in dot notation and byte format).

> **NOTE** When **HSM policy 12:** Allow non-FIPS algorithms is disabled (FIPS mode):
> - curves with length 224-bits or greater can be used for all operations
> - curves >= 160-bits and < 224-bits can be used for signature verification only
> - curves less than 160-bits are not permitted
>
> These restrictions comply with the recommendations in NIST SP 800-131A Rev2.

Curve Name(s)	OID (dot)	OID (byte)	Security Strength (bits)
brainpoolP160r1	1.3.36.3.3.2.8.1.1.1	06 09 2B 24 03 03 02 08 01 01 01	80
brainpoolP160t1	1.3.36.3.3.2.8.1.1.2	06 09 2B 24 03 03 02 08 01 01 02	80
brainpoolP192r1	1.3.36.3.3.2.8.1.1.3	06 09 2B 24 03 03 02 08 01 01 03	96
brainpoolP192t1	1.3.36.3.3.2.8.1.1.4	06 09 2B 24 03 03 02 08 01 01 04	96
brainpoolP224r1	1.3.36.3.3.2.8.1.1.5	06 09 2B 24 03 03 02 08 01 01 05	112
brainpoolP224t1	1.3.36.3.3.2.8.1.1.6	06 09 2B 24 03 03 02 08 01 01 06	112
brainpoolP256r1	1.3.36.3.3.2.8.1.1.7	06 09 2B 24 03 03 02 08 01 01 07	128
brainpoolP256t1	1.3.36.3.3.2.8.1.1.8	06 09 2B 24 03 03 02 08 01 01 08	128
brainpoolP320r1	1.3.36.3.3.2.8.1.1.9	06 09 2B 24 03 03 02 08 01 01 09	160
brainpoolP320t1	1.3.36.3.3.2.8.1.1.10	06 09 2B 24 03 03 02 08 01 01 0a	160
brainpoolP384r1	1.3.36.3.3.2.8.1.1.11	06 09 2B 24 03 03 02 08 01 01 0b	192
brainpoolP384t1	1.3.36.3.3.2.8.1.1.12	06 09 2B 24 03 03 02 08 01 01 0c	192
brainpoolP512r1	1.3.36.3.3.2.8.1.1.13	06 09 2B 24 03 03 02 08 01 01 0d	256
brainpoolP512t1	1.3.36.3.3.2.8.1.1.14	06 09 2B 24 03 03 02 08 01 01 0e	256
c2pnb163v1 (X9.62 c2pnb163v1)	1.2.840.10045.3.0.1	06 08 2A 86 48 CE 3D 03 00 01	81
c2pnb163v2 (X9.62 c2pnb163v2)	1.2.840.10045.3.0.2	06 08 2A 86 48 CE 3D 03 00 02	81
c2pnb163v3 (X9.62 c2pnb163v3)	1.2.840.10045.3.0.3	06 08 2A 86 48 CE 3D 03 00 03	81
c2pnb176w1 (X9.62 c2pnb176w1) **c2pnb176v1** (X9.62 c2pnb176v1)	1.2.840.10045.3.0.4	06 08 2A 86 48 CE 3D 03 00 04	88
c2pnb208w1 (X9.62 c2pnb208w1)	1.2.840.10045.3.0.10	06 08 2A 86 48 CE 3D 03 00 0A	104
c2pnb272w1 (X9.62 c2pnb272w1)	1.2.840.10045.3.0.16	06 08 2A 86 48 CE 3D 03 00 10	136

Curve Name(s)	OID (dot)	OID (byte)	Security Strength (bits)
c2pnb304w1 (X9.62 c2pnb304w1)	1.2.840.10045.3.0.17	06 08 2A 86 48 CE 3D 03 00 11	152
c2pnb368w1 (X9.62 c2pnb368w1)	1.2.840.10045.3.0.19	06 08 2A 86 48 CE 3D 03 00 13	184
c2tnb191v1 (X9.62 c2tnb191v1)	1.2.840.10045.3.0.5	06 08 2A 86 48 CE 3D 03 00 05	96
c2tnb191v2 (X9.62 c2tnb191v2)	1.2.840.10045.3.0.6	06 08 2A 86 48 CE 3D 03 00 06	96
c2tnb191v3 (X9.62 c2tnb191v3)	1.2.840.10045.3.0.7	06 08 2A 86 48 CE 3D 03 00 07	96
c2tnb239v1 (X9.62 c2tnb239v1)	1.2.840.10045.3.0.11	06 08 2A 86 48 CE 3D 03 00 0B	120
c2tnb239v2 (X9.62 c2tnb239v2)	1.2.840.10045.3.0.12	06 08 2A 86 48 CE 3D 03 00 0C	120
c2tnb239v3 (X9.62 c2tnb239v3)	1.2.840.10045.3.0.13	06 08 2A 86 48 CE 3D 03 00 0D	120
c2tnb359v1 (X9.62 c2tnb359v1)	1.2.840.10045.3.0.18	06 08 2A 86 48 CE 3D 03 00 12	180
c2tnb431r1 (X9.62 c2tnb431r1)	1.2.840.10045.3.0.20	06 08 2A 86 48 CE 3D 03 00 14	215
Ed25519 (edwards25519)	1.3.6.1.4.1.11591.15.1	06 09 2B 06 01 04 01 DA 47 0F 01	128
prime192v1 (X9.62 prime192v1, secp192r1)	1.2.840.10045.3.1.1	06 08 2A 86 48 CE 3D 03 01 01	96
prime192v2 (X9.62 prime192v2)	1.2.840.10045.3.1.2	06 08 2A 86 48 CE 3D 03 01 02	96

Curve Name(s)	OID (dot)	OID (byte)	Security Strength (bits)
prime192v3 (X9.62 prime192v3)	1.2.840.10045.3.1.3	06 08 2A 86 48 CE 3D 03 01 03	96
prime239v1 (X9.62 prime239v1)	1.2.840.10045.3.1.4	06 08 2A 86 48 CE 3D 03 01 04	120
prime239v2 (X9.62 prime239v2)	1.2.840.10045.3.1.5	06 08 2A 86 48 CE 3D 03 01 05	120
prime239v3 (X9.62 prime239v3)	1.2.840.10045.3.1.6	06 08 2A 86 48 CE 3D 03 01 06	120
prime256v1 (X9.62 prime256v1, secp256r1)	1.2.840.10045.3.1.7	06 08 2A 86 48 CE 3D 03 01 07	128
secp112r1	1.3.132.0.6	06 05 2B 81 04 00 06	56
secp112r2	1.3.132.0.7	06 05 2B 81 04 00 07	56
secp128r1	1.3.132.0.28	06 05 2B 81 04 00 1C	64
secp128r2	1.3.132.0.29	06 05 2B 81 04 00 1D	64
secp160k1	1.3.132.0.9	06 05 2B 81 04 00 09	80
secp160r1	1.3.132.0.8	06 05 2B 81 04 00 08	80
secp160r2	1.3.132.0.30	06 05 2B 81 04 00 1E	80
secp192k1	1.3.132.0.31	06 05 2B 81 04 00 1F	96
secp224k1	1.3.132.0.32	06 05 2B 81 04 00 20	112
secp224r1	1.3.132.0.33	06 05 2B 81 04 00 21	112
secp256k1	1.3.132.0.10	06 05 2B 81 04 00 0A	128
secp384r1	1.3.132.0.34	06 05 2B 81 04 00 22	192
secp521r1	1.3.132.0.35	06 05 2B 81 04 00 23	260
sect113r1	1.3.132.0.4	06 05 2B 81 04 00 04	56
sect113r2	1.3.132.0.5	06 05 2B 81 04 00 05	56
sect131r1	1.3.132.0.22	06 05 2B 81 04 00 16	64
sect131r2	1.3.132.0.23	06 05 2B 81 04 00 17	64
sect163k1	1.3.132.0.1	06 05 2B 81 04 00 01	80
sect163r1	1.3.132.0.2	06 05 2B 81 04 00 02	80
sect163r2	1.3.132.0.15	06 05 2B 81 04 00 0F	80

Curve Name(s)	OID (dot)	OID (byte)	Security Strength (bits)
sect193r1	1.3.132.0.24	06 05 2B 81 04 00 18	96
sect193r2	1.3.132.0.25	06 05 2B 81 04 00 19	96
sect233k1	1.3.132.0.26	06 05 2B 81 04 00 1A	112
sect233r1	1.3.132.0.27	06 05 2B 81 04 00 1B	112
sect239k1	1.3.132.0.3	06 05 2B 81 04 00 03	115
sect283k1	1.3.132.0.16	06 05 2B 81 04 00 10	128
sect283r1	1.3.132.0.17	06 05 2B 81 04 00 11	128
sect409k1	1.3.132.0.36	06 05 2B 81 04 00 24	192
sect409r1	1.3.132.0.37	06 05 2B 81 04 00 25	192
sect571k1	1.3.132.0.38	06 05 2B 81 04 00 26	256
sect571r1	1.3.132.0.39	06 05 2B 81 04 00 27	256
X25519 (curve25519)	1.3.6.1.4.1.3029.1.5.1	06 0a 2b 06 01 04 01 97 55 01 05 01	128

For additional information about the Elliptic Curve specification, refer to this article:

http://www.ietf.org/rfc/rfc4492.txt

ECDH with Key Derive Function

HSM mechanisms supporting key derivation include DH, ECDH, ECIES.

For the concatenation KDF the X9 and NIST variants differ. For example, the X9 variant is not compliant with KDFs defined in SP800-56.

You will need to determine with which standard your solution should comply.

The mechanism parameter for CKM_ECDH1_DERIVE (in most cases, probably CKM_ECDH1_COFACTOR_DERIVE) has a “kdf” parameter that defines the type of KDF to use.

At the PKCS#11 level, both variants are supported by the Luna HSMs and the Luna Cloud HSM; for example CKD_SHA256_KDF vs CKD_SHA256_NIST_KDF.

PKCS#11 standard KDFs supported in Luna HSM

PKCS#11		Available since
#define CKD_NULL	0x00000001UL	Luna HSM Firmware 7.0.1
#define CKD_SHA1_KDF	0x00000002UL	Luna HSM Firmware 7.0.1
#define CKD_SHA1_KDF_ASN1	0x00000003UL	Luna HSM Firmware 7.0.1

PKCS#11		Available since
#define CKD_SHA1_KDF_CONCATENATE	0x00000004UL	Luna HSM Firmware 7.0.1
#define CKD_SHA224_KDF	0x00000005UL	Luna HSM Firmware 7.0.1
#define CKD_SHA256_KDF	0x00000006UL	Luna HSM Firmware 7.0.1
#define CKD_SHA384_KDF	0x00000007UL	Luna HSM Firmware 7.0.1
#define CKD_SHA512_KDF	0x00000008UL	Luna HSM Firmware 7.0.1
#define CKD_SHA3_224_KDF	0x0000000AUL	Luna HSM Firmware 7.4.2
#define CKD_SHA3_256_KDF	0x0000000BUL	Luna HSM Firmware 7.4.2
#define CKD_SHA3_384_KDF	0x0000000CUL	Luna HSM Firmware 7.4.2
#define CKD_SHA3_512_KDF	0x0000000DUL	Luna HSM Firmware 7.4.2
#define CKD_SHA1_KDF_SP800	0x0000000EUL	Luna HSM Client 10.5.0
#define CKD_SHA224_KDF_SP800	0x0000000FUL	Luna HSM Client 10.5.0
#define CKD_SHA256_KDF_SP800	0x00000010UL	Luna HSM Client 10.5.0
#define CKD_SHA384_KDF_SP800	0x00000011UL	Luna HSM Client 10.5.0
#define CKD_SHA512_KDF_SP800	0x00000012UL	Luna HSM Client 10.5.0
#define CKD_SHA3_224_KDF_SP800	0x00000013UL	Luna HSM Client 10.5.0
#define CKD_SHA3_256_KDF_SP800	0x00000014UL	Luna HSM Client 10.5.0
#define CKD_SHA3_384_KDF_SP800	0x00000015UL	Luna HSM Client 10.5.0
#define CKD_SHA3_512_KDF_SP800	0x00000016UL	Luna HSM Client 10.5.0

Vendor defined KDFs

The "_NIST_" KDFs map to the SP800 KDFs above.

The "_OLD" KDFs map to their non-"_OLD" equivalents above.

Vendor Defined	Value
#define CKD_SHA224_KDF_OLD	0x80000003

Vendor Defined	Value
#define CKD_SHA256_KDF_OLD	0x80000004
#define CKD_SHA384_KDF_OLD	0x80000005
#define CKD_SHA512_KDF_OLD	0x80000006
#define CKD_RIPEMD160_KDF	0x80000007
#define CKD_SHA1_NIST_KDF	0x00000012
#define CKD_SHA224_NIST_KDF	0x80000013
#define CKD_SHA256_NIST_KDF	0x80000014
#define CKD_SHA384_NIST_KDF	0x80000015
#define CKD_SHA512_NIST_KDF	0x80000016
#define CKD_RIPEMD160_NIST_KDF	0x80000017
#define CKD_SHA3_224_NIST_KDF	0x8000001A
#define CKD_SHA3_256_NIST_KDF	0x8000001B
#define CKD_SHA3_384_NIST_KDF	0x8000001C
#define CKD_SHA3_512_NIST_KDF	0x8000001D
#define CKD_SHA1_SES_KDF	0x82000000
#define CKD_SHA224_SES_KDF	0x83000000
#define CKD_SHA256_SES_KDF	0x84000000
#define CKD_SHA384_SES_KDF	0x85000000
#define CKD_SHA512_SES_KDF	0x86000000
#define CKD_RIPEMD160_SES_KDF	0x87000000
#define CKD_SHA3_224_SES_KDF	0x8A000000

Vendor Defined	Value
#define CKD_SHA3_256_SES_KDF	0x8B000000
#define CKD_SHA3_384_SES_KDF	0x8C000000
#define CKD_SHA3_512_SES_KDF	0x8D000000
#define CKD_SHA1_KDF_CONCATENATE_X9_42 CKD_SHA1_KDF_CONCATENATE	
#define CKD_SHA1_KDF_CONCATENATE_NIST	0x80000001

JCPROV	Value	Equivalent Available in JSP
public static final long CKD_NULL =	0x00000001	Yes
public static final long CKD_SHA1_KDF =	0x00000002	Yes
public static final long CKD_SHA224_KDF =	0x00000005	Yes
public static final long CKD_SHA256_KDF =	0x00000006	Yes
public static final long CKD_SHA384_KDF =	0x00000007	Yes
public static final long CKD_SHA512_KDF =	0x00000008	Yes
public static final long CKD_RIPEMD160_KDF =	0x80000007	No
public static final long CKD_SHA1_NIST_KDF =	0x80000012	No
public static final long CKD_SHA224_NIST_KDF =	0x80000013	No
public static final long CKD_SHA256_NIST_KDF =	0x80000014	Yes
public static final long CKD_SHA384_NIST_KDF =	0x80000015	No
public static final long CKD_SHA512_NIST_KDF =	0x80000016	No
public static final long CKD_RIPEMD160_NIST_KDF=	0x80000017	No

JCPROV	Value	Equivalent Available in JSP
public static final long CKD_SHA1_SES_KDF =	0x82000000	No
public static final long CKD_SHA224_SES_KDF =	0x83000000	No
public static final long CKD_SHA256_SES_KDF =	0x84000000	No
public static final long CKD_SHA384_SES_KDF =	0x85000000	No
public static final long CKD_SHA512_SES_KDF =	0x86000000	No
public static final long CKD_RIPEMD160_SES_KDF=	0x87000000	No
/* counter values for TR-03111 session keys */		
public static final long CKD_SES_ENC_CTR =	0x00000001	No
public static final long CKD_SES_AUTH_CTR =	0x00000002	No
public static final long CKD_SES_ALT_ENC_CTR =	0x00000003	No
public static final long CKD_SES_ALT_AUTH_CTR =	0x00000004	No
public static final long CKD_SES_MAX_CTR =	0x0000FFFF	No
public static final long CKD_SHA3_224_KDF	0x0000000A	No
public static final long CKD_SHA3_256_KDF	0x0000000B	No
public static final long CKD_SHA3_384_KDF	0x0000000C	No
public static final long CKD_SHA3_512_KDF	0x0000000D	No
public static final long CKD_SHA1_KDF_SP800 =	0x0000000E	No
public static final long CKD_SHA224_KDF_SP800 =	0x0000000F	No
public static final long CKD_SHA256_KDF_SP800 =	0x00000010	No
public static final long CKD_SHA384_KDF_SP800 =	0x00000011	No
public static final long CKD_SHA512_KDF_SP800 =	0x00000012	No

JCPROV	Value	Equivalent Available in JSP
public static final long CKD_SHA3_224_KDF_SP800 =	0x00000013	No
public static final long CKD_SHA3_256_KDF_SP800 =	0x00000014	No
public static final long CKD_SHA3_384_KDF_SP800 =	0x00000015	No
public static final long CKD_SHA3_512_KDF_SP800 =	0x00000016	No
public static final long CKD_RIPEMD160_KDF =	0x80000007	No

ECIES general

EC IES mechanism (X9.63)

```
#define CKM_ECIES                           (CKM_VENDOR_DEFINED + 0xA00)
#define CKM_XOR_BASE_AND_DATA_W_KDF         (CKM_VENDOR_DEFINED + 0xA01)
#define CKM_NIST_PRF_KDF                    (CKM_VENDOR_DEFINED + 0xA02)
#define CKM_PRF_KDF                         (CKM_VENDOR_DEFINED + 0xA03)
#define CKM_AES_XTS_OLD                     (CKM_VENDOR_DEFINED + 0xA04)
```

Mechanism parameters for CKM_ECIES.

EC Diffie-Hellman (DH) primitive to use for shared secret derivation

```
typedef CK_ULONG CK_EC_DH_PRIMITIVE;
```

EC DH primitives

```
#define CKDHP_STANDARD        0x00000001
#define CKDHP_ECDH1_COFACTOR  0x00000001
#define CKDHP_MODIFIED        0x00000002 /* Not implemented */
#define CKDHP_ECDH1           0x00000003
```

Inner encryption scheme to use for ECIES

```
typedef CK_ULONG CK_EC_ENC_SCHEME;
```

Inner encryption schemes

```
#define CKES_XOR           0x00000001
#define CKES_DES3_CBC_PAD  0x00000002
#define CKES_AES_CBC_PAD   0x00000003
#define CKES_DES3_CBC      0x00000004
#define CKES_AES_CBC       0x00000005
#define CKES_AES_CTR       0x00000006
#define CKES_AES_GCM       0x00000007
#define CKES_AES_KW        0x00000008
#define CKES_AES_KWP       0x00000009
```

 Copyright 2001-2024 Thales Group

Message Authentication Code (MAC) scheme to use for ECIES */

```
typedef CK_ULONG CK_EC_MAC_SCHEME;
```

MAC schemes

```
#define CKMS_HMAC_SHA1        0x00000001
#define CKMS_SHA1             0x00000002
#define CKMS_HMAC_SHA224      0x00000003
#define CKMS_SHA224           0x00000004
#define CKMS_HMAC_SHA256      0x00000005
#define CKMS_SHA256           0x00000006
#define CKMS_HMAC_SHA384      0x00000007
#define CKMS_SHA384           0x00000008
#define CKMS_HMAC_SHA512      0x00000009
#define CKMS_SHA512           0x0000000a
#define CKMS_HMAC_RIPEMD160   0x0000000b
#define CKMS_RIPEMD160        0x0000000c
```

Mechanism parameter structure for ECIES

```
typedef struct CK_ECIES_PARAMS
{
```

Diffie-Hellman primitive used to derive the shared secret value

```
   CK_EC_DH_PRIMITIVE dhPrimitive;
```

Key derivation function used on the shared secret value

```
   CK_EC_KDF_TYPE kdf;
```

The length in bytes of the key derivation shared data

```
   CK_ULONG ulSharedDataLen1;
```

The key derivation padding data shared between the two parties

```
   CK_BYTE_PTR pSharedData1;
```

The encryption scheme used to transform the input data

```
   CK_EC_ENC_SCHEME encScheme;
```

The bit length of the key to use for the encryption scheme

```
   CK_ULONG ulEncKeyLenInBits;
```

The MAC scheme used for MAC generation or validation

```
   CK_EC_MAC_SCHEME macScheme;
```

The bit length of the key to use for the MAC scheme

```
   CK_ULONG ulMacKeyLenInBits;
```

The bit length of the MAC scheme output

```
   CK_ULONG ulMacLenInBits;
```

The length in bytes of the MAC shared data

```
   CK_ULONG ulSharedDataLen2;
```

The MAC padding data shared between the two parties

```
   CK_BYTE_PTR pSharedData2;
} CK_ECIES_PARAMS;
typedef CK_ECIES_PARAMS CK_PTR CK_ECIES_PARAMS_PTR;
typedef struct CK_ECIES_PARAMS_EXT
{
```

Legacy ECIES parameters

```
   CK_ECIES_PARAMS eciesParams;
```

Reference encryption scheme structure extension

```
   CK_VOID_PTR pEncSchemeMechanismParameter;
```

Length encryption scheme structure extension

```
   CK_ULONG    ulEncSchemeMechanismParameterLen;
} CK_ECIES_PARAMS_EXT;
typedef CK_ECIES_PARAMS_EXT CK_PTR CK_ECIES_PARAMS_EXT_PTR;
typedef struct CK_ECIES_PARAMS_EXT2
{
```

Legacy ECIES parameters

```
   CK_ECIES_PARAMS eciesParams;
```

Reference encryption scheme structure extension

```
   CK_VOID_PTR pEncSchemeMechanismParameter;
```

Length encryption scheme structure extension

```
   CK_ULONG    ulEncSchemeMechanismParameterLen;
```

Flags for KDF additional shared data (sharedData1)

0 = no addition to shared data

1 = shared data | ephemeral public key

2 = shared data | compressed ephemeral public key

3 = ephemeral public key | shared data

4 = compressed ephemeral public key | shared data

```
   CK_ULONG    ulKDFSharedDataFlags;
} CK_ECIES_PARAMS_EXT2;
typedef CK_ECIES_PARAMS_EXT2 CK_PTR CK_ECIES_PARAMS_EXT2_PTR;
```

Parameter and values used with CKM_PRF_KDF and CKM_NIST_PRF_KDF

```
typedef CK_ULONG CK_KDF_PRF_TYPE;
typedef CK_ULONG CK_KDF_PRF_ENCODING_SCHEME;
```

Pseudorandom Function(PRF) Key Derivation Function (KDF) schemes

```
#define CK_NIST_PRF_KDF_DES3_CMAC      0x00000001
#define CK_NIST_PRF_KDF_AES_CMAC       0x00000002
#define CK_PRF_KDF_ARIA_CMAC           0x00000003
#define CK_PRF_KDF_SEED_CMAC           0x00000004
#define CK_NIST_PRF_KDF_HMAC_SHA1      0x00000005
#define CK_NIST_PRF_KDF_HMAC_SHA224    0x00000006
#define CK_NIST_PRF_KDF_HMAC_SHA256    0x00000007
#define CK_NIST_PRF_KDF_HMAC_SHA384    0x00000008
#define CK_NIST_PRF_KDF_HMAC_SHA512    0x00000009
#define CK_PRF_KDF_HMAC_RIPEMD160      0x0000000A
#define CK_NIST_PRF_KDF_HMAC_SHA3_224  0x0000000B
#define CK_NIST_PRF_KDF_HMAC_SHA3_256  0x0000000C
#define CK_NIST_PRF_KDF_HMAC_SHA3_384  0x0000000D
#define CK_NIST_PRF_KDF_HMAC_SHA3_512  0x0000000E
```

Mask Generation Function MGF SHA3

```
#define CKG_MGF1_SHA3_224     0x80000006
#define CKG_MGF1_SHA3_256     0x80000007
#define CKG_MGF1_SHA3_384     0x80000008
#define CKG_MGF1_SHA3_512     0x80000009
```

Affects the format of the fixed data passed to the PRF.

Scheme #3 is the one described in NIST SP 800-108.

```
#define LUNA_PRF_KDF_ENCODING_SCHEME_1     0x00000000 // Context || 0x00 || Label || Length
#define LUNA_PRF_KDF_ENCODING_SCHEME_2     0x00000001 // Context || Label
#define LUNA_PRF_KDF_ENCODING_SCHEME_3     0x00000002 // Label || 0x00 || Context || Length
#define LUNA_PRF_KDF_ENCODING_SCHEME_4     0x00000003 // Label || Context
#define LUNA_PRF_KDF_ENCODING_SCHEME_SCP03 0x00000004
#define LUNA_PRF_KDF_ENCODING_SCHEME_HID_KD 0x00000005
typedef struct CK_KDF_PRF_PARAMS {
   CK_KDF_PRF_TYPE            prfType;
   CK_BYTE_PTR                pLabel;
   CK_ULONG                   ulLabelLen;
   CK_BYTE_PTR                pContext;
   CK_ULONG                   ulContextLen;
   CK_ULONG                   ulCounter;
   CK_KDF_PRF_ENCODING_SCHEME ulEncodingScheme;
} CK_PRF_KDF_PARAMS;
typedef CK_PRF_KDF_PARAMS CK_PTR CK_KDF_PRF_PARAMS_PTR;
```

ECIES for 5G

Prior to Luna HSM Firmware 7.7.2, for the Luna implementation of ECIES AES-CTR, it was required to supply the Initial Counter Block (ICB) or else all zeroes for that parameter. Luna HSM Firmware 7.7.2 onward adds the derivation (Key Derivation Function) of the ICB to the encryption scheme. This supports 5G 3GPP TS 33.501 standard by enabling the Luna HSM to process 5G 3GPP SUCI (SUbscription Concealed Identifier) de-concealement requests.

Profiles Supported

The ECIES decryption operation maps to the “home network side” (HN), while encryption maps to User Equipment (UE) such as phones and other devices. Two profiles each with specific ECIES parameters are supported:

> Profile A:

- ECC: Curve25519

> Profile B:

- ECC: secp256r1 (same as prime256v1)

Both require Shared Data for KDF: compressed ephemeral public key.

Test vectors, included in the 33.501 standard, can be used to verify firmware compliance.

Decryption

In firmware older than Luna HSM Firmware 7.7.2, the ECIES mechanism parameter structure CK_ECIES_PARAMS uses a 0-byte ICB or IV for all encryption schemes.

> For AES-CTR this results in an ICB of 16 0x0 bytes and counterBits = 0x0.

In Luna HSM Firmware 7.7.2 and onward, CK_ECIES_PARAMS interpretation has been changed for AES-CTR in firmware:

> Firmware now derives an extra 16 bytes for the ICB in the KDF step and uses a 32-bit counter for decryption.

> To derive the ICB using CK_ECIES_PARAMS_EXT, set the encryption scheme mechanism parameters to no parameters (NULL, 0 length).

The ephemeral public key can now be in compressed format.

- It is uncompressed for the Key Agreement step (Curve25519 is already compressed).

The 3GPP TS 33.501 standard requires the compressed ephemeral public key to be passed into the KDF via “sharedData1” mechanism parameter.

The ECIES encryption mechanism was changed for AES-CTR to derive an ICB.

The ephermal public key is generated each time so it cannot be sent into the KDF derivation through “sharedData1”.

To test ECIES encryption with the new ECIES decryption a new CK_ECIES_PARAMS_EXT2 structure (Luna HSM Client 10.4 onward) takes parameter kdfSharedDataFlags added:

> 0 = no addition to shared data (sharedData1)

> 1 = shared data | ephemeral public key

> 2 = shared data | compressed ephemeral public key

> 3 = ephemeral public key | shared data

> 4 = compressed ephemeral public key | shared data

This allows the encryption or decryption to add the ephemeral public key to the KDF without using the “sharedData1” parameter.

Luna 5G OP flags and message response TLV tag definitions

```
#define LUNA_5G_OPC                 0x00000001      // OPC is provided rather than OP
#define LUNA_5G_ENCRYPTED_OP        0x00000002      // OP or OPC is passed in encrytped by
Storage Key
#define LUNA_5G_OP_OBJECT           0x00000004      // OP or OPC is an object in HSM partition
#define LUNA_5G_USE_TLV             0x00000008      // Use the Tag/Length/Value format in
return string
#define LUNA_5G_USER_DEFINED_RC     0x00000010      // User has defined his own R and C
constants for Milenage
#define LUNA_5G_TAG_RANDOM          0x00000001      // Random data generated in HSM
#define LUNA_5G_TAG_RES             0x00000002      // Response string
#define LUNA_5G_TAG_CK              0x00000003      // Confidentiality Key
#define LUNA_5G_TAG_IK              0x00000004      // Integrity Key
#define LUNA_5G_TAG_SQN_XOR_AK      0x00000005      // Sequence # xor'd with Anonymity Key (AK)
#define LUNA_5G_TAG_AMF             0x00000006      // Authentication Management Field
#define LUNA_5G_TAG_MAC             0x00000007      // MAC-A for authentication
#define LUNA_5G_TAG_SEQUENCE        0x00000008      // Sequence number for response to resynch
operation
typedef struct CK_MILENAGE_SIGN_PARAMS {
   CK_ULONG             ulMilenageFlags;
   CK_ULONG             ulEncKiLen;
   CK_BYTE_PTR          pEncKi;
   CK_ULONG             ulEncOPcLen;
   CK_BYTE_PTR          pEncOPc;
   CK_OBJECT_HANDLE     hSecondaryKey;
   CK_OBJECT_HANDLE     hRCKey;
   CK_BYTE              sqn[6];
   CK_BYTE              amf[2];
} CK_MILENAGE_SIGN_PARAMS;
typedef CK_MILENAGE_SIGN_PARAMS CK_PTR CK_MILENAGE_SIGN_PARAMS_PTR;
```

```
typedef struct CK_TUAK_SIGN_PARAMS {
   CK_ULONG             ulTuakFlags;
   CK_ULONG             ulEncKiLen;
   CK_BYTE_PTR          pEncKi;
   CK_ULONG             ulEncTOPcLen;
   CK_BYTE_PTR          pEncTOPc;
   CK_ULONG             ulIterations;
   CK_OBJECT_HANDLE     hSecondaryKey;
   CK_ULONG             ulResLen;
   CK_ULONG             ulMacALen;
   CK_ULONG             ulCkLen;
   CK_ULONG             ulIkLen;
   CK_BYTE              sqn[6];
   CK_BYTE              amf[2];
} CK_TUAK_SIGN_PARAMS;
typedef CK_TUAK_SIGN_PARAMS CK_PTR CK_TUAK_SIGN_PARAMS_PTR;
typedef struct CK_COMP128_SIGN_PARAMS {
   CK_ULONG            ulVersion;
   CK_ULONG            ulEncKiLen;
   CK_BYTE_PTR         pEncKi;
} CK_COMP128_SIGN_PARAMS;
typedef CK_COMP128_SIGN_PARAMS CK_PTR CK_COMP128_SIGN_PARAMS_PTR;
typedef struct CK_KEY_TRANSLATE_PARAMS {
   CK_FLAGS  ulFlags;
   CK_MECHANISM mWrapMech;
   CK_MECHANISM mUnWrapMech;
   CK_BYTE_PTR pData;
   CK_ULONG ulDataLen;
   CK_OBJECT_HANDLE hUnwrapKey;
} CK_KEY_TRANSLATE_PARAMS;
typedef CK_KEY_TRANSLATE_PARAMS CK_PTR CK_KEY_TRANSLATE_PARAMS_PTR;
```

Summary

The decision comes down to using the CKM_ECIES mechanism while passing

> either CK_ECIES_PARAMS

> or CK_ECIES_PARAMS_EXT

parameter structures filled out to select the AES-CTR encryption scheme.

For CK_ECIES_PARAMS_EXT specify no extra encryption scheme parameters:

```
pEncSchemeMechanismParameter = NULL
ulEncSchemeMechanismParameterLen = 0
```

This causes the AES-CTR ICB to be derived.

Tools

For Luna HSM Client 10.4 onward:

> CKDEMO was changed to allow zero-length AES-CTR bits and to support KDF flags selection.

> MULTITOKEN and FMULTITOKEN changed to allow selecting AES-CTR or AES_CBC_PAD for all the “eciesaes…” modes.

Capability and Policy Configuration Control Using the Luna API

The configuration and control of the Luna PCIe HSM 7 is provided by a set of capabilities and policies which you can query and set using the Luna API. See for more information.

HSM Capabilities and Policies

Each HSM has a set of capabilities. An HSM's capability set defines and controls the behavior of the HSM.

HSM behavior can be further modified through changing policies. The HSM SO can refine the behavior of an HSM by changing the policy settings.

HSM Partition Capabilities and Policies

Each HSM can support one-or-more virtual HSMs called application partitions (may also be called “containers” in some areas of the API), which are used by properly authenticated remote clients to perform cryptographic operations.

Each application partition has a set of capabilities. An application partition's capability set defines and controls the behavior of the partition.

application partition behavior can be further modified through changing policies. The HSM SO can refine the behavior of an application partition by changing the policy settings. Different partitions can have different values for the configuration elements which apply to specific application partitions – in other words, if a policy is set to a given value for one partition, the policy can be set to a different value for another partition on the same HSM.

In some cases, a partition policy change is destructive.

Policy Refinement

For every policy set element, there is a corresponding capability set element (the reverse is not true – there can be some capability set elements that do not have corresponding policy set elements). The value of a policy set element can be modified by the HSM SO, but only within the limitations imposed by the corresponding capability set element.

For example, there is a policy set element which determines how many failed login attempts may be made before a Partition is deleted or locked out. There is also a corresponding capability set element for the same purpose. The policy element may be modified by the HSM SO, but may only be set to a value less than or equal to that of the capability set element. So if the capability set element has a value of 10, the HSM SO can set the policy to a value less than or equal to 10.

In general, the HSM SO may modify policy set elements to make the HSM or partition policy more restrictive than that imposed by the capability set elements. The HSM SO can not make the HSM or application partition policy less restrictive or enable functionality that is disabled through capability settings.

Policy Types

There are three types of policy elements, as follows:

Normal policy elements	May be set at any time by the HSM SO. The values which may be set are limited only by the corresponding capability element as described in the previous section (i.e. the policy element can be set only to a value less than or equal to the capability set element).
Destructive policy elements	May be set at any time, but setting them results in the erasure of any partitions and their contents. Policy elements are destructive if changing them significantly affects the security policy of the HSM.
Write-restricted policy elements	Cannot be modified directly, but instead are affected by other actions that can be taken.

Querying and Modifying HSM Configuration

The following are the relevant functions (found in **sfnt_extensions.h**):

```
> CK_RV CK_ENTRY CA_GetConfigurationElementDescription(
> CK_SLOT_ID slotID,
> CK_ULONG ulIsContainerElement,
> CK_ULONG ulIsCapabilityElement,
> CK_ULONG ulElementId,
> CK_ULONG_PTR pulElementBitLength,
> CK_ULONG_PTR pulElementDestructive,
> CK_ULONG_PTR pulElementWriteRestricted,
> CK_CHAR_PTR pDescription);
> CK_RV CK_ENTRY CA_GetHSMCapabilitySet(
> CK_SLOT_ID uPhysicalSlot,
> CK_ULONG_PTR pulCapIdArray,
> CK_ULONG_PTR pulCapIdSize,
> CK_ULONG_PTR pulCapValArray,
> CK_ULONG_PTR pulCapValSize );
> CK_RV CK_ENTRY CA_GetHSMCapabilitySetting (
> CK_SLOT_ID slotID,
> CK_ULONG ulPolicyId,
> CK_ULONG_PTR pulPolicyValue);
> CK_RV CK_ENTRY CA_GetHSMPolicySet(
> CK_SLOT_ID uPhysicalSlot,
> CK_ULONG_PTR pulPolicyIdArray,
> CK_ULONG_PTR pulPolicyIdSize,
```

```
> CK_ULONG_PTR pulPolicyValArray,
> CK_ULONG_PTR pulPolicyValSize );
> CK_RV CK_ENTRY CA_GetHSMPolicySetting (
> CK_SLOT_ID slotID,
> CK_ULONG ulPolicyId,
> CK_ULONG_PTR pulPolicyValue);
> CK_RV CK_ENTRY CA_GetContainerCapabilitySet(
> CK_SLOT_ID uPhysicalSlot,
> CK_ULONG ulContainerNumber,
> CK_ULONG_PTR pulCapIdArray,
> CK_ULONG_PTR pulCapIdSize,
> CK_ULONG_PTR pulCapValArray,
> CK_ULONG_PTR pulCapValSize );
> CK_RV CK_ENTRY CA_GetContainerCapabilitySetting (
> CK_SLOT_ID slotID,
> CK_ULONG ulContainerNumber,
> CK_ULONG ulPolicyId,
> CK_ULONG_PTR pulPolicyValue);
> CK_RV CK_ENTRY CA_GetContainerPolicySet(
> CK_SLOT_ID uPhysicalSlot,
> CK_ULONG ulContainerNumber,
> CK_ULONG_PTR pulPolicyIdArray,
> CK_ULONG_PTR pulPolicyIdSize,
> CK_ULONG_PTR pulPolicyValArray,
> CK_ULONG_PTR pulPolicyValSize );
> CK_RV CK_ENTRY CA_GetContainerPolicySetting(
> CK_SLOT_ID uPhysicalSlot,
> CK_ULONG ulContainerNumber,
> CK_ULONG ulPolicyId,
> CK_ULONG_PTR pulPolicyValue);
> CK_RV CK_ENTRY CA_SetHSMPolicy (
> CK_SESSION_HANDLE hSession,
> CK_ULONG ulPolicyId,
> CK_ULONG ulPolicyValue);
> CK_RV CK_ENTRY CA_SetDestructiveHSMPolicy (
```

```
> CK_SESSION_HANDLE hSession,
> CK_ULONG ulPolicyId,
> CK_ULONG ulPolicyValue);
> CK_RV CK_ENTRY CA_SetContainerPolicy (
> CK_SESSION_HANDLE hSession,
> CK_ULONG ulContainer,
> CK_ULONG ulPolicyId,
> CK_ULONG ulPolicyValue);
```

The CA_GetConfigurationElementDescription() Function

The **CA_GetConfigurationElementDescription()** function requires that you pass in a zero or one value to indicate whether the element you are querying is an application partition (container) element or an HSM element, and another zero/one value to define whether it is a capability or policy that you are interested in. You also pass in the id of the element and a character buffer of at least 60 characters. The function then returns the size of the element value (in bits), an indication of whether the element is destructive, an indication of whether the policy (if it is a policy) is write-restricted, and it also writes the description string into the buffer that you provided.

The CA_Get{HSM|Container}{Capability|Policy}Set() Functions

The various **CA_Get{HSM|Container}{Capability|Policy}Set()** functions all return (in the word arrays provided) a complete list of element id/value pairs for the set specified. For example, **CA_GetHSMCapabilitySet()** returns a list of all HSM capability elements and their values. The parameters for these functions include a list pointer and length pointer for each of the element ids and element values. On calling the function, you should provide a buffer or a null pointer for each of the lists, and the length value should be initialized to the size of the buffer. On return, the buffer (if given) is populated, and the length is updated to the real length of the list. If the buffer is given but is not large enough, an error results.

Typically you would invoke the function twice: call the function the first time with null buffer pointers so that the real length necessary is returned, then allocate the necessary buffers and call the function a second time, giving the real buffers.

The various **CA_Get{HSM|Container}{Capability|Policy}Setting()** functions allow you to query a specific element value. Provide the element id and the function returns the value.

The CA_Set...() Functions

The various **CA_Set...()** functions allow you to set individual HSM and partition policies. There are two varieties for setting HSM policies, because changing the value of a destructive HSM policy results in the HSM being cleared of any partitions and their contents. To make it clear when this is going to happen, the appropriate set function must be called based on whether the HSM policy is destructive or not (which you can determine with the **CA_GetConfigurationElementDescription()** function).

Connection Timeout

The connection timeout is not configurable.

Linux and Unix Connection Timeout

On Unix platforms, the client performs a **connect** on the socket. If the socket is busy or unavailable, the client performs a **select** on the socket with the timeout set to 10 seconds (hardcoded). If the **select** call returns before the timeout, then the client is able to connect. If not then it fails. This prevents the situation where some Unix operating systems can block for several minutes when Luna PCIe HSM 7 is unavailable.

Windows Connection Timeout

On Windows platforms, **connect** is called without **select**, relying upon the default Windows timeout of approximately 20 seconds.

 Copyright 2001-2024 Thales Group

CHAPTER 7: Design Considerations

This chapter provides guidance for creating applications that use specific Luna PCIe HSM 7 configurations or features. It contains the following topics:

> "Multifactor Quorum-Authenticated HSMs" below
> "High Availability Implementations" on page 617
> "Key Attribute Defaults" on page 619
> "Object Usage Count" on page 621
> "Migrating Keys From Software to a Luna PCIe HSM 7" on page 624
> "Audit Logging" on page 647

Multifactor Quorum-Authenticated HSMs

In normal use, Luna PED supplies PINs and certain other critical security parameters to the token/HSM, invisibly to the user. This prevents other persons from viewing PINs, etc. on a computer screen or watching them typed on a keyboard, which in turn prevents such persons from illicitly cloning token or HSM contents.

Two classes of users operate Luna PED: the HSM Security Officer, and an application partition user (Partition SO, Crypto Officer/User). The person handling new HSMs and using Luna PED is normally the HSM SO, who:

> Initializes the HSM
> Conducts HSM maintenance, such as firmware and capability upgrades
> Initializes HSM Partitions and tokens
> Creates users (sets PINs)
> Changes policy settings
> Changes passwords

Following these initial activities, the Luna PED may be required to present the HSM Partition Owner's PED key or PED keys (in case of MofN operations) to enable ordinary signing cryptographic operations carried out by your applications.

With the combination of Activation and AutoActivation, the black PED key is required only upon initial authentication and then not again unless the authentication is interrupted by power failure or by deliberate action on the part of the PED key holders.

About CKDemo with PED key

As its name suggests, CKDemo (CryptoKi Demonstration) is a demonstration program, allowing you to explore the capabilities and functions of several Luna products. The demo program breaks out a number of PKCS 11 functions, as well as the Luna extensions to Cryptoki that allow the enhanced capabilities of our HSMs. However the flexibility, combined with the bare-bones nature of the program, can result in some confusion as to whether

certain operations and combinations are permissible. Where these come up, in the explanation of CKDemo with Luna PCIe HSM 7 with multifactor quorum authentication, and PED key, they are mentioned and explained if necessary.

The demo program appears to make it optional to permit several of the security operations via the keyboard and program interface, or to require that they be done only via the Luna PED keypad. In fact, the option is dictated by the Luna PCIe HSM 7, as it was configured and shipped from the factory, and cannot be changed by you. That is, you can use CKDemo to work/experiment with either type of Luna PCIe HSM 7 (password or multifactor quorum-authenticated), but you cannot make one type behave like the other.

Security and design requirements, enforced by the multifactor quorum-authenticated Luna PCIe HSM 7, dictate that use of Luna PED be mandatory within the applications that you develop for it.

Interchangeability

As mentioned above, several secrets and security parameters related to HSMs are imprinted on PED keys which provide "something you have" access control, as opposed to the "something you know" access control provided by password-authenticated HSMs. The HSM can create each type of secret, which is then also imprinted on a suitably labeled PED key. Alternatively, the secret can be accepted from a PED key (previously imprinted by another HSM) and imprinted on the current HSM. This is mandatory for the cloning domain, when HSMs (or application partitions) are to clone objects one to the other. It is optional for the other HSM secrets, as a matter of convenience or of your security policy, allowing more than one HSM to be accessed for administration by a single SO (blue PED key holder) or more than one application partition to be administered by a single Crypto Officer/User.

PED keys that have never been imprinted are completely interchangeable. They can be used with any modern Luna PCIe HSM 7, and can be imprinted with any of the various secrets. The self-stick labels are provided as a visual identifier of which type of secret has been imprinted on a PED key, or is about to be imprinted. Imprinted PED keys are tied to their associated HSMs and cannot be used to access HSMs or partitions that have been imprinted with different secrets.

Any Luna PED can be used with any Luna PCIe HSM 7 - the PED itself contains no secrets; it simply provides the interface between you and your HSM(s). The exception is that only some Luna PEDs have the capability to be used remotely from the HSM. Any Remote-capable Luna PED is interchangeable with any other Remote-capable Luna PED, and any Luna PED (remote-capable or not) is interchangeable with any other when locally connected to a Luna PCIe HSM 7.

Application partitions and Backup Tokens and PED keys can be “re-cycled” for use in different combinations, but this reuse requires re-initializing the HSM(s) and re-imprinting the PED keys with new secrets or security parameters. Re-initializing a token or HSM wipes previous information from it. Re-imprinting a PED key overwrites any previous information it carried (PIN, domain, etc.).

Startup

Luna PED expects to be connected to a multifactor quorum-authenticated Luna PCIe HSM 7. At power-up, it presents a message showing its firmware version. After a few seconds, the message changes to "Awaiting command..." The Luna PED is waiting for a command from the token/HSM.

The Luna PED screen remains in this status until the CKDemo program, or your own application, initiates a command through the token/HSM.

For the purposes of demonstration, you would now go ahead and create some objects and perform other transactions with the HSM.

> **NOTE** To perform most actions you must be logged in. CKDemo may not remind you before you perform actions out-of-order, but it generates error messages after such attempts. If you receive an error message from the program, review your recent actions to determine if you have logged out or closed sessions and then not formally logged into a new session before attempting to create an object or perform other token/HSM actions. When you do wish to end activities, be sure to formally log out and close sessions. An orderly shutdown of your application should include logging out any users and closing all sessions on HSMs.

Cloning of Tokens

To securely copy the contents of an application partition to another application partition, you must perform a backup to a Luna Backup HSM from the source partition followed by a restore operation from the Backup HSM to the new destination partition. This is done via LunaCM command line, and cannot be accomplished via CKDemo.

High Availability Implementations

If you use the Luna PCIe HSM 7 HA feature then the calls to the Luna PCIe HSM 7s are load-balanced. The session handle that the application receives when it opens a session is a virtual one and is managed by the HA code in the library. The actual sessions with the HSM are established by the HA code in the library and hidden from the application and will come and go as necessary to fulfill application level requests.

Before the introduction of HA AutoRecovery, bringing a failed/lost group member back into the group (recovery) was a manual procedure.

The Administration & Maintenance section contains a general description of the how the HA AutoRecovery function works, in practice.

For every PKCS#11 call, the HA recover logic will check to see if we need to perform auto recovery to a disconnected appliance. If there is a disconnected appliance then it will try to reconnect to that appliance before it proceeds with the current PKCS#11 call.

The HA recovery logic is designed in such a way that it will try to reconnect to an appliance only every X secs and N number of times where X is pre-set to one minute, and N is configurable via LunaCM.

For HA recovery attempts:

- The default retry interval is 60 seconds.
- The default number of retries is effectively infinite.
- The HA configuration section in the **Chrystoki.conf/crystoki.ini** file is created and populated when either the interval or the number of retries is specified in the LunaCM commands hagroup retry and hagroup interval.

The following is the pseudo code of the HA logic

```
if (disconnected_member > 0  and recover_attempt_count < N and time_now - last_recover_attempt
> X) then
   performance auto recovery
   set last_recover_attempt  equal to time_now
   if  (recovery failed)  then
      increment recover_attempt_count  by 1
   else
      decrement disconnected_member by 1
```

```
      reset recover_attempt_count to 0
   end if
end if
```

The HA auto recovery design runs within a PKCS#11 call. The responsiveness of recovering a disconnected member is greatly influenced by the frequency of PKCS#11 calls from the user application. Although the logic shows that it will attempt to recover a disconnected client in X secs, in reality, it will not run until the user application makes the next PKCS#11 call.

Detecting the Failure of an HA Member

When an HA Group member first fails, the HA status for the group shows "device error" for the failed member. All subsequent calls return "token not present", until the member (HSM Partition or PKI token) is returned to service.

Here is an example of two such calls using CKDemo:

```
Enter your choice : 52
Slots available:
   slot#1 - LunaNet Slot
   slot#2 - LunaNet Slot
   slot#3 - HA Virtual Card Slot

Select a slot: 3

HA group 1599447001 status:
   HSM 599447001      - CKR_DEVICE_ERROR
   HSM 78665001       - CKR_OK
Status: Doing great, no errors (CKR_OK)

TOKEN FUNCTIONS
( 1) Open Session  ( 2) Close Session  ( 3) Login
( 4) Logout        ( 5) Change PIN     ( 6) Init Token
( 7) Init Pin      ( 8) Mechanism List ( 9) Mechanism Info
(10) Get Info      (11) Slot Info      (12) Token Info
(13) Session Info  (14) Get Slot List  (15) Wait for Slot Event
(16) InitToken(ind)(17) InitPin (ind)  (18) Login (ind)
(19) CloneMofN

OBJECT MANAGEMENT FUNCTIONS
(20) Create object (21) Copy object    (22) Destroy object
(23) Object size   (24) Get attribute  (25) Set attribute
(26) Find object   (27) Display Object

SECURITY FUNCTIONS
(40)  Encrypt file (41) Decrypt file   (42)  Sign
(43)  Verify       (44) Hash file      (45)  Simple Generate Key
(46)  Digest Key

HIGH AVAILABILITY RECOVERY FUNCTIONS
(50) HA Init       (51) HA Login       (52) HA Status

KEY FUNCTIONS
(60) Wrap key      (61) Unwrap key     (62) Generate random number
(63) Derive Key    (64) PBE Key Gen    (65) Create known keys
(66) Seed RNG      (67) EC User Defined Curves

CA FUNCTIONS
(70) Set Domain    (71) Clone Key      (72) Set MofN
(73) Generate MofN (74) Activate MofN  (75) Generate Token Keys
(76) Get Token Cert(77) Sign Token Cert(78) Generate CertCo Cert
```

```
(79) Modify MofN   (86) Dup. MofN Keys (87) Deactivate MofN

CCM FUNCTIONS
(80) Module List   (81) Module Info    (82) Load Module
(83) Load Enc Mod  (84) Unload Module  (85) Module function Call

OTHERS
(90) Self Test     (94) Open Access    (95) Close Access
(97) Set App ID    (98) Options

OFFBOARD KEY STORAGE:
(101) Extract Masked Object    (102) Insert Masked Object
(103) Multisign With Value     (104) Clone Object
(105) SIMExtract               (106) SIMInsert
(107) SimMultiSign

SCRIPT EXECUTION:
(108) Execute Script
(109) Execute Asynchronous Script
(110) Execute Single Part Script
(0) Quit demo
Enter your choice : 52

Slots available:
   slot#1 - LunaNet Slot
   slot#2 - LunaNet Slot
   slot#3 - HA Virtual Card Slot

Select a slot: 3

HA group 1599447001 status:
   HSM 599447001      - CKR_TOKEN_NOT_PRESENT
   HSM 78665001       - CKR_OK
Status: Doing great, no errors (CKR_OK)
--- end ---
```

Key Attribute Defaults

The following default attribute settings are applied to generated keys/keypairs, and to unwrapped private/secret keys, unless your application specifies different values.

Management Attributes

Attribute	Default Value			
	Generated Public Keys	**Generated Private Keys**	**Unwrapped Private/Secret Keys**	**Derived Secret Keys**
CKA_TOKEN	**0** (FALSE)	**0** (FALSE)	**0** (FALSE)	**0** (FALSE)

Attribute	Default Value			
	Generated Public Keys	**Generated Private Keys**	**Unwrapped Private/Secret Keys**	**Derived Secret Keys**
CKA_PRIVATE	**1** (TRUE) if Crypto Officer logged in **0** (FALSE) if Crypto Officer not logged in	**1** (TRUE) if Crypto Officer logged in **0** (FALSE) if Crypto Officer not logged in	**1** (TRUE) if Crypto Officer logged in **0** (FALSE) if Crypto Officer not logged in	**1** (TRUE) if Crypto Officer logged in **0** (FALSE) if Crypto Officer not logged in
CKA_SENSITIVE	N/A	**1** (TRUE)	**1** (TRUE)	**0** (FALSE)
CKA_MODIFIABLE	**1** (TRUE)	**1** (TRUE)	**1** (TRUE)	**1** (TRUE)
CKA_ EXTRACTABLE	N/A	**0** (FALSE)	**0** (FALSE)	**0** (FALSE)
CKA_ALWAYS_ SENSITIVE	N/A	Always the same value as CKA_ SENSITIVE	Always **0** (FALSE)	Inherited from base key(s) depending on CKA_SENSITIVE history*
CKA_NEVER_ EXTRACTABLE	N/A	Always the opposite value of CKA_ EXTRACTABLE	Always **0** (FALSE)	Inherited from base key(s) depending on CKA_ EXTRACTABLE history**

> **NOTE** If using a Luna Cloud HSM service you must specify both CKA_PRIVATE=1 and CKA_ SENSITIVE=1 Key Attributes for all Generated, Derived and Unwrapped keys.

* CKA_ALWAYS_SENSITIVE=1 assures that the key and the key(s) from which it was derived have always been sensitive (CKA_SENSITIVE=1). If a newly-derived key has CKA_ALWAYS_SENSITIVE=0, it means the key(s) from which it derives has had CKA_SENSITIVE=0 at some point.

** CKA_NEVER_EXTRACTABLE=1 assures that the key and the key(s) from which it was derived have never been extractable (CKA_EXTRACTABLE has always been set to 0). If a newly-derived key has CKA_NEVER_ EXTRACTABLE=0, it means the key(s) from which it derives has had CKA_EXTRACTABLE=1 at some point.

Key Usage Attributes

Attribute	Default Value			
	Generated Public Keys	**Generated Private Keys**	**Unwrapped Private/Secret Keys**	**Derived Secret Keys**
CKA_ENCRYPT	**0** (FALSE)	N/A	**0** (FALSE)	**0** (FALSE)
CKA_DECRYPT	N/A	**0** (FALSE)	**0** (FALSE)	**0** (FALSE)
CKA_WRAP	**0** (FALSE)	N/A	**0** (FALSE)	**0** (FALSE)
CKA_UNWRAP	N/A	**0** (FALSE)	**0** (FALSE)	**0** (FALSE)
CKA_SIGN	N/A	**0** (FALSE)	**0** (FALSE)	**0** (FALSE)
CKA_VERIFY	**0** (FALSE)	N/A	**0** (FALSE)	**0** (FALSE)
CKA_DERIVE	**0** (FALSE)	N/A	**0** (FALSE)	**0** (FALSE)

Vendor-defined key attributes

KEY ATTRIBUTE	DESCRIPTION
CKA_CCM_PRIVATE	Not used by current Luna HSMs; it does not affect any of the HSM functionality.
CKA_OUID	This is a 12-byte unique identifier for the object, unique across all Luna HSMs. It can be used to identify the object across multiple HSM.
CKA_EKM_UID	This is not used by the Luna HSM, it does not affect any of the HSM functionality. It is intended to be used by our EKM Key Manager SHIM to store a KEY ID, so that the key manager can track keys efficiently. Customer applications should not use this (they should use the CKA_GENERIC_1/2/3 attributes defined below).
CKA_GENERIC_1/2/3	These are not used by the Luna HSM, and do not affect any of the HSM functionality. They are variable length attributes that store an array of CK_BYTE and are provided for customer applications to make use of, to store whatever data they want.

Object Usage Count

You may wish to create keys that have a limited number of uses. You can set attributes on a key object to track and limit the number of cryptographic operations that object may perform. The relevant attributes are:

> CKA_USAGE_COUNT: the number of operations that have been performed using the key

> CKA_USAGE_LIMIT: the maximum number of operations allowed for the key.

When the limit set by CKA_USAGE_LIMIT is reached, attempts to use the key for operations like encrypt/decrypt, sign/verify, etc. will return an error (CKR_KEY_NOT_ACTIVE).

Setting CKA_USAGE_LIMIT on a key using CKDEMO

You can use CKDEMO to set this limit for a specific key on the HSM.

To set CKA_USAGE_LIMIT on a key:

1. Navigate to the Luna HSM Client directory and run CKDEMO.
2. Select **Option 1 (Open Session)**.
3. Select **Option 3 (Login)**, select the partition where the key is located, and present the Crypto Officer login credential.
4. If you do not know the key's object handle, select **Option 27 (Display Object)** and enter 0 to view a list of available objects.
5. Select **Option 25 (Set Attribute)** and enter the key's object handle when prompted.
6. Select **Sub-option 1 (Add Attribute)**, and **53 (CKA_USAGE_LIMIT)** from the list of attributes.
7. Enter the desired maximum number of uses in hexadecimal (Allowable range: 1 - FFFFFFFF).
8. Select **Option 27** and enter the key's object handle to view the key attributes. When you set CKA_USAGE_LIMIT in step 7, CKA_USAGE_COUNT is also set, with a value of 0:

```
Enter your choice: 27

Enter handle of object to display (0 to list available objects) : 247
Object handle=247
CKA_CLASS=0003 (3)
CKA_TOKEN=01
CKA_PRIVATE=01
CKA_LABEL=Generated RSA Private Key
CKA_KEY_TYPE=0000 (0)
CKA_SUBJECT=
CKA_ID=
CKA_SENSITIVE=01
CKA_DECRYPT=01
CKA_UNWRAP=01
CKA_SIGN=01
CKA_SIGN_RECOVER=00
CKA_DERIVE=00
CKA_START_DATE=
CKA_END_DATE=
CKA_MODULUS=bc613525ae8c5b30ca086c0e688f2f0ed6928805bf007d4fc...
CKA_MODULUS_BITS=0400 (1024)
CKA_PUBLIC_EXPONENT=010001
CKA_LOCAL=01
CKA_MODIFIABLE=01
CKA_EXTRACTABLE=01
CKA_ALWAYS_SENSITIVE=01
CKA_NEVER_EXTRACTABLE=00
CKA_CCM_PRIVATE=00
CKA_FINGERPRINT_SHA1=6beddef34f9f5c8023e3422daecd6bd91c2dc40d
CKA_OUID=b00800000300000d1b030100
CKA_X9_31_GENERATED=00
```

```
CKA_EKM_UID=
CKA_USAGE_LIMIT=000e (15)
CKA_USAGE_COUNT=0000 (0)
CKA_GENERIC_1=
CKA_GENERIC_2=
CKA_GENERIC_3=
CKA_FINGERPRINT_SHA256=a8293ea9ddb578bcca644279c9753de4df772958563d259bed28c5d2a2e04e7d

Status: Doing great, no errors (CKR_OK)
```

Using this key to perform cryptographic operations will now increment the value of CKA_USAGE_COUNT.

Creating multiple keys with CKA_USAGE_LIMIT using CKDEMO

If you are creating multiple, usage-limited keys in CKDEMO, you can simplify this procedure by changing a CKDEMO setting. You will then have the option to set a usage limit for all new keys created in that session.

To create multiple keys with CKA_USAGE_LIMIT set:

1. Navigate to the Luna HSM Client directory and run CKDEMO.
2. Select **Option 98 (Options)**.
3. Select **Option 10 (Object Usage Counters)**.

   Note that the option value has changed from "disabled" to "selectable".
4. Enter **0** to exit the **(Options)** menu.
5. Open a session and begin creating your new keys. In addition to setting the attributes governing key capabilities, you will be prompted to enter a value for CKA_USAGE_LIMIT (in hexadecimal):

```
Select type of key to generate
[ 1] DES     [ 2] DES2    [ 3] DES3             [ 5]  CAST3
[ 6] Generic [ 7] RSA     [ 8] DSA    [ 9] DH    [10]  CAST5
[11] RC2     [12] RC4     [13] RC5    [14] SSL3  [15]  ECDSA
[16] AES     [17] SEED    [18] KCDSA-1024   [19] KCDSA-2048
[20] DSA Domain Param     [21] KCDSA Domain Param
[22] RSA X9.31            [23] DH X9.42         [24] ARIA
[25] DH PKCS Domain Param [26] RSA 186-3 Aux Primes
[27] RSA 186-3 Primes      [28] DH X9.42 Domain Param
[29] ECDSA with Extra Bits [30] EC Edwards
[31] EC Montgomery
> 7

Enter Key Length in bits: 1024

Enter Is Token Attribute [0-1]: 1

Enter Is Sensitive Attribute [0-1]: 1

Enter Is Private Attribute [0-1]: 1

Enter Is Modifiable Attribute [0-1]: 1

Enter Extractable Attribute [0-1]: 1

Enter Encrypt/Decrypt Attribute [0-1]: 1
```

```
Enter Sign/Verify Attribute [0-1]: 1

Enter Wrap/Unwrap Attribute [0-1]: 1

Enter Derive Attribute [0-1]: 1
Would you like to specify a usage count limit? [0-no, 1-yes]: 1
Please enter the limit in HEX: 0E
Generated RSA Public Key:        160 (0x000000a0)
Generated RSA Private Key:        247 (0x000000f7)

Status: Doing great, no errors (CKR_OK)
```

Migrating Keys From Software to a Luna PCIe HSM 7

Luna PCIe HSM 7s expect key material to be in PKCS#8 format. PKCS#8 format follows BER (Basic encoding rules)/DER (distinguished encoding rules) encoding. An example of this format can be found in the document "Some examples of PKCS standards" produced by RSA, and available on their web site (http://www.rsasecurity.com/rsalabs/pkcs/index.html at the bottom of the page, under “Related Documents”).

Here is an example of a formatted key:

```
0x30,
0x82, 0x04, 0xbc, 0x02, 0x01, 0x00, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86,
0x48, 0x86, 0xf7, 0x0d, 0x01, 0x01, 0x01, 0x05, 0x00, 0x04, 0x82, 0x04,
0xa6, 0x30, 0x82, 0x04, 0xa2, 0x02, 0x01, 0x00, 0x02, 0x82, 0x01, 0x01,
0x00, 0xb8, 0xb5, 0x0f, 0x49, 0x46, 0xb5, 0x5d, 0x58, 0x04, 0x8e, 0x52,
0x59, 0x39, 0xdf, 0xd6, 0x29, 0x45, 0x6b, 0x6c, 0x96, 0xbb, 0xab, 0xa5,
0x6f, 0x72, 0x1b, 0x16, 0x96, 0x74, 0xd5, 0xf9, 0xb4, 0x41, 0xa3, 0x7c,
0xe1, 0x94, 0x73, 0x4b, 0xa7, 0x23, 0xff, 0x61, 0xeb, 0xce, 0x5a, 0xe7,
0x7f, 0xe3, 0x74, 0xe8, 0x52, 0x5b, 0xd6, 0x5d, 0x5c, 0xdc, 0x98, 0x49,
0xfe, 0x51, 0xc2, 0x7e, 0x8f, 0x3b, 0x37, 0x5c, 0xb3, 0x11, 0xed, 0x85,
0x91, 0x15, 0x92, 0x24, 0xd8, 0xf1, 0x7b, 0x3d, 0x2f, 0x8b, 0xcd, 0x1b,
0x30, 0x14, 0xa3, 0x6b, 0x1b, 0x4d, 0x27, 0xff, 0x6a, 0x58, 0x84, 0x9e,
0x79, 0x94, 0xca, 0x78, 0x64, 0x01, 0x33, 0xc3, 0x58, 0xfc, 0xd3, 0x83,
0xeb, 0x2f, 0xab, 0x6f, 0x85, 0x5a, 0x38, 0x41, 0x3d, 0x73, 0x20, 0x1b,
0x82, 0xbc, 0x7e, 0x76, 0xde, 0x5c, 0xfe, 0x42, 0xd6, 0x7b, 0x86, 0x4f,
0x79, 0x78, 0x29, 0x82, 0x87, 0xa6, 0x24, 0x43, 0x39, 0x74, 0xfe, 0xf2,
0x0c, 0x08, 0xbe, 0xfa, 0x1e, 0x0a, 0x48, 0x6f, 0x14, 0x86, 0xc5, 0xcd,
0x9a, 0x98, 0x09, 0x2d, 0xf3, 0xf3, 0x5a, 0x7a, 0xa4, 0xe6, 0x8a, 0x2e,
0x49, 0x8a,  0xde,
0x73, 0xe9, 0x37, 0xa0, 0x5b, 0xef, 0xd0, 0xe0, 0x13, 0xac, 0x88, 0x5f,
0x59, 0x47, 0x96, 0x7f, 0x78, 0x18, 0x0e, 0x44, 0x6a, 0x5d,  0xec,
0x6e, 0xed, 0x4f, 0xf6, 0x6a, 0x7a, 0x58, 0x6b, 0xfe, 0x6c, 0x5a, 0xb9,
0xd2, 0x22, 0x3a, 0x1f, 0xdf, 0xc3, 0x09, 0x3f, 0x6b, 0x2e, 0xf1, 0x6d,
0xc3, 0xfb, 0x4e, 0xd4, 0xf2, 0xa3, 0x94, 0x13, 0xb0, 0xbf, 0x1e, 0x06,
0x2e, 0x29, 0x55, 0x00, 0xaa, 0x98, 0xd9, 0xe8, 0x77, 0x84, 0x8b, 0x3f,
0x5f, 0x5e, 0xf7, 0xf8, 0xa7, 0xe6, 0x02, 0xd2, 0x18, 0xb0, 0x52, 0xd0,
0x37, 0x2e, 0x53, 0x02, 0x03, 0x01, 0x00, 0x01, 0x02, 0x82, 0x01, 0x00,
0x0c, 0xdf, 0xd1, 0xe8, 0xf1, 0x9c, 0xc2, 0x9c, 0xd7, 0xf4, 0x73, 0x98,
0xf4, 0x87, 0xbd, 0x8d, 0xb2, 0xe1, 0x01, 0xf8, 0x9f, 0xac, 0x1f, 0x23,
0xdd, 0x78, 0x35, 0xe2, 0xd6, 0xd1, 0xf3, 0x4d, 0xb5, 0x25, 0x88, 0x16,
0xd1, 0x1a, 0x18, 0x33, 0xd6, 0x36, 0x7e, 0xc4, 0xc8, 0xe5, 0x5d, 0x2d,
0x74, 0xd5, 0x39, 0x3c, 0x44, 0x5a, 0x74, 0xb7, 0x7c, 0x48, 0xc1, 0x1f,
0x90, 0xe3, 0x55, 0x9e, 0xf6, 0x29, 0xad, 0xb4, 0x6d, 0x93, 0x78, 0xb3,
0xdc, 0x25, 0x0b, 0x9c, 0x73, 0x78, 0x7b, 0x93, 0x4c, 0xd3, 0x47, 0x09,
0xda, 0xe6, 0x69, 0x18, 0xc6, 0x0f, 0xfb, 0xa5, 0x95, 0xf5, 0xe8, 0x75,
0xe1, 0x01, 0x1b, 0xd3, 0x1c, 0xa2, 0x57, 0x03, 0x64, 0xdb, 0xf9, 0x5d,
```

```
0xf3, 0x3c, 0xa7, 0xd1, 0x4b, 0xb0, 0x90, 0x1b, 0x90, 0x62, 0xb4, 0x88,
0x30, 0x4b, 0x40, 0x4d, 0xcf, 0x7d, 0x89, 0x7a, 0xfb, 0x29, 0xc9, 0x64,
0x34, 0x0a, 0x52, 0xf6, 0x70, 0x7c, 0x76, 0x5a, 0x2e, 0x8f, 0x50, 0xd4,
0x92, 0x15, 0x97, 0xed, 0x4c, 0x2e, 0xf2, 0x3a, 0xd0, 0x58, 0x7e, 0xdb,
0xf1, 0xf4, 0xdd, 0x07, 0x76, 0x04, 0xf0, 0x55, 0x8b, 0x72, 0x2b, 0xa7,
0xa8, 0x78, 0x78, 0x67, 0xe6, 0xd8, 0xa5, 0xde, 0xe7, 0xc9, 0x1f, 0x5a,
0xa0, 0x89, 0xc7, 0x24, 0xa2, 0x71, 0xb6, 0x7b, 0x3b, 0xe6, 0x92, 0x69,
0x22, 0xaa, 0xe2, 0x47, 0x4b, 0x80, 0x3f, 0x6a, 0xab, 0xce, 0x4e, 0xcd,
0xe8, 0x94, 0x6c, 0xf7, 0x84, 0x73, 0x85, 0xfd, 0x85, 0x1d, 0xae, 0x81,
0xf7, 0xec, 0x12, 0x31, 0x7d, 0xc1, 0x99, 0xc0, 0x3c, 0x51, 0xb0, 0xdc,
0xb0, 0xba, 0x9c, 0x84, 0xb8, 0x70, 0xc2, 0x09, 0x7f, 0x96, 0x3d, 0xa1,
0xe2, 0x64, 0x27, 0x7a, 0x22, 0xb8, 0x75, 0xb9, 0xd1, 0x5f, 0xa5, 0x23,
0xf9, 0x62, 0xe0, 0x41, 0x02, 0x81, 0x81, 0x00, 0xf4, 0xf3, 0x08, 0xcf,
0x83, 0xb0, 0xab, 0xf2, 0x0f, 0x1a, 0x08, 0xaf, 0xc2, 0x42, 0x29, 0xa7,
0x9c, 0x5e, 0x52, 0x19, 0x69, 0x8d, 0x5b, 0x52, 0x29, 0x9c, 0x06, 0x6a,
0x5a, 0x32, 0x8f, 0x08, 0x45, 0x6c, 0x43, 0xb5, 0xac, 0xc3, 0xbb, 0x90,
0x7b, 0xec, 0xbb, 0x5d, 0x71, 0x25, 0x82, 0xf8, 0x40, 0xbf, 0x38, 0x00,
0x20, 0xf3, 0x8a, 0x38, 0x43, 0xde, 0x04, 0x41, 0x19, 0x5f, 0xeb, 0xb0,
0x50, 0x59, 0x10, 0xe1, 0x54, 0x62, 0x5c, 0x93, 0xd9, 0xdc, 0x63, 0x24,
0xd0, 0x17, 0x00, 0xc0, 0x44, 0x3e, 0xfc, 0xd1, 0xda, 0x4b, 0x24, 0xf7,
0xcb, 0x16, 0x35, 0xe6, 0x9f, 0x67, 0x96, 0x5f, 0xb0, 0x94, 0xde, 0xfa,
0xa1, 0xfd, 0x8c, 0x8a, 0xd1, 0x5c, 0x02, 0x8d, 0xe0, 0xa0, 0xa0,  0x02,
0x1d, 0x56, 0xaf, 0x13, 0x3a, 0x65, 0x5e, 0x8e, 0xde, 0xd1, 0xa8, 0x28,
0x8b, 0x71, 0xc9, 0x65, 0x02, 0x81, 0x81, 0x00, 0xc1, 0x0a,  0x47,
0x39, 0x91, 0x06, 0x1e, 0xb9, 0x43, 0x7c, 0x9e, 0x97, 0xc5, 0x09, 0x08,
0xbc, 0x22, 0x47, 0xe2, 0x96, 0x8e, 0x1c, 0x74, 0x80, 0x50, 0x6c, 0x9f,
0xef, 0x2f, 0xe5, 0x06, 0x3e, 0x73, 0x66, 0x76, 0x02, 0xbd, 0x9a, 0x1c,
0xfc, 0xf9, 0x6a, 0xb8, 0xf9, 0x36, 0x15, 0xb5, 0x20, 0x0b, 0x6b, 0x54,
0x83, 0x9c, 0x86, 0xba, 0x13, 0xb7, 0x99, 0x54, 0xa0, 0x93, 0x0d, 0xd6,
0x1e, 0xc1, 0x12, 0x72, 0x0d, 0xea, 0xb0, 0x14, 0x30, 0x70, 0x73, 0xef,
0x6b, 0x4c, 0xae, 0xb6, 0xff, 0xd4, 0xbb, 0x89, 0xa1, 0xec, 0xca, 0xa6,
0xe9, 0x95, 0x56, 0xac, 0xe2, 0x9b, 0x97, 0x2f, 0x2c, 0xdf, 0xa3, 0x6e,
0x59, 0xff, 0xcd, 0x3c, 0x6f, 0x57, 0xcc, 0x6e, 0x44, 0xc4, 0x27, 0xbf,
0xc3, 0xdd, 0x19, 0x9e, 0x81, 0x16, 0xe2, 0x8f, 0x65, 0x34, 0xa7, 0x0f,
0x22, 0xba, 0xbf, 0x79, 0x57, 0x02, 0x81, 0x80, 0x2e, 0x21, 0x0e, 0xc9,
0xb5, 0xad, 0x31, 0xd4, 0x76, 0x0f, 0x9b, 0x0f, 0x2e, 0x70, 0x33, 0x54,
0x03, 0x58, 0xa7, 0xf1, 0x6d, 0x35, 0x57, 0xbb, 0x53, 0x66, 0xb4, 0xb6,
0x96, 0xa1, 0xea, 0xd9, 0xcd, 0xe9, 0x23, 0x9f, 0x35, 0x17, 0xef, 0x5c,
0xb8, 0x59, 0xce, 0xb7, 0x3c, 0x35, 0xaa, 0x42, 0x82, 0x3f, 0x00, 0x96,
0xd5, 0x9d, 0xc7, 0xab, 0xec, 0xec, 0x04, 0xb5, 0x15, 0xc8, 0x40, 0xa4,
0x85, 0x9d, 0x20, 0x56, 0xaf, 0x03, 0x8f, 0x17, 0xb0, 0xf1, 0x96, 0x22,
0x3a, 0xa5, 0xfa, 0x58, 0x3b, 0x01, 0xf9, 0xae, 0xb3, 0x83, 0x6f, 0x44,
0xd3, 0x14, 0x2d, 0xb6, 0x6e, 0xd2, 0x9d, 0x39, 0x0c, 0x12, 0x1d, 0x23,
0xea, 0x19, 0xcb, 0xbb, 0xe0, 0xcd, 0x89, 0x15, 0x9a, 0xf5, 0xe4, 0xec,
0x41, 0x06, 0x30, 0x16, 0x58, 0xea, 0xfa, 0x31, 0xc1, 0xb8, 0x8e, 0x08,
0x84, 0xaa, 0x3b, 0x19, 0x02, 0x81, 0x80, 0x70, 0x4c, 0xf8, 0x6e, 0x86,
0xed, 0xd6, 0x85, 0xd4, 0xba, 0xf4, 0xd0, 0x3a, 0x32, 0x2d, 0x40, 0xb5,
0x78, 0xb8, 0x5a, 0xf9, 0xc5, 0x98, 0x08, 0xe5, 0xc0, 0xab, 0xb2, 0x4c,
0x5c, 0xa2, 0x2b, 0x46, 0x9b, 0x3e, 0xe0, 0x0d, 0x49, 0x50, 0xbf, 0xe2,
0xa1, 0xb1, 0x86, 0x59, 0x6e, 0x7b, 0x76, 0x6e, 0xee, 0x3b, 0xb6, 0x6d,
0x22, 0xfb, 0xb1, 0x68, 0xc7, 0xec, 0xb1, 0x95, 0x9b, 0x21, 0x0b, 0xb7,
0x2a, 0x71, 0xeb, 0xa2, 0xb2, 0x58, 0xac, 0x6d, 0x5f, 0x24, 0xd3, 0x79,
0x42, 0xd2, 0xf7, 0x35, 0xdc, 0xfc, 0x0e, 0x95, 0x60, 0xb7, 0x85, 0x7f,
0xf9, 0x72, 0x8e, 0x4a, 0x11, 0xc3, 0xc2, 0x09, 0x40, 0x5c, 0x7c, 0x43,
0x12, 0x34, 0xac, 0x59, 0x99, 0x76, 0x34,  0xcf,
0x20, 0x88, 0xb0, 0xfb, 0x39, 0x62, 0x3a, 0x9b, 0x03, 0xa6, 0x84, 0x2c,
0x03, 0x5c, 0x0c, 0xca, 0x33, 0x85, 0xf5, 0x02, 0x81, 0x80,  0x56,
0x99, 0xe9, 0x17, 0xdc, 0x33, 0xe1, 0x33, 0x8d, 0x5c, 0xba, 0x17, 0x32,
```

```
 0xb7, 0x8c, 0xbd, 0x4b, 0x7f, 0x42, 0x3a, 0x79, 0x90, 0xe3,  0x70,
 0xe3, 0x27, 0xce, 0x22, 0x59, 0x02, 0xc0, 0xb1, 0x0e, 0x57, 0xf5, 0xdf,
 0x07, 0xbf, 0xf8, 0x4e, 0x10, 0xef, 0x2a, 0x62, 0x30, 0x03,  0xd4,
 0x80, 0xcf, 0x20, 0x84, 0x25, 0x66, 0x3f, 0xc7, 0x4f, 0x56, 0x8c, 0x1e,
 0xe1, 0x18, 0x91, 0xc1, 0xfd, 0x71, 0x5f, 0x65, 0x9b, 0xe4,  0x4f,
 0xe0, 0x1a, 0x3a, 0xf8, 0xc1, 0x69, 0xdb, 0xd3, 0xbb, 0x8d, 0x91, 0xd1,
 0x11, 0x4f, 0x7e, 0x91, 0x1b, 0xb4, 0x27, 0xa5, 0xab, 0x7c,  0x7b,
 0x76, 0xd4, 0x78, 0xfe, 0x63, 0x44, 0x63, 0x7e, 0xe3, 0xa6, 0x60, 0x4f,
 0xb9, 0x55, 0x28, 0xba, 0xba, 0x83, 0x1a, 0x2d, 0x43, 0xd5,  0xf7,
 0x2e, 0xe0, 0xfc, 0xa8, 0x14, 0x9b, 0x91, 0x2a, 0x36, 0xbf, 0xc7, 0x14
```

The example above contains the exponent, the modulus, and private key material.

Other Formats of Key Material

The format of key material depends on the application, and is therefore unpredictable. Key material commonly exists in any of the following formats; ASN1, PEM, P12, PFX, etc. Key material in those formats, or in another format, can likely be re-formatted to be acceptable for moving onto the Luna PCIe HSM 7.

Sample Program

The sample program below encrypts a known RSA private key, then unwraps the key pair onto the Luna PCIe HSM 7 partition.

```
/*****************************************************************************\
*
* File: UnwrapKey.cpp*
* Encrypts a PrivateKeyInfo structure with a generated DES
 key and then
* unwraps the RSA key onto a token.
*
* This file is provided as an example only.
*
*
* Copyright (C) 2020, Thales Group.
*
* All rights reserved. This file contains information that
 is
* proprietary to SafeNet, Inc. and may not be
* distributed or copied without written consent from
* SafeNet, Inc.
*
\*************************************************************/
#ifdef UNIX
#define _POSIX_SOURCE 1
#endif
#ifdef USING_STATIC_CHRYSTOKI
# define STATIC  ckdemo_cpp
#endif
#include <assert.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <time.h>
#ifdef _WINDOWS
#include <conio.h>
#include <io.h>
```

```
#include <windows.h>
#endif
#ifdef UNIX
#include <unistd.h>
#endif
#include "source/cryptoki.h"
#include "source/Ckbridge.h"
#define DIM(a)                    (sizeof(a)/sizeof(a[0]))
CK_BBOOL no = FALSE;
CK_BBOOL yes = TRUE;
const int MAX =100;
// Function Prototypes
CK_RV  Pinlogin(CK_SESSION_HANDLE
 hSession);
int getPinString(CK_CHAR_PTR pw);
// Main
int main( void )
{
   int
               error
 = 0;
   CK_RV
            retCode
 = CKR_OK;
   CK_SESSION_HANDLE
 hSessionHandle;
   CK_CHAR_PTR
       myuserpin
 = (CK_CHAR_PTR)"default";
   CK_USHORT
         lenmyuserpin
  = 7;
   CK_CHAR_PTR
       soPIN
 = (CK_CHAR_PTR)"default";
   CK_USHORT
         lensoPIN
 = 7;
   CK_USHORT
         usNumberOfSlots;
   CK_SLOT_ID_PTR
    pSlotList;
   CK_OBJECT_HANDLE
 hKey;
   CK_MECHANISM
 mech;
   CK_VERSION
 version;
   struct
   {
      CK_INFO
 info;
      char
 reserved[100]; // This is in case the library that we are
                        //
 talking to requires a larger info structure
                        //
 then the one defined.
   }
 protectedInfo;
//Disclaimer
```

```
   cout
 << "\n\n\n\n";
cout << "THE SOFTWARE IS PROVIDED BY SAFENET INCORPORATED
 (SAFENET) ON AN 'AS IS' BASIS, \n";
cout << "WITHOUT ANY OTHER WARRANTIES OR CONDITIONS,
 EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED \n";
cout << "TO, WARRANTIES OF MERCHANTABLE QUALITY,
 SATISFACTORY QUALITY, MERCHANTABILITY OR FITNESS FOR\n";
cout << "A PARTICULAR PURPOSE, OR THOSE ARISING
 BY LAW, STATUTE, USAGE OF TRADE, COURSE OF DEALING OR\n";
cout << "OTHERWISE.  SAFENET
 DOES NOT WARRANT THAT THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR \n";
cout << "THAT OPERATION OF THE SOFTWARE WILL BE
 UNINTERRUPTED OR THAT THE SOFTWARE WILL BE ERROR-FREE.\n";
cout << "YOU ASSUME THE ENTIRE RISK AS TO THE
 RESULTS AND PERFORMANCE OF THE SOFTWARE.  NEITHER
 \n";
cout << "SAFENET NOR OUR LICENSORS, DEALERS OR
 SUPPLIERS SHALL HAVE ANY LIABILITY TO YOU OR ANY\n";
cout << "OTHER PERSON OR ENTITY FOR ANY INDIRECT,
 INCIDENTAL, SPECIAL, CONSEQUENTIAL, PUNITIVE, \n";
cout << "EXEMPLARY OR AY OTHER DAMAGES WHATSOEVER,
 INCLUDING, BUT NOT LIMITED TO, LOSS OF REVENUE OR \n";
cout << "PROFIT, LOST OR DAMAGED DATA, LOSS OF
 USE OR OTHER COMMERCIAL OR ECONOMIC LOSS, EVEN IF \n";
cout << "SAFENET HAS BEEN ADVISED OF THE POSSIBILITY
 OF SUCH DAMAGES, OR THEY ARE FORESEEABLE. \n";
cout << "SAFENET IS ALSO NOT RESPONSIBLE FOR CLAIMS
 BY A THIRD PARTY.  THE
 MAXIMUM AGGREGATE \n";
cout << "LIABILITY OF SAFENET TO YOU AND THAT
 OF SAFENET'S LICENSORS, DEALERS AND SUPPLIERS \n";
cout << "SHALL NOT EXCEED FORTY DOLLARS ($40.00CDN).
  THE LIMITATIONS
 IN THIS SECTION SHALL APPLY \n";
cout << "WHETHER OR NOT THE ALLEGED BREACH OR
 DEFAULT IS A BREACH OF A FUNDAMENTAL CONDITION OR TERM \n";
cout << "OR A FUNDAMENTAL BREACH.  SOME
 STATES/COUNTRIES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF\n";
cout << "LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL
 DAMAGES, SO THE ABOVE LIMITATION MAY NOT APPLY TO \n";
cout << "YOU.\n";
cout << "THE LIMITED WARRANTY, EXCLUSIVE REMEDIES
 AND LIMITED LIABILITY SET OUT HEREIN ARE FUNDAMENTAL \n";
cout << "ELEMENTS OF THE BASIS OF THE BARGAIN
 BETWEEN YOU AND SAFENET.  \n";
cout << "NO SUPPORT.  YOU
 ACKNOWLEDGE AND AGREE THAT THERE ARE NO SUPPORT SERVICES PROVIDED BY SAFENET\n";
cout << "INCORPORATED FOR THIS SOFTWARE\n"
 << endl;
   //
 Display Generic Warning
   cout
 << "\nInsert a token for the test...";
   cout
 << "\n\nWARNING!!! This test initializes the first ";
   cout
 << " token detected in the card reader.";
   cout
 << "\nDo not use a token that you don't want erased.";
```

```
   cout
 << "\nYou can use CTRL-C to abort now...Otherwise...";
   cout
 << "\n\n... press <Enter> key to continue ...\n";
   cout.flush();
   getchar();
 // Wait for keyboard hit
#ifndef STATIC
   //
 Connect to Chrystoki
if(!CrystokiConnect())
{
cout << "\n" "Unable to connect to Chrystoki.
  Error =
 " << LibError() << "\n";
error = -1;
      goto
 exit_routine_1;
}
#endif
   //
 Verify this is the version of the library required
   retCode
 = C_GetInfo(&protectedInfo.info);
   if(
 retCode != CKR_OK )
   {
      cout
 << endl << "Unable to call C_GetInfo() before C_Initialize()\n";
error = -2;
      goto
 exit_routine_2;
   }
   else
   {
      CK_BYTE
 majorVersion = protectedInfo.info.version.major;
      CK_BYTE
 expectedVersion;
#ifndef PKCS11_2_0
      expectedVersion
 = 1;
#else
      expectedVersion
 = 2;
#endif
      if(
 expectedVersion != majorVersion )
      {
         cout
 << endl << "This version of the program was built for
 Cryptoki version "
              <<
 (int)expectedVersion << ".\n"
              <<
 "The loaded Cryptoki library reports its version to be "
              <<
 (int)majorVersion << ".\n"
              <<
 "Program will terminate.\n";
```

```
        //
 Wait to exit until user read message and acknowledges
        cout
 << endl << "Press <Enter> key to end.";
        getchar();
 // Wait for keyboard hit
   error
 = -3;
        goto
 exit_routine_2;
      }
   }
   //
 Initialize the Library
retCode = C_Initialize(NULL);
if(retCode != CKR_OK)
{
cout << "\n" "Error 0x" <<
 hex << retCode << " initializing cryptoki.\n";
error = -4;
      goto
 exit_routine_3;
}
// Get the number of tokens possibly available
retCode = C_GetSlotList(TRUE, NULL, &usNumberOfSlots);
if(retCode != CKR_OK)
{
cout << "\n" "Error 0x" <<
 hex << retCode << " getting slot list.\n";
error = -5;
      goto
 exit_routine_3;
}
// Are any tokens present?
if(usNumberOfSlots == 0)
{
cout << "\n" "No tokens found\n";
error = -6;
      goto
 exit_routine_3;
   }
   //
 Get a list of slots
pSlotList = new CK_SLOT_ID[usNumberOfSlots];
retCode = C_GetSlotList(TRUE, pSlotList, &usNumberOfSlots);
if(retCode != CKR_OK)
{
cout << "\n" "Error 0x" <<
 hex << retCode << " getting slot list.\n";
error = -7;
      goto
 exit_routine_4;
}
   //
 Open a session
retCode = C_OpenSession(pSlotList[0], CKF_RW_SESSION | CKF_SERIAL_SESSION,

      NULL,
 NULL, &hSessionHandle);
if(retCode != CKR_OK)
{
```

```
cout << "\n" "Error 0x" <<
 hex << retCode << " opening session.\n";
error = -9;
      goto
 exit_routine_4;
}
Pinlogin(hSessionHandle);
if(retCode != CKR_OK)
{
cout << "\n" "Error 0x" <<
 hex << retCode << " Calling PinLogin fn";
exit(hSessionHandle);
}
  //
 Encrypt an RSA Key and then unwrap it onto the token
   {
      //
 The following is an RSA Key that is formatted as a PrivateKeyInfo structure
   //BER
 encoded format
      const
 CK_BYTE pRsaKey[] = {
 0x30,
 0x82, 0x04, 0xbc, 0x02, 0x01, 0x00, 0x30, 0x0d, 0x06, 0x09, 0x2a, 0x86,
 0x48, 0x86, 0xf7, 0x0d, 0x01, 0x01,
0x01, 0x05, 0x00, 0x04,
 0x82,
 0x04, 0xa6, 0x30, 0x82, 0x04, 0xa2, 0x02, 0x01, 0x00, 0x02, 0x82, 0x01,
 0x01, 0x00, 0xb8, 0xb5, 0x0f, 0x49,
0x46, 0xb5, 0x5d, 0x58,
 0x04,
 0x8e, 0x52, 0x59, 0x39, 0xdf, 0xd6, 0x29, 0x45, 0x6b, 0x6c, 0x96, 0xbb,
 0xab, 0xa5, 0x6f, 0x72, 0x1b, 0x16,
0x96, 0x74, 0xd5, 0xf9,
 0xb4,
 0x41, 0xa3, 0x7c, 0xe1, 0x94, 0x73, 0x4b, 0xa7, 0x23, 0xff, 0x61, 0xeb,
 0xce, 0x5a, 0xe7, 0x7f, 0xe3, 0x74,
0xe8, 0x52, 0x5b, 0xd6,
 0x5d,
 0x5c, 0xdc, 0x98, 0x49, 0xfe, 0x51, 0xc2, 0x7e, 0x8f, 0x3b, 0x37, 0x5c,
 0xb3, 0x11, 0xed, 0x85, 0x91, 0x15,
0x92, 0x24, 0xd8, 0xf1,
 0x7b,
 0x3d, 0x2f, 0x8b, 0xcd, 0x1b, 0x30, 0x14, 0xa3, 0x6b, 0x1b, 0x4d, 0x27,
 0xff, 0x6a, 0x58, 0x84, 0x9e, 0x79,
0x94, 0xca, 0x78, 0x64,
 0x01,
 0x33, 0xc3, 0x58, 0xfc, 0xd3, 0x83, 0xeb, 0x2f, 0xab, 0x6f, 0x85, 0x5a,
 0x38, 0x41, 0x3d, 0x73, 0x20, 0x1b,
0x82, 0xbc, 0x7e, 0x76,
 0xde,
 0x5c, 0xfe, 0x42, 0xd6, 0x7b, 0x86, 0x4f, 0x79, 0x78, 0x29, 0x82, 0x87,
 0xa6, 0x24, 0x43, 0x39, 0x74, 0xfe,
0xf2, 0x0c, 0x08, 0xbe,
 0xfa,
 0x1e, 0x0a, 0x48, 0x6f, 0x14, 0x86, 0xc5, 0xcd, 0x9a, 0x98, 0x09, 0x2d,
 0xf3, 0xf3, 0x5a, 0x7a, 0xa4, 0xe6,
0x8a, 0x2e, 0x49, 0x8a, 0xde, 0x73, 0xe9, 0x37, 0xa0, 0x5b,
 0xef,
```

```
 0xd0,
 0xe0, 0x13, 0xac, 0x88, 0x5f, 0x59, 0x47, 0x96, 0x7f, 0x78, 0x18, 0x0e,
 0x44, 0x6a, 0x5d, 0xec, 0x6e, 0xed,
0x4f, 0xf6, 0x6a, 0x7a,
 0x58,
 0x6b, 0xfe, 0x6c, 0x5a, 0xb9, 0xd2, 0x22, 0x3a, 0x1f, 0xdf, 0xc3, 0x09,
 0x3f, 0x6b, 0x2e, 0xf1, 0x6d, 0xc3,
0xfb, 0x4e, 0xd4, 0xf2,
 0xa3,
 0x94, 0x13, 0xb0, 0xbf, 0x1e, 0x06, 0x2e, 0x29, 0x55, 0x00, 0xaa, 0x98,
 0xd9, 0xe8, 0x77, 0x84, 0x8b, 0x3f,
0x5f, 0x5e, 0xf7, 0xf8,
 0xa7,
 0xe6, 0x02, 0xd2, 0x18, 0xb0, 0x52, 0xd0, 0x37, 0x2e, 0x53, 0x02, 0x03,
 0x01, 0x00, 0x01, 0x02, 0x82, 0x01,
0x00, 0x0c, 0xdf, 0xd1,
 0xe8,
 0xf1, 0x9c, 0xc2, 0x9c, 0xd7, 0xf4, 0x73, 0x98, 0xf4, 0x87, 0xbd, 0x8d,
 0xb2, 0xe1, 0x01, 0xf8, 0x9f, 0xac,
0x1f, 0x23, 0xdd, 0x78,
 0x35,
 0xe2, 0xd6, 0xd1, 0xf3, 0x4d, 0xb5, 0x25, 0x88, 0x16, 0xd1, 0x1a, 0x18,
 0x33, 0xd6, 0x36, 0x7e, 0xc4, 0xc8,
0xe5, 0x5d, 0x2d, 0x74,
 0xd5,
 0x39, 0x3c, 0x44, 0x5a, 0x74, 0xb7, 0x7c, 0x48, 0xc1, 0x1f, 0x90, 0xe3,
 0x55, 0x9e, 0xf6, 0x29, 0xad, 0xb4,
0x6d, 0x93, 0x78, 0xb3,
 0xdc,
 0x25, 0x0b, 0x9c, 0x73, 0x78, 0x7b, 0x93, 0x4c, 0xd3, 0x47, 0x09, 0xda,
 0xe6, 0x69, 0x18, 0xc6, 0x0f, 0xfb,
0xa5, 0x95, 0xf5, 0xe8,
 0x75,
 0xe1, 0x01, 0x1b, 0xd3, 0x1c, 0xa2, 0x57, 0x03, 0x64, 0xdb, 0xf9, 0x5d,
 0xf3, 0x3c, 0xa7, 0xd1, 0x4b, 0xb0,
0x90, 0x1b, 0x90, 0x62,
 0xb4,
 0x88, 0x30, 0x4b, 0x40, 0x4d, 0xcf, 0x7d, 0x89, 0x7a, 0xfb, 0x29, 0xc9,
 0x64, 0x34, 0x0a, 0x52, 0xf6, 0x70,
0x7c, 0x76, 0x5a, 0x2e,
 0x8f,
 0x50, 0xd4, 0x92, 0x15, 0x97, 0xed, 0x4c, 0x2e, 0xf2, 0x3a, 0xd0, 0x58,
 0x7e, 0xdb, 0xf1, 0xf4, 0xdd, 0x07,
0x76, 0x04, 0xf0, 0x55,
 0x8b,
 0x72, 0x2b, 0xa7, 0xa8, 0x78, 0x78, 0x67, 0xe6, 0xd8, 0xa5, 0xde, 0xe7,
 0xc9, 0x1f, 0x5a, 0xa0, 0x89, 0xc7,
0x24, 0xa2, 0x71, 0xb6,
 0x7b,
 0x3b, 0xe6, 0x92, 0x69, 0x22, 0xaa, 0xe2, 0x47, 0x4b, 0x80, 0x3f, 0x6a,
 0xab, 0xce, 0x4e, 0xcd, 0xe8, 0x94,
0x6c, 0xf7, 0x84, 0x73,
 0x85,
 0xfd, 0x85, 0x1d, 0xae, 0x81, 0xf7, 0xec, 0x12, 0x31, 0x7d, 0xc1, 0x99,
 0xc0, 0x3c, 0x51, 0xb0, 0xdc, 0xb0,
0xba, 0x9c, 0x84, 0xb8,
 0x70,
 0xc2, 0x09, 0x7f, 0x96, 0x3d, 0xa1, 0xe2, 0x64, 0x27, 0x7a, 0x22, 0xb8,
 0x75, 0xb9, 0xd1, 0x5f, 0xa5, 0x23,
0xf9, 0x62, 0xe0, 0x41,
```

```
 0x02,
 0x81, 0x81, 0x00, 0xf4, 0xf3, 0x08, 0xcf, 0x83, 0xb0, 0xab, 0xf2, 0x0f,
 0x1a, 0x08, 0xaf, 0xc2, 0x42, 0x29,
0xa7, 0x9c, 0x5e, 0x52,
 0x19,
 0x69, 0x8d, 0x5b, 0x52, 0x29, 0x9c, 0x06, 0x6a, 0x5a, 0x32, 0x8f, 0x08,
 0x45, 0x6c, 0x43, 0xb5, 0xac, 0xc3,
0xbb, 0x90, 0x7b, 0xec,
 0xbb,
 0x5d, 0x71, 0x25, 0x82, 0xf8, 0x40, 0xbf, 0x38, 0x00, 0x20, 0xf3, 0x8a,
 0x38, 0x43, 0xde, 0x04, 0x41, 0x19,
0x5f, 0xeb, 0xb0, 0x50,
 0x59,
 0x10, 0xe1, 0x54, 0x62, 0x5c, 0x93, 0xd9, 0xdc, 0x63, 0x24, 0xd0, 0x17,
 0x00, 0xc0, 0x44, 0x3e, 0xfc, 0xd1,
0xda, 0x4b, 0x24, 0xf7,
 0xcb,
 0x16, 0x35, 0xe6, 0x9f, 0x67, 0x96, 0x5f, 0xb0, 0x94, 0xde, 0xfa, 0xa1,
 0xfd, 0x8c, 0x8a, 0xd1, 0x5c, 0x02,
0x8d, 0xe0, 0xa0, 0xa0,
 0x02,
 0x1d, 0x56, 0xaf, 0x13, 0x3a, 0x65, 0x5e, 0x8e, 0xde, 0xd1, 0xa8, 0x28,
 0x8b, 0x71, 0xc9, 0x65, 0x02, 0x81,
0x81, 0x00, 0xc1, 0x0a,
 0x47,
 0x39, 0x91, 0x06, 0x1e, 0xb9, 0x43, 0x7c, 0x9e, 0x97, 0xc5, 0x09, 0x08,
 0xbc, 0x22, 0x47, 0xe2, 0x96, 0x8e,
0x1c, 0x74, 0x80, 0x50,
 0x6c,
 0x9f, 0xef, 0x2f, 0xe5, 0x06, 0x3e, 0x73, 0x66, 0x76, 0x02, 0xbd, 0x9a,
 0x1c, 0xfc, 0xf9, 0x6a, 0xb8, 0xf9,
0x36, 0x15, 0xb5, 0x20,
 0x0b,
 0x6b, 0x54, 0x83, 0x9c, 0x86, 0xba, 0x13, 0xb7, 0x99, 0x54, 0xa0, 0x93,
 0x0d, 0xd6, 0x1e, 0xc1, 0x12, 0x72,
0x0d, 0xea, 0xb0, 0x14,
 0x30,
 0x70, 0x73, 0xef, 0x6b, 0x4c, 0xae, 0xb6, 0xff, 0xd4, 0xbb, 0x89, 0xa1,
 0xec, 0xca, 0xa6, 0xe9, 0x95, 0x56,
0xac, 0xe2, 0x9b, 0x97,
 0x2f,
 0x2c, 0xdf, 0xa3, 0x6e, 0x59, 0xff, 0xcd, 0x3c, 0x6f, 0x57, 0xcc, 0x6e,
 0x44, 0xc4, 0x27, 0xbf, 0xc3, 0xdd,
0x19, 0x9e, 0x81, 0x16,
 0xe2,
 0x8f, 0x65, 0x34, 0xa7, 0x0f, 0x22, 0xba, 0xbf, 0x79, 0x57, 0x02, 0x81,
 0x80, 0x2e, 0x21, 0x0e, 0xc9, 0xb5,
0xad, 0x31, 0xd4, 0x76,
 0x0f,
 0x9b, 0x0f, 0x2e, 0x70, 0x33, 0x54, 0x03, 0x58, 0xa7, 0xf1, 0x6d, 0x35,
 0x57, 0xbb, 0x53, 0x66, 0xb4, 0xb6,
0x96, 0xa1, 0xea, 0xd9,
 0xcd,
 0xe9, 0x23, 0x9f, 0x35, 0x17, 0xef, 0x5c, 0xb8, 0x59, 0xce, 0xb7, 0x3c,
 0x35, 0xaa, 0x42, 0x82, 0x3f, 0x00,
0x96, 0xd5, 0x9d, 0xc7,
 0xab,
 0xec, 0xec, 0x04, 0xb5, 0x15, 0xc8, 0x40, 0xa4, 0x85, 0x9d, 0x20, 0x56,
 0xaf, 0x03, 0x8f, 0x17, 0xb0, 0xf1,
0x96, 0x22, 0x3a, 0xa5,
```

```
 0xfa,
 0x58, 0x3b, 0x01, 0xf9, 0xae, 0xb3, 0x83, 0x6f, 0x44, 0xd3, 0x14, 0x2d,
 0xb6, 0x6e, 0xd2, 0x9d, 0x39, 0x0c,
0x12, 0x1d, 0x23, 0xea,
 0x19,
 0xcb, 0xbb, 0xe0, 0xcd, 0x89, 0x15, 0x9a, 0xf5, 0xe4, 0xec, 0x41, 0x06,
 0x30, 0x16, 0x58, 0xea, 0xfa, 0x31,
0xc1, 0xb8, 0x8e, 0x08,
 0x84,
 0xaa, 0x3b, 0x19, 0x02, 0x81, 0x80, 0x70, 0x4c, 0xf8, 0x6e, 0x86, 0xed,
 0xd6, 0x85, 0xd4, 0xba, 0xf4, 0xd0,
0x3a, 0x32, 0x2d, 0x40,
 0xb5,
 0x78, 0xb8, 0x5a, 0xf9, 0xc5, 0x98, 0x08, 0xe5, 0xc0, 0xab, 0xb2, 0x4c,
 0x5c, 0xa2, 0x2b, 0x46, 0x9b, 0x3e,
0xe0, 0x0d, 0x49, 0x50,
 0xbf,
 0xe2, 0xa1, 0xb1, 0x86, 0x59, 0x6e, 0x7b, 0x76, 0x6e, 0xee, 0x3b, 0xb6,
 0x6d, 0x22, 0xfb, 0xb1, 0x68, 0xc7,
0xec, 0xb1, 0x95, 0x9b,
 0x21,
 0x0b, 0xb7, 0x2a, 0x71, 0xeb, 0xa2, 0xb2, 0x58, 0xac, 0x6d, 0x5f, 0x24,
 0xd3, 0x79, 0x42, 0xd2, 0xf7, 0x35,
0xdc, 0xfc, 0x0e, 0x95,
 0x60,
 0xb7, 0x85, 0x7f, 0xf9, 0x72, 0x8e, 0x4a, 0x11, 0xc3, 0xc2, 0x09, 0x40,
 0x5c, 0x7c, 0x43, 0x12, 0x34, 0xac,
0x59, 0x99, 0x76, 0x34,
 0xcf,
 0x20, 0x88, 0xb0, 0xfb, 0x39, 0x62, 0x3a, 0x9b, 0x03, 0xa6, 0x84, 0x2c,
 0x03, 0x5c, 0x0c, 0xca, 0x33, 0x85,
0xf5, 0x02, 0x81, 0x80,
 0x56,
 0x99, 0xe9, 0x17, 0xdc, 0x33, 0xe1, 0x33, 0x8d, 0x5c, 0xba, 0x17, 0x32,
 0xb7, 0x8c, 0xbd, 0x4b, 0x7f, 0x42,
0x3a, 0x79, 0x90, 0xe3,
 0x70,
 0xe3, 0x27, 0xce, 0x22, 0x59, 0x02, 0xc0, 0xb1, 0x0e, 0x57, 0xf5, 0xdf,
 0x07, 0xbf, 0xf8, 0x4e, 0x10, 0xef,
0x2a, 0x62, 0x30, 0x03,
 0xd4,
 0x80, 0xcf, 0x20, 0x84, 0x25, 0x66, 0x3f, 0xc7, 0x4f, 0x56, 0x8c, 0x1e,
 0xe1, 0x18, 0x91, 0xc1, 0xfd, 0x71,
0x5f, 0x65, 0x9b, 0xe4,
 0x4f,
 0xe0, 0x1a, 0x3a, 0xf8, 0xc1, 0x69, 0xdb, 0xd3, 0xbb, 0x8d, 0x91, 0xd1,
 0x11, 0x4f, 0x7e, 0x91, 0x1b, 0xb4,
0x27, 0xa5, 0xab, 0x7c,
 0x7b,
 0x76, 0xd4, 0x78, 0xfe, 0x63, 0x44, 0x63, 0x7e, 0xe3, 0xa6, 0x60, 0x4f,
 0xb9, 0x55, 0x28, 0xba, 0xba, 0x83,
0x1a, 0x2d, 0x43, 0xd5,
 0xf7,
 0x2e, 0xe0, 0xfc, 0xa8, 0x14, 0x9b, 0x91, 0x2a, 0x36, 0xbf, 0xc7, 0x14
  };
CK_BYTE
  knownRSA1Modulus[]
 = {
0xb8, 0xb5, 0x0f, 0x49, 0x46, 0xb5, 0x5d, 0x58, 0x04, 0x8e,
 0x52, 0x59, 0x39, 0xdf, 0xd6,
```

```
0x29,
0x45, 0x6b, 0x6c, 0x96, 0xbb, 0xab, 0xa5, 0x6f, 0x72, 0x1b,
 0x16, 0x96, 0x74, 0xd5, 0xf9,
0xb4,
0x41, 0xa3, 0x7c, 0xe1, 0x94, 0x73, 0x4b, 0xa7, 0x23, 0xff,
 0x61, 0xeb, 0xce, 0x5a, 0xe7,
0x7f,
0xe3, 0x74, 0xe8, 0x52, 0x5b, 0xd6, 0x5d, 0x5c, 0xdc, 0x98,
 0x49, 0xfe, 0x51, 0xc2, 0x7e,
0x8f,
0x3b, 0x37, 0x5c, 0xb3, 0x11, 0xed, 0x85, 0x91, 0x15, 0x92,
 0x24, 0xd8, 0xf1, 0x7b, 0x3d,
0x2f,
0x8b, 0xcd, 0x1b, 0x30, 0x14, 0xa3, 0x6b, 0x1b, 0x4d, 0x27,
 0xff, 0x6a, 0x58, 0x84, 0x9e,
0x79,
0x94, 0xca, 0x78, 0x64, 0x01, 0x33, 0xc3, 0x58, 0xfc, 0xd3,
 0x83, 0xeb, 0x2f, 0xab, 0x6f,
0x85,
0x5a, 0x38, 0x41, 0x3d, 0x73, 0x20, 0x1b, 0x82, 0xbc, 0x7e,
 0x76, 0xde, 0x5c, 0xfe, 0x42,
0xd6,
0x7b, 0x86, 0x4f, 0x79, 0x78, 0x29, 0x82, 0x87, 0xa6, 0x24,
 0x43, 0x39, 0x74, 0xfe, 0xf2,
0x0c,
0x08, 0xbe, 0xfa, 0x1e, 0x0a, 0x48, 0x6f, 0x14, 0x86, 0xc5,
 0xcd, 0x9a, 0x98, 0x09, 0x2d,
0xf3,
0xf3, 0x5a, 0x7a, 0xa4, 0xe6, 0x8a, 0x2e, 0x49, 0x8a, 0xde,
 0x73, 0xe9, 0x37, 0xa0, 0x5b,
0xef,
0xd0, 0xe0, 0x13, 0xac, 0x88, 0x5f, 0x59, 0x47, 0x96, 0x7f,
 0x78, 0x18, 0x0e, 0x44, 0x6a,
0x5d,
0xec, 0x6e, 0xed, 0x4f, 0xf6, 0x6a, 0x7a, 0x58, 0x6b, 0xfe,
 0x6c, 0x5a, 0xb9, 0xd2, 0x22,
0x3a,
0x1f, 0xdf, 0xc3, 0x09, 0x3f, 0x6b, 0x2e, 0xf1, 0x6d, 0xc3,
 0xfb, 0x4e, 0xd4, 0xf2, 0xa3,
0x94,
0x13, 0xb0, 0xbf, 0x1e, 0x06, 0x2e, 0x29, 0x55, 0x00, 0xaa,
 0x98, 0xd9, 0xe8, 0x77, 0x84,
0x8b,
0x3f, 0x5f, 0x5e, 0xf7, 0xf8, 0xa7, 0xe6, 0x02, 0xd2, 0x18,
 0xb0, 0x52, 0xd0, 0x37, 0x2e,
0x53,
      },
      knownRSA1PubExponent[]
 = { 0x01, 0x00, 0x01 };
      char
 *pPlainData = 0;
      unsigned
 long ulPlainDataLength;
      char
 *pEncryptedData = 0;
      unsigned
 long ulEncryptedDataLength = 0;
      CK_MECHANISM
 mech;
      CK_USHORT
   usStatus=0,
```

```
                  usKeyLength;
      CK_OBJECT_HANDLE
 hKey;
      CK_OBJECT_CLASS
    SymKeyClass
  = CKO_SECRET_KEY;
      CK_BBOOL
           bTrue
 = 1,
                         bFalse
 = 0,
                         bToken
 = bTrue,
                         bSensitive
 = bTrue,
                         bPrivate
 = bTrue,
                         bEncrypt
 = bTrue,
                         bDecrypt
 = bTrue,
                         bSign
 = bFalse, // "..."
                         bVerify
 = bFalse, //Will not allow sign/verify operation.
                         bWrap
 = bTrue,
                         bUnwrap
 = bTrue,
#ifdef EXTRACTABLE
                         bExtract
 = bTrue,
#endif //EXTRACTABLE
                         bDerive
 = bTrue;
      CK_KEY_TYPE
        keyType;
      CK_USHORT
          usValueBits;
      char
               pbPublicKeyLabel[128];
      CK_ATTRIBUTE_PTR
 pPublicTemplate;
      CK_USHORT
 usPublicTemplateSize = 0;
      char
 iv[8] = { '1', '2', '3', '4', '5', '6', '7', '8' };
      CK_ATTRIBUTE
 SymKeyTemplate[] = {
         {CKA_CLASS,
 0, sizeof(SymKeyClass)},
         {CKA_KEY_TYPE,
 0, sizeof(keyType)},
         {CKA_TOKEN,
 0, sizeof(bToken)},
         {CKA_SENSITIVE,
 0, sizeof(bSensitive)},
         {CKA_PRIVATE,
 0, sizeof(bPrivate)},
```

```
        {CKA_ENCRYPT,
 0, sizeof(bEncrypt)},
        {CKA_DECRYPT,
 0, sizeof(bDecrypt)},
        {CKA_SIGN,
 0, sizeof(bSign)},
        {CKA_VERIFY,
 0, sizeof(bVerify)},
        {CKA_WRAP,
 0, sizeof(bWrap)},
        {CKA_UNWRAP,
 0, sizeof(bUnwrap)},
        {CKA_DERIVE,
 0, sizeof(bDerive)},
        {CKA_VALUE_LEN,0,
  sizeof(usKeyLength)
 },
        {CKA_LABEL,
 0, 0}  //
 Always keep last!!!
#ifdef EXTRACTABLE      //Conditional
 stuff must be at the end!!!!!
        {CKA_EXTRACTABLE,
 0, sizeof(bExtract)},
#endif //EXTRACTABLE
      };
      CK_OBJECT_HANDLE
 hUnWrappedKey, hPublicRSAKey;
      char
             *pbWrappedKey;
      unsigned
 long    ulWrappedKeySize;
      CK_OBJECT_CLASS
  privateKey
 = CKO_PRIVATE_KEY,
publicKey = CKO_PUBLIC_KEY;
      CK_KEY_TYPE
      rsaType
   =
 CKK_RSA;
      CK_BYTE
          pLabel[]
  = "RSA
 private Key",
pbPublicRSAKeyLabel[] = "RSA Public Key";
      CK_ATTRIBUTE
 *pTemplate;
      CK_ULONG
     usTemplateSize,
ulPublicRSAKeyTemplateSize;
      CK_ATTRIBUTE
 pPublicRSAKeyTemplate[] = {
        {CKA_CLASS,
    0,
 sizeof(publicKey) },
        {CKA_KEY_TYPE,
 0,    sizeof(rsaType)
    },
```

```
        {CKA_TOKEN,
   0,
    sizeof(bToken)
    },
        {CKA_PRIVATE,
 0,   sizeof(bPrivate)
  },
        {CKA_ENCRYPT,
 0,   sizeof(bEncrypt)
  },
        {CKA_VERIFY,
  0,
     sizeof(bSign)
     },
        {CKA_WRAP,
    0,
     sizeof(bWrap)
     },
{CKA_MODULUS,
0, sizeof(knownRSA1Modulus) },
{CKA_PUBLIC_EXPONENT,
0, sizeof(knownRSA1PubExponent) },
        {CKA_LABEL,
   0,
     sizeof(pbPublicRSAKeyLabel)
    }
     };
     CK_ATTRIBUTE
pPrivateKeyTemplate[] = {
        {CKA_CLASS,
   &privateKey,
sizeof(privateKey) },
        {CKA_KEY_TYPE,
&rsaType,    sizeof(rsaType)
   },
        {CKA_TOKEN,
   &bToken,
    sizeof(bToken)
    },
        {CKA_SENSITIVE,&bSensitive,
sizeof(bSensitive) },
        {CKA_PRIVATE,
 &bPrivate,
  sizeof(bPrivate)
  },
        {CKA_DECRYPT,
 &bEncrypt,
  sizeof(bEncrypt)
  },
        {CKA_SIGN,
    &bSign,
     sizeof(bSign)
     },
//{CKA_SIGN_RECOVER,
&bTrue, sizeof(bTrue)     },
        {CKA_UNWRAP,
  &bWrap,
     sizeof(bWrap)
     },
```

```
{CKA_EXTRACTABLE, &bFalse, sizeof(bFalse)      },
{CKA_LABEL,       pLabel,
  sizeof(pLabel)
     }
      };
      //
 Generate a DES3 Key
      SymKeyTemplate[0].pValue
 = &SymKeyClass;
      SymKeyTemplate[1].pValue
 = &keyType;
      SymKeyTemplate[2].pValue
 = &bToken;
      SymKeyTemplate[3].pValue
 = &bSensitive;
      SymKeyTemplate[4].pValue
 = &bPrivate;
      SymKeyTemplate[5].pValue
 = &bEncrypt;
      SymKeyTemplate[6].pValue
 = &bDecrypt;
      SymKeyTemplate[7].pValue
 = &bSign;
      SymKeyTemplate[8].pValue
 = &bVerify;
      SymKeyTemplate[9].pValue
 = &bWrap;
      SymKeyTemplate[10].pValue
 = &bUnwrap;
      SymKeyTemplate[11].pValue
 = &bDerive;
      SymKeyTemplate[12].pValue
 =  &usKeyLength;
      SymKeyTemplate[13].pValue
 = pbPublicKeyLabel;
#ifdef EXTRACTABLE
      SymKeyTemplate[14].pValue
 = &bExtract;
#endif //EXTRACTABLE
      mech.mechanism
 = CKM_DES3_KEY_GEN;
      mech.pParameter
 = 0;
      mech.usParameterLen
 = 0;
      keyType
 = CKK_DES3;
      usKeyLength
 = 24;
      strcpy(
 pbPublicKeyLabel, "Generated DES3 Key" );
      pPublicTemplate
 = SymKeyTemplate;
      usPublicTemplateSize
 = DIM(SymKeyTemplate);
      //
 Adjust size of label (ALWAYS LAST ENTRY IN ARRAY)
      pPublicTemplate[usPublicTemplateSize-1].usValueLen
 = strlen(
pbPublicKeyLabel );
```

```
     retCode
 = C_GenerateKey(   hSessionHandle,
                                 (CK_MECHANISM_PTR)&mech,
                                 pPublicTemplate,
                                 usPublicTemplateSize,
                                 &hKey);
     if(retCode
 == CKR_OK)
     {
        cout
 << pbPublicKeyLabel << ": " << hKey <<
 endl;
     }
     else
     {
   cout
 << "\n" "Error 0x" << hex << retCode;
        cout
 << " generating the DES3 Key.\n";
   error
 = -11;
        goto
 exit_routine_6;
     }
     //
 Encrypt the RSA Key
     mech.mechanism
 = CKM_DES3_CBC;
     mech.pParameter
 = iv;
     mech.usParameterLen
 = sizeof(iv);
     pPlainData
 = (char *)(pRsaKey);
     ulPlainDataLength
 = sizeof(pRsaKey);
     //
 Allocate memory for output buffer
     if(
 retCode == CKR_OK )
     {
        pEncryptedData
 = new char [ulPlainDataLength + 2048]; // Leave
// extra room for
// RSA Operations
        if(
 !pEncryptedData )
        {
           retCode
 = CKR_DEVICE_ERROR;
        }
     }
     //
 Start encrypting
     if(
 retCode == CKR_OK )
     {
        retCode
 = C_EncryptInit(hSessionHandle, &mech, hKey);
     }
```

```
     //
Continue encrypting
     if(
retCode == CKR_OK )
     {
        CK_USHORT
usInDataLen,
                  usOutDataLen
= (CK_USHORT) (ulPlainDataLength + 2048);
        CK_ULONG
 ulBytesRemaining
= ulPlainDataLength;
        char
*    pPlainTextPointer
= pPlainData;
        char
*    pEncryptedDataPointer
= pEncryptedData;
        while
(ulBytesRemaining > 0)
        {
           if
(ulBytesRemaining > 0xfff0) // We are longer than a USHORT can handle
           {
              usInDataLen
= 0xfff0;
              ulBytesRemaining
-= usInDataLen;
           }
           else
           {
              usInDataLen
= (CK_USHORT) ulBytesRemaining;
              ulBytesRemaining
-= usInDataLen;
           }
           retCode
= C_EncryptUpdate( hSessionHandle,
                                      (CK_BYTE_PTR)pPlainTextPointer,
                                      usInDataLen,
                                      (CK_BYTE_PTR)pEncryptedDataPointer,
                                      &usOutDataLen
);
           pPlainTextPointer
+= usInDataLen;
           pEncryptedDataPointer
+= usOutDataLen;
           ulEncryptedDataLength
+= usOutDataLen;
        }
     }


     //
Finish encrypting
     if(
retCode == CKR_OK )
     {
        CK_USHORT
usOutDataLen;
```

```
        CK_BYTE_PTR
pOutData = (CK_BYTE_PTR)pEncryptedData;
        pOutData
+= ulEncryptedDataLength;
        retCode
= C_EncryptFinal(hSessionHandle, pOutData, &usOutDataLen);
        ulEncryptedDataLength
+= usOutDataLen;
     }
     else
     {
  cout
<< "\n" "Error 0x" << hex << retCode;
        cout
<< " somewhere in the encrypting.\n";
        if(
pEncryptedData )
        {
           delete
pEncryptedData;
        }
  error
= -12;
        goto
exit_routine_6;
     }
     mech.mechanism
     =
CKM_DES3_CBC;
     mech.pParameter
    =
(void*) "12345678"; // 8 byte IV
     mech.usParameterLen
= 8;
     pTemplate
= pPrivateKeyTemplate;
     usTemplateSize
= DIM(pPrivateKeyTemplate);
    pbWrappedKey
= pEncryptedData;
     ulWrappedKeySize
= ulEncryptedDataLength;
     if(
retCode == CKR_OK )
     {
        retCode
= C_UnwrapKey( hSessionHandle,
                                &mech,
                                hKey,
                                (CK_BYTE_PTR)pbWrappedKey,
                                (CK_USHORT)ulWrappedKeySize,
                                pTemplate,
                                usTemplateSize,
                                &hUnWrappedKey);
     }
     //
Report unwrapped key handle
     if(
retCode == CKR_OK )
     {
```

```
        cout
<< "\n Private key Unwrapped key is:" << hUnWrappedKey
<<"\n\n";
     }
     else
     {
  cout
<< "\n" "Error 0x" << hex << retCode;
        cout
<< " unwrapping.\n";
        if(
pEncryptedData )
        {
           delete
pEncryptedData;
        }
  error
= -13;
        goto
exit_routine_6;
     }
     //
Release temporary memory
     if(
pEncryptedData )
     {
        delete
pEncryptedData;
     }
  //
Create the Public Key that goes with the Private Key
  if(
retCode == CKR_OK )
  {
 //
Unwrap it onto the token
 pPublicRSAKeyTemplate[0].pValue
= &publicKey;
 pPublicRSAKeyTemplate[1].pValue
= &rsaType;
 pPublicRSAKeyTemplate[2].pValue
= &bToken;
 pPublicRSAKeyTemplate[3].pValue
= &bPrivate;
 pPublicRSAKeyTemplate[4].pValue
= &bEncrypt;
 pPublicRSAKeyTemplate[5].pValue
= &bSign;
 pPublicRSAKeyTemplate[6].pValue
= &bWrap;
 pPublicRSAKeyTemplate[7].pValue
= knownRSA1Modulus;
 pPublicRSAKeyTemplate[8].pValue
= knownRSA1PubExponent;
 pPublicRSAKeyTemplate[9].pValue
= pbPublicRSAKeyLabel;
 pTemplate
= pPublicRSAKeyTemplate;
 usTemplateSize
= DIM(pPublicRSAKeyTemplate);
```

```
  retCode
 = C_CreateObject(  hSessionHandle,
pTemplate,
 usTemplateSize,
 &hPublicRSAKey);
  if(retCode
 == CKR_OK)
  {
 cout
 << pbPublicRSAKeyLabel << ": " << hPublicRSAKey
 << endl;
  }
  else
  {
 cout
 << "\n" "Error 0x" << hex << retCode;
 cout
 << " creating the RSA Public Key.\n";
 error
 = -14;
 goto
 exit_routine_6;
  }
   }
if( retCode == CKR_OK )
   {
CK_CHAR label[] = "RSA Key";
CK_ATTRIBUTE RSAFindPriTemplate[] =
{
CKA_LABEL, label, sizeof(label)
};
CK_ULONG numHandles;
CK_OBJECT_HANDLE handles[1000];
retCode = C_FindObjectsInit( hSessionHandle, RSAFindPriTemplate,
 1 );
if(retCode != CKR_OK)
{
cout << "C_FindObjectsInit not returning OK ("
 << hex << retCode << ")\n\n";
goto exit_routine_6;
}
retCode =C_FindObjects( hSessionHandle , handles, 90,
&numHandles );
if(retCode != CKR_OK)
{
cout << "C_FindObjects not returning OK ("
 << hex <<
retCode << ")\n\n";
goto exit_routine_6;
}
cout << "Everything's GOOD\n\n";
for(int i=0; i < numHandles; i++)
{
cout << handles[i] << "\n";
}
}
}
//CJM-> END OF TEST CODE
   //
 Beginning of exit routines
exit_routine_6:
```

```
   //
 Logout
   retCode
 = C_Logout(hSessionHandle);
   if(retCode
 != CKR_OK)
{
cout << "\n" "Error 0x" <<
 hex << retCode << " logging out.";
}
exit_routine_5:
// Close the session
   retCode
 = C_CloseSession(hSessionHandle);
if(retCode != CKR_OK)
{
cout << "\n" "Error 0x" <<
 hex << retCode << " closing session.";
}
exit_routine_4:
   delete
 pSlotList;
exit_routine_3:
#ifdef PKCS11_2_0
   C_Finalize(0);
#else
   C_Terminate();
#endif
exit_routine_2:
#ifndef STATIC
   //
 No longer need Chrystoki
   CrystokiDisconnect();
#endif
exit_routine_1:
   cout
 << "\nDone. (" << dec << error << ")\n";
   cout.flush();
   return
 error;
}
CK_RV  Pinlogin(CK_SESSION_HANDLE
 hSession)
{

CK_RV retCode;
unsigned char buffer[MAX];
int count =0;
cout << "Please enter the USER password : "
 << endl;
//calling get PinString to mask input, variable "count"

//holds length of "buffer"(password)
//needed for Login call
count = getPinString(buffer);
//Login as user on  token
 in slot
retCode = C_Login(hSession, CKU_USER, buffer, count);
if(retCode != CKR_OK)
{
```

```
cout << "\n" "Error 0x" <<
 hex << retCode;
      cout
 << " logging in as user.";
  exit(hSession);
  return
 -3;
}
cout << "logging into the token....";
cout << "\nlogged into token " << endl;
return retCode;
}
//////////////////////////////////////////////////////////////////////
// getPinString()
// ==============
//
// This function retrieves a pin string from the user.  It
 modifies the
// console mode before starting so that the characters the
 user types are
// not echoed, and a '*' character is displayed for each
 typed character
// instead.
//
// Backspace is supported, but we don't get any fancier than
 that.
//////////////////////////////////////////////////////////////////////
int getPinString(CK_CHAR_PTR pw)
{
    int
 len=0;
    char
 c=0;
    //
 Unfortunately, the method of turning off character echo is
// different for Windows and Unix platforms.  So
 we have to
// conditionally compile the appropriate section.  Even
 the basic
// password retrieval is slightly different, since
    //
 Windows and Unix use different character codes for the return key.
#ifdef WIN32
    DWORD
 mode;
    //
 This console mode stuff only applies to windows.  We'll
 have to
// do something else when it comes to unix.
    if
 (GetConsoleMode(GetStdHandle(STD_INPUT_HANDLE), &mode)) {
        if
 (SetConsoleMode(GetStdHandle(STD_INPUT_HANDLE), mode & (!ENABLE_ECHO_INPUT)))
 {
            while
 (c  != '\r')
 {
                //
 wait for a character to be hit
                while
 (!_kbhit()) {
```

```
                    Sleep(100);
                }
                //
 get it
                c
 = _getch();
                //
 check for carriage return
                if
 (c != '\r') {
                    //
 check for backspace
                    if
 (c!='\b') {
                        //
 neither CR nor BS -- add it to the password string
                        printf("*");
                        *pw++
 = c;
                        len++;
                    }
 else {
// handle backspace -- delete the last character &
// erase it from the screen
                        if
 (len > 0) {
                            pw--;
                            len--;
                            printf("\b
 \b");
}
}
}
}
            //
 Add the zero-termination
            *pw
 = '\0';
            SetConsoleMode(GetStdHandle(STD_INPUT_HANDLE),
 mode);
            printf("\n");
}
    }
#endif
    return
 len;
}
```

Audit Logging

By default, the HSM logs select events. See Audit Logging for more information.

The HSM creates a log secret unique to the HSM, computed during the first initialization after manufacture. The log secret resides in flash memory (permanent, non-volatile memory), and is used to create log records that are sent to a log file. Later, the log secret is used to prove that a log record originated from a legitimate HSM and has not been tampered with.

Audit Log Records

A log record consists of two fields – the log message and the HMAC for the previous record. When the HSM creates a log record, it uses the log secret to compute the SHA256-HMAC of all data contained in that log message, plus the HMAC of the previous log entry. The HMAC is stored in HSM flash memory. The log message is then transmitted, along with the HMAC of the previous record, to the host. The host has a logging daemon to receive and store the log data on the host hard drive.

For the first log message ever returned from the HSM to the host there is no previous record and, therefore, no HMAC in flash. In this case, the previous HMAC is set to zero and the first HMAC is computed over the first log message concatenated with 32 zero-bytes. The first record in the log file then consists of the first log message plus 32 zero-bytes. The second record consists of the second message plus HMAC1 = HMAC (message1 || 0x0000). This results in the organization shown below.

MSG 1	HMAC 0
	. . .
MSG n-1	HMAC n-2
MSG n	HMAC n-1
. . .	
MSG n+m	HMAC n+m-1
MSG n+m+1	HMAC n+m
. . .	
MSG end	HMAC n+m-1

Recent HMAC in NVRAM	HMAC end

To verify a sequence of *m* log records which is a subset of the complete log, starting at index *n*, the host must submit the data illustrated above. The HSM calculates the HMAC for each record the same way as it did when the record was originally generated, and compares this HMAC to the value it received. If all of the calculated HMACs match the received HMACs, then the entire sequence verifies. If an HMAC doesn't match, then the associated record and all following records can be considered suspect. Because the HMAC of each message depends on the HMAC of the previous one, inserting or altering messages would cause the calculated HMAC to be invalid.

The HSM always stores the HMAC of the most-recently generated log message in flash memory. When checking truncation, the host would send the newest record in its log to the HSM; and, the HSM would compute the HMAC and compare it to the one in flash. If it does not match, then truncation has occurred.

Audit Log Message Format

Each message is a fixed-length, comma delimited, and newline-terminated string. The table below shows the width and meaning of the fields in a message.

Offset	Length (Chars)	Description
0	10	Sequence number
10	1	Comma
11	17	Timestamp
28	1	Comma
29	256	Message text, interpreted from raw data
285	1	Comma
286	64	HMAC of previous record as ASCII-HEX
350	1	Comma
351	96	Data for this record as ASCII-HEX (raw data)
447	1	Newline '\n'

The raw data for the message is stored in ASCII-HEX form, along with a human-readable version. Although this format makes the messages larger, it simplifies the verification process, as the HSM expects to receive raw data records.

Example

The following example shows a sample log record. It is separated into multiple lines for readability even though it is a single record. Some white spaces are also omitted.

```
38,12/08/13 15:30:50,session 1 Access 2147483651:22621 operation LUNA_CREATE_CONTAINER
returned LUNA_RET_SM_UNKNOWN_TOSM_STATE(0x00300014) (using PIN (entry=LUNA_ENTRY_DATA_AREA)),
29C51014B6F131EC67CF48734101BBE301335C25F43EDF8828745C40755ABE25,
2600001003600B00EA552950140030005D58000003000080010000000000000000000000000000000000000000
```

The sequence number is “`38`”. The time is “`12/08/13 15:30:50`”.

The log message is “`session 1 Access 2147483651:22621 operation LUNA_CREATE_CONTAINER returned LUNA_RET_SM_UNKNOWN_TOSM_STATE(0x00300014) (using PIN (entry=LUNA_ENTRY_DATA_AREA))`”.

In the message text, the “who” is the session identified by “`session 1 Access 2147483651:22621`” (the application is identified by the access ID major = 2147483651, minor = 22621).

The “what” is “`LUNA_CREATE_CONTAINER`”.

The operation status is “`LUNA_RET_SM_UNKNOWN_TOSM_STATE(0x00300014)`”.

The HMAC of previous record is “`29C51014B6F131EC67CF48734101BBE301335C25F43EDF8828745C40755ABE25`”.

The remainder is the raw data for this record as ASCII-HEX.

Log External

An important element of the security audit logging feature is the Log External function. This Luna extension to PKCS #11 allows a user application to insert text of the user's choice into the log record stream. The function call is **CA_LogExternal ( )**. It can be used, for example, to insert an application name or the name of the user who is logged into the application and have the inserted text string protected as part of the audit log in the same way as records that have been generated by the HSM itself. It is recommended that applications use the **CA_LogExternal ( )** function when the application starts to insert the application name and also to insert the user name each time an individual user logs into or out of the application. The function is called as:

CA_LogExternal(CK_SLOT_ID slotID, CK_SESSION_HANDLE hSession, CK_CHAR_PTR pData, CK_ULONG puldataLen);

where:

> **slotID** is PKCS #11 slot containing the HSM or partition being addressed

> **hSession** is the handle of the session with which the record is to be associated

> **pData** is the pointer to the character array containing the external message

> **puldataLen** is the length of the character array

Note that the input character array is limited to a maximum of 100 characters and it will be truncated at 100 characters if **puldataLen** > 100.

For applications that cannot add this function call, it is possible to use lunacm:> **audit logmsg** within a startup script to insert a text record at the time the application is started.

When a user logs in to the Luna PCIe HSM 7 lunash:> session, the **CA_LogExternal ( )** function is automatically called to register the user name and access ID. Subsequent HSM operations can be tracked by the access ID.

You must configure the "log external" event category in order for the HSM to log the **CA_LogExternal ( )** messages.

CHAPTER 8: Java Interfaces

This chapter describes the Java interfaces to the PKCS#11 API. It contains the following topics:

> "Luna JSP Overview and Installation" below
> "Luna JSP Configuration" on page 657
> "The JCPROV PKCS#11 Java Wrapper" on page 661
> "Java or JSP Errors" on page 667
> "Re-Establishing a Connection Between Your Java Application and Luna PCIe HSM 7" on page 668
> "Recovering From the Loss of All HA Members" on page 669
> "Using Java Keytool with Luna PCIe HSM 7" on page 671
> "LunaKeyStore Reference" on page 676

Luna JSP Overview and Installation

The Luna JSP is part of an application program interface (API) that allows Java applications to make use of certain Luna products.

As with other APIs, some existing Java-based applications might have generic requirements and calls that can already work with Luna products. In other cases, it might be necessary for you or your vendor to create an application or to adapt one, using the JSP API.

You have the choice of:

> using a previously integrated third-party application, known to work with this Luna product
> performing your own integration with a Java-based application supplied by you or a third party, or
> developing your own application using our Java API.

Develop your own Java apps using our included Software Development Kit, which includes Luna Java API usage notes for developers, as well as development support by Thales. A standard Java development environment is required, in addition to the API provided by Thales.

Please refer to the current-version Luna PCIe HSM 7 Customer Release Notes (CRN) for the most up-to-date list of supported platforms and APIs.

> **NOTE** **Java Provider (JSP)** - both GMC and GMAC are supported. **GmacAesDemo.java** provides a sample for using GMAC with Java.
>
> Java Parameter Specification class **LunaGmacParameterSpec.java** defines default values recommended by the NIST specification.

This following sections describe the tasks required to set up the JSP API:

> "Installation" on the next page

> "JSP Registration" on the next page
> "Post-Installation Tasks" on page 654

Installation

To use the Luna JavaSP service providers three main components are needed:

> The Java SDK
> The Java Cryptographic JCE Policy files (optional)
> The Luna JavaSP artifacts in the Luna HSM Client

Java SDK Installation

Acquire and install the JDK or JRE (available from the Java site, not included with the Luna software). Refer to the Customer Release Notes for supported Java versions.

Java Cryptographic JCE Policy Files Installation (optional)

If you intend to generate large key sizes, you might need to apply the unlimited strength ciphers policy. You will need two cryptographic JCE Policy files v 7/8/9/10/11 (available from the Oracle Java web site): local_policy.jar and US_export_policy.jar.

Copy these files to JAVA_HOME/jre/lib/security (or the equivalent directory that applies to your setup).

```
[root@my-client]# echo $JAVA_HOME
/usr/java/default
[root@my-client]# cp -p local_policy.jar /usr/java/default/jre/lib/security/
[root@my-sclient]# cp -p US_export_policy.jar /usr/java/default/jre/lib/security/
```

If you see errors like "Invalid Key size", that is usually an indication that the JCE is not properly installed.

Luna JavaSP included in the Luna HSM Client

Follow the installation procedure for the Luna HSM Client as described in the *Installation Guide*. When installing the Luna HSM Client software, choose the option to install Luna JSP. There are two SafeNet files: the **LunaProvider.jar** file, and the Java library file (**libLunaAPI.so** in Unix based systems or **LunaAPI.dll** in Windows systems). To ensure that the Java Environment can find these files, follow the instructions for either Java 7/8 or Java 9+.

Operating System	JSP Install directory
AIX	**/usr/safenet/lunaclient/jsp/lib**
Linux	**/usr/safenet/lunaclient/jsp/lib**
Solaris	**/opt/safenet/lunaclient/jsp/**
Windows	**C:\Program Files\LunaClient\JSP\lib**

To configure Java 7/8 for Luna JSP

To ensure that Luna PCIe HSM 7 and Luna JSP can work with the JRE, copy the JSP files from the default installation location to the Java environment. The exact destination directory might differ depending on where you obtained your Java system, the version, and any choices that you made while installing and configuring it.

Operating System	Destination directory example
AIX	**/usr/jre/lib/ext**
Linux	**/usr/jre/lib/ext**
Solaris	**/opt/jre/lib/ext**
Windows	<java_install_dir>**\bin** **C:\Program Files\Java\jdk1.8.0_121\bin** **NOTE** Java 7/8/9 for Windows has removed the **<java_install_dir>\lib\ext** directory from the Java library path.

IBM Java 7/8 has a .jar authentication issue that requires a patch from IBM. See APAR IJ25459 for details.

To configure Java 9+ for Luna JSP

Add **LunaProvider.jar** to the Java classpath and specify the Luna Java library location (**libLunaAPI.so** in Unix based systems or **LunaAPI.dll** in Windows systems) in the Java library path.

For example:

> **java -cp /**<directory_location>**/LunaProvider.jar -Djava.library.path=**<Luna_Java_library_location> <class name>

> **TIP** In Windows, you can also put **LunaAPI.dll** in an arbitrary folder and add that folder to the system path. Java will search the system path for **LunaAPI.dll**.

The exact directory might differ depending on where you obtained your Java system, the version, and any choices that you made while installing and configuring it.

JSP Registration

Before Java can use Luna JSP, you must register it with the Java Runtime Enviroment. You can choose either a static registration or a dynamic registration. A static registration defaults all Java applications to default to the Luna provider, while a dynamic registration allows you to set the provider for Java applications individually.

JSP Static Registration

> **NOTE** This section applies to JSP, not to JCPROV.

You would choose static registration of providers if you want all applications to default to the Luna provider.

Once your client has externally logged in using **salogin** or your own HSM-aware utility, any application would be able to use Luna product without being designed to log in to the HSM Partition.

Edit the **java.security** file located in the **/jre/lib/security** directory of your Java SDK/JRE installation to read as follows:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
security.provider.3=com.safenetinc.luna.provider.LunaProvider
security.provider.4=com.sun.rsajca.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
```

You can set our provider in first position for efficiency if Luna HSM operations are your primary mode. However, if your application needs to perform operations not supported by the LunaProvider (secure random generation or random publickey verification, for example) then it would receive error messages from the HSM and would need to handle those gracefully before resorting to providers further down the list. We have found that having our provider in third position works well for most applications.

The modifications in the **java.security** file are global, and they might result in the breaking of another application that uses the default KeyPairGenerator without logging into the Luna Network HSM first. This consideration might argue for using dynamic registration, instead.

JSP Dynamic Registration

You might prefer to employ dynamic registration of Providers, in order to avoid possible negative impacts on other applications running on the same machine. As well, the use of dynamic registration allows you to keep installation as straightforward as possible for your customers.

This sample code shows an example of dynamic registration with the Luna provider. The Luna provider is registered in position 2, ensuring that the "SUN" provider is still the default. If you want the Luna provider to be used when no provider is explicitly specified, it should be registered at position 1.

```
try {
  com.safenetinc.luna.LunaSlotManager.getInstance().login("<HSM Partition Password>");
  java.security.Provider provider = new com.safenetinc.luna.provider.LunaProvider();
  // removing the provider is only necessary if it is already registered
  // and you want to change its position
  java.security.Security.removeProvider(provider.getName());
  java.security.Security.insertProviderAt(provider, 2);
  com.safenetinc.luna.LunaSlotManager.getInstance().logout();
} catch (Exception e) {
  System.out.println("Exception caught during loading of the providers: "
      + ex.getMessage());
```

Post-Installation Tasks

Making Private and Secret Keys Extractable

By default, all generated private and secret keys have their CKA_EXTRACTABLE attribute set to **0** (see "Key Attribute Defaults" on page 619). These keys are stored in the HSM hardware and cannot be extracted, only cloned to a partition on another HSM. This attribute cannot be modified later. If you want the ability to wrap private and/or secret keys and export them off the HSM, you must use one of the following two methods to set CKA_EXTRACTABLE to **1** (TRUE) when the key is created:

Global configuration:

Configure **java.security** as follows to have JSP create all future private/secret keys with CKA_EXTRACTABLE=1:

> To make all private keys extractable, add the following line to **java.security**:

com.safenetinc.luna.provider.createExtractablePrivateKeys=true

> To make all secret keys extractable, add the following line to **java.security**:

com.safenetinc.luna.provider.createExtractableSecretKeys=true

Local configuration:

Configure CKA_EXTRACTABLE on a key-by-key basis by using the following methods in your Java application:

> To make the next generated private key extractable using the **LunaSlotManager.setPrivateKeysExtractable()** method:

```
LunaSlotManager.getInstance().setPrivateKeysExtractable(true); // Set CKA_EXTRACTABLE=1 on
upcoming private keys
kpg = KeyPairGenerator.getInstance("RSA", "LunaProvider");
kpg.initialize(2048);
myPair = kpg.generateKeyPair();
LunaSlotManager.getInstance().setPrivateKeysExtractable(false); // Set CKA_EXTRACTABLE=0 on
upcoming private keys
```

> **NOTE** To wrap and export private keys, the partition must have partition policy 1: Allow private key wrapping set to 1 (ON). See Configuring the Partition for Cloning or Export of Private Keys.

> To make the next generated secret key extractable using the **LunaSlotManager.setSecretKeysExtractable()** method:

```
LunaSlotManager.getInstance().setSecretKeysExtractable(true); // Set CKA_EXTRACTABLE=1 on
upcoming secret keys
kg = KeyGenerator.getInstance("AES");
kg.init(256);
aesKey = kg.generateKey();
LunaSlotManager.getInstance().setPrivateKeysExtractable(false); // Set CKA_EXTRACTABLE=0 on
upcoming secret keys
```

Using Luna JCE/JCA with 64-bit Libraries

If you are using Luna JCE/JCA with the 64-bit libraries for Luna Network HSM, you must include the **-d64** switch in the Java command-line.

java -d64 -jar jMultitoken.jar

For most 64-bit platforms, 64-bit is supported. Some 64-bit platforms support the option of running in 32-bit mode), as a backward compatibility feature.

> **NOTE** Luna HSM Client 10.1.0 and newer includes libraries for 64-bit operating systems only.

If you use the 64-bit installation and do not use the **-d64** command-line switch in your Java command lines, the system attempts (by default) to use the 32-bit library (which is not installed, because you installed 64-bit in this example...), and the result is an error message complaining about the kernel model.

Using ECC Keys for TLS with Java 7

For optimal Java performance when using Elliptic Curve keys to perform TLS with Java 7, where those keys reside in the HSM, you must configure the SunEC security provider (sun.security.ec.SunEC) to be below the LunaProvider in your java.security file.

We suggest that you not attempt to resolve a performance issue by having the LunaProvider as the default because that would result in the symmetric keys also being used in the HSM which is not optimal for performance.

Managing Security for Java Developers

The Luna JSP is a Java API that is intended to be used as an interface between customer-written or third-party Java applications and the Luna PCIe HSM 7. Managing security issues associated with the overall operational environment in which the application is running, including the user interface, is the responsibility of the application.

A common example would be input and capture of user name and password. The application, or a set of organizational procedures, is responsible for making the access control decision regarding whether the user has the necessary permissions (at the organizational level) to access the HSM's services and then must provide protection for the password as it is entered, and erasure from memory after the operation is completed. The Luna JSP will control access to the HSM based on the correct password being input from the application via the Login method, but security outside the HSM is your responsibility.

Non-standard ECDSA Mapping

The Luna provider maps the "ECDSA" signature algorithm to "NONEwithECDSA". The Java convention is to map it to "SHA1withECDSA". This is noted here in case you wish to use it in provider inter-operability testing. This mapping is noted in the Javadoc as well.

For comparison, "RSA" maps to "NONEwithRSA" while "DSA" maps to "SHA1withDSA".

Notes about thread safe, session safe, and multi-threading

PKCS#11 (the standard, and Thales's implementation) requires that a session can be used only by a single thread at a time. That is, multiple threads cannot access the same session simultaneously. Threads can share a session; however the application must ensure that only one thread accesses the session at a time. It is simpler for an application to assign a unique session for each thread, but applications do not need to follow that pattern.

Our LunaProvider endeavors to be thread safe in the way it uses our PKCS#11 library. But customer Java applications must follow the threading model defined by Java. For example, Java Cipher objects (essentially all crypto-related objects) are not thread safe according to the JSP specification. Similar to PKCS#11 sessions, only one thread should use a cipher object at a time. Our LunaProvider requires that the Java application follows that JSP approach.

Therefore, it is very possible, and expected, to see sessions being used by multiple threads, all in legitimate and thread-safe ways according to both JSP and PKCS#11.

Luna JSP Configuration

Luna JSP consists of a single JCA/JCE service provider, that allows a Java-based application to use Luna PCIe HSM 7 products for secure cryptographic operations. Please refer to the Javadocs accompanying the toolkit, for the most current information regarding the Luna JSP packages and LunaProvider functionality.

To install JSP, refer to "Luna JSP Overview and Installation" on page 651.

Luna Java Security Provider

In general, you should use the standard JCA/JCE classes and methods to work with Luna PCIe HSM 7. The following sections provide examples of when you may wish to use the special Luna methods.

Class Hierarchy

All public classes in the Luna Java crypto provider are included in the com.safenetinc.luna package or subpackages of that package. Thus the full class names are (for example):

> com.safenetinc.luna.LunaSlotManager

> com.safenetinc.luna.provider.key.LunaKey

If your application is compliant with the JCA/JCE spec, you will generally not need to directly reference any SafeNet implementation classes. Use the interfaces defined in the java.security packages instead. The exception is if you need to perform an HSM-specific operation, such as modifying PKCS#11 attributes.

Throughout the rest of this document, the short form of the class names is used for convenience and readability. The full class names (of SafeNet or other classes) are used only where necessary to resolve ambiguity.

Special Classes/Methods

The JCA/JCE interfaces were not designed with hardware security modules (HSMs) in mind and do not include methods for managing aspects of a hardware module. Luna JSP provides some additional functions in addition to the standard JCA/JCE API.

The LunaSlotManager class provides custom methods that allow some HSM-specific information to be retrieved. It also provides a way to log in to the HSM if your application cannot make use of the standard KeyStore interface. For details please check the Javadoc which comes with the product.

It is not always necessary to use the LunaSlotManager class. With proper use of the JCE API provided in Luna JSP, your code can be completely hardware-agnostic.

The LunaKey class implements the Key interface and provides all of the methods of that class along with custom methods for manipulating key objects on Luna hardware.

> **NOTE** Sensitive attributes cannot be retrieved from keys stored on Luna hardware. Thus certain JCE-specified methods (such as PrivateKeyRSA.getPrivateExponent()) will throw an exception.

The LunaCertificateX509 class implements the X509Certificate methods along with custom methods for manipulating certificate objects on Luna hardware.

Signature Verification in Software

Using Luna HSM Client 10.7.0 or newer, you can enable the following settings in the java.Security file to allow some algorithms to perform signature verification in software:

> com.safenetinc.luna.provider.verifyInSoftware=<setting>

When <setting> is **true**, signature verification is performed in software. When <setting> is **false** (default), it is performed using the HSM.

> com.safenetinc.luna.provider.verifyInSoftwareProvider=<provider>

In this setting, <provider> specifies the provider to be used for the signature verification (default: **SunRsaSign**).

> **NOTE** This default setting requires minimum Java JDK 1.8.0-251.

Examples

The Luna JSP comes with several sample applications that show you how to use the Luna provider. The samples include detailed comments.

To compile on Windows without an IDE (Administrator privileges may be required):

```
cd <Luna PCIe HSM 7 install>/jsp/samples
javac com\safenetinc\luna\sample\*.java
```

To run:

```
java com.safenetinc.luna.sample.KeyStoreLunaDemo (or any other sample class in that package)
```

> **NOTE** The Luna Keystore is not a physical file like a regular JKS. It is a virtual interface to the HSM and contains only handles for the private key objects.

Authenticating to the HSM

In order to make use of an HSM, it is necessary to activate the device through a login. Depending on the security level of the device, the login will require a plain-text password and/or a PED key.

The preferred method of logging in to the module is through the Java KeyStore interface. The store type is “Luna” and the password for the key store is the challenge for the partition specified.

KeyStore files for the Luna KeyStore must be created manually. The content of the KeyStore file differs if you wish to reference the partition by the slot number or label (preferred). Details of authenticating to the HSM via the KeyStore interface are explained in the Javadoc for LunaKeyStore and in the KeyStoreLunaDemo sample application.

> **NOTE**
> - Thales strongly recommends that you *use the application partition's label* as the identifier for the cryptographic slot on the HSM. That designator never changes, unless you explicitly change label. The slot number, on the other hand, might change, and therefore should not be used in your code.
> - If you are using OpenJDK 9 or newer, you can configure the KeyStore file to allow the Crypto User to log in to the HSM. This is useful in circumstances where you would like the user to be able to use the Java Keytool utility without the risk of wiping, modifying, or adding cryptographic objects. For more information about this utility, refer to "Keytool" below.

Keys in a Luna KeyStore cannot have individual passwords. Only the KeyStore password is used. If your HSM requires PED keys to be presented for authentication and the partition is not already activated, loading the KeyStore will cause the PED to prompt you to present this key.

Other than the KeyStore interface your application may also make use of the LunaSlotManager class or by using a login state created outside of the application through a utility called ‘salogin’. Use of salogin is strongly discouraged unless you have a very specific need.

LunaKeyStoreMP is Deprecated

LunaKeyStoreMP is deprecated for Luna JSP, and may be discontinued in a future release. LunaKeyStoreMP was used in previous releases to allow logical partitioning of the key space on HSMs that have only one partition. This allowed you to create a separate MP key store for each individual client that accessed the partition. Recent SafeNet releases, however, support multiple partitions, and dedicating a partition per client is a superior solution for management and security reasons.

> **NOTE** LunaKeyStoreMP is retained for backwards compatibility reasons only. Do not use LunaKeyStoreMP when creating new applications.

Logging Out

Logging out of the HSM is performed implicitly when the application is terminated normally. Logging out of the HSM while the application is running can be done with the LunaSlotManager class. Please note that any ephemeral (non-persistent) key material present on the HSM will be destroyed when the session is logged out. Because the link to the HSM will be severed, cryptographic objects that were created by the LunaProvider will no longer be usable. Attempting to use these objects after logging out will result in undefined behavior.

All key material which was persisted on the HSM (either through the KeyStore interface or using the proprietary Make Persistent method) will remain on the HSM after a logout and will be accessible again when the application logs back in to the HSM.

Keytool

The Luna JSP may be used in combination with Java’s keytool utility to store and use keys on a Luna PCIe HSM 7, see "Using Java Keytool with Luna PCIe HSM 7" on page 671.

Cleaning Up

Keys that are made persistent will continue to exist on the HSM until they are explicitly destroyed, or until the HSM is reinitialized. Persistent keys that are no longer needed can be explicitly destroyed to free resources on the HSM.

Keys may be removed using the Keytool, or programmatically through the KeyStore interface or other methods available through the API.

LunaSlotManager contains methods that report the number of objects that exist on the HSM. See the Javadoc for LunaSlotManager for more information.

PKCS#11/JCA Interaction

Keys created using the Luna PKCS#11 API can be used with the Luna JSP; the inverse is also true.

Certificate Chains

The PKCS#11 standard does not provide a certificate chain representation. When a Java certificate chain is stored on a Luna token, the certificates of the chain appear as individual objects when viewed through the PKCS#11 API. In order for the LunaProvider to properly identify PKCS#11-created certificates as part of a chain attached to a private key, the certificates must follow the labeling scheme described below.

Java Aliases and PKCS#11 Labels

The PKCS#11 standard defines a large set of object attributes, including the object label. This label is analogous to the Object alias in a java KeyStore.

The Luna KeyStore key entry or a Luna KeyStore certificate entry will have a PKCS#11 object label exactly equal to the Java alias. Similarly, a key created through PKCS#11 will have a Java alias equal to the PKCS#11 label.

Because a java certificate chain cannot be represented as a single PKCS#11 object, the individual certificates in the chain will each appear as individual PKCS#11 objects. The labels of these PKCS#11 objects will be composed of the alias of the corresponding key entry, concatenated with "--certX", where 'X' is the index of the certificate in the java certificate chain.

For example, consider a token that has a number of objects created through the Java API. The objects consist of the following:

> A key entry with alias "signing key", consisting of a private key and a certificate chain of length 2

> A trusted certificate entry with alias "root cert"

> A secret key with alias "session key"

If all objects on the token were viewed through a PKCS#11 interface, 5 objects would be seen:

> A private key with label "signing key"

> A certificate with label "signing key--cert0"

> A certificate with label "signing key--cert1"

> A certificate with label "root cert"

> A secret key with label "session key"

> **NOTE** PKCS#11 labels (strings of ascii characters) and Java aliases (of the java.lang.String type) are usually fully compatible, but problems can arise if non-printable characters are used. To maintain compatibility between Java and PKCS#11, avoid embedding non-printable or non-ascii characters in aliases or object labels.

RSA Cipher

Previously, by default, the Luna JSP RSA cipher mode used raw RSA X.509 encryption, with no padding.

For improved security and compatibility, default padding for RSA cipher has been changed from NoPadding to PKCS1v1_5.

The JCPROV PKCS#11 Java Wrapper

This section describes how to install and use the JCPROV Java wrapper for the PKCS#11 API. It contains the following topics:

> "JCPROV Overview" below

> "Installing JCPROV" on the next page

> "JCPROV Sample Programs" on page 663

> "JCPROV Sample Classes" on page 664

> "JCPROV API Documentation" on page 667

JCPROV Overview

JCPROV is a Java wrapper for the PKCS#11 API. JCPROV is designed to be as similar to the PKCS#11 API as the Java language allows, allowing developers who are familiar with the PKCS#11 API to rapidly develop Java-based programs that exercise the PKCS#11 API.

AES-GMAC and AES-GCM are supported in JCPROV. Use CK_AES_GCM_PARAMS.java to define the GMAC operation. Implementation is the same as for PKCS#11.

JDK compatibility

The JCPROV Java API is compatible with JDK 1.5.0 or higher.

The JCPROV library

The JCPROV library is implemented in **jcprov.jar**, under the namespace **com.safenetinc.jcprov**. It is accompanied by a shared library that provides the native methods used to access the appropriate PKCS#11 library. The name of the shared library is platform dependent, as follows:

Operating system	Shared library
Windows (64 bit)	jcprov.dll
Linux	libjcprov.so

Operating system	Shared library
Solaris	libjcprov.so
AIX	libjcprov.so

Installing JCPROV

Use the Luna HSM Client Installer to install the JCPROV software (runtime and SDK packages). The software is installed in the location specified in the following table:

Operating system	Installation location
Windows	C:\Program Files\safenet\lunaclient\jcprov
Linux	/usr/safenet/lunaclient/jcprov
Solaris	/opt/safenet/lunaclient/jcprov
AIX	/usr/safenet/lunaclient/jcprov

The installation includes a **samples** subdirectory and a **javadocs** subdirectory.

Changing the Java JNI libraries (AIX only)

The Java VM on AIX does not support mixed mode JNI libraries. Mixed mode libraries are shared libraries that provide both 32-bit and 64-bit interfaces. It is therefore essential that you select the correct JNI library to use with your Java VM.

> **NOTE** Luna HSM Client 10.1.0 and newer includes libraries for 64-bit operating systems only.

> **CAUTION!** When JCPROV is installed, links are automatically created to use the 32-bit versions of the JNI libraries. You need to update the links if you are using a 64-bit operating system.

To configure the JNI library for use with a 32-bit Java VM

1. Ensure that the **/usr/safenet/lunaclient/jcprov/lib/libjcprov.a** symbolic link points to a 32-bit version of the library (**libjcprov_32.a**), for example **/usr/safenet/lunaclient/jcprov/lib/libjcprov_32.a**.
2. Ensure that the **/usr/safenet/lunaclient/jcprov/lib/libjcryptoki.a** symbolic link points to a 32-bit version of the library(**libjcryptoki_32.a**), for example **/usr/safenet/lunaclient/jcprov/lib/libjcryptoki_32.a**.

To configure the JNI library for use with a 64-bit Java VM

1. Ensure that the **/usr/safenet/lunaclient/jcprov/lib/libjcprov.a** symbolic link points to a 64-bit version of the library (**libjcprov_64.a**), for example **/usr/safenet/lunaclient/jcprov/lib/libjcprov_64.a**.

2. Ensure that the **/usr/safenet/lunaclient/jcprov/lib/libjcryptoki.a** symbolic link points to a 64-bit version of the library (**libjcryptoki_64.a**), for example **/usr/safenet/lunaclient/jcprov/lib/libjcryptoki_64.a**.

JCPROV Sample Programs

Several sample programs are included to help you become familiar with JCPROV. The binaries for the sample programs are included in the **jcprovsamples.jar** file. You must compile the binaries before you can use the sources provided.

Compiling and running the JCPROV sample programs

> **CAUTION!** You require JDK 1.5.0 or newer to compile the JCPROV sample programs.

It is recommended that you compile the samples in their installed locations, so that the path leading to the samples directory in the installation location will allow them to be executed as documented below.

Prerequisites

For best results, perform the following actions before attempting to compile the sample programs:

> Add **jcprov.jar** to your **CLASSPATH** environment variable

> Add a path to the **CLASSPATH** environment variable that allows JCPROV to use the **com.safenetinc.jcprov.sample** namespace. This is required since all of the applications are registered under this namespace.

To compile the JCPROV sample programs on UNIX/Linux:

1. Set the **CLASSPATH** environment variable to point to **jcprov.jar** and the root path for the sample programs.
   export CLASSPATH=<jcprov_installation_directory>**/***
2. Change directory to the sample programs path.
   cd /usr/safenet/lunaclient/jcprov/samples/com/safenetinc/jcprov/sample
3. Use the **javac** program to compile the examples.
   javac GetInfo.java
4. Use the **java** program to run the samples.
   java com.safenetinc.jcprov.sample.GetInfo -slot 0 -info

To compile the JCPROV sample programs on Windows:

1. Set the **CLASSPATH** environment variable to point to **jcprov.jar** and the root path for the sample programs:
   C:\> **set "CLASSPATH= C:\Program Files\safenet\lunaclient\jcprov\jcprov.jar; C:\program files\safenet\jcprov\samples"**
2. Use the **javac** program to compile the examples:
   C:\Program Files\safenet\lunaclient\jcprov\samples> **javac GetInfo.java**
3. Use the **java** program to run the samples:
   C:\Program Files\safenet\lunaclient\jcprov\samples> **java com.safenetinc.jcprov.sample.GetInfo -info -slot 0**

JCPROV Sample Classes

JCPROV provides sample classes in the <jcprov_installation_directory>/**samples** directory. These include:

> "DeleteKey" below

> "EncDec" on the next page

> "GenerateKey" on the next page

> "GetInfo" on page 666

> "Threading" on page 666

Other samples contained in the **samples** directory may be more or less useful to you depending on what you need. Each relevant sample has a description of both its purpose and its parameters in the header section of its file.

DeleteKey

Demonstrates the deletion of keys.

A generated key is required to use this script. To generate a key, use "GenerateKey" on the next page or refer to "Using Java Keytool with Luna PCIe HSM 7" on page 671

Usage

java com.safenetinc.jcprov.sample.DeleteKey -keyType <keytype> **-keyName** <keyname> [**-slot** <slotId>] [**-password** <password>]

Parameters

Parameter	Description
-keytype	Specifies the type of key you want to delete. Enter this parameter followed by one of the following supported key types: > **des** - single DES key > **des2** - double-length, triple-DES key > **des3** - triple-length, triple-DES key > **rsa** - RSA key pair
-keyName	Specifies the name (label) of the key you want to delete. Enter this parameter followed by the name (label) of the key you want to delete.
-slot	Specifies the slot for the HSM or partition that contains the key you want to delete. Optionally enter this parameter followed by the slot identifier for the HSM or partition that contains the key you want to delete. If this parameter is not specified, the default slot is used. **Default:** 1
-password	Specifies the password for the slot. Optionally enter this parameter followed by the slot password to delete a private key.

EncDec

Demonstrates encryption and decryption operations by encrypting and decrypting a string.

A generated key is required to use this script. To generate a key, use "GenerateKey" below or refer to "Using Java Keytool with Luna PCIe HSM 7" on page 671

Usage

java com.safenetinc.jcprov.sample.EncDec -keyType <keytype> **-keyName** <keyname> [**-slot** <slotId>] [**-password** <password>]

Parameters

Parameter	Description
-keytype	Specifies the type of key you want to use to perform the encryption/decryption operation. Enter this parameter followed by one of the following supported key types: > **des** - single DES key > **des2** - double-length, triple-DES key > **des3** - triple-length, triple-DES key > **rsa** - RSA key pair
-keyName	Specifies the name (label) of the key you want to use to perform the encryption/decryption operation. Enter this parameter followed by the name (label) of the key you want to use to perform the encryption/decryption operation.
-slot	Specifies the slot for the HSM or partition that contains the key you want to use to perform the encryption/decryption operation. Optionally enter this parameter followed by the slot identifier for the HSM or partition that contains the key you want to use to perform the encryption/decryption operation. If this parameter is not specified, the default slot is used. **Default:** 1
-password	Specifies the password for the slot. Optionally enter this parameter followed by the slot password to encrypt/decrypt a private key.

GenerateKey

Demonstrates the generation of keys.

Usage

java com.safenetinc.jcprov.sample.GenerateKey -keyType <keytype> **-keyName** <keyname> [**-slot** <slotId>] [**-password** <password>]

Parameters

Parameter	Description
-keytype	Specifies the type of key you want to generate. Enter this parameter followed by one of the following supported key types: > **des** - single DES key > **des2** - double-length, triple-DES key > **des3** - triple-length, triple-DES key > **rsa** - RSA key pair
-keyName	Specifies the name (label) of the key you want to generate. Enter this parameter followed by the name (label) of the key you want to generate.
-slot	Specifies the slot for the HSM or partition where you want to generate the key. Optionally enter this parameter followed by the slot identifier for the HSM or partition where you want to generate the key. If this parameter is not specified, the default slot is used. **Default:** 1
-password	Specifies the password for the slot. Optionally enter this parameter followed by the slot password to generate a private key.

GetInfo

Demonstrates the retrieval of slot and token information.

Usage

java com.safenetinc.jcprov.sample.GetInfo {**-info** | **-slot** [<slotId>] | **-token** [<slotId>]}

Parameters

Parameter	Description
-info	Retrieve general information.
-slot	Retrieve slot information for the specified slot. Enter this parameter followed by the slot identifier for the slot you want to retrieve information from. If <slotId> is not specified, information is retrieved for all available slots.
-token	Retrieve token information for the HSM or partition in the specified slot. Enter this parameter followed by the slot identifier for the HSM or partition you want to retrieve information from. If <slotId> is not specified, information is retrieved for all available slots.

Threading

This sample program demonstrates different ways to handle multi-threading.

This program initializes the Cryptoki library according to the specified locking model. Then a shared handle to the specified key is created. The specified number of threads is started, where each thread opens a session and then enters a loop which does a triple DES encryption operation using the shared key handle.

It is assumed that the key exists in slot 1, and is a Public Token object.

A generated key is required to use this script. To generate a key, use "GenerateKey" on page 665 or refer to "Using Java Keytool with Luna PCIe HSM 7" on page 671

Usage

java com.safenetinc.jcprov.sample.Threading -numThreads <numthreads> **-keyName** <keyname> **-locking** { **none** | **os** | **functions** } [**-v**]

Parameters

Parameter	Description
-numthreads	Specifies the number of threads you want to start. Enter this parameter followed by an integer that specifies the number of threads you want to start.
-keyName	Specifies the triple-DES key to use for the encryption operation. Enter this parameter followed by the name (label) of the key to use for the encryption operation.
-locking	Specifies the locking model used when initializing the Cryptoki library. Enter this parameter followed by one of the following locking models: > **none** - do not use locking when initializing the Cryptoki library. If you choose this option, some threads should report failures. > **os** - use the native operating system mechanisms to perform locking. > **functions** - use Java functions to perform locking
-v	Specifies the password for the slot. Optionally enter this parameter followed by the slot password to generate a private key.

JCPROV API Documentation

The JCPROV API is documented in a series of javadocs. The documentation is located in the <jcprov_installation_directory>**/javadocs** directory.

Java or JSP Errors

In the process of using our JSP (Java Service Provider) or programming for Java clients, you might encounter a variety of errors generated by various levels of the system. In rare cases those might be actual problems with the system, but in the vast majority of cases the errors are the system (or the Client-side libraries) telling you that you (or your application) have done something "wrong". In other words, the error messages are guidance to ensure that your actions and your programs are giving the system what it needs (in the right order and format) to complete the tasks that you ask of it.

Keep in mind that there are several levels involved. The Luna appliance and its HSM cryptographic module have both software and firmware built in. Among other things, the system software handles the system side of communication between you (either as administrator or as Client) and the HSM on the appliance. In general, a client-side program (or programmer) would not encounter error messages directly from the system. If an error condition arises on the system, the most likely visibility would be error messages in the system logs - viewed by the appliance administrator - or else client-side messages based upon the interaction of the client-side software (ours and yours) with the appliance.

On the client side, the JSP and any Java programs that you use would be overlaid on, and using, the Luna library, which is an extended version of PKCS#11, customized to make use of our HSM (the standard itself and the cryptoki library are oriented toward in-software implementation of cryptographic functions, with some generic support of generic HSM functions, leaving room for each HSM supplier to support their own special functions by extending the standard). PKCS#11 is an RSA Laboratories cryptographic standard, and our libraries are a C-language implementation of that standard. You can view all that is known about PKCS #11 error conditions and messages at the RSA website.

See Library Codes for a summary of error codes and their meanings, which includes the Luna extensions to the PKCS#11 standard that are specific to our HSM. Note that "error codes" do not usually indicate a problem with the appliance or HSM - they indicate an exception condition has been encountered, possibly because you (or your application) stopped/canceled a requested action before it could complete, provided incorrect or incomplete or wrongly-formatted input data, and so on, or possibly because a network connection has been disrupted, power has failed, or any of a variety of situations has been detected.

The JSP and your Java programming are overlaid on top of the PKCS#11 and Luna libraries. An error reported by a Java application might refer to a problem at the Java or JSP level, or the error might have been passed through from a lower level.

If you receive a cryptic error that looks something like:

```
Exception in thread "main"
com.safenetinc.crypto.LunaCryptokiException: function 'C_Initialize' returns 0x30
```

then this error has been passed through from a lower layer and is not a Java or JSP error. You should look in the Error Codes page (link above) or in the PKCS#11 standard for the meaning of any error in a similar format.

In general, we wrap cryptoki exception codes. Most exceptions thrown by the JSP are in accordance with the specification. Check the Javadoc for the API call that threw the exception.

> LunaException is used to report a LunaProvider-specific exception.

> LunaCryptokiException reports errors returned by the HSM. Those might be wrapped in other Exceptions.

Re-Establishing a Connection Between Your Java Application and Luna PCIe HSM 7

Thales provides Java code samples for performing various application functions. For the proper method for performing a reconnect between a Java application and the Luna PCIe HSM 7 in the event of a disconnect, see **MiscReconnectDemo.java** in the Samples folder.

Recovering From the Loss of All HA Members

The reinitialize method of the **LunaSlotManager** class takes the role of the PKCS#11 functions **C_Finalize** and **C_Initialize**. It is intended to be used when a complete loss of communication happens with all the members of your High Availability (HA) group.

This section describes the situations in which you should use this method, the effect this method has on a running application, and how to use this method safely. It is assumed that the auto-recovery features of the HA group are enabled.

You should read this section if you are developing an application that uses the LunaProvider in an environment that leverages an HA group of Luna Network HSM 7 appliances, so that you can safely recover an entire HA group.

When to Use the reinitialize Method

When using the high-availability (HA) features of Luna PCIe HSM 7, the auto-recovery feature will resolve situations where connectivity is lost to a subset of members for a brief time. However, if you lose connection to all members then the connection cannot be automatically recovered. Finalizing the library and initializing it again is the only way to recover other than restarting the application.

Why the Method Must Be Used

In an HA group, we rely on having at least one member present in order to maintain state. If all of the members have been lost, then we cannot make any determination of which member has a known good state. Also, when a connection to a member is lost, the authenticated state is lost. When an individual member returns, we can use the authenticated state from another member to authenticate to the one that has returned. When all members are lost, then the authenticated state is lost on all members.

What Happens on the HSM

The Network Trust Link Service (NTLS) on the HSM appliance is responsible for cleaning up any cryptographic resources, such as session objects, and cryptographic operation contexts when a connection to the client is lost. This happens when the socket closes.

Effect on Running Applications

All resources created within the LunaProvider must be treated as junk after the library is finalized. Sessions will no longer be valid, session objects will point to non-existent objects or worse to a wrong object, and **Signature/Cipher/Mac/etc** objects will have invalid data.

Even **LunaKey** objects, which represent persistent objects, may contain invalid data. When the virtual slot is constructed in the library, the virtual object table is built from the objects present on each individual member. There is no guarantee that objects will have the same handle from one initialization to the next. This is true from the moment the connection to the group is severed. All these resources must be released before calling the reinitialize method. Beyond causing undesirable behavior when used, if these objects are garbage collected after cryptographic operations resume, they can result in the deletion of new objects or sessions.

Using the Method Safely

The first indication that all communications may have been lost with the group is a **LunaException** reporting an error code of **0x30** (Device Error). Other possible error codes that can indicate this status are **0xE0** (Token not present) and **0xB3** (Session Handle invalid). The **LunaException** class does not provide the error code as a discrete value and you will have to parse the message string to determine this value.

At this point, you should validate that the group has been lost. The **com.safenetinc.luna.LunaHAStatus** object is best suited for this. Your application should know the slot number of the HA slot that you are using because it may not be able to query this information from the label when the slot is missing.

Example

```
LunaHAStatus status = new LunaHAStatus(haSlotNumber);
```

You can query the object for detailed information or just use the **isOK()** method to determine if the group has been lost. The **isOK()** method will return true if all members are still present. If all members are gone, an exception will be thrown.

If no application is thrown, the application should be able to proceed operating, and any individual members of the HA group that have been lost will be recovered by the library. Further details on failed members can be queried through the LunaHAStatus object.

In many highly threaded applications, such as web applications, it is desirable to have a singleton, which is responsible for keeping track of the health of the HSM connection. This can be done by having worker threads report information to this singleton, by having a specific health check thread, or through a combination of the two.

Once the error state is discovered, all worker threads should be stopped or allowed to return an error. It may take up to 40 seconds from the time the group was lost for all threads to discover that there is an error. It can take 20 seconds for any given command to time out as a result of network failure. Once this happens, new commands will not be sent to that HSM, but a command may have just been sent and that command will have its own 20-second timeout. As mentioned above, in the section on application effects, all of the objects created or managed by the LunaProvider must be considered at this point to contain junk data. Operating after recovery with this junk data can cause undesired effects. This means all keys, signature, cipher, Mac, KeyGenerator, KeyPairGenerator, X509Certificate, and similar objects must be released to the garbage collector. Instances of most non-SPI (LunaAPI, LunaSlotManager, LunaTokenManager, etc.) objects do not pose a problem, but any instances of LunaSession held in the application during the course of the reinitialize can cause problems if they are returned to the session pool after the reinitialization takes place.

Cryptographic processing in the application should be halted until connection with the HSMs is back to a known good state. It may be appropriate to hold operations in a queue for processing later or to return an Out of Service message.

Once the objects have been released and no further processing will occur, the application should attempt recovery of the connection. This is done through the **com.safenetinc.luna.LunaSlotManager.reinitialize** method. This method will first clear session objects held within the provider before finalizing the library. After the library is finalized, it will initialize it again by invoking the **C_Initialize** method. This method will establish a connection with all the HSMs if possible. The same **isOK()** method of **LunaHAStatus** can be used to determine if the group has been recovered successfully.

It is also important to only have a single thread call the **reinitialize** method. When multiple threads try to unload or load the library at the same time, errors can occur.

Using Java Keytool with Luna PCIe HSM 7

This page describes how to use the Java KeyTool application with the LunaProvider.

Limitations

The following limitations apply:

> You cannot use the importkeystore command to migrate keys from a Luna KeyStore to another KeyStore.

> Private keys cannot be extracted from the KeyStore unless you have the Key Export model of the HSM.

> By default secret keys created with the LunaProvider are non-extractable.

The example below uses a KeyStore file containing only the line “slot:0”. This tells the Luna KeyStore to use the token in slot 0.

> **NOTE** The Luna Keystore is not a physical file like a regular JKS. It is a virtual interface to the HSM and contains only handles for the private key objects.

For information on creating keys through Key Generator or Key Factory classes please see the LunaProvider Javadoc or the JCA/JCE API documentation.

Keys (with self signed certificates) can be generated using the keytool by specifying a valid Luna KeyStore file and specifying the KeyStore type as “Luna”. The password presented to authenticate to the KeyStore is the challenge password of the partition.

Example

```
keytool -genkeypair -alias myKey -keyalg RSA -sigalg SHA256withRSA -keystore keystore.luna -
storetype Luna
Enter keystore password:
What is your first and last name?
[Unknown]:  test
What is the name of your organizational unit?
[Unknown]:  codesigning
What is the name of your organization?
[Unknown]:  Thales
What is the name of your City or Locality?
[Unknown]:  Ottawa
What is the name of your State or Province?
[Unknown]:  ON
What is the two-letter country code for this unit?
[Unknown]:  CA
Is CN=test, OU=codesigning, O=Thales, L=Ottawa, ST=ON, C=CA correct?
[no]:  yes
Enter key password for <myKey>
(RETURN if same as keystore password):
```

Keytool Usage and Examples

The LunaProvider is unable to determine which PKCS#11 slot to use without providing a keystore file. This file can be manually created to specify the desired slot by either the slot number or partition label. The naming of the files is not important - only the contents.

The keytool examples below refer to a keystore file named bylabel.keystore. Its content is just one line:

```
tokenlabel:a-partition-name
```

where a-partition-name is the name of the partition you want the Java client to use.

Here is the (one line) content of a keystore file that specifies the partition by slot number:

```
slot:0
```

where 1 is the slot number of the partition you want the Java client to use.

To test that the Java configuration is correct, execute:

```
my-lunaclient:~/luna-keystores$ keytool -list -v -storetype Luna -keystore bylabel.keystore
```

The system requests the Crypto Officer password of the partition and shows its contents.

> **NOTE** To log in to the partition with a different role, you must add a line to the keystore file to specify that role:
>
> > Crypto User:
>
> ```
> usertype:CKU_CRYPTO_USER
> ```
>
> > Limited Crypto Officer:
>
> ```
> usertype:CKU_LIMITED_CRYPTO_OFFICER
> ```
>
> See also "LunaKeyStore Reference" on page 676.

Here is a sample command to create an RSA 2048 bit key with SHA256withRSA self-signed certificate. This example uses java 6, other versions might be slightly different.

keytool -genkeypair -alias keyLabel -keyalg RSA -keysize 2048 -sigalg SHA256withRSA -storetype Luna -keystore bylabel.keystore -validity 365

```
Enter keystore password:
What is your first and last name?
  [Unknown]:  mike
What is the name of your organizational unit?
  [Unknown]:  appseng
What is the name of your organization?
  [Unknown]:  Thales
What is the name of your City or Locality?
  [Unknown]:  ottawa
What is the name of your State or Province?
  [Unknown]:  on
What is the two-letter country code for this unit?
  [Unknown]:  ca
Is CN=mike, OU=appseng, O=Thales, L=ottawa, ST=on, C=ca correct?
  [no]:  yes
Enter key password for <keyLabel>
  (RETURN if same as keystore password):
```

With the Luna provider there is no concept of a key password and anything entered is ignored.

The following is a more elaborate sequence of keytool usage where the final goal is to have the private key generated in the HSM through keytool “linked” to its certificate.

Import CA certificate

It is mandatory to import the CA certificate – keytool verifies the chain before importing a client certificate:

```
my-lunaclient:~/luna-keystores$ keytool -importcert -storetype Luna -keystore bylabel.keystore
-alias root-mikeca -file mike_CA.crt
```

It is not required to import this certificate in the Java default cacerts keystore.

Generate private key

Generate the private key. It is not important that the sigalg specified matches the one used by the CA. You can also have OU, O, L, ST, and C different from the ones in the CA certificate.

```
my-lunaclient:~/luna-keystores$ keytool -genkeypair -alias java-client2-key -keyalg RSA -
keysize 2048 -sigalg SHA256withRSA -storetype Luna -keystore bylabel.keystore
Enter keystore password:
What is your first and last name?
[Unknown]:  java-client2
What is the name of your organizational unit?
[Unknown]:  SE
What is the name of your organization?
[Unknown]:  SFNT
What is the name of your City or Locality?
[Unknown]:  bgy
What is the name of your State or Province?
[Unknown]:  bg
What is the two-letter country code for this unit?
[Unknown]:  IT
Is CN=java-client2, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT correct?
[no]:  yes
Enter key password for <java-client2-key>
(RETURN if same as keystore password):
```

Verify that the private key is in the partition:

```
my-lunaclient:~/luna-keystores$ keytool -list -v -storetype Luna -keystore bylabel.keystore
Enter keystore password:
Keystore type: LUNA
Keystore provider: LunaProvider
Your keystore contains 2 entries
Alias name: root-mikeca
Creation date: Oct 4, 2012
Entry type: trustedCertEntry
Owner: EMAILADDRESS=username@thales.com, CN=mike CA, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
Issuer: EMAILADDRESS=username@thales.com, CN=mike CA, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
Serial number: 1
Valid from: Thu Oct 04 09:02:00 CEST 2012 until: Tue Oct 04 09:02:00 CEST 2022
Certificate fingerprints:
      MD5:  A2:15:4F:94:70:2B:D2:F7:C0:96:B1:47:F2:1D:03:E9
      SHA1: B3:4A:68:0A:8D:12:39:86:11:CE:EF:22:1B:D1:DE:8D:E9:19:2B:F4
      Signature algorithm name: SHA256withRSA
      Version: 3
*******************************************
*******************************************
Alias name: java-client2-key
Creation date: Oct 4, 2012
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=java-client2, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
```

```
Issuer: CN=java-client2, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
Serial number: 506d42dd
Valid from: Thu Oct 04 10:03:41 CEST 2012 until: Wed Jan 02 09:03:41 CET 2013
Certificate fingerprints:
      MD5:  7A:37:72:6B:8A:05:B6:49:91:70:0F:C4:04:1F:69:D9
      SHA1: 05:CD:9F:A5:37:0B:A6:A3:65:24:56:40:5E:29:2D:95:2D:53:8F:5F
      Signature algorithm name: SHA256withRSA
      Version: 3
```

Create the CSR

Create the CSR to be submitted to the CA.

```
my-lunaclient:~/luna-keystores$ keytool -certreq -alias java-client2-key -file client2-
mikeca.csr -storetype Luna -keystore bylabel.keystore
Enter keystore password:
```

Now have the CSR signed by the CA. Have the issued certificate exported to include the certificate chain. Without the chain, keytool fails with the error:

```
java.lang.Exception: Failed to establish chain from reply
```

If you do not have the chain, you can use the steps in the section below to build the chain yourself.

To translate a PKCS#7 exported certificate from DER format to PEM format use the following:

```
my-lunaclient $ openssl pkcs7 -inform der -in Luna_Key.p7b -outform pem -out Luna_Key-pem.p7b
```

Microsoft CA exports certificates with chain only in PKCS#7 PEM encoded format.

Import client certificate

Now import the client certificate:

```
user@myserver:~/luna-keystores$ keytool -importcert -storetype Luna -keystore bylabel.keystore
-alias java-client2-key -file java-client2.crt
Enter keystore password:
Certificate reply was installed in keystore
```

Ensure that it is linked to the private key generated previously – the chain length is not 1 (Certificate chain length: 2)

```
user@myserver:~/luna-keystores$ keytool -list -v -storetype Luna -keystore bylabel.keystore
Enter keystore password:
Keystore type: LUNA
Keystore provider: LunaProvider
Your keystore contains 2 entries
Alias name: root-mikeca
Creation date: Oct 4, 2012
Entry type: trustedCertEntry
Owner: EMAILADDRESS=username@thales.com, CN=mike CA, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
Issuer: EMAILADDRESS=username@thales.com, CN=mike CA, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
Serial number: 1
Valid from: Thu Oct 04 09:02:00 CEST 2012 until: Tue Oct 04 09:02:00 CEST 2022
Certificate fingerprints:
      MD5:  A2:15:4F:94:70:2B:D2:F7:C0:96:B1:47:F2:1D:03:E9
      SHA1: B3:4A:68:0A:8D:12:39:86:11:CE:EF:22:1B:D1:DE:8D:E9:19:2B:F4
      Signature algorithm name: SHA256withRSA
      Version: 3
*******************************************
*******************************************
Alias name: java-client2-key
Creation date: Oct 4, 2012
Entry type: PrivateKeyEntry
```

```
Certificate chain length: 2
Certificate[1]:
Owner: CN=java-client2, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
Issuer: EMAILADDRESS=username@thales.com, CN=mike CA, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
Serial number: 5
Valid from: Thu Oct 04 10:07:00 CEST 2012 until: Fri Oct 04 10:07:00 CEST 2013
Certificate fingerprints:
       MD5:  4B:F0:9E:BC:EB:6A:88:2B:87:3A:76:35:7C:DE:4B:B4
       SHA1: F1:0C:BC:E3:A1:97:E4:8B:24:2D:44:43:7A:EA:71:52:B3:C3:20:D7
       Signature algorithm name: SHA256withRSA
       Version: 3
Certificate[2]:
Owner: EMAILADDRESS=username@thales.com, CN=mike CA, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
Issuer: EMAILADDRESS=username@thales.com, CN=mike CA, OU=SE, O=SFNT, L=bgy, ST=bg, C=IT
Serial number: 1
Valid from: Thu Oct 04 09:02:00 CEST 2012 until: Tue Oct 04 09:02:00 CEST 2022
Certificate fingerprints:
       MD5:  A2:15:4F:94:70:2B:D2:F7:C0:96:B1:47:F2:1D:03:E9
       SHA1: B3:4A:68:0A:8D:12:39:86:11:CE:EF:22:1B:D1:DE:8D:E9:19:2B:F4
       Signature algorithm name: SHA256withRSA
       Version: 3
```

How to build a certificate with chain ...

When you receive the client certificate without the chain, it is possible to build a PKCS#7 certificate that includes the chain (and then feed it to keytool -importcert). In short, the “single” certificates without the chain can be “stacked” together by manually editing a PEM cert file; this PEM cert file can then be translated into a PKCS#7 cert. How? Like this:

1. Prerequisites. Have all the certs in .crt format. The cert in this format is represented as an ASCII file starting with the line

   ```
   -----BEGIN CERTIFICATE-----
   ```

   and ending with

   ```
   -----END CERTIFICATE-----
   ```

   For example, if the client cert is issued by a subCA and the subCA is signed by a root CA, you will have 3 cert files – the client cert, the subCA cert, and the root CA cert. If the certs are not in .crt format, openssl can be used to transform the format that you have into .crt format. See notes below.

2. Open a new text file, calling it, for example, cert-with-chain.crt. Insert into this file the content of the certificates in the chains. For the above example, you must first insert the client cert, then the subCA cert, then the root CA cert. The content of the file would then resemble the following:

   ```
   -----BEGIN CERTIFICATE-----
                <-- client cert goes here
   -----END CERTIFICATE-----
   -----BEGIN CERTIFICATE-----
                <-- subCA cert goes here
   -----END CERTIFICATE-----
   -----BEGIN CERTIFICATE-----
                <-- root CA cert goes here
   -----END CERTIFICATE-----
   ```

3. Use the following openssl command to convert the new certificate with chain, that you just created above, to a PKCS#7 certificate with chain:

```
my-sa $ openssl crl2pkcs7 -nocrl -certfile HSM_Luna-manual-chain.crt -out HSM_Luna-manual-chain.p7b -certfile root_CA.crt
```

4. Keytool is then able to import this .p7b certificate into the Luna keystore and correctly validate the chain.

Additional minor notes

1. Command to add a CA to the default CA cert store “cacerts”:

```
root@myserver:~# keytool -importcert -trustcacerts -alias root-mikeca -file /home/mike/luna-keystores/mike_CA.crt -keystore /etc/java-6-sun/security/cacerts
```

2. Use the following openssl command to convert a PKCS#7 certificate DER-encoded into a PKCS#7 PEM-encoded certificate:

```
user@myserver:~/tmp/$ openssl pkcs7 -inform der -in java-client2.p7b -out java-client2-pem.p7b
```

3. Use the following openssl command to convert a PKCS#7 DER-encoded certificate into a .crt PEM certificate:

```
user@myserver:~/tmp/$ openssl pkcs7 -print_certs -inform der -in mike_CA.p7b -out mike_CA-p7-2-crt.crt
```

4. Use the following openssl command to convert a PEM certificate with chain to a PKCS#7 with chain:

```
user@myserver:~/tmp/$ openssl crl2pkcs7 -nocrl -certfile HSM_Luna-manual-chain.crt -out HSM_Luna-manual-chain.p7b -certfile mike_CA.crt
```

LunaKeyStore Reference

This page reproduces information found in the Luna JSP javadocs.

```
java.lang.Object
    java.security.KeyStoreSpi
        com.safenetinc.luna.provider.LunaKeyStore
public class LunaKeyStore
extends java.security.KeyStoreSpi
```

This is the preferred means of managing Luna HSM access via the LunaProvider. This is the KeyStore engine class for storing objects on Luna hardware. The data in a Luna KeyStore corresponds to the objects in a hardware token. Like a JKS KeyStore, a Luna KeyStore must be loaded before being used. Unlike a JKS KeyStore, setting a key/certificate entry causes the key/certificate to be immediately written to the HSM as a token (permanent) object with the specified alias; the Luna provider does not wait until store() is called.

When no InputStream is specified, the KeyStore acts essentially as a front- end to the default HSM slot.

```
KeyStore ks = KeyStore.getInstance("Luna"); ks.load(null, "mypasswd".toCharArray());
```

The code above is the bare minimum necessary to get a Luna KeyStore up and running. This KeyStore is backed by the HSM partition that is at the currently specified default slot in LunaSlotManager. If no password is supplied in load, the user must log in via LunaSlotManager before using the keystore.

When the InputStream is backed by a file, the file should specify the slot to use in one of two formats. Using the string "tokenlabel:label" will attempt to open the KeyStore against the token with the provided label. Using "slot:<slotNum>" will attempt to open the KeyStore against the token at the provided slot. It is recommended that the token label be used, as the slot number of a given token may change but the label will not.

As well, the user type can be specified by adding a line with "usertype:<user type>" with possible values of CKU_CRYPTO_USER or CKU_CRYPTO_OFFICER.

Object Caching can be enabled for the LunaKeyStore by adding a line with "caching:true". If Caching is enabled the number of loading threads can be specified by adding a line with "loadingthreads:<number of threads>". If caching is enabled, adding a line with "cachingstrict:true" will prevent the LunaKeystore from accessing the HSM to search for the object if the object isn't found in the cache. If caching is enabled, adding a line with "clearcache:false" will prevent the object cache from being cleared when the LunaKeyStore is loaded. If caching is enabled, adding a line with "loadcache:false" will prevent the object cache from being loaded when the LunaKeyStore is loaded.

Using a file to back the InputStream in the load() method is optional. If there is no existing KeyStore file, a new KeyStore can be loaded by creating an InputStream backed by a String in one of the two formats above.

```
ByteArrayInputStream slot = new ByteArrayInputStream("slot:2".getBytes()); KeyStore ks =
KeyStore.getInstance("Luna"); ks.load(slot, "mypasswd".toCharArray());
```

The code above will attempt to open a KeyStore on slot 2 with the partition password "mypasswd". Multiple KeyStores can be opened on the same slot, but they are not guaranteed to be thread-safe. External synchronization is recommended.

If an InputStream is provided that contains anything other than a string in one of the two formats above, the KeyStore will attempt to use the default slot.

 Copyright 2001-2024 Thales Group

CHAPTER 9: Microsoft Interfaces

This chapter describes the Microsoft interfaces to the PKCS#11 API. It contains the following topics:

> "Luna CSP Registration Utilities" below

> "Luna KSP for CNG Registration Utilities" on page 683

> "Run a Windows CNG application as Crypto Officer limited to key handling ability at Crypto User level" on page 689

> "Luna CSP Calls and Functions" on page 692

Luna CSP Registration Utilities

This section describes how to use the Luna CSP registration tool and related utilities to configure the Luna HSM client to use a Luna PCIe HSM 7 with Microsoft Certificate Services. You must be the client Administrator or a member of the Administrators group to run the Luna CSP tools.

The Luna CSP can be used by any application that acquires the context of the Luna CSP. All users who log in and use the applications that acquired the context have access to the Luna CSP. After you register the Luna PCIe HSM 7 partitions with Luna CSP, your CSP and KSP code should work the same whether the Luna PCIe HSM 7 (crypto provider) or the default provider is selected.

The Luna CSP is an optional client feature. During client installation, select **CSP (CAPI) / KSPCNG)** to install it. To install the feature later, run the client installer again, select the option, and click **Modify**.

By default, the Luna CSP utilities are installed in <client_install_dir>/**CSP**. The installation includes **LunaCSP.dll**, the library used by CSP to interact with **Cryptoki.dll**, and the following utilities:

> **"register" below**

- "Registering Partitions/HA Groups to CSP" on page 680
- "Registering Cryptographic Algorithms to be Used in Software" on page 681
- "Enabling Key Counting" on page 681

> **"ms2Luna" on page 681** — Used to migrate Microsoft CSP keys to a Luna PCIe HSM 7 partition

> **"keymap" on page 682** — Used to manage keys on the partition for use with Microsoft CSP

register

You can use the CSP registration tool (<client_install_dir>/**CSP**/**register.exe**) to perform the following functions:

> Register application partitions/HA groups and their passwords/challenge secrets for use with the Luna CSP (see "Registering Partitions/HA Groups to CSP" on page 680).

> Register which non-RSA cryptographic algorithms you want performed in software only (see "Registering Cryptographic Algorithms to be Used in Software" on page 681).

> Enable key counting in KSP/CSP (see "Enabling Key Counting" on page 681).

> Register the provider library with the Windows OS to make it available for applications.

NOTE CSP or KSP registration includes a step that verifies the DLLs are signed by our certificate that chains back to the DigiCert root of trust G4 (in compliance with industry security standards).

This step can fail if your Windows operating system does not have the required certificate. If you have been keeping your Windows OS updated, you should already have that certificate.

If your Luna HSM Client host is connected to the internet, use the following commands to update the certificate manually:

certutil -urlcache -f http://cacerts.digicert.com/DigiCertTrustedRootG4.crt

certutil -addstore -f root DigiCertTrustedRootG4.crt

To manually update a non-connected host

1. Download the DigiCert Trusted Root G4 (http://cacerts.digicert.com/DigiCertTrustedRootG4.crt) to a separate internet-connected computer.
2. Transport the certificate, using your approved means, to the Luna HSM Client host into a <downloaded cert path> location of your choice
3. Add the certificate to the certificate store using the command:

   certutil -addstore -f root <downloaded cert path>

Syntax

register.exe [**/partition** | **/algorithms** | **/library** | **/usagelimit**] [**/password**] [**/highavail**] [**/strongprotect**] [**/cryptouser**] [**/?**]

Argument	Shortcut	Description
/algorithms	**/a**	Register algorithms that will be used in software by Microsoft CSP (i.e. not on the HSM). Only non-RSA algorithms can be configured to run in software; RSA algorithms will always run on the HSM hardware.
/cryptouser	**/c**	Register the password/challenge for the Crypto User (read-only crypto role). If this option is not specified, the Crypto Officer password/challenge is registered.
/highavail	**/h**	Register the virtual partition of a high-availability (HA) group.

Argument	Shortcut	Description
/library	**/l**	Register the library and associated provider names for use with CSP. The following providers are registered: > Luna enhanced RSA and AES provider for Microsoft Windows > Luna Cryptographic Services for Microsoft Windows > Luna SChannel Cryptography Services for Microsoft Windows **NOTE** This operation is required only for 32-bit client libraries, which have been discontinued in Luna HSM Client 10.1.0 and newer.
/partition	**/p**	Register a partition and its password/challenge. You are prompted to select which available partitions to register to the CSP. This is the default option. If you type **register** with no additional parameters, then **/partition** is assumed. For example, **register /strongprotect** is the same as **register /partition /strongprotect**.
/password		Specify the user password or challenge for the desired role. By default, this is the Crypto Officer. This option requires minimum Luna HSM Client 10.5.1.
/strongprotect	**/s**	Strongly protect the challenge for registered partitions. This option ensures that only existing client users can access the CSP partitions. After running **register /strongprotect**, new users are not allowed to use CSP.
/usagelimit	**/u**	Set the maximum usage limit for RSA keys using CSP. Enter **0** to register unlimited uses.

Registering Partitions/HA Groups to CSP

Use the **"register" on page 678** utility to register application partitions or HA virtual slots to the CSP. The Crypto Officer or Crypto User must complete this procedure, depending on which role you wish to use.

> **NOTE** You cannot register a combination of HA groups and application partitions; either physical or virtual slots may be registered to the CSP at one time.

To register an application partition or HA group to the CSP

1. In a command prompt, navigate to the Luna CSP install directory and register the desired application partition (s) or HA group(s). Specify **/cryptouser** to register the CU role. Otherwise, the CO role will be registered. If you want to register both roles, you can run the command twice, once with **/cryptouser** and once without.

   "register" on page 678 [**/highavail**] [**/cryptouser**]

   You are prompted (**y**/**n**) to decide whether to register each available partition or HA virtual slot.
2. Install and/or configure your application(s).
3. Run each of your applications once to use Luna CSP.
4. Ensure the security of the registered role passwords/challenges by specifying **/strongprotect**.

"register" on page 678 **/strongprotect**

5. If you are using a 32-bit CSP provider, register the library. If you are using a 64-bit CSP provider, this is done automatically.

   "register" on page 678 **/library**

   You can now run all applications as usual.

Registering Cryptographic Algorithms to be Used in Software

Certain symmetric operations such as hashing may be completed faster in software than on the Luna PCIe HSM 7. The **register /algorithms** command allows you to choose which algorithms to de-register from the Luna PCIe HSM 7. This may improve performance for operations that use these algorithms, but there is a security cost (exposing the operation in software). Signing and other asymmetric operations are always done on the HSM.

To register algorithms for software-only use

1. In a command prompt, navigate to the Luna CSP install directory and register the desired algorithms to be used in software.

   "register" on page 678 **/algorithms**

   You are prompted (**y**/**n**) to decide whether each available algorithm should be used in software.

Enabling Key Counting

Key counting allows you to specify the maximum number of times that a key can be used.

To enable key counting

1. In a command prompt, navigate to the Luna CSP install directory and register the key usage limit.

   "register" on page 678 **/usagelimit**

   You are prompted to enter a key usage limit. You can turn the feature off (unlimited uses) by entering **0**.

ms2Luna

Use the **ms2Luna** utility (<client_install_dir>/**CSP**/**ms2Luna.exe**) to migrate existing Microsoft CSP keys held in software to a registered partition/HA group on the Luna PCIe HSM 7. It requires the thumbprint of a certificate held in the client's keystore.

Prerequisites

> You must already have registered a partition/HA group using the "register" on page 678 utility.

> Private keys must be exportable to be migrated to the HSM.

To migrate Microsoft CSP keys to the Luna PCIe HSM 7

1. In a command prompt, navigate to the Luna CSP install directory and migrate your existing keys to the HSM.

   ms2Luna

   You are prompted for the CSP certificate thumbprint.

keymap

Use the **keymap** utility (<client_install_dir>/**CSP**/**keymap.exe**) to manage keys for use with CSP. CSP needs three objects for a certificate to work:

> Private key

> Public key

> A container: data object containing the certificate's association with the keys

A container is automatically created for all keypairs created using the CSP. For existing keypairs that were created outside the CSP, you must create a container and associate it with each keypair to make them available to the CSP.

When you run the **keymap** utility and select an available slot, the following options are available:

Option	Name	Description
1	**Browse Objects**	List the objects on the slot (public keys, private keys, and containers) that can be used by the CSP.
2	**Create Key Container**	Create a key container that can be used by the CSP.
3	**View Key Container**	Display information about a key container and the keys associated with it.
4	**Associate Keys With Container**	Map a keypair to an existing container. There are two possible algorithm mappings, depending on the intended purpose of the keypair: > **Signature**: keypair will be used for signing operations > **Exchange**: keypair will be used for key exchange
5	**Do Nothing**	Take no action.
99	**Destroy Key Container**	Destroy a key container object. This has no effect on the keys associated with a container.
0	**Exit**	Exit the **keymap** utility.

Luna KSP for CNG Registration Utilities

CNG (Cryptography Next Generation) is Microsoft's cryptographic application programming interface (API), replacing the older Windows cryptoAPI (CAPI). CNG adds new algorithms along with additional flexibility and functionality. Thales provides Luna CSP for applications running in older Windows crypto environments (running CAPI), and Luna KSP for newer Windows clients (running CNG). Consult Microsoft documentation to determine which one is appropriate for your client operating system.

KSP must be installed on any computer that is intended to act via CNG as a client of the HSM, running crypto operations in hardware. You need KSP to integrate Luna cryptoki with CNG and to use the newer functions and algorithms in Microsoft IIS.

After you register the Luna PCIe HSM 7 partitions with Luna KSP, your KSP code should work the same whether a Luna HSM (crypto provider) or the default provider is selected.

> **NOTE** Be aware when working in a mixed environment or updating applications that previously used CAPI and the Luna CSP - the new algorithms supported by CNG (such as SHA512 and ECDSA) in Certificate Services are not recognized by systems that use CAPI. If Certificate Services is configured to use any of these new algorithms then the signed certificates can be installed only on systems that are aware of these new algorithms. Any of the systems that use CAPI will not be able to use this feature and certificate installation will fail.

The Luna KSP is an optional client feature. During client installation, select **CSP (CAPI) / KSP (CNG)** to install it. To install the feature later, run the client installer again, select the option, and click **Modify**.

By default, the Luna KSP utilities are installed in <client_install_dir>/**KSP**. The installation includes the following utilities:

> **"kspcmd" on the next page**
- "Configuring the KSP Using the Command Line" on page 685

> **"KspConfig" on page 685**
- "Configuring the KSP Using the GUI" on page 686

> **"ms2Luna" on page 687** — Used to migrate Microsoft CSP keys to a Luna PCIe HSM 7 partition

> **"ksputil" on page 688** — Used to display and manage partition keys that are visible to the KSP

> **NOTE** *KSP works with Crypto Officer only*.
>
> For management and security and compliance reasons, you might prefer to limit your applications to read-only usage of keys such as the Crypto User role provides. However, since KSP cannot function as CU, you can simulate the CO/CU role separation - see "Run a Windows CNG application as Crypto Officer limited to key handling ability at Crypto User level" on page 689.
>
> This allows you to use the full capability of Crypto Officer for partition and object management tasks, whenever necessary, and then resume running your CNG/KSP-using application as CO, but with reduced, read-only permissions.

kspcmd

You can use this utility (<client_install_dir>**/KSP/kspcmd.exe**) to register the KSP library and partitions via the Windows command line.

> **NOTE** To register the library and partitions using a GUI, use **"KspConfig" on the next page**. It is unnecessary to use both utilities.

Syntax

kspcmd.exe

library <path**cryptoki.dll**>
nonAdminuser
password /s <slot_label> [**/u** <username>] [**/c** <co_password/challenge>] [**/d** <domain>]
usagelimit
viewslots

Argument	Shortcut	Description
library <path**cryptoki.dll**>	**l**	Register the library and associated provider names with KSP.
nonAdminUser	**n**	Enable non-administrator users on the client to use Luna KSP. This feature requires minimum Luna HSM Client 10.4.0.
password	**p**	Register the designated slot and its Crypto Officer password/challenge to the KSP. You can specify the following options: > **/s** <slot_label> [Mandatory] The label of the partition being registered to the KSP. > **/u** <username> [Optional] The username to register for this partition. If this is not specified, the currently logged-in user is registered. > **/c** <co_password/challenge> [Optional] The Crypto Officer password/challenge. You require minimum Luna HSM Client 10.4.0 to specify this option. > **/d** <domain> [Optional] The domain to register for this partition.
usagelimit	**u**	Set the maximum usage limit for RSA keys using KSP. Enter **0** to register unlimited uses.
viewslots	**v**	Display the registered slots by user/domain.

Configuring the KSP Using the Command Line

You can use the **"kspcmd" on the previous page** command-line tool to configure the KSP for use with your partitions. The Crypto Officer must complete this procedure using Administrator privileges on the client.

You can register the following user/domain combinations with the KSP:

> **Administrator** user with the domain specific to the client. Default Windows domains are in the format **WIN-XXXXXXXXXXX**.

> **SYSTEM** user with the **NT-AUTHORITY** domain

The configuration tool registers a Crypto Officer password/challenge to a specific user, so that only that user can unlock the partition.

To configure the KSP using the command line

1. In a command line, navigate to the Luna KSP install directory and register the **cryptoki.dll** library to the KSP.

   "kspcmd" on the previous page library /s <path**cryptoki.dll**> [**/u** <username>] [**/d** <domain>]

2. Register the designated slot and its Crypto Officer password/challenge to the KSP.

   "kspcmd" on the previous page password /s <slot_label> [**/u** <username>] [**/c** <co_ password/challenge>] [**/d** <domain>]

   You are prompted to enter the CO password/challenge for the slot, unless you specified it using the **/c** option (minimum Luna HSM Client 10.4.0 required).

3. [Optional] Display the registered slots to ensure that registration is complete.

   "kspcmd" on the previous page viewslots

4. [Optional] Set the maximum usage limit for RSA keys using KSP.

   "kspcmd" on the previous page usagelimit

   You are prompted to enter a usage limit. Enter **0** to register unlimited uses.

5. [Optional] Enable non-administrator users on the client to use Luna KSP. This feature requires minimum Luna HSM Client 10.4.0.

   "kspcmd" on the previous page nonAdminUser

   You are prompted to confirm this action. When the action succeeds, the following entry is added to the Windows registry with a value of **1**:

   HKEY_LOCAL_MACHINE\SOFTWARE\Safenet\SafeNetKSP\CurrentConfig\NAUaccess

   To restrict non-admin users from Luna KSP in the future, set the value of this entry to **0**, or delete the key from the registry.

KspConfig

You can use this tool (<client_install_dir>**\KSP\KspConfig.exe**) to register the KSP library and partitions using a GUI.

> **NOTE** To register the library and partitions using the command line, use **"kspcmd" on the previous page**. It is unnecessary to use both utilities.

> **NOTE** CSP or KSP registration includes a step that verifies the DLLs are signed by our certificate that chains back to the DigiCert root of trust G4 (in compliance with industry security standards).
>
> This step can fail if your Windows operating system does not have the required certificate. If you have been keeping your Windows OS updated, you should already have that certificate.
>
> If your Luna HSM Client host is connected to the internet, use the following commands to update the certificate manually:
>
> **certutil -urlcache -f** http://cacerts.digicert.com/DigiCertTrustedRootG4.crt
>
> **certutil -addstore -f root** DigiCertTrustedRootG4.crt
>
> **To manually update a non-connected host**
>
> 1. Download the DigiCert Trusted Root G4 (http://cacerts.digicert.com/DigiCertTrustedRootG4.crt) to a separate internet-connected computer.
> 2. Transport the certificate, using your approved means, to the Luna HSM Client host into a <downloaded cert path> location of your choice
> 3. Add the certificate to the certificate store using the command:
>
> **certutil -addstore -f root** <downloaded cert path>

Configuring the KSP Using the GUI

You can use the **"KspConfig" on the previous page** utility to configure the KSP for use with your partitions. The Crypto Officer must complete this procedure using Administrator privileges on the client.

You can register the following user/domain combinations with the KSP:

> **Administrator** user with the domain specific to the client. Default Windows domains are in the format **WIN-XXXXXXXXXXX**.

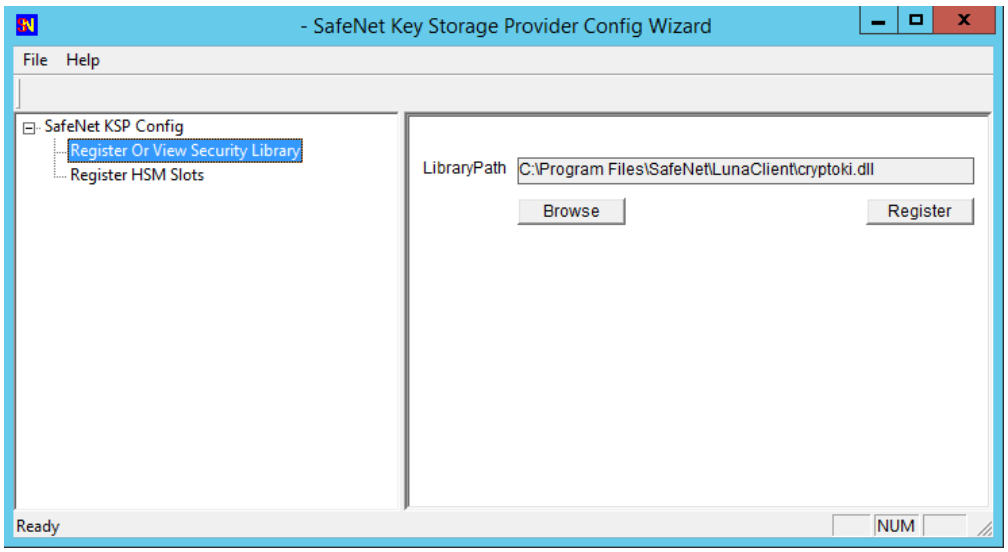
> **SYSTEM** user with the **NT-AUTHORITY** domain

The configuration tool registers a Crypto Officer password/challenge to a specific user, so that only that user can unlock the partition.

To configure the KSP using the GUI

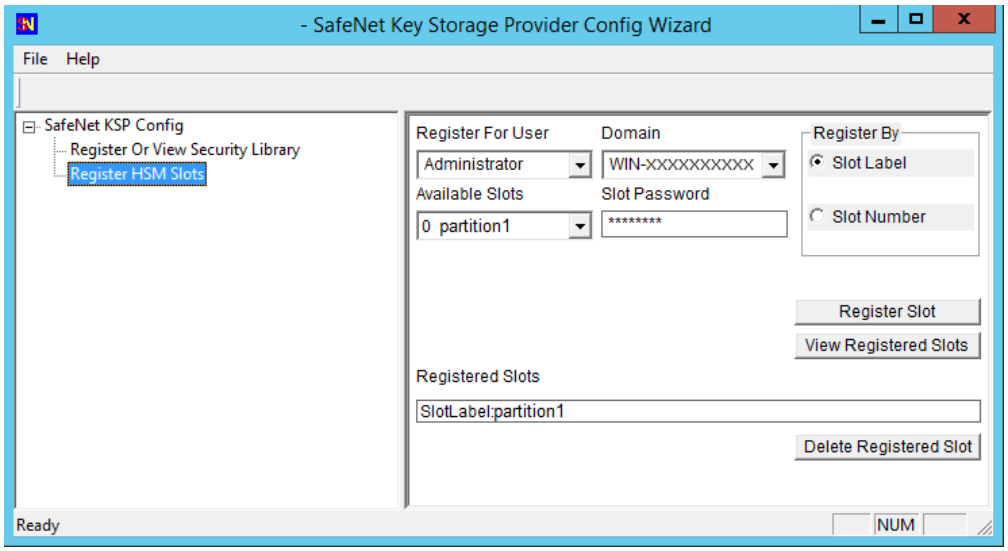
1. In Windows Explorer, navigate to the Luna KSP install directory and launch **"KspConfig" on the previous page** as the **Administrator** user.
2. In the left panel, double-click **Register or View Security Library**. Enter the filepath to **cryptoki.dll** or click Browse to locate it.

   <client_install_dir>**\cryptoki.dll**



Click **Register** to complete the registration.

3. In the left panel, double-click **Register HSM Slots**. Select the **Administrator** user, client domain, and an available slot to register. Enter the CO password/challenge and click **Register Slot**.



4. Select the **SYSTEM** user and **NT-AUTHORITY** domain and register for the slot.
5. Repeat steps 3-4 for any other available slots you want to register with the KSP.

   You can now begin using your applications to perform crypto operations on the registered slots.

ms2Luna

Use the **ms2Luna** utility (<client_install_dir>/**KSP**/**ms2Luna.exe**) to migrate existing Microsoft KSP keys held in software to a registered partition/HA group on the Luna PCIe HSM 7. It requires the thumbprint of a certificate held in the client's keystore.

Prerequisites

> You must already have registered a partition/HA group using the **"kspcmd" on page 684** or **"KspConfig" on page 685** utility.

> Private keys must be exportable to be migrated to the HSM.

To migrate Microsoft KSP keys to the Luna PCIe HSM 7

1. In a command prompt, navigate to the Luna KSP install directory and migrate your existing keys to the HSM.

   ms2Luna

   You are prompted for the KSP certificate thumbprint.

ksputil

KSP binds machine keys to the hostname of the crypto server that created the keys. You can use the **"ksputil" above** utility to display and manage keys that are visible to the KSP.

Syntax

ksputil

clusterkeys /s <slotnum> **/n** <keyname> **/t** <target>
listkeys /s <slotnum> [**/user**]

Argument	Shortcut	Description
clusterkeys	**c**	Bind a specified keypair to a different server domain. Note that this does not change the bindings of existing keys; it creates a copy of the original keypair that is bound to the new domain. Available options: **/s** <slotnum> [Mandatory] The slot number of the partition where the key(s) are located. **/n** <keyname> [Mandatory] The name of the key(s) to bind to the new domain. **/d** <domain> [Mandatory] The domain to which keys will be bound.
listkeys	**l**	DIsplay a list of KSP-visible keys. Available options: **/s** <slotnum> [Mandatory] The slot number of the partition where the key(s) are located. **/user** [Optional] List keys bound to the currently logged-in user/hostname.

Algorithms Supported

Here, for comparison, are the algorithms supported by our CSP and KSP APIs.

Algorithms supported by the Luna CSP

CALG_RSA_SIGN

CALG_RSA_KEYX

CALG_RC2

CALG_RC4

CALG_RC5

CALG_DES

CALG_3DES_112

CALG_3DES

CALG_MD2

CALG_MD5

CALG_SHA

CALG_SHA_256

CALG_SHA_384

CALG_SHA_512

CALG_MAC

CALG_HMAC

Algorithms supported by the Luna KSP

NCRYPT_RSA_ALGORITHM

NCRYPT_DSA_ALGORITHM

NCRYPT_ECDSA_P256_ALGORITHM

NCRYPT_ECDSA_P384_ALGORITHM

NCRYPT_ECDSA_P521_ALGORITHM

NCRYPT_ECDH_P256_ALGORITHM

NCRYPT_ECDH_P384_ALGORITHM

NCRYPT_ECDH_P521_ALGORITHM

NCRYPT_DH_ALGORITHM

NCRYPT_RSA_ALGORITHM

Run a Windows CNG application as Crypto Officer limited to key handling ability at Crypto User level

> **NOTE** KSP works with Crypto Officer only.

You might wish to implement the following scenario:

> create keys (requires the CO to have read/write capability on the partition, meaning that Partition Policy 28 - Allow Key Management Functions is set to ON)

> but also permit an application or service to make ongoing use of those keys, *without* an ability to change/create/delete them, (meaning that Partition Policy 28 - Allow Key Management Functions is set to OFF)

> from time to time, create new keys or delete old ones, etc. (requires the CO to have read/write capability on the partition, meaning that Partition Policy 28 - Allow Key Management Functions is set to ON again)

> then resume using your application, still as CO, *without* an ability to change/create/delete keys, (meaning that Partition Policy 28 - Allow Key Management Functions is set to OFF again)

> but Partition Policy 28 is destructive when set from OFF (0) to ON (1)

It is possible to implement such a scheme by initializing the partition using a Partition Policy Template file (see Setting Partition Policies Using a Template) which allows you to override the destructiveness setting, as follows:

Prerequisites

> The partition is created (see partition create for Luna PCIe HSM 7 or for Luna USB HSM 7.

To restrict Crypto Officer power for KSP-using application, while temporarily enabling full capability for partition management

1. Create a basic default policy template file.

   lunacm:> **partition showpolicies -slot** <slotnum> **-verbose -exporttemplate** <filename.policy>

2. In that file, edit policy 28,

   FROM its default value of

   28:"Allow Key Management Functions":1:**1**:0

   (This means that setting KeyManagement OFF to ON will zeroize the partition.)

   TO a new value of

   28:"Allow Key Management Functions":1:**0**:0

   (This removes destructiveness.)

   > **TIP** In the file, you can delete any of the other entries where you intend to use the default values.

3. Initialize a partition with the newly created template

   lunacm:> **partition init -label** <label string> **-password** <password> **-domain** <domainstring if PW-auth> **-applytemplate** <filename.policy> -force

   Complete the mandatory steps of creating new roles and changing the initial password.

4. Log in as Partition Security Officer.

   lunacm:> **role login -name** PO

5. Initialize the Crypto Officer role.

   lunacm:> **role init -name** CO

6. Log in as Crypto Officer.

   lunacm:> **role logout**

   lunacm:> **role login -name** CO

7. Partition policy 28 should currently be ON (from partition initialization with the template file). Generate keys by whatever means you normally employ.

8. Log in as Partition Security Officer.

   lunacm:> **role logout**

   lunacm:> **role login -name** PO

9. Switch policy 28 to OFF, to *disallow* key management.

   lunacm:> **partition changepolicy -policy** 28 -value 0

   In this state, the CO (or an application authenticated with CO credential) can now use any key that is currently in the partition, but cannot delete them or add new ones. This permits the CO, when logged in, to facilitate application access via KSP/CNG but with key-use capability at read-only Crypto User level.

10. Log in as Crypto Officer.

    lunacm:> **role logout**

    lunacm:> **role login -name** CO

    Run your KSP/CNG-using application that needs CO credentials (but not full CO read-write capability) to make use of existing keys.

11. Whenever it is necessary to manage keys in the partition (delete existing ones or create new ones, etc.), first shut down the application or service (for example code-signing) and then switch Partition Policy 28 to ON.

    lunacm:> **role logout**

    lunacm:> **role login -name** PO

    lunacm:> **partition changepolicy -policy** 28 **-value** 1

    lunacm:> **role logout**

    lunacm:> **role login -name** CO

    In this state, CO has full ability to add, delete, change objects in the partition, and the application is paused so that it does not make use of the expanded powers.

12. Generate a new keypair or perform other management function in the partition.

13. When key management action is done, switch Partition Policy 28 back to OFF.

    lunacm:> **role logout**

    lunacm:> **role login -name** PO

    lunacm:> **partition changepolicy -policy** 28 **-value** 0

    lunacm:> **role logout**

    lunacm:> **role login -name** CO

14. Resume operation of your application or service, via KSP/CNG, as the CO role, but with CU role limitations.

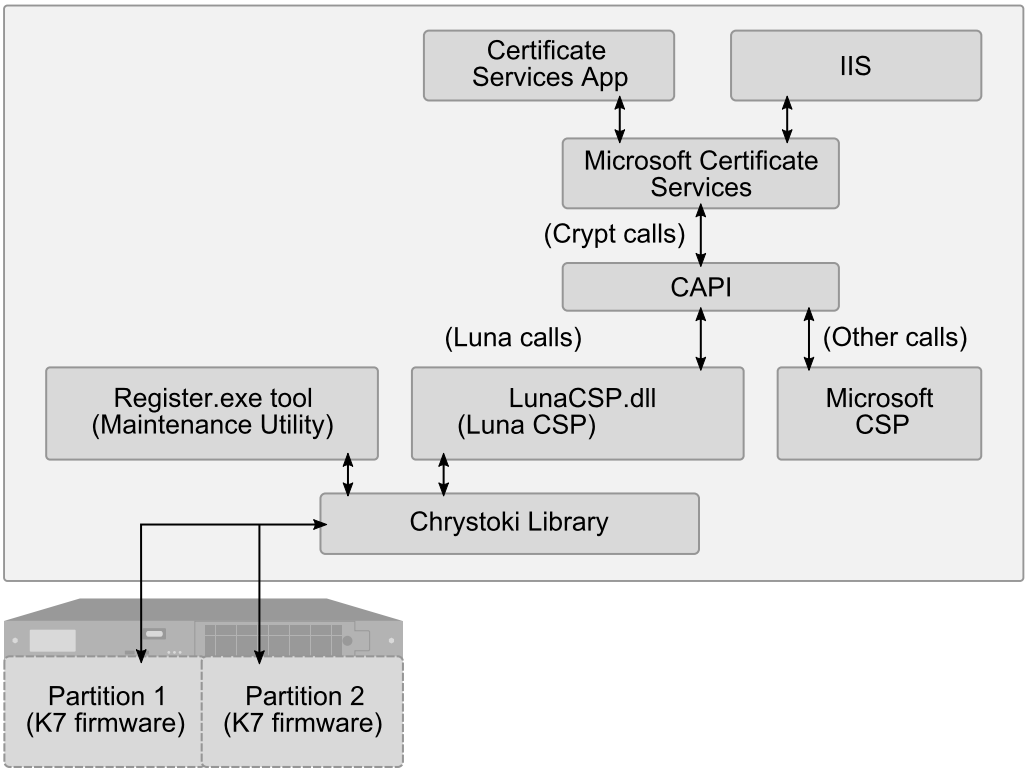
 Copyright 2001-2024 Thales Group

Luna CSP Calls and Functions

For integration with Microsoft Certificate Services and other applications, the LunaCSP.dll library accepts Crypt calls and gives access to token functions (via CP calls) as listed in this section. Key pairs and certificates are generated, stored and used on the Luna PCIe HSM 7.

The diagram below depicts the relationship of the Luna components to the other layers in the certificate system.

Figure 1: Luna CSP architecture



Note, in the diagram, that the Luna CSP routes relevant calls through the statically linked Crystoki library to the HSM via CP calls. Other calls from the application layer – those not directed at the token/HSM, and not matching the Luna CSP supported functions (see next section) – are passed to the Microsoft CSP.

Programming for Luna PCIe HSM 7 with Luna CSP

The Luna CSP DLL exports the following functions, each one corresponding to an equivalent (and similarly named) Crypt call from the application layer:

- CPAcquireContext
- CPGetProvParam
- CPSetProvParam
- CPReleaseContext
- CPDeriveKey
- CPDestroyKey
- CPDuplicateKey

- CPExportKey
- CPGenKey
- CPGenRandom
- CPGetKeyParam
- CPGetUserKey
- CPImportKey
- CPSetKeyParam
- CPDecrypt
- CPEncrypt
- CPCreateHash
- CPDestroyHash
- CPGetHashParam
- CPHashData
- CPHashSessionKey
- CPSetHashParam
- CPSignHash
- CPVerifySignature

> **NOTE** The CPVerifySignature function is able to verify signatures of up to 2048 bits, regardless of the size of the signatures produced by CPSignHash. This ensures that the CSP is able to validate all compatible certificates, even those signed with large keys.

The MSDN (Microsoft Developers Network) web site provides syntax and descriptions of the corresponding Crypt calls that invoke the functions in the above list.

Algorithms

Luna CSP supports the following algorithms:

- CALG_RSA_SIGN [RSA Signature] [256 - 4096 bits]. The CSP uses the RSA Public-Key Cipher for digital signatures.
- CALG_RSA_KEYX [RSA Key Exchange] [256- 4096 bits] The CSP must use the RSA Public-Key Cipher key exchange. The exchange key pair can be used both to exchange session keys and to verify digital signatures.
- CALG_RC2 [RSA Data Securities RC2 (block cipher)] [8 - 1024 bits].
- CALG_RC4 [RSA Data Securities RC4 (stream cipher)] [8 - 2048 bits].
- CALG_RC5 [RSA Data Securities RC5 (block cipher)] [8 - 2048 bits].
- CALG_DES [Data Encryption Standard (block cipher)] [56 bits].
- CALG_3DES_112 [Double DES (block cipher)] [112 bits].
- CALG_3DES [Triple DES (block cipher)] [168 bits].
- CALG_MAC [Message Authentication Code] (with RC2 only).

- CALG_HMAC [Hash-based MAC].
- CALG_MD2 [Message Digest 2 (MD2)] [128 bits].
- CALG_MD5 [Message Digest 5 (MD5)] [128 bits].
- CALG_SHA [Secure Hash Algorithm (SHA-1)] [160 bits].
- CALG_SHA224 [Secure Hash Algorithm (SHA-2)] [224 bits].
- CALG_SHA256 [Secure Hash Algorithm (SHA-2)] [256 bits].
- CALG_SHA384 [Secure Hash Algorithm (SHA-2)] [384 bits].
- CALG_SHA512 [Secure Hash Algorithm (SHA-2)] [512 bits].

NOTE If you intend to perform key exchanges between the Luna CSP and the Microsoft CSP with RC2 keys, the attribute KP_EFFECTIVE_KEYLEN must be set to 128 bits. For RC2 and RC4, the salt value of the keys must be transferred by making a call to get the salt value of the original key and to set the salt value of an imported key. This is done with the CryptGetKeyParam(KP_SALT) and CryptSetKeyParam(KP_SALT) functions respectively.

 Copyright 2001-2024 Thales Group