CAE Specification

Common Security: CDSA and CSSM

The Open Group

CAE Specification

Common Security: CDSA and CSSM

ISBN: 1-85912-194-2
Document Number: C707

Published in the U.K. by The Open Group, December 1997.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

Part 1 Common Data Security Architecture (CDSA) **1**

Chapter 1 Introduction **3**
1.1 The Threat Model........ 4
1.2 Common Data Security Architecture 5
1.2.1 Architectural Assumptions........ 5
1.2.2 Architectural Overview 6
1.2.3 Layered Security Services 7
1.2.4 Common Security Services Manager Layer 8
1.2.5 Security Add-In Modules Layer 10
1.2.5.1 Cryptographic Service Providers (CSPs) 10
1.2.5.2 Trust Policy Modules (TPs)........ 11
1.2.5.3 Certificate Library Modules (CLs)........ 11
1.2.5.4 Data Storage Library Modules (DLs)........ 11
1.2.5.5 Multi-Service Library Module........ 12
1.3 Interoperability Goals 12

Chapter 2 Common Security Services Manager **15**
2.1 Overview 15
2.2 General Module Management Services 16
2.3 Elective Module Managers 17
2.3.1 Transparent, Dynamic Attach 17
2.3.2 Registering Module Managers........ 18
2.3.3 State Sharing Among Module Managers........ 18
2.4 Basic Module Managers 19
2.5 Dispatching Application Calls for Security Services 19
2.6 Integrity Services 20
2.6.1 CSSM-Enforced Integrity Verification 21
2.7 Creating Checkable Components........ 22
2.7.1 Verifying Components........ 22
2.8 Security Context Services........ 23

Chapter 3 Cryptographic Service Provider Modules **25**
3.1 CSP Form Factor........ 26
3.2 Legacy CSPs 26
3.3 Cryptographic Service Provider Registration........ 27
3.4 Cryptographic Services API 27
3.5 Additional CSP Services........ 28

Chapter 4 Trust Policy Modules **31**
4.1 Trust Policy Services API 32

Chapter 5 Certificate Library Modules **35**
5.1 Certificate Library API 35

Chapter 6 Data Storage Library Modules **39**
6.1 Data Storage Library Registration 40
6.2 Data Storage Library API 40

Chapter 7 Multi-Service Modules **43**
7.1 Application Developer's View of a Multi-Service Add-in Module.. 44
7.2 Service Provider's View of a Multi-Service Add-in Module 45
7.3 Companion Modules 45

Chapter 8 System Security Services **47**

Part 2 Common Security Services Manager (CSSM) **49**

Chapter 9 Introduction **51**
9.1 Common Data Security Architecture 51

Chapter 10 Core Services API **55**
10.1 Overview 55
10.2 Core Services for CSSM Management 55
10.2.1 Module Management Services 56
10.2.2 Memory Management Support 58
10.2.3 Integrity of the CSSM Environment 58
10.2.4 Module-Defined Usage Policies 59
10.2.5 Application-Authenticated Add-In Modules 60
10.2.6 Application Exemptions 62
10.3 Data Structures for Core Services 63
10.3.1 CSSM_BOOL 63
10.3.2 CSSM_RETURN 63
10.3.3 CSSM_STRING 63
10.3.4 CSSM_DATA 63
10.3.5 CSSM_GUID 64
10.3.6 CSSM_VERSION 64
10.3.7 CSSM_SUBSERVICE_UID 65
10.3.8 CSSM_HANDLE 65
10.3.9 CSSM_MODULE_HANDLE 65
10.3.10 CSSM_LIST_ITEM 65
10.3.11 CSSM_LIST 66
10.3.12 CSSM_CSSMINFO 66
10.3.13 CSSM_EVENT_TYPE 67
10.3.14 CSSM_SERVICE_MASK 67
10.3.15 CSSM_SERVICE_TYPE 67
10.3.16 CSSM_SERVICE_FLAGS 67

	10.3.17	CSSM_SERVICE_INFO	68
	10.3.18	CSSM_MODULE_FLAGS	69
	10.3.19	CSSM_MODULE_INFO	69
	10.3.20	CSSM_ALL_SUBSERVICES	70
	10.3.21	CSSM_INFO_LEVEL	71
	10.3.22	CSSM_NET_ADDRESS_TYPE	71
	10.3.23	CSSM_NET_ADDRESS	71
	10.3.24	CSSM_NET_PROTOCOL	71
	10.3.25	CSSM_APP_SERVICE_FLAGS	72
	10.3.26	CSSM_APP_KEYS	72
	10.3.27	CSSM_APP_SERVICE_INFO	73
	10.3.28	CSSM_EXEMPTION_MASK	74
	10.3.29	CSSM_USER_AUTHENTICATION_MECHANISM	74
	10.3.30	CSSM_CALLBACK	74
	10.3.31	CSSM_CRYPTO_DATA	74
	10.3.32	CSSM_USER_AUTHENTICATION	75
	10.3.33	CSSM_NOTIFY_CALLBACK	75
	10.3.34	CSSM_MEMORY_FUNCS and CSSM_API_MEMORY_FUNCS	76
	10.4	Core Functions	77
		CSSM_Init	78
		CSSM_GetInfo	79
		CSSM_FreeInfo	80
		CSSM_Load	81
		CSSM_RequestCssmExemption	82
		CSSM_VerifyComponents	84
	10.5	Module Management Functions	85
		CSSM_ModuleInstall	86
		CSSM_ModuleUninstall	88
		CSSM_ModuleAttach	89
		CSSM_ModuleDetach	91
		CSSM_ListModules	92
		CSSM_GetModuleInfo	93
		CSSM_SetModuleInfo	95
		CSSM_FreeModuleInfo	96
		CSSM_GetGUIDUsage	97
		CSSM_GetHandleUsage	98
		CSSM_GetModuleGUIDFromHandle	99
		CSSM_GetSubserviceUIDFromHandle	100
	10.6	Utility Functions	101
		CSSM_FreeList	102
		CSSM_Free	103
		CSSM_GetAPIMemoryFunctions	104
Chapter	**11**	**Cryptographic Services API**	**105**
	11.1	Overview	105
	11.1.1	Key Formats for Public Key-Based Algorithms	106
	11.2	Data Structures	107
	11.2.1	CSSM_CC_HANDLE	107

11.2.2 CSSM_CSP_HANDLE 107
11.2.3 CSSM_DATE 107
11.2.4 CSSM_RANGE 107
11.2.5 CSSM_QUERY_SIZE_DATA 107
11.2.6 CSSM_HEADERVERSION 108
11.2.7 CSSM_KEY_SIZE 108
11.2.8 CSSM_KEYHEADER 108
11.2.9 CSSM_KEY 113
11.2.10 CSSM_WRAP_KEY 114
11.2.11 CSSM_CSP_TYPE 114
11.2.12 CSSM_CSP_SESSION_TYPE 114
11.2.13 CSSM_PADDING 114
11.2.14 CSSM_CONTEXT_ATTRIBUTE 114
11.2.15 CSSM_CONTEXT 117
11.2.16 CSSM_CSP_CAPABILITY 122
11.2.17 CSSM_SOFTWARE_CSPSUBSERVICE_INFO 122
11.2.18 CSSM_HARDWARE_CSPSUBSERVICE_INFO 122
11.2.19 CSSM_HYBRID_CSPSUBSERVICE_INFO 125
11.2.20 CSSM_CSP_WRAPPEDPRODUCTINFO 125
11.2.21 CSSM_CSP_FLAGS 126
11.2.22 CSSM_CSPSUBSERVICE 126
11.3 Cryptographic Context Operations 128
CSSM_CSP_CreateSignatureContext 129
CSSM_CSP_CreateSymmetricContext 130
CSSM_CSP_CreateDigestContext 132
CSSM_CSP_CreateMacContext 133
CSSM_CSP_CreateRandomGenContext 134
CSSM_CSP_CreateAsymmetricContext 135
CSSM_CSP_CreateDeriveKeyContext 137
CSSM_CSP_CreateKeyGenContext 139
CSSM_CSP_CreatePassThroughContext 141
CSSM_GetContext 142
CSSM_FreeContext 143
CSSM_SetContext 144
CSSM_DeleteContext 145
CSSM_GetContextAttribute 146
CSSM_UpdateContextAttributes 147
CSSM_DeleteContextAttributes 148
11.4 Cryptographic Sessions and Logon 149
CSSM_CSP_Login 150
CSSM_CSP_Logout 151
CSSM_CSP_ChangeLoginPassword 152
11.5 Cryptographic Operations 153
CSSM_SignData 154
CSSM_SignDataInit 157
CSSM_SignDataUpdate 159
CSSM_SignDataFinal 161
CSSM_VerifyData 163

CSSM_VerifyDataInit 165
CSSM_VerifyDataUpdate 167
CSSM_VerifyDataFinal 168
CSSM_DigestData 169
CSSM_DigestDataInit 171
CSSM_DigestDataUpdate 172
CSSM_DigestDataClone 174
CSSM_DigestDataFinal 175
CSSM_GenerateMac 177
CSSM_GenerateMacInit 179
CSSM_GenerateMacUpdate 181
CSSM_GenerateMacFinal 182
CSSM_VerifyMac 184
CSSM_VerifyMacInit 186
CSSM_VerifyMacUpdate 188
CSSM_VerifyMacFinal 189
CSSM_QuerySize 190
CSSM_EncryptData 192
CSSM_EncryptDataInit 195
CSSM_EncryptDataUpdate 197
CSSM_EncryptDataFinal 199
CSSM_DecryptData 201
CSSM_DecryptDataInit 204
CSSM_DecryptDataUpdate 206
CSSM_DecryptDataFinal 208
CSSM_QueryKeySizeInBits 210
CSSM_GenerateKey 212
CSSM_GenerateKeyPair 215
CSSM_GenerateRandom 218
CSSM_ObtainPrivateKeyFromPublicKey 220
CSSM_WrapKey 221
CSSM_UnwrapKey 224
CSSM_DeriveKey 227
CSSM_FreeKey 230
CSSM_GenerateAlgorithmParams 231
11.6 Miscellaneous Functions 233
CSSM_RetrieveUniqueId 234
CSSM_RetrieveCounter 235
CSSM_VerifyDevice 236
11.7 Extensibility Functions 237
CSSM_CSP_PassThrough 238

Chapter 12 Trust Policy Services API 239
12.1 Overview 239
12.2 Data Structures 240
12.2.1 CSSM_TP_HANDLE 240
12.2.2 CSSM_TP_ACTION 240
12.2.3 CSSM_REVOKE_REASON 240

12.2.4 CSSM_TP_STOP_ON ... 241
12.2.5 CSSM_CERTGROUP ... 241
12.2.6 CSSM_EVIDENCE_FORM ... 242
12.2.7 CSSM_VERIFYCONTEXT ... 242
12.2.8 CSSM_TP_WRAPPEDPRODUCTINFO ... 244
12.2.9 CSSM_TPSUBSERVICE ... 245
12.3 Trust Policy Operations ... 247
CSSM_TP_CertRequest ... 248
CSSM_TP_CertRetrieve ... 251
CSSM_TP_CertGroupVerify ... 253
CSSM_TP_CertSign ... 256
CSSM_TP_CertRevoke ... 258
CSSM_TP_CrlVerify ... 260
CSSM_TP_CrlSign ... 262
CSSM_TP_ApplyCrlToDb ... 264
12.4 Group Functions ... 266
CSSM_TP_CertGroupConstruct ... 267
CSSM_TP_CertGroupPrune ... 269
12.5 Extensibility Functions ... 271
CSSM_TP_PassThrough ... 272

Chapter 13 Certificate Library Services API ... 275
13.1 Overview ... 275
13.1.1 Certificate Life Cycle ... 275
13.1.2 Application and Certificate Library Interaction ... 277
13.1.3 Operations on Certificates ... 277
13.2 Data Structures ... 278
13.2.1 CSSM_CL_HANDLE ... 278
13.2.2 CSSM_CERT_TYPE ... 278
13.2.3 CSSM_CERT_ENCODING ... 279
13.2.4 CSSM_CERT_BUNDLE_TYPE ... 279
13.2.5 CSSM_CERT_BUNDLE_ENCODING ... 279
13.2.6 CSSM_CERT_BUNDLE_HEADER ... 279
13.2.7 CSSM_CERT_BUNDLE ... 280
13.2.8 CSSM_OID ... 280
13.2.9 CSSM_CRL_TYPE ... 281
13.2.10 CSSM_CRL_ENCODING ... 281
13.2.11 CSSM_FIELD ... 281
13.2.12 CSSM_ESTIMATED_TIME_UNKNOWN ... 281
13.2.13 CSSM_CA_SERVICES ... 282
13.2.14 CSSM_CL_CA_CERT_CLASSINFO ... 282
13.2.15 CSSM_CL_CA_PRODUCTINFO ... 282
13.2.16 CSSM_CL_ENCODER_PRODUCTINFO ... 284
13.2.17 CSSM_CL_WRAPPEDPRODUCTINFO ... 285
13.2.18 CSSM_CLSUBSERVICE ... 285
13.3 Certificate Operations ... 287
CSSM_CL_CertRequest ... 288
CSSM_CL_CertRetrieve ... 291

CSSM_CL_RegistrationFormRequest ... 293
CSSM_CL_RegistrationFormSubmit ... 294
CSSM_CL_CertMultiSignRequest ... 296
CSSM_CL_CertMultiSignRetrieve ... 299
CSSM_CL_CertRecoveryRequest ... 301
CSSM_CL_CertRecoveryRetrieve ... 304
CSSM_CL_CertRecover ... 306
CSSM_CL_CertKeyRecover ... 307
CSSM_CL_CertAbortRecovery ... 309
CSSM_CL_CertVerify ... 310
CSSM_CL_CertGetFirstFieldValue ... 312
CSSM_CL_CertGetNextFieldValue ... 314
CSSM_CL_CertAbortQuery ... 315
CSSM_CL_CertGetKeyInfo ... 316
CSSM_CL_CertGetAllFields ... 317
CSSM_CL_CertGroupToSignedBundle ... 318
CSSM_CL_CertGroupFromVerifiedBundle ... 320
CSSM_CL_CertImport ... 322
CSSM_CL_CertExport ... 324
CSSM_CL_CertDescribeFormat ... 325
13.4 Certificate Revocation List Operations ... 326
CSSM_CL_CrlCreateTemplate ... 327
CSSM_CL_CrlSetFields ... 328
CSSM_CL_CrlRequest ... 329
CSSM_CL_CrlRetrieve ... 332
CSSM_CL_CrlAddCert ... 334
CSSM_CL_CrlRemoveCert ... 336
CSSM_CL_CrlSign ... 337
CSSM_CL_CrlVerify ... 339
CSSM_CL_IsCertInCrl ... 341
CSSM_CL_CrlGetFirstFieldValue ... 342
CSSM_CL_CrlGetNextFieldValue ... 344
CSSM_CL_CrlAbortQuery ... 345
CSSM_CL_CrlDescribeFormat ... 346
13.5 Extensibility Functions ... 347
CSSM_CL_PassThrough ... 348

Chapter 14 Data Storage Library Services API ... 349
14.1 Overview ... 349
14.2 Data Storage Data Structures ... 350
14.2.1 CSSM_DL_HANDLE ... 350
14.2.2 CSSM_DB_HANDLE ... 350
14.2.3 CSSM_DL_DB_HANDLE ... 350
14.2.4 CSSM_DL_DB_LIST ... 350
14.2.5 CSSM_DB_ATTRIBUTE_NAME_FORMAT ... 351
14.2.6 CSSM_DB_ATTRIBUTE_INFO ... 351
14.2.7 CSSM_DB_ATTRIBUTE_DATA ... 351
14.2.8 CSSM_DB_RECORDTYPE ... 352

14.2.9 CSSM_DB_CERTRECORD_SEMANTICS 353
14.2.10 CSSM_DB_RECORD_ATTRIBUTE_INFO 354
14.2.11 CSSM_DB_RECORD_ATTRIBUTE_DATA 354
14.2.12 CSSM_DB_RECORD_PARSING_FNTABLE 355
14.2.13 CSSM_DB_PARSING_MODULE_INFO 356
14.2.14 CSSM_DB_INDEX_TYPE 356
14.2.15 CSSM_DB_INDEXED_DATA_LOCATION 356
14.2.16 CSSM_DB_INDEX_INFO 357
14.2.17 CSSM_DB_UNIQUE_RECORD 357
14.2.18 CSSM_DB_RECORD_INDEX_INFO 357
14.2.19 CSSM_DB_ACCESS_TYPE 358
14.2.20 CSSM_DBINFO 358
14.2.21 CSSM_DB_OPERATOR 359
14.2.22 CSSM_DB_CONJUNCTIVE 360
14.2.23 CSSM_SELECTION_PREDICATE 360
14.2.24 CSSM_QUERY_LIMITS 360
14.2.25 CSSM_QUERY_FLAGS 361
14.2.26 CSSM_QUERY 361
14.2.27 CSSM_DLTYPE 362
14.2.28 CSSM_DL_PKCS11_ATTRIBUTES 362
14.2.29 CSSM_DB_DATASTORES_UNKNOWN 362
14.2.30 CSSM_DL_WRAPPEDPRODUCT_INFO 363
14.2.31 CSSM_NAME_LIST 364
14.2.32 CSSM_DLSUBSERVICE 364
14.3 Data Storage Functions 367
CSSM_DL_DbOpen 368
CSSM_DL_DbClose 370
CSSM_DL_DbCreate 371
CSSM_DL_DbDelete 373
CSSM_DL_DbImport 375
CSSM_DL_DbExport 378
CSSM_DL_Authenticate 380
CSSM_DL_DbSetRecordParsingFunctions 381
CSSM_DL_DbGetRecordParsingFunctions 382
CSSM_DL_GetDbNames 383
CSSM_DL_GetDbNameFromHandle 384
CSSM_DL_FreeNameList 385
14.4 Data Record Operations 386
CSSM_DL_DataInsert 387
CSSM_DL_DataDelete 389
CSSM_DL_DataModify 390
CSSM_DL_DataGetFirst 392
CSSM_DL_DataGetNext 394
CSSM_DL_DataAbortQuery 396
CSSM_DL_DataGetFromUniqueRecordId 397
CSSM_DL_FreeUniqueRecord 398
14.5 Extensibility Functions 399
CSSM_DL_PassThrough 400

Appendix A CSSM Error-Handling **401**
A.1 Introduction 401
A.2 Data Structures 402
A.3 Error Handling Functions 403
CSSM_GetError 404
CSSM_SetError 405
CSSM_ClearError 406
CSSM_InitError 407
CSSM_DestroyError 408
CSSM_IsCSSMError 409
CSSM_IsCLError 410
CSSM_IsDLError 411
CSSM_IsTPError 412
CSSM_IsCSPError 413
CSSM_CompareGuids 414

Appendix B Application Memory Functions **415**
B.1 Introduction 415
B.1.1 CSSM_API_MEMORY_FUNCS Data Structure 415

Part 3 CSSM Key Recovery API **417**

Chapter 15 Overview **419**
15.1 Key Recovery Nomenclature 419
15.1.1 Key Recovery Types 419
15.1.2 Key Recovery Phases 421
15.1.3 Lifetime of Key Recovery Fields 422
15.1.4 Key Recovery Policy 422
15.1.5 Operational Scenarios for Key Recovery 422
15.2 Key Recovery in the Common Data Security Architecture 424

Chapter 16 Key Recovery Enablement in CSSM **425**
16.1 Functionality Definition 425
16.2 Extensions to the Cryptographic Module Manager 426
16.3 Key Recovery Module Manager 427
16.3.1 Operational Scenarios 427
16.3.2 Key Recovery Profiles 428
16.3.3 Key Recovery Context 429
16.3.4 Key Recovery Policy 429
16.3.5 Key Recovery Enablement Operations 430
16.3.6 Key Recovery Registration and Request Operations 430

Chapter 17 Key Recovery APIs **431**
17.1 Module Management Operations 431
17.2 Key Recovery Context Operations 431
17.3 Key Recovery Registration Operations 432
17.4 Key Recovery Enablement Operations 432
17.5 Key Recovery Request Operations 432

17.6 Extensibility Functions 433
17.7 An Example Application Using Key Recovery APIs 433
17.8 Data Structures 435
17.8.1 CSSM_KR_HANDLE 435
17.8.2 CSSM_KR_NAME 435
17.8.3 CSSM_KR_PROFILE 436
17.8.4 CSSM_EXEMPTION_MASK 437
17.8.5 CSSM_CERT_LIST 437
17.8.6 CSSM_CONTEXT_ATTRIBUTE Extensions 438
17.8.7 CSSM_ATTRIBUTE_TYPE Additions 438
17.8.8 CSSM_KRSUBSERVICE 438
17.8.9 CSSM_KRINFO 439
17.9 Key Recovery Module Management Operations 440
CSSM_KR_SetEnterpriseRecoveryPolicy 441
17.10 Key Recovery Context Operations 442
CSSM_KR_CreateRecoveryRegistrationContext 443
CSSM_KR_CreateRecoveryEnablementContext 444
CSSM_KR_CreateRecoveryRequestContext 445
CSSM_KRPolicyInfo 446
17.11 Key Recovery Registration Operations 447
CSSM_KR_RegistrationRequest 448
CSSM_KR_RegistrationRetrieve 450
17.12 Key Recovery Enablement Operations 451
CSSM_KR_GenerateRecoveryFields 452
CSSM_KR_ProcessRecoveryFields 454
17.13 Key Recovery Request Operations 455
CSSM_KR_RecoveryRequest 456
CSSM_KR_RecoveryRetrieve 458
CSSM_KR_GetRecoveredObject 460
CSSM_KR_RecoveryRequestAbort 462
17.14 Extensibility Functions 463
CSSM_KR_PassThrough 464

Part 4 CDSA Embedded Integrity Services Library API 467

Chapter 18 Introduction 469
18.1 Problem Statement 469
18.2 Extending Trust 469
18.3 Why an Embedded Library? 470
18.4 A Phased Approach 470
18.4.1 Phase I. Establishing a Foothold: Self-Check 470
18.4.2 Phase II. Finding our Friends: Bilateral Authentication 471
18.4.3 Phase III. Secure Linkage Check 471
18.5 Using Library Services 471
18.5.1 Location of Modules and Credentials 472
18.5.2 Verification of Modules and their Credentials 472
18.5.3 Secure Linkage 472
18.5.4 Integrity Credentials 472

18.6 EISL Uses Other Standards or Specifications........ 473

Chapter 19 Data Structures........ 475
19.1 Object Pointers........ 475
19.1.1 Iterator Objects........ 475
19.1.2 Verified Signature Root Object........ 475
19.1.3 Verified Certificate Chain Object........ 476
19.1.4 Verified Certificate Object........ 476
19.1.5 Manifest Section Object........ 476
19.1.6 Verified Module Object........ 476
19.1.7 EISL Object Relationships and Life Cycle........ 476
19.2 Low-Level Data Structures Used in API Functions........ 478
19.2.1 ISL_DATA........ 478
19.2.2 ISL_CONST_DATA........ 478

Chapter 20 EISL Functions........ 479
20.1 Locator Services........ 479
ISL_FindRegistryAttribute........ 480
20.2 Credential and Attribute Verification Services........ 481
ISL_SelfCheck........ 482
ISL_VerifyAndLoadModuleAndCredentials........ 483
ISL_VerifyLoadedModuleAndCredentials........ 485
ISL_GetCertficateChain........ 486
ISL_ContinueVerification........ 487
ISL_RecycleVerifiedModuleCredentials........ 488
20.3 Signature Root Methods........ 489
ISL_CreateVerifiedSignatureRoot........ 490
ISL_CreateVerifiedSignatureRootWithCertificate........ 491
ISL_FindManifestSection........ 492
ISL_CreateManifestSectionEnumerator........ 493
ISL_GetNextManifestSection........ 494
ISL_RecycleManifestSectionEnumerator........ 495
ISL_FindSignatureAttribute........ 496
ISL_CreateSignatureAttributeEnumerator........ 497
ISL_GetNextSignatureAttribute........ 498
ISL_RecycleSignatureAttributeEnumerator........ 499
ISL_RecycleVerifiedSignatureRoot........ 500
20.4 Certificate Chain Methods........ 501
ISL_CreateCertificateChain........ 502
ISL_CopyCertificateChain........ 503
ISL_RecycleCertificateChain........ 504
20.5 Certificate Attribute Methods........ 505
ISL_FindCertificateAttribute........ 506
ISL_CreateCertificateAttributeEnumerator........ 507
ISL_GetNextCertificateAttribute........ 508
ISL_RecycleCertificateAttributeEnumerator........ 509
20.6 Manifest Section Object Methods........ 510
ISL_GetManifestSignatureRoot........ 511

ISL_VerifyAndLoadModule 512
ISL_VerifyLoadedModule 513
ISL_VerifyData 514
ISL_FindManifestSectionAttribute 515
ISL_CreateManifestSectionAttributeEnumerator 516
ISL_GetNextManifestSectionAttribute 517
ISL_RecycleManifestSectionAttributeEnumerator 518
ISL_GetModuleManifestSection 519
20.7 Secure Linkage Services 520
ISL_LocateProcedureAddress 521
ISL_GetReturnAddress 522
ISL_CheckAddressWithinModule 523
ISL_GetLibHandle 524

Part 5 CDSA Signed Manifest 525

Chapter 21 Introduction 527
21.1 Signed Manifests—An Overview 527
21.2 Overview of the Common Data Security Architecture 527

Chapter 22 Signed Manifests—Requirements 529
22.1 Requirements 529

Chapter 23 Signed Manifests—The Architecture 531

Chapter 24 Format Specification 535
24.1 The Manifest 535
24.1.1 Manifest Header Specification 535
24.1.2 Manifest Sections 535
24.1.3 Format Specification 536
24.1.4 MAGIC—A Flagging Mechanism 537
24.1.5 Metadata 538
24.1.6 Ordering Metadata Values 538
24.1.7 Manifest Examples 538
24.2 Signer's Information 539
24.2.1 Signing Information Header 539
24.2.2 Signer's Information Sections 539
24.2.3 Signing Information Examples 540
24.3 Signature Blocks 540

Chapter 25 Signed Manifests—Verifying Signatures 541
25.1 Verifying the Manifest 541
25.2 Verifying Referents in the Manifest 541

Chapter 26 File-Based Representation of Signed Manifests 543
26.1 The META-INF Directory—First File-Based Signed Manifest Representation 543
26.2 The ESW File—Archive-Based Signed Manifest Representation 543

	26.3	Representation Constraints	544
Chapter	**27**	**Signed Manifests—Examples**	**547**
	27.1	Static Referent Objects	547
	27.2	Dynamic Referent Objects with Verified Source	548
	27.2.1	Stock Quote Service	548
	27.3	Embedded or Nested Referent Objects	549
	27.3.1	Signed Objects Whose Signatures Serve to Carry the Object	549
	27.3.2	Signed Objects Whose Signature Blocks are Embedded	549
	27.3.3	Nested Manifests	550
	27.3.4	Signed Portion of an HTML Page	553
	27.3.5	Foreign Language Support / Multiple Hash Values for a Referent	554
	27.3.6	Dynamic Sources with no Associated Data	554
	27.3.7	Resources that Transform Locations	554
Appendix	**C**	**Signed Manifests**	**557**
	C.1	Extensions to the JavaSoft / Netscape Specification	557
	C.2	Core Set of Name:Value Pairs	557
	C.3	Metadata	558
	C.3.1	Integrity Core	558
	C.3.2	Dublin Core	559
	C.3.3	PKWARE Archive File Format Specification	559
Part	**6**	**CSSM Elective Module Manager**	**561**
Chapter	**28**	**Introduction**	**563**
Chapter	**29**	**Overview of Elective Module Managers**	**565**
	29.1	Built-In Policies and Application Exemptions	565
	29.2	Transparent, Dynamic Attach	566
	29.3	Registering Module Managers	567
	29.4	State Sharing Among Module Managers	567
Chapter	**30**	**Administration of Elective Module Managers**	**569**
	30.1	Integrity Verification	569
	30.2	Module Manager Credentials	569
	30.3	Installing an Elective Module Manager	571
	30.3.1	Global Unique Identifiers (GUIDs)	572
	30.4	Loading an Elective Module Manager	572
	30.4.1	Elective Module Manager Entry Point	573
	30.4.2	Bilateral Authentication	573
	30.4.3	Module Manager Function Table Registration	574
	30.5	Error Handling	575
Chapter	**31**	**Elective Module Manager Operations**	**577**
	31.1	Data Structures	577
	31.1.1	CSSM_BOOL	577
	31.1.2	CSSM_RETURN	577

31.1.3 CSSM_DATA 577
31.1.4 CSSM_GUID 578
31.1.5 CSSM_MODULE_HANDLE 578
31.1.6 CSSM_SERVICE_MASK 578
31.1.7 CSSM_EXEMPTION_MASK 579
31.1.8 CSSM_MODULE_MANAGER_INFO 579
31.1.9 CSSM_MEMORY_FUNCS 580
31.1.10 CSSM_MODULE_FUNCS 581
31.1.11 CSSM_MANAGER_EVENT_TYPES 581
31.1.12 CSSM_MANGER_EVENT_NOTIFICATION 581
31.1.13 CSSM_MANAGER_REGISTRATION_INFO 582
31.2 Elective Module Manager Functions 584
Initialize 585
Terminate 586
ModuleManagerAuthenticate 587
RegisterDispatchTable 588
DeregisterDispatchTable 589
EventNotifyManager 590

Chapter 32 Managing Elective Module Managers 591
32.1 Installation Functions 591
CSSM_ModuleManagerInstall 592
CSSM_ModuleManagerUninstall 594
32.2 Information Functions 595
CSSM_GetModuleManagerInfo 596
CSSM_ListAttachedModuleManagers 597
32.3 Registration Functions 598
CSSM_RegisterManagerServices 599
CSSM_DeregisterManagerServices 600
32.4 Notification Functions 601
CSSM_DeliverModuleManagerEvent 602

Part 7 CSSM Add-In Module Structure and Administration 605

Chapter 33 Introduction 607
33.1 Common Data Security Architecture 607
33.2 Add-In Module Structure 609
33.3 Add-In Module Usage 610
33.3.1 Application Interaction 610
33.3.2 CSSM Interaction 611
33.3.3 Module to Module Interaction 611

Chapter 34 Add-In Module Structure 613
34.1 Security Services 613
34.2 Module Administration Components 614
34.2.1 Integrity Verification 614
34.2.2 Module-Defined Usage Policies 615
34.2.3 Initialization and Cleanup 616

Chapter 35 Add-In Module Administration.. 617
35.1 Manufacturing an Add-In Module.. 617
35.1.1 Authenticating to Multiple CSSM Vendors....................................... 620
35.1.2 Obtaining an Add-In Module Manufacturing Certificate............... 620
35.1.3 Issuing an Add-In Module Product Certificate 621
35.1.4 Manufacturing Add-In Modules 621
35.2 Installing an Add-In Module.. 622
35.2.1 Global Unique Identifiers (GUIDs) 622
35.2.2 The Module Description.. 623
35.3 Attaching an Add-In Module 623
35.3.1 Module Entry Point 623
35.3.2 Bilateral Authentication 624
35.3.3 Module Function Table Registration 624
35.3.4 Memory Management Upcalls... 625
35.4 Error Handling 625
35.5 Install Example... 626
35.5.1 CL Module Install 626
35.6 Attach / Detach and AddInAuthenticate Example.............................. 627
35.6.1 DLLMain.. 627

Chapter 36 Add-In Module Interface Functions.. 631
Initialize 632
Terminate 633
EventNotify 634
AddInAuthenticate 636

Appendix D Relevant CSSM API Functions... 637
D.1 Overview 637
D.2 Data Structures 637
D.2.1 CSSM_BOOL... 637
D.2.2 CSSM_RETURN.. 637
D.2.3 CSSM_STRING 638
D.2.4 CSSM_DATA.. 638
D.2.5 CSSM_GUID 638
D.2.6 CSSM_VERSION 639
D.2.7 CSSM_SUBSERVICE_UID.. 639
D.2.8 CSSM_HANDLE... 640
D.2.9 CSSM_MODULE_HANDLE... 640
D.2.10 CSSM_EVENT_TYPE.. 640
D.2.11 CSSM_SERVICE_MASK 640
D.2.12 CSSM_SERVICE_TYPE 641
D.2.13 CSSM_SERVICE_FLAGS 641
D.2.14 CSSM_SERVICE_INFO 641
D.2.15 CSSM_MODULE_FLAGS.. 642
D.2.16 CSSM_MODULE_INFO.. 642
D.2.17 CSSM_ALL_SUBSERVICES 644
D.2.18 CSSM_INFO_LEVEL... 644
D.2.19 CSSM_NET_ADDRESS_TYPE... 644

D.2.20 CSSM_NET_ADDRESS 645
D.2.21 CSSM_NET_PROTOCOL 645
D.2.22 CSSM_USER_AUTHENTICATION_MECHANISM 646
D.2.23 CSSM_CALLBACK 646
D.2.24 CSSM_CRYPTO_DATA 646
D.2.25 CSSM_USER_AUTHENTICATION 647
D.2.26 CSSM_NOTIFY_CALLBACK 647
D.2.27 CSSM_MEMORY_FUNCS / CSSM_API_MEMORY_FUNCS 648
D.2.28 CSSM_SPI_MEMORY_FUNCS 649
D.2.29 CSSM_MODULE_FUNCS 649
D.2.30 CSSM_HANDLEINFO 650
D.2.31 CSSM_REGISTRATION_INFO 651
D.3 Function Definitions 653
CSSM_ModuleInstall 654
CSSM_ModuleUninstall 656
CSSM_ModuleAttach 657
CSSM_ModuleDetach 659
CSSM_GetModuleInfo 660
CSSM_SetModuleInfo 662
CSSM_FreeModuleInfo 663
CSSM_RegisterServices 664
CSSM_DeregisterServices 665
CSSM_GetHandleInfo 666
CSSM_GetError 667
CSSM_SetError 668
CSSM_ClearError 669

Part 8 CDSA Mechanisms for Policy Compliance 671

Chapter 37 Introduction 673
37.1 Overview of CDSA 674

Chapter 38 Goals and General Approach 677
38.1 Goals 677
38.2 Requirements 677
38.3 Specifying a System-Wide Policy 678
38.4 Assumptions and Architectural Approach 678

Chapter 39 CSSM Integrity Services—The Foundation 681
39.1 A Module's Certificate Chain 681
39.2 Checking a Module's Credentials 682

Chapter 40 Defining the Local, System-Wide Policy 683

Chapter 41 Screening Requests Based on Simple Policies 685
41.1 Simple Policies 685
41.2 CSSM Mechanisms Supporting Simple Policies 686

Chapter 42 Screening Requests Based on Complex Policies ... 687
42.1 Complex Policies ... 687
42.2 Evaluation of a Sequence of Events ... 687
42.3 Services that Establish Pre-Conditions ... 687

Part 9 CSSM Cryptographic Service Provider Interface ... 689

Chapter 43 Introduction ... 691
43.1 CDSA Add-In Module Overview ... 691
43.2 Cryptographic Service Provider Overview ... 692

Chapter 44 Service Provider Interface ... 695
44.1 Overview ... 695
44.1.1 Cryptographic Operations ... 696
44.1.2 Cryptographic Sessions and Logon ... 698
44.1.3 Extensibility Functions ... 698
44.1.4 Key Formats for Public Key-Based Algorithms ... 698
44.2 Data Structures ... 699
44.2.1 CSSM_CSP_HANDLE ... 699
44.2.2 CSSM_DATA ... 700
44.2.3 CSSM_CRYPTO_DATA ... 700
44.2.4 CSSM_DATE ... 700
44.2.5 CSSM_RANGE ... 701
44.2.6 CSSM_QUERY_SIZE_DATA ... 701
44.2.7 CSSM_HEADERVERSION ... 701
44.2.8 CSSM_KEY_SIZE ... 701
44.2.9 CSSM_KEYHEADER ... 702
44.2.10 CSSM_KEY ... 707
44.2.11 CSSM_WRAP_KEY ... 707
44.2.12 CSSM_CALLBACK ... 707
44.2.13 CSSM_CSP_TYPE ... 708
44.2.14 CSSM_CSP_SESSION_TYPE ... 708
44.2.15 CSSM_NOTIFY_CALLBACK ... 708
44.2.16 CSSM_HANDLEINFO ... 709
44.2.17 CSSM_PADDING ... 709
44.2.18 CSSM_CONTEXT_ATTRIBUTE ... 709
44.2.19 CSSM_CONTEXT ... 711
44.2.20 CSSM_CSP_CAPABILITY ... 717
44.2.21 CSSM_SOFTWARE_CSPSUBSERVICE_INFO ... 717
44.2.22 CSSM_HARDWARE_CSPSUBSERVICE_INFO ... 717
44.2.23 CSSM_HYBRID_CSPSUBSERVICE_INFO ... 720
44.2.24 CSSM_CSP_WRAPPEDPRODUCTINFO ... 720
44.2.25 CSSM_CSP_FLAGS ... 721
44.2.26 CSSM_CSPSUBSERVICE ... 721
44.2.27 CSSM_SERVICE_INFO ... 722
44.2.28 CSSM_MODULE_INFO ... 723
44.3 Cryptographic Operations ... 725
CSP_SignData ... 726

CSP_SignDataInit 728
CSP_SignDataUpdate 730
CSP_SignDataFinal 732
CSP_VerifyData 734
CSP_VerifyDataInit 736
CSP_VerifyDataUpdate 738
CSP_VerifyDataFinal 740
CSP_DigestData 741
CSP_DigestDataInit 743
CSP_DigestDataUpdate 744
CSP_DigestDataClone 746
CSP_DigestDataFinal 748
CSP_GenerateMac 750
CSP_GenerateMacInit 752
CSP_GenerateMacUpdate 754
CSP_GenerateMacFinal 756
CSP_VerifyMac 758
CSP_VerifyMacInit 760
CSP_VerifyMacUpdate 762
CSP_VerifyMacFinal 764
CSP_QuerySize 765
CSP_EncryptData 767
CSP_EncryptDataInit 770
CSP_EncryptDataUpdate 772
CSP_EncryptDataFinal 774
CSP_DecryptData 776
CSP_DecryptDataInit 779
CSP_DecryptDataUpdate 781
CSP_DecryptDataFinal 783
CSP_GenerateKey 785
CSP_GenerateKeyPair 788
CSP_GenerateRandom 791
CSP_FreeKey 793
CSP_ObtainPrivateKeyFromPublicKey 794
CSP_WrapKey 795
CSP_UnwrapKey 798
CSP_DeriveKey 801
CSP_GenerateAlgorithmParams 804
CSP_QueryKeySizeInBits 806
44.4 Cryptographic Sessions and Logon 808
CSP_Login 809
CSP_Logout 810
CSP_ChangeLoginPassword 811
44.5 Extensibility Functions 812
CSP_PassThrough 813
44.6 Module Management Functions 814
CSP_GetCapabilities 815
CSP_EventNotify 816

Part 10 CSSM Trust Policy Interface **819**

Chapter 45 Introduction **821**
45.1 CDSA Add-In Module Overview 821
45.2 Trust Policy Overview 822
45.2.1 Using Trust Policy Modules 822

Chapter 46 Trust Policy Interface **823**
46.1 Overview 823
46.1.1 Trust Policy Services API 825
46.1.2 Trust Policy Module Operations 826
46.2 Data Structures 827
46.2.1 CSSM_DATA 827
46.2.2 CSSM_OID 827
46.2.3 CSSM_FIELD 827
46.2.4 CSSM_REVOKE_REASON 828
46.2.5 CSSM_CRL_TYPE 828
46.2.6 CSSM_CRL_ENCODING 828
46.2.7 CSSM_DL_DB_HANDLE 828
46.2.8 CSSM_DL_DB_LIST 829
46.2.9 CSSM_CERTGROUP 829
46.2.10 CSSM_EVIDENCE_FORM 830
46.2.11 CSSM_VERIFYCONTEXT 830
46.2.12 CSSM_TP_WRAPPEDPRODUCTINFO 832
46.2.13 CSSM_TPSUBSERVICE 833
46.2.14 CSSM_SPI_TP_FUNCS 834
46.3 Trust Policy Operations 837
TP_CertRequest 838
TP_CertRetrieve 841
TP_CertGroupVerify 843
TP_CertSign 846
TP_CertRevoke 848
TP_CrlVerify 850
TP_CrlSign 852
TP_ApplyCrlToDb 854
TP_CertGroupConstruct 856
TP_CertGroupPrune 858
46.4 Extensibility Functions 860
TP_PassThrough 861

Part 11 CSSM Certificate Library Interface **863**

Chapter 47 Introduction **865**
47.1 CSSM Add-In Module Overview 865
47.2 Certificate Library Overview 866
47.2.1 Certificate Life Cycle 866

Chapter 48 Certificate Library Interface **869**
48.1 Overview 869
48.1.1 Certificate Operations 870
48.1.2 Certificate Revocation List Operations 872
48.1.3 Extensibility Functions 874
48.2 Data Structures 874
48.2.1 CSSM_CL_HANDLE 875
48.2.2 CSSM_CERT_TYPE 875
48.2.3 CSSM_CERT_ENCODING 875
48.2.4 CSSM_CERT_BUNDLE_TYPE 876
48.2.5 CSSM_CERT_BUNDLE_ENCODING 876
48.2.6 CSSM_CERT_BUNDLE_HEADER 876
48.2.7 CSSM_CERT_BUNDLE 877
48.2.8 CSSM_OID 877
48.2.9 CSSM_CRL_TYPE 877
48.2.10 CSSM_CRL_ENCODING 877
48.2.11 CSSM_FIELD 878
48.2.12 CSSM_ESTIMATED_TIME_UNKNOWN 878
48.2.13 CSSM_CA_SERVICES 878
48.2.14 CSSM_CL_CA_CERT_CLASSINFO 879
48.2.15 CSSM_CL_CA_PRODUCTINFO 879
48.2.16 CSSM_CL_ENCODER_PRODUCTINFO 880
48.2.17 CSSM_CL_WRAPPEDPRODUCTINFO 881
48.2.18 CSSM_CLSUBSERVICE 882
48.2.19 Certificate Operations 884
CL_CertRequest 885
CL_CertRetrieve 888
CL_RegistrationFormRequest 890
CL_RegistrationFormSubmit 891
CL_CertMultiSignRequest 893
CL_CertMultiSignRetrieve 896
CL_CertRecoveryRequest 898
CL_CertRecoveryRetrieve 901
CL_CertRecover 903
CL_CertKeyRecover 904
CL_CertAbortRecovery 906
CL_CertVerify 907
CL_CertGetFirstFieldValue 909
CL_CertGetNextFieldValue 911
CL_CertAbortQuery 912
CL_CertGetKeyInfo 913

		CL_CertGetAllFields	914
		CL_CertGroupToSignedBundle	915
		CL_CertGroupFromVerifiedBundle	917
		CL_CertImport	919
		CL_CertExport	920
		CL_CertDescribeFormat	921
	48.3	Certificate Revocation List Operations	922
		CL_CrlCreateTemplate	923
		CL_CrlSetFields	924
		CL_CrlRequest	926
		CL_CrlRetrieve	929
		CL_CrlAddCert	931
		CL_CrlRemoveCert	933
		CL_CrlSign	934
		CL_CrlVerify	936
		CL_IsCertInCrl	938
		CL_CrlGetFirstFieldValue	939
		CL_CrlGetNextFieldValue	941
		CL_CrlAbortQuery	942
		CL_CrlDescribeFormat	943
	48.4	Extensibility Functions	944
		CL_PassThrough	945
Part	**12**	**CSSM Data Storage Library Interface**	**947**
Chapter	**49**	**Introduction**	**949**
	49.1	CDSA Add-In Module Overview	949
	49.2	Data Storage Library Overview	950
Chapter	**50**	**Data Storage Library Interface**	**951**
	50.1	Overview	951
	50.1.1	Categories of Operations	952
	50.1.2	Data Storage Library Operations	953
	50.1.3	Data Store Operations	953
	50.1.4	Data Record Operations	954
	50.1.5	Extensibility Functions	955
	50.2	Data Storage Data Structures	956
	50.2.1	CSSM_DL_HANDLE	956
	50.2.2	CSSM_DB_HANDLE	956
	50.2.3	CSSM_DL_DB_HANDLE	956
	50.2.4	CSSM_DL_DB_LIST	956
	50.2.5	CSSM_DB_ATTRIBUTE_NAME_FORMAT	957
	50.2.6	CSSM_DB_ATTRIBUTE_FORMAT	957
	50.2.7	CSSM_DB_ATTRIBUTE_INFO	957
	50.2.8	CSSM_DB_ATTRIBUTE_DATA	958
	50.2.9	CSSM_DB_RECORDTYPE	959
	50.2.10	CSSM_DB_CERTRECORD_SEMANTICS	959
	50.2.11	CSSM_DB_RECORD_ATTRIBUTE_INFO	960

50.2.12 CSSM_DB_RECORD_ATTRIBUTE_DATA 961
50.2.13 CSSM_DB_RECORD_PARSING_FNTABLE 961
50.2.14 CSSM_DB_PARSING_MODULE_INFO 962
50.2.15 CSSM_DB_INDEX_TYPE 962
50.2.16 CSSM_DB_INDEXED_DATA_LOCATION 962
50.2.17 CSSM_DB_INDEX_INFO 963
50.2.18 CSSM_DB_UNIQUE_RECORD.. 963
50.2.19 CSSM_DB_RECORD_INDEX_INFO.. 964
50.2.20 CSSM_DB_ACCESS_TYPE.. 964
50.2.21 CSSM_DBINFO.. 965
50.2.22 CSSM_DB_OPERATOR... 966
50.2.23 CSSM_DB_CONJUNCTIVE.. 966
50.2.24 CSSM_SELECTION_PREDICATE 966
50.2.25 CSSM_QUERY_LIMITS 967
50.2.26 CSSM_QUERY_FLAGS.. 967
50.2.27 CSSM_QUERY.. 968
50.2.28 CSSM_DLTYPE 968
50.2.29 CSSM_DL_PKCS11_ATTRIBUTES 969
50.2.30 CSSM_DB_DATASTORES_UNKNOWN 969
50.2.31 CSSM_DL_WRAPPEDPRODUCT_INFO 969
50.2.32 CSSM_NAME_LIST 970
50.2.33 CSSM_DLSUBSERVICE... 970
50.3 Data Storage Library Operations... 973
DL_Authenticate 974
50.4 Data Store Operations... 975
DL_DbOpen 976
DL_DbClose 978
DL_DbCreate.. 979
DL_DbDelete 981
DL_DbImport 983
DL_DbExport 986
DL_DbSetRecordParsingFunctions... 988
DL_DbGetRecordParsingFunctions 989
DL_GetDbNames 990
DL_GetDbNameFromHandle 991
DL_FreeNameList 992
50.5 Data Record Operations 993
DL_DataInsert 994
DL_DataDelete.. 996
DL_DataModify 997
DL_DataGetFirst 999
DL_DataGetNext 1001
DL_DataAbortQuery 1003
DL_DataGetFromUniqueRecordId.. 1004
DL_FreeUniqueRecord 1005
50.6 Extensibility Functions 1006
DL_PassThrough 1007

Part 13 CSSM Key Recovery Interface **1009**

Chapter 51 Introduction **1011**
51.1 CDSA Add-In Module Overview 1011
51.2 Key Recovery Overview 1012
51.2.1 Key Recovery Nomenclature 1012
51.2.2 Key Recovery Types 1012
51.2.3 Lifetime of Key Recovery Fields 1013
51.2.4 Key Recovery Policy 1014
51.2.5 Operational Scenarios for Key Recovery 1014
51.3 Key Recovery in the Common Data Security Architecture 1015

Chapter 52 Key Recovery Service Provider Interface **1017**
52.1 Overview 1017
52.1.1 Key Recovery Phases 1018
52.1.2 Key Recovery Registration Operations 1019
52.1.3 Key Recovery Enablement Operations 1019
52.1.4 Key Recovery Request Operations 1019
52.1.5 Privileged Context Functions 1020
52.1.6 Extensibility Functions 1020
52.2 Data Structures 1020
52.2.1 CSSM_KR_HANDLE 1020
52.2.2 CSSM_KR_NAME 1020
52.2.3 CSSM_KR_PROFILE 1021
52.2.4 CSSM_CERT_LIST 1022
52.2.5 CSSM_CONTEXT_ATTRIBUTE Extensions 1022
52.2.6 CSSM_ATTRIBUTE_TYPE Additions 1023
52.2.7 CSSM_KRSUBSERVICE 1023
52.2.8 CSSM_KRINFO 1024
52.2.9 CSSM_PRIV_FUNC_PTR 1024
52.3 Key Recovery Registration Operations 1025
KRSP_RegistrationRequest 1026
KRSP_RegistrationRetrieve 1028
52.4 Key Recovery Enablement Operations 1029
KRSP_GenerateRecoveryFields 1030
KRSP_ProcessRecoveryFields 1032
52.5 Key Recovery Request Operations 1034
KRSP_RecoveryRequest 1035
KRSP_RecoveryRetrieve 1037
KRSP_GetRecoveredObject 1038
KRSP_RecoveryRequestAbort 1040
52.6 Privileged Context Operations 1041
KRSP_PassPrivFunc 1042
52.7 Extensibility Functions 1043
KRSP_PassThrough 1044

Glossary **1045**

Index **1051**

List of Figures

1-1 The Common Data Security Architecture for all Platforms 7
2-1 Services Provided by CSSM 15
2-2 Processing Steps to Attach an Add-In Module 17
2-3 CSSM Dispatches Calls to Selected Add-In Security Modules 20
2-4 Indirect Creation of a Security Context 24
7-1 A Multi-Service, Add-In Module 43
7-2 A Single Handle References a Multi-Service Add-In Module 44
9-1 The Common Data Security Architecture for all Platforms 52
10-1 Application Using Cryptographic Services and Persistent Storage Services 57
10-2 Three Module-Specific Certificate Chains 60
10-3 Module credentials with app-specific chain 61
13-1 Certificate Life Cycle States and Actions 276
15-1 Key Recovery Phases 421
16-1 Elective Key Recovery Services in the CSSM 425
17-1 Encrypted Communications without Key Recovery 433
17-2 Encrypted Communications with Key Recovery Enablement 434
18-1 Bilateral Authentication Using Software Credentials 471
21-1 The Common Data Security Architecture for all Platforms 528
23-1 Signed Manifest Architectural View 531
23-2 Relationships of Manifest, Signer′s Information and Signature Block 532
27-1 Relationship of Publisher’s Archive and Signed Manifest 551
27-2 Relationship of Distributor’s Archive to Publisher’s Archive 552
27-3 Relationship of Reseller to Distributor to Publisher 553
28-1 Common Data Security Architecture for all Platforms 564
29-1 Steps to Attach an Add-In Module and load its EMM 567
30-1 Certificate Chain for an Elective Module Manager 570
33-1 Common Data Security Architecture for all Platforms 608
33-2 CDSA Add-In Module Structure 609
34-1 Three Module-Specific Certificate Chains 616
35-1 Credentials of an Add-In Service Module 618
35-2 Certificate Chain for an Add-In Service Module 619
35-3 Signature File for an Add-In Module 620
37-1 Common Data Security Architecture for all Platforms 675
38-1 Enhanced Common Data Security Architecture 679
43-1 CDSA Add-In Module Structure 691
45-1 CDSA Add-In Module Structure 821
47-1 CDSA Add-In Module Structure 865
47-2 Certificate Life Cycle States and Actions 867
49-1 CDSA Add-In Module Structure 949
51-1 CDSA Add-In Module Structure 1011
52-1 Key Recovery Phases 1018

List of Tables

36-1 Module Event Types 634
36-2 Module Event Parameters 635
D-1 Notification Reasons 648
D-2 Service Access Tables 650
44-1 Keyblob Type Identifiers 703
44-2 Keyblob Format Identifiers 704
44-3 Key Class Identifiers 704
44-4 Key Usage Flags 706
44-5 Attribute Types 711
44-6 Context Types 712
44-7 Algorithms for a Session Context 715
44-8 PKCS 11 CSP Reader Flags 717
44-9 PKCS 11 CSP Reader Flags 719
44-10 PKCS 11 CSP Token Flags 719
44-11 CSP Information Type Identifiers and Associated Structure Types 722

Preface

The Open Group

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the *IT DialTone*. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- Consolidating, prioritizing, and communicating customer requirements to vendors
- Conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- Managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- Adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- Licensing and promoting the Open Brand, represented by the ''X'' mark, that designates vendor products which conform to Open Group Product Standards
- Promoting the benefits of the IT DialTone to customers, vendors, and the public

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

The Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product.

The ‘‘X’’ device is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the X/Open Trade Mark License Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our Product Standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group’s rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation—programmer's guides, user manuals, and so on—relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.

- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/corrigenda**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/pubs**.

This Document

This document is a CAE Specification (see above).

The CDSA specification is divided into thirteen parts in order to address the needs of a number of distinct audiences. Most of the parts are normative (they define programming interfaces), and a small number are descriptive and/or informative.

The 13 parts are as follows:

- **Part 1**: Common Data Security Architecture (CDSA)

  Presents the overall CDSA architecture, with emphasis on the Common Security Services Manager. It explains the four-layer architecture as consisting of:

  1. Applications
  2. Layered services and middleware
  3. Common Security Services Manager (CSSM) infrastructure
  4. Security Service Provider Modules

- **Part 2**: Common Security Services Manager (CSSM) API

  Defines the base application programming interfaces available in all CSSM implementations, as follows:

  — CSSM Core Services API

  — Cryptographic Services API

  — Trust Policy Services API

  — Certificate Library Services API

  — Data Storage Library Services API

  These are a subset of the APIs that can be supported and available to applications and add-in service modules through the CSSM.

- **Part 3**: CSSM Key Recovery API

  Defines the elective application programming interface that applications and other add-in service modules can use to access key recovery services. Applications use these services explicitly. CSSM dynamically incorporates extended services when required. From the application's perspective, basic services and elective services are accessed through the CSSM in the same manner.

- **Part 4**: CDSA Embedded Integrity Services Library API

  Defines the application programming interfaces provided by the static Embedded Integrity Services Library (EISL). These services are available to applications, add-in security service modules, and to CSSM itself. This also includes documentation of the bilateral authentication procedure for integrity and identity checks between two parties, and the specification of manifests as an aggregator of heterogeneous signed objects.

- **Part 5**: CDSA Signed Manifest Specification

  This is a Descriptive/Informational specification document.

  It defines the structure and use of signed manifests. A manifest aggregates the description of the integrity of a set of heterogeneous signed objects. A manifest is one of the credentials required for each dynamic component of the CDSA.

- **Part 6**: CSSM Elective Module Manager

  Defines CSSM-internal interfaces for elective module managers. These interfaces include installation, dynamic attach, function registration, and mechanisms for state sharing among module managers.

- **Part 7**: CSSM Add-In Module Structure and Administration

  Defines the architecture and management interfaces for all add-in security service modules. Modules must implement this interface to dynamically attach to CSSM and provide their services to applications through the CSSM APIs.

- **Part 8**: CDSA Mechanisms for Policy Compliance

  This is a Descriptive / Informational specification document.

  It defines architectural extensions and feature enhancements to support a wide range of government-specified and enterprise-specified policies that control the use of cryptography or other security services. These extensions affect all components in the four-layer architecture: applications, layered security services, the Common Security Services Manager (CSSM), and add-in service provider modules.

- **Part 9**: CSSM Cryptographic Service Provider Interface

  Defines the interface that cryptographic service providers must conform to in order to be accessible via CSSM. Individuals interested in making cryptographic services available under the CSSM interface will need to be familiar with the CSSM SPI. This part also provides key information regarding the expected behavior of a cryptographic service provider as well as implementation examples, which may be of use to the cryptographic service provider developer.

- **Part 10**: CSSM Trust Policy Interface

  Defines the interface that trust policy modules must conform to in order to be accessible via CSSM. Individuals interested in developing trust policy features available under the CSSM interface will need to be familiar with the CSSM TPI. This part also provides key information regarding the expected behavior of a trust policy module, as well as implementation examples which may be of use to the trust policy module developer.

- **Part 11**: CSSM Certificate Library Interface

  Defines the interface that certificate libraries must conform to in order to be accessible via CSSM. Individuals interested in developing certificate library features available under the CSSM interface will need to be familiar with the CSSM CLI. This part also provides key information regarding the expected behavior of a certificate library module, as well as implementation examples which may be of use to the certificate library module developer.

- **Part 12**: CSSM Data Storage Library Interface

  Defines the interface that a data storage library must conform to in order to be accessible via CSSM. Individuals interested in developing data storage library features available under the CSSM interface will need to be familiar with the CSSM DLI. This part also provides key information regarding the expected behavior of a data storage library module, as well as implementation examples which may be of use to the data storage library module developer.

- **Part 13**: CSSM Key Recovery Interface

  Defines the service provider interface that key recovery modules must conform to in order to be accessible as an elective service via CSSM. Individuals interested in developing key recovery mechanisms and making them accessible through the CSSM interface will need to be familiar with the CSSM KRI. This part also provides critical information regarding the expected behavior of a key recovery module, as well as implementation examples which may be of use to the key recovery module developer.

A glossary and index are also provided.

Intended Audience

Part 1 provides an overview of the CDSA for Independent Software Vendors (ISVs), Independent Hardware Vendors (IHVs), and platform vendors who develop security products as complete applications in a monolithic environment. This audience includes:

- Experienced software and hardware designers
- Security architects who work in high-end cryptography
- Advanced programmers
- Sophisticated integrators familiar with numerous forms of network computing

This audience understands their requirements and the advantages of a ubiquitous, extensible security infrastructure upon which they can build security-aware application products, or through which they can offer their plug-in security service products.

The CDSA specifications are partitioned to address the needs and perspectives of three audiences—application developers, security service providers, and infrastructure providers.

Developers and providers, having read Part 1, may choose to selectively read other parts of the document, since particular specifications will satisfy the needs of the different categories of reader:

- *Application Developers* who implement applications, layered services and middleware, will find Parts 2, 3, 4, and 5 useful.
- *CSSM Infrastructure Providers* who implement the Common Security Services Manager will find Parts 2, 5, 6, 7, and 8 useful.
- *Security Service Module Providers* who implement dynamic plug-in security services will find Parts 5, 9, 10, 11, 12, and 13 useful.

The intended audience for various parts of the book is summarized here:

Part 2 This part is intended for use by Independent Software Vendors (ISVs) who will develop their own application code to interact with CSSM services. These ISVs are highly experienced software and security architects, advanced programmers, and sophisticated users. They are familiar with network operating systems and high-end cryptography. We assume that this audience is familiar with the basic capabilities and features of the protocols they are considering.

Part 3 This part is intended for use by Independent Software Vendors (ISVs) who will develop exportable and importable application code to interact with CSSM services. These ISVs are highly experienced software and security architects, advanced programmers, and sophisticated users. They are also familiar with local and foreign government regulations on the use of cryptography and the implication of those regulations for their applications and products.

Part 4 This part should be used by Platform Vendors and Independent Software Vendors (ISVs) who want to enhance product security by including integrity and authentication checks in the core of their products. These developers must have a good understanding of:

- The principles of integrity and authentication in an executing software environment
- The role of digital credentials in authenticating a software object
- The structure of a secured manufacturing environment for authenticate-able software products

It is also assumed that these developers have a working knowledge of signed manifests as digital credentials.

Part 5 This part is essential for all developers whose products involve the expression and/or validation of the integrity of a collection of digital objects. This includes those developing:

- Add-in security service modules for CSSM
- Elective Module Managers for CSSM
- Applications that:
  - Package a collection of digital objects or services
  - Make assertions about a collection of digital objects or services
  - Verify the integrity of a collection of digital objects or services
  - Establish trust in the assertions being made about a collection of digital objects or services

Part 6 This part should be used by Independent Software Vendors (ISVs) who want to develop categories of security services different from the four basic CSSM service categories: trust policy, certificate library, data storage library, and cryptographic services. These ISVs should be highly experienced software and security architects and advanced programmers. This audience is familiar with high-end cryptography, digital certificates, and features of the security protocols they are considering.

Part 7 This part should be used by Independent Software Vendors (ISVs) who want to develop their own add-in modules to support one or more of the CSSM Service Provider Interfaces. These ISVs should be highly experienced software and security architects, advanced programmers, and sophisticated users. They are familiar with data storage systems, high-end cryptography, and digital certificates. It is assumed that this audience is familiar with the basic capabilities and features of the protocols they are considering.

Part 8 This part should be of interest to a broad audience of CDSA developers and CDSA system administrators.

- CSSM Developers—engineers who implement or port CSSM can use this document as a design specification for enhanced mechanisms.
- Component Developers—engineers who design and develop add-in security service modules can use this document as the definition of software procedures that add-in modules must incorporate to inter-operate with an enhanced CSSM. These procedures are the responsibility of a CDSA add-in module manufacturer.
- CSSM and Component Vendors—product deployment engineers can use this document as a guide to required business practices and procedures for packaging and shipping an enhanced product.

- Policy Definers—corporate and government entities that define system-wide policies on the use of security services can use this document as a guide to determine whether the enhanced CSSM meets their compliance requirements.

Part 9 This part should be used by Independent Software Vendors (ISVs) who want to develop CSSM add-in service modules providing cryptographic services such as digital signing and verification, encryption and decryption, digesting, key generation, and random number generation. These developers must have a very strong understanding of:

- The cryptographic algorithms they intend to implement
- Standard formats for cryptographic keys
- All legal constraints on their product defined by the government of their local jurisdiction

It is also assumed that these developers have a working knowledge of how the cryptographic services they provide can be used to provide integrity, authentication, confidentiality, and non-repudiation of data and actions.

Part 10 This part is directed toward security software developers who want to develop their own Trust Policy module. These developers should be familiar with cryptography and digital certificates. This document assumes the reader is familiar with the basic capabilities and features of security protocols associated with authentication, integrity and privacy. These developers should be highly experienced software architects, advanced programmers, or sophisticated users, who have a strong understanding of public-key infrastructures.

Part 11 This part should be used by Independent Software Vendors (ISVs) who want to develop add-in service modules providing creation and manipulation of digital certificates and certificate revocation lists through the CSSM APIs. These developers should have a strong understanding of:

- One or more digital certificate standards
- Public-key infrastructures
- certification Authority (CA) protocols
- Certificate life cycle services

It is also assumed that these developers are knowledgeable users of cryptographic services.

Part 12 This part should be used by Independent Software Vendors (ISVs) who want to develop CSSM add-in service modules providing persistent storage for security-related objects, such as digital certificates, certificate revocation lists, cryptographic keys, and security policy statements. These developers should have a strong understanding of:

- Underlying storage mechanisms to be used in an implementation
- Traditional database-implementation techniques
- Data server interface protocols

It is also assumed that these developers have a working knowledge of cryptographic services.

Part 13 This part is intended for use by Independent Software Vendors (ISVs) who will develop products that provide key recovery services through the CSSM APIs. These ISVs are highly experienced software and security architects and advanced programmers. They are also familiar with local and foreign government regulations on the use of cryptography and the implication of those regulations for their products.

Trademarks

Acknowledgements

The Open Group gratefully acknowledges that this document is the result of a co-operative effort and exchange of ideas of participating industry leaders. The specification was initiated by Intel Architecture Labs, and led to the development efforts of CDSA, having attained the support and participation of organizations such as Entrust, Hewlett-Packard, IBM, Motorola, Netscape, Sun, and Trusted Information Systems, together with the many member organizations of the PKI (Public Key Infrastructure) Task Group, meeting regularly under the auspices of The Open Group.

Referenced Documents

The following documents are referenced in this specification:

ASN.1

ITU-T Recommendation X.200: Abstract Syntax Notation One (ASN.1).

ITU was formerly CCITT (Comité Consultatif Internationale Telegraphique et Telephonique).

BER

ITU-T Recommendation X.209: Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

BSAFE

BSAFE Cryptographic Toolkit, RSA Data Security, Inc., Redwood City, CA.

Cryptography

Applied Cryptography, Second Edition, Protocols, Algorithms, and Source Code in C, Bruce Schneier: John Wiley & Sons, Inc., 1996.

Cryptography Usage

Handbook of Applied Cryptography, Menezes, A., Van Oorschot, P., and Vanstone, S., CRC Press, Inc., 1997.

CSSM Java

CSSM Java Application Programming Interface (API) Specification, Intel Architecture Labs, 1996.

DER

ITU-T Recommendation X.690: Distinguished Encoding Rules.

DSA

Federal Information Procurement Standard (FIPS) 186, Digital Signature Standard.

Key Escrow

A Taxonomy for Key Escrow Encryption Systems, Denning, Dorothy E., and Branstad, Dennis, Communications of the ACM, Vol 39, No. 3, March 1996.

OIW

Stable Implementation Agreements, Open Systems Environment Implementors Workshop, June 1995.

PKCS

The Public-Key Cryptography Standards, RSA Laboratories, RSA Data Security, Inc., Redwood City, CA.

SDSI

SDSI: A Simple Distributed Security Infrastructure, R. Rivest and B. Lampson, 1996.

SHA

Federal Information Procurement Standard (FIPS) 180, Secure Hash Algorithm.

SPKI

Simple Public Key Infrastructure, Internet Draft: draft-ietf-spki-cert-structure-03.txt (Expires 26th May 1998).

X.509
ITU-T Recommendation X.509: The Directory — Authentication Framework, 1988.

License Agreement for CDSA Specifications

CAE Specification

Part 1:

Common Data Security Architecture (CDSA)

The Open Group

Chapter 1

Introduction

The Common Data Security Architecture (CDSA) is a set of layered security services that address communications and data security problems in the emerging Internet and Intranet application space. Intel Architecture Labs (IAL) defined the CDSA to:

- Encourage interoperable, horizontal security standards
- Offer essential components of security capability to the industry at large

The motivation for a robust, broadly diffused, multi-platform, industry-standard security infrastructure is clear. The definition of such an infrastructure, however, must accommodate the emerging Internet and Intranet business opportunities and address the requirements unique to the most popular client systems, namely personal computers (PCs) and networked application servers. CDSA focuses on security in peer-to-peer distributed systems with homogeneous and heterogeneous platform environments. The architecture also applies to the components of a client-server application. The CDSA addresses security issues in a broad range of applications, including:

- Electronic commerce in business-to-business and home-to-business applications—this implies a selectable range of security solutions
- Content distribution of software, reference information, educational material, or entertainment content requiring new algorithms and protocols
- Metering of content, service, or both, and the requirement for secure storage of state and value
- Securing business or personal activity for private email, home banking, and monetary transactions where the value, and thus the threat, may be quite varied.

The architecture addresses the security requirements of this broad range of applications by:

- Providing layered security mechanisms (not policies)
- Supporting application-specific policies by providing an extensibility mechanism that manages add-in (policy-specific) modules
- Supporting distinct user markets and product needs by providing a dynamically-extensible security framework that securely adds new categories of security service
- Exposing flexible service provider interfaces that can accommodate a broad range of formats and protocols for certificates, cryptographic keys, policies, and documents
- Supporting existing secure protocols such as Secure Sockets Layer (SSL), Secure Multipurpose Internet Mail Extensions (SMIME), Secure ElectronicTransaction protocol (SET)

1.1 The Threat Model

The need for a security infrastructure like CDSA has been fueled by the desire to provide new applications in the face of increasing incidents of unauthorized access and manipulation of computer systems, data, and communications. Malicious observation and manipulation of computer systems can be classified into three categories, based on the origins of threats. The origins of threats are expressed in terms of the security perimeter that's been breached in order to effect the malicious act:

Category I The malicious threat originates outside of the computer system. The perpetrator breaches communications access controls, but still operates under the constraints of the communications protocols. This is the standard hacker attack.

Category I attacks are best defended against by correctly designed and implemented access-control protocols and mechanisms, and proper system administration, rather than by the use of secured software.

Frequently, the goal of a Category I attack is to mount a Category II attack.

Category II The malicious attack originates as software running on the platform. The perpetrator introduces malicious code onto the platform and the operating system executes it. The attack moves inside the communications perimeter, but remains bounded by the operating system and BIOS (Basic Input-Output System), using their interfaces. The malicious software may have been introduced with or without the user's consent. This is the common virus attack.

Examples include viruses, Trojan horses, and software used to discover secrets stored in other software (such as another user's access control information). Category II attacks tend to attack classes of software. Viruses are a good example. Viruses must assume certain coding characteristics to be constant among the target population, such as the format of the execution image. It's the consistency of software across individual computer systems and even across platforms that enables Category II attacks.

Category III The perpetrator completely controls the platform, may substitute hardware or system software, and may observe any communications channel (such as using a bus analyzer). This attack faces no security perimeter and is limited only by technical expertise and financial resources.

In an absolute sense, Category III attacks are impossible to prevent on the computer system. Defense against a Category III attack merely raises a technological bar to a height sufficient to deter a perpetrator by providing a poor return on investment. That investment might be measured in terms of the tools necessary, or the skills required to observe and modify the software's behavior. The technological bar, from low to high would be:

- No special analysis tools required (such as debuggers and system diagnostic tools)
- Specialized software analysis tools (such as SoftIce)
- Specialized hardware analysis tools (such as processor emulators and bus-logic analyzers)

The CSSM's goal is to defend against Category II and Category III attacks, up to but not including the level of specialized hardware analysis tools. This provides a reasonable compromise. As threat follows value, this level of security is adequate for low-to-medium-value applications and high-value applications where the user is unlikely to be a willing perpetrator (such as applications involving the user's personal property).

1.2 Common Data Security Architecture

The Common Data Security Architecture is a set of layered services and associated programming interfaces, providing an integrated, but dynamic set of security services to applications. The lowest layers begins with fundamental components such as cryptographic algorithms, random numbers, and unique identification information. The layers build up to digital certificates, key management mechanisms, and secure transaction protocols in higher layers.

1.2.1 Architectural Assumptions

The CDSA design follows five architectural principles:

- **A layered service provider model**.

  CDSA is built up from a set of horizontal layers, each providing services to the layer above it. This approach is portable, adaptable, and modular.

- **Open architecture**.

  The CDSA is fully disclosed for peer review, standardization, and adoption by the industry.

- **Modularity and extensibility**.

  Components of each layer can be chosen as separate modules. An extensible framework supports inserting module managers for new, elective categories of security services. Extensibility fosters industry growth by encouraging development of incremental functionality and performance-competitive implementations of each add-in module.

- **Value in managing the details**.

  The CDSA can manage security details, so individual applications do not need to be concerned with security-related details. The CSSM APIs define logical categories of security services to assist developers in easily adding security to their application.

- **Embrace emerging technologies**.

  The CDSA incorporates emerging technologies for data security. Fundamental technologies include portable digital tokens and digital certificates.

The architecture is built on two fundamental models:

- **Portable digital tokens**

  These are used as a person's digital persona for commerce, communications, and access control. These digital tokens are encryption modules, with some amount of encrypted storage. They can be software or hardware, depending on the application's security needs. They come in various form factors, and may have multiple functions aggregated into a single device, such as a digital wallet.

- **Digital certificates**

These can be used to embody trust. These certificates do not create any new trust models or relationships. They are the digital form for current trust models. A person may have a certificate for each trust relationship (such as multiple credit cards, checkbooks, employer ID). Certificates are used for identity. They can also carry authorization information.

The ability of client platforms to accommodate these two new technologies is critical to the success of such platforms for digital commerce and information management as the Intranet extends seamlessly into the Internet.

1.2.2 Architectural Overview

CDSA defines an open, extensible architecture in which applications can selectively and dynamically access security services. Figure 1-1 shows the three basic layers of the Common Data Security Architecture:

- System Security Services
- The Common Security Services Manager (CSSM)
- Security Add-in Modules (cryptographic service providers, trust policy modules, certificate library modules, and data storage library modules)

It is the goal of CDSA to be a leading, multi platform security architecture that is horizontally broad and vertically robust. Horizontal breadth is achieved by an extensible design that can incorporate new categories of security services and the application programming interfaces for accessing those services. A vertically robust architecture defines layers that can support a full range of applications from security-naive to security-aware to a security service.

The Common Security Services Manager (CSSM) is the core of CDSA. CSSM manages categories of security services and multiple discrete implementations of those services as add-in security modules. CSSM:

- Defines the application programming interface for accessing security services
- Defines the service providers interface for security service modules
- Dynamically extends the security services available to an application, while maintaining an extended security perimeter for that application

Applications request security services through the CSSM security API, or via layered security services and tools implemented over the CSSM API. The requested security services are performed by add-in security modules. Four basic types of module managers are defined:

- Cryptographic Services Manager
- Trust Policy Services Manager
- Certificate Library Services Manager
- Data Storage Library Services Manager

Over time, new categories of security services will be defined, and new module managers will be required. CSSM supports elective module managers that dynamically extend the system with new categories of security services. Again, CSSM manages the extended security perimeter.

Below CSSM are add-in security modules that perform cryptographic operations and manipulate certificates. Add-in security modules may be provided by independent software and hardware vendors as competitive products. Applications use CSSM to direct their requests to modules from specific vendors or to any module that performs the required services. Add-in modules augment the set of available security services.

CDSA's extensible architecture allows new module types to be included that accommodate prudent division of labor. Signing services and key management services can be added at the System Security Services Layer and the Security Add-in Modules layer in CDSA. An appropriate degree of visibility of lower layers may be reflected at higher layers, such that a complete security profile can be managed uniformly. Independent software and hardware vendors may specialize in their chosen area of expertise and package their products as appropriate. For example, hardware-specific cryptographic device vendors can also provide tamper-resistant storage facilities in the same add-in module.

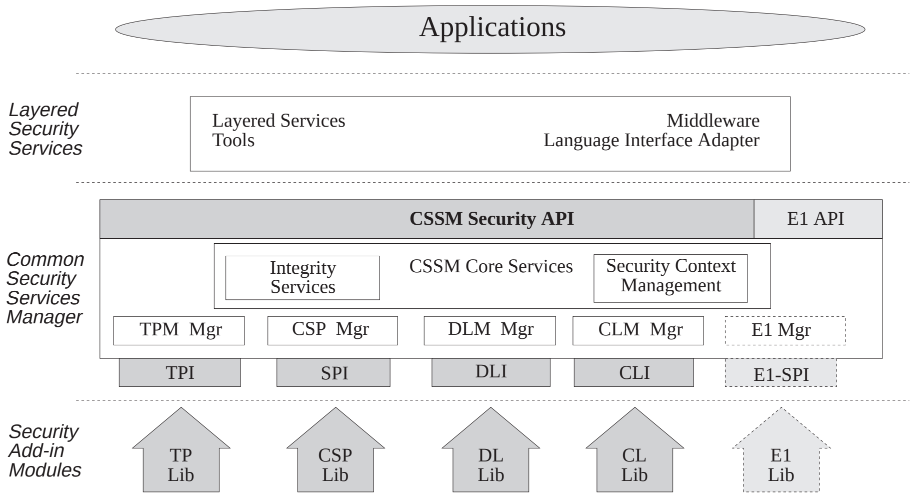

Figure 1-1 The Common Data Security Architecture for all Platforms

1.2.3 Layered Security Services

Layered Security Services are between application and basic CSSM services. Software at this layer may:

- Define high-level security abstractions (such as secure electronic mail services)
- Provide transparent security services (such as secure file systems or private communication)
- Make CSSM security services accessible to applications developed in languages other than the C language
- Provide tools to manage the security infrastructure

Applications can invoke the CSSM APIs directly, or use layered services to access security services on a platform. The use of security services through a layered service can be opaque. Legacy layered services, such as the Sockets protocol and HTTP, can be enhanced with security features for privacy and authentication. If this can be accomplished without changing the service interface, then applications can benefit from these services without change to the application code. Examples include:

- Hypertext Transfer Protocol (HTTP) over the Secure Sockets Layer (SSL) for secured network communications
- Pretty Good Privacy (PGP) for secured files

Additionally, new security-related layered services that define new interfaces can be developed. Applications that have only a high-level conceptual awareness of security can use these services with some modification of the application code. Examples include:

- Secure Electronic Transaction (SET) protocol for secure electronic commerce
- PGP for secure and private electronic mail

Another category of layered service is the language interface adapter. A language adapter extends the CSSM API calls (defined in the C language) to other programming languages and programming environments. These language-specific wrappers may export the CSSM C language API calls directly to another language, or may abstract the CSSM concepts and present them through the target language. For example, a Java package defines object-oriented classes and methods by which Java applications and Java applets can use security functionality provided by and through CSSM.

CSSM accommodates many new and existing standards as layered services. The broad spectrum of layered security services is easier to implement by virtue of CSSM's modularity. Layered service developers are included in the category of application developers for purposes of this document.

1.2.4 Common Security Services Manager Layer

The second level of CDSA is the Common Security Services Manager (CSSM). CSSM, the essential component in the CDSA, integrates and manages all categories of security service. It enables tight integration of individual services, while allowing those services to be provided by interoperable modules. The CSSM defines a rich, extensible API to support developing secure applications and system services, and an extensible SPI supporting add-in security modules that implement building blocks for secure operations.

CSSM provides a set of core services that are common to all categories of security services. Examples include:

- Dynamic attach of an add-in security module
- Enforced verification and identification procedures when dynamically extending services
- General integrity services

Module managers within CSSM are responsible for matching API calls to one or more SPI calls that result in an add-in service module performing the requested service.

CSSM APIs are logically partitioned into functional subsets. The goal of this logical partitioning is to assist application developers in understanding and making effective use of the security APIs. CSSM itself is partitioned into a set of core services, context management services, integrity services, and a set of basic module managers. There is one module manager (MM) for each functional subset of the CSSM API. Each MM manages implementations of add-in modules that service the manager's respective functional category. CSSM defines four basic categories of service and their corresponding managers:

- Cryptographic Services Manager
- Trust Policy Services Manager
- Certificate Services Manager
- Data Store Services Manager

The **Cryptographic Services Manager** administers the Cryptographic Service Providers that may be installed on the local system. It defines a common API for accessing all of the Cryptographic Service Providers that may be installed beneath it. All cryptography functions are implemented by the CSPs.

The **Trust Policy Services Manager** administers the trust policy modules that may be installed on the local system. It defines a common API for these libraries. The API allows applications to request security services that require "policy review and approval" as the first step in performing the operation. Approval can be based on the identity, integrity, and authorization represented in a digital certificate. All policy-specific tests and decisions are implemented by the add-in trust policy module.

The **Certificate Services Manager** administers the Certificate Libraries that may be installed on the local system. It defines a common API for these libraries. The API allows applications to manipulate memory-resident certificates and certificate revocation lists. Operations must include creating, signing, verifying, and extracting field values from certificates. All certificate operations are implemented by the add-in certificate libraries. Each library incorporates knowledge of certificate data formats and how to manipulate that format.

The **Data Store Services Manager** defines an API for secure, persistent storage of certificates, certificate revocation lists (CRLs) and other security objects. The API must allow applications to search and select stored data objects, and to query meta-information about each data store (such as its name, date of last modification, size of the data store, and so on). Data store operations are implemented by add-in data storage library modules.

CSSM also extends to dynamically include elective module managers. These module managers define additional APIs for a new category of service. An example of an elective category of security services is Key Recovery. A Key Recovery Module Manager (KRMM) defines a set of APIs that provide applications access to key recovery services and SPIs, that allow vendors to implement competitive key recovery service modules. If an application chooses to use an elective service API, CSSM extends the services available to that application by dynamically attaching the appropriate module manager and an add-in service module to the running CSSM.

Two additional CSSM core services include:

- Integrity services
- Security context management

As the foundation of the security framework, CSSM must provide a set of integrity services that can be used by CSSM, module managers, add-in modules, and applications to verify the integrity of themselves and other components in the CSSM environment.

CSSM's minimal set of self-contained security services establishes its security perimeter. These self-contained services incorporate techniques to protect against category II and most category III attacks. Because application and add-in security service modules are dynamic components in the system, CSSM uses and requires the use of a strong verification mechanism to screen all components as they are added to the CSSM environment.

Applications can extend CSSM's security perimeter to include themselves by using bilateral authentication, integrity verification, and authorization checks during dynamic binding. These procedures and interfaces are defined in the *CSSM Elective Module Management* specification. By extending the security perimeter, CSSM helps applications address category II and category III attacks.

CSSM provides security context services to assist applications in specifying and managing the numerous parameters required for cryptographic operations. CSSM assists by providing default parameter values (when appropriate) and by managing the data structures used to hold these parameters.

1.2.5 Security Add-In Modules Layer

CSSM supports an extensible set of add-in service modules. The four basic service categories are:

- Cryptographic Service Providers (CSPs)
- Trust Policy Modules (TPs)
- Certificate Library Modules (CLs)
- Data Storage Library Modules (DLs)

Every instance of an add-in module must be installed with CSSM making the module accessible for use on the local system. The installation process records, in a persistent, CSSM-managed registry, the module's identifying name, a description of the services it provides, and the information required to dynamically load the module. Applications may query the registry and select one or more modules based on their capabilities.

Module implementors can provide multiple categories of service in a single module. These multi-service add-in modules separate module packaging from the application developer's functional view of CSSM APIs. The module simply registers interfaces in multiple categories. For example, a hardware cryptographic token vender may register CSP and DL interfaces which may capitalize on the vendor's tamper-resistant persistent storage technology. Other vendors may find synergy in supporting both TP and CL modules.

1.2.5.1 Cryptographic Service Providers (CSPs)

Cryptographic service providers (CSPs) are modules equipped to perform cryptographic operations and to securely store cryptographic keys. A CSP may implement one or more of these cryptographic functions:

- Bulk encryption algorithm
- Digital signature algorithm
- Cryptographic hash algorithm
- Unique identification number
- Random number generator
- Secure key storage
- Custom facilities unique to the CSP

A CSP may be instantiated in software, hardware, or both.

CSPs can be constructed to provide a subset of the services listed above. These subsets should be self-consistent. If an operation O is supported then related operations, such as the inverse of operation O should also be supported. CSPs must also provide:

- Key generation or key import
- Secure storage for cryptographic keys and variables that have been entrusted to the CSP for use or storage

It is highly desirable that CSPs support key import and key export. A primary goal of key export is portability of keys. Some CSPs can achieve this goal by physical portability of the cryptographic device versus logical portability of a key. CSPs should not reveal key material unless it's been wrapped. A CSP or an independent module can also deliver key management services, such as key escrow, key archive, or key recovery.

1.2.5.2 *Trust Policy Modules (TPs)*

Trust policy modules implement policies defined by authorities and institutions. Policies define the level of trust required before certain actions can be performed. Three basic action categories exist for all certificate-based trust domains:

- Actions on certificates
- Actions on certificate revocation lists
- Domain-specific actions (such as issuing a check or writing to a file)

The CSSM Trust Policy API defines the generic operations that should be supported by every TP module. Each module may choose to implement the subset of these operations that are required for its policy.

When a TP function has determined the trustworthiness of performing an action, the TP function may invoke certificate library functions and data storage library functions to carry out the mechanics of the approved action.

1.2.5.3 *Certificate Library Modules (CLs)*

Certificate library modules implement syntactic manipulation of memory-resident certificates and certificate revocation lists. The CSSM Certificate API defines the generic operations that should be supported by every CL module. Each module may choose to implement only those operations required to manipulate a specific certificate data format, such as X.509, SDSI, and so on.

The implementation of these operations should be semantic-free. Semantic interpretation of certificate values should be implemented in TP modules, layered services, and applications.

The CSSM architecture makes manipulation of certificates and certificate revocation lists orthogonal to persistence of those objects. Hence, it is not recommended that CL modules invoke the services of data storage library modules. Decisions regarding persistence should be made by TP modules, layered security services, and applications.

CL modules may implement their services locally or remotely. For example, remote signing requests may use standard message protocols such as PKCS#10 and may be transport-independent.

1.2.5.4 *Data Storage Library Modules (DLs)*

A Data storage library module provides stable storage for security-related data objects. These objects can be certificates, certificate revocation lists (CRLs), cryptographic keys, policy objects or application-specific objects. Stable storage could be provided by a

- Commercially-available database management system product
- Native file system
- Custom hardware-based storage devices
- Remote directory services
- In-memory storage

Each DL module may choose to implement only those operations required to provide persistence under its selected model of service.

The implementation of DL operations should be semantic-free. Semantic interpretation of stored objects such as certificate values, CRL values, key values, and policy should be interpreted by layered services, TP modules and applications. An extensible function interface is defined in the DL API. This mechanism allows each DL to provide additional functions to store and retrieve security objects, such as performance-enhancing retrieval functions or unique administrative functions required by the nature of the implementation.

1.2.5.5 Multi-Service Library Module

Vendors building add-in security module products can provide services for CSSM APIs from multiple CSSM-functional categories. The result is a multi-service add-in module A multi-service module is a single, dynamic add-in module that implements CSSM functions from two or more functional categories of the CSSM APIs.

Applications use a single identifier to reference the module for all categories of service. Multi-service add-in modules separate module/product packaging from the application developer's functional view of CSSM APIs.

1.3 Interoperability Goals

Interoperability is essential among CDSA components and among instances of CDSA's interoperability goals include:

- Applications written to the CSSM APIs will operate using add-in service modules from multiple vendors without major code changes or numerous special checks.
- Applications will run on different CSSM implementations without major code changes.
- Applications can use a particular add-in service module through different CSSM implementations and obtain the same results.
- Applications can use different implementations of the same add-in services and obtain the same results.

These goals could be achieved by the following combined efforts:

- Standard security service APIs and SPIs that define predictable behavior and allow distinct implementations
- Well-documented security service API and SPI specifications
- The use of conformance test suites for security services APIs and SPIs
- Publication of developer guides, porting guides, sample application code, and other tutorial materials
- Organizing working conferences for service providers to achieve and demonstrate levels of interoperability
- Specification of a standard object code signing mechanism for each platform hosting CDSA

Results of the first two efforts can be seen in the CDSA specification set. The CDSA conformance test suite checks API conformance of a CSSM implementation and SPI conformance for add-in service modules. All adopters of CDSA specifications, in whole or in part, are expected to use the conformance test suite to determine conformance of their products and to increase interoperability with other CDSA-based products.

The CDSA conformance test suite for a CSSM implementation checks:

- Correct behavior of CSSM core service functions, including module installation, registry queries, and the ability to attach a dummy add-in module

- Support for all of the basic APIs, by correct dispatching of calls and parameters to attached, dummy service providers, which must be included as part of the test suite
- Correct implementation of architecture features such as dynamic and transparent attachment of a dummy elective module manager, and attach-time authentication of a dummy add-in service, module
- Support for optional application authentication at module attach-time

The CDSA conformance test suite for an add-in service module must use a conformant real (or dummy) CSSM implementation to test add-in service modules for:

- Correct behavior during module attach, including bilateral authentication and proper handshakes to register services with CSSM
- Support for the service provider interface as recorded in the module's capabilities list
- Self consistent operation of logically-related functions, such as inverse operations (sign and verify), or life cycle operations (certificate creation followed by field value extraction, and persistent record creation followed by record retrieval and record deletion)

The conformance test suites contribute to interoperability, but they are not the complete solution. The conformance test suites are not intended to be:

- Complete correctness tests
- Multi-vendor interoperability tests
- Performance tests
- Stress tests

Complete multi-vendor interoperability is outside the scope of typical conformance testing. Industry support could be demonstrated by voluntary participation in interoperability testing events organized by standards organizations, or a committee of active, CDSA developers.

CDSA bases integrity of the runtime environment on signature verification of a CDSA component's object code and other signed credentials. Object code signing is inherently platform-specific. To ensure that the signature of a service provider module can be correctly checked on all instances of a specific base-platform type, there must be a standard signing mechanism defined and used to sign all object code modules for that base-platform type. Without this standard, executable modules must be platform-provider specific, which is more constraining that being specific only to the base-platform type. CDSA defines the integrity service interfaces to perform signing and verifying on all platforms. CDSA reference implementations pave the way for the standardization of object code signing mechanisms for each base-platform type.

Chapter 2

Common Security Services Manager

This section provides details on the main infrastructure component of the CDSA, the Common Security Services Manager (CSSM).

2.1 Overview

The Common Security Services Manager integrates the security functions required by applications to use cryptographic service provider modules (or tokens) and certificate libraries. In particular, it facilitates linking digital certificates to cryptographic actions and trust protocols. Tokens and certificate libraries plug into the CSSM as add-in modules.

Functionally, CSSM provides the services shown in Figure 2-1:

- General module management services—install, dynamically attach, and dynamically locate module managers and add-in modules.
- Elective module managers—dynamically extend the APIs and security services available to applications implemented to use those services.
- Basic module managers—define a minimal set of security services APIs.
- Multi-service modules—allow a single add-in service module to implement services to functionally separate sets of CSSM APIs.
- Integrity Services—verify signed credentials to ensure trusted identification and authorizations.
- Security context management—aggregate and manage input and output parameters required when performing cryptographic operations.

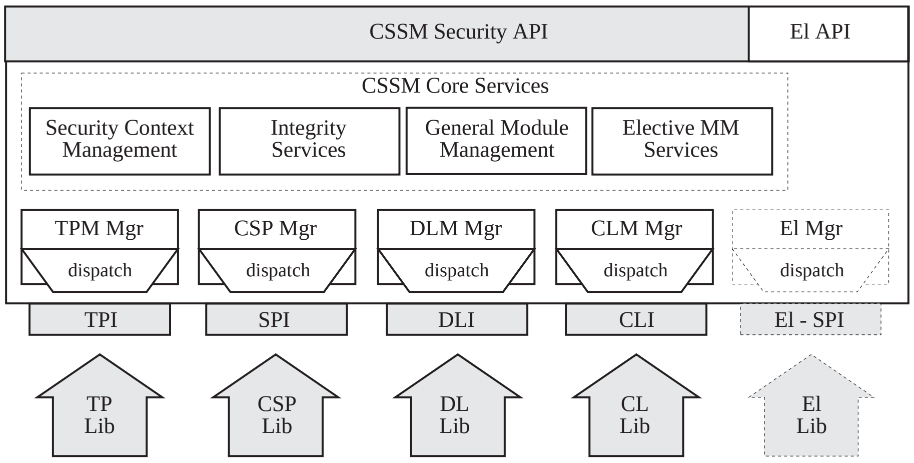

Figure 2-1 Services Provided by CSSM

2.2 General Module Management Services

CSSM manages a registry that records each component's logical name, attached components in the CSSM environment. CSSM manages a registry that records information about each installed add-in module and elective module manager. This information can be queried by applications, add-in modules, and components of CSSM. The registry is CSSM's critical information base. CSSM must protect this information base by controlling access to the information, (particularly write access), and checking the integrity of stored values upon retrieval.

The CSSM registry records the logical name of each add-in module and elective module manager, the information required to locate and dynamically initiate the component, and some minimal meta-data describing the capabilities and services implemented by the component. An add-in module may or may not implement all of the APIs defined by CSSM. Unimplemented functions are registered as null. For extensibility, an add-in module can implement additional functions outside of the CSSM-defined API calls. CSSM defines a single *pass-through* function, which an add-in module can overload with multiple custom functions. The meaning and use of these functions is documented outside of CSSM by the module vendor.

CSSM APIs allow an application to query the registry of installed (known) add-in modules to determine the availability of various security services. Applications must be able to search the registry by module name and by features and capabilities. This allows applications to select specific vendor's modules or to select any module that provides the desired services. Once an applicable module is located in the registry information, an application uses the CSSM *attach* operation to load and initiate the module.

For each attach call, CSSM creates a unique attach handle to identify the logical connection between the application and the add-in module. CSSM maintains a separate context state for each attach operation. This enables non-cooperating threads of execution to maintain their independence, even though they may share the same process space.

When dynamically loading components, CSSM ensures the integrity of the expanding system using the CSSM Integrity Services Library. (The complete set of integrity services are described in more detail later in this section.) When a module is loaded and initiated, it must present a digitally-signed credential, such as a certificate, to identify its author and publisher. The signature represents the module provider's attestation of ownership and a guarantee that the module conforms to the CSSM specification. CSSM checks the authenticity of the module's credentials and the integrity of the module's code before attaching the module to the CSSM execution environment.

Once the module has been loaded into the CSSM runtime environment, CSSM exchanges state information with the application and with the module. This allows CSSM to act as a broker between the application and a set of add-in modules. An excellent example of this brokerage service is CSSM's memory usage model. Often cryptographic operations and operations on certificates make pre-calculation of memory block sizes difficult and inefficient. CSSM rectifies this problem through registration of application memory allocation callback functions. CSSM and attached add-in modules use the applications memory functions to create complex or opaque objects in the application's memory space. Memory blocks allocated by an add-in module and returned to the application can be freed by the application using its chosen free routine.

When an application no longer requires a module's services, the add-in module can be detached. An application should not invoke this operation unless all requests to the target module have been completed. Modules can also be uninstalled. This operation removes the module name and its associated attributes from the CSSM's registry. Uninstall must be performed before a new version of the module is installed in the CSSM registry.

2.3 Elective Module Managers

To ensure long-lived utility of CDSA and CSSM APIs, the architecture includes several extensibility mechanisms. Elective module managers is a transparent mechanism supporting the dynamic addition of new categories of service. For example, key recovery can be an elective service. Some applications will use key recovery services (by explicit invocation) and other applications will not use it. Audit logs can be an elective service. Applications wishing to maintain a log can do so, other applications will not use that facility.

2.3.1 Transparent, Dynamic Attach

Applications are not explicitly aware of module managers. Applications are aware of instances of add-in modules. Before requesting services from an add-in service provider (via CSSM API calls), the application invokes *attach* to obtain an instance of the add-in service provider. Figure 2-2 shows the sequence of processing steps. If the module is of an elective category, then CSSM transparently attaches the module manager for that category of service (if that manager is not currently loaded). Once the manager is loaded, the APIs defined by that module are available to the application.

The dynamic nature of the elective module manager is transparent to the add-in module also. This is important. It means that an add-in module vendor should not need to modify their module implementation to work with an elective module manager, versus a basic module manager.

There is at most one module manager for each category of service loaded in CSSM at any given time. When an elective module manager is dynamically added to service an application, that module is a peer of all other module managers and can cooperate with other managers as appropriate.

When an attached application detaches from an add-in service module, CSSM will also unload the associated module manager if it is not in use by another application.

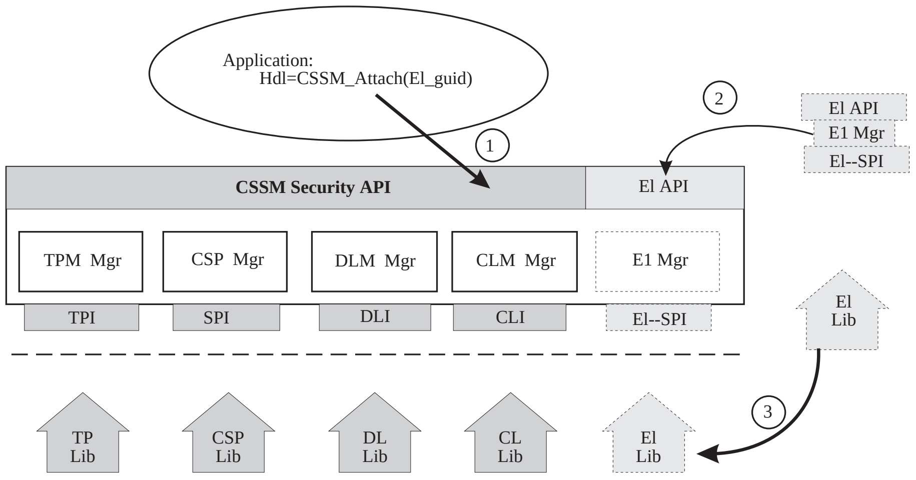

Figure 2-2 Processing Steps to Attach an Add-In Module

and Load its Elective Module Manager

2.3.2 Registering Module Managers

Module managers are installed and registered with CSSM in a similar manner to add-in modules. CSSM records module manager information in the CSSM registry. This information can be queried, but typically only system administration applications will use registry information about module managers. For example, a smart installer for an add-in module may check to see that the corresponding module manager is also installed on the local system. If not, then the installer can also install the required module manager. This does not effect the implementation of the add-in module itself, just the install program for that module.

2.3.3 State Sharing Among Module Managers

Module managers may be required to share state information in order to correctly perform their services. When two or more module managers share state, each manager must be able to:

- Inform the other module managers of its presence in the system
- Request notification of certain states or activities taking place in the domain of another module manager
- Gather event information from other module managers
- Inform the other module managers of its imminent removal from the system

The other module managers must be able to:

- Change their behavior based on the presence or non-presence of another module manager in the system
- Accept and honor requests from other module managers for ongoing state and activity information
- Issue event notifications to other module managers when selected events occurs

When module managers share state information they must implement conditional logic to interact with each other. Different mechanisms can be used to share state information:

- Invoking known, internal, module manager interfaces
- Using operating system supported state-sharing mechanisms, such as shared memory, RPC, event notification, and general interrupts
- Using a CSSM supported event notification service

The first two mechanisms depend on platform services outside of CDSA. Module managers that share state information can use all of these mechanisms.

CSSM-supported event notifications requires that all module managers implement and register with CSSM an event notification entry point. Module managers issue notifications by invoking a CSSM function, specifying:

- The source manager
- The destination manager
- The event type
- Notification ID (optional)
- Data Values (optional)

CSSM delivers the notification to the destination module manager by invoking the manager's notification entry point.

Typical event types include:

- Module manager loaded
- Module manager unloaded
- Selected Service Request
- Reply

Module managers that share state information are not required to use the CSSM event notification mechanism. These types of events, requests, and notifications can be shared using the other platform dependent mechanisms. CSSM provides this simple mechanism specifically for situations where other platform services are not readily available.

2.4 Basic Module Managers

CDSA defines module managers for four basic types of service:

- Cryptographic Services Module Manager
- Certificate Library Module Manager
- Data Storage Library Module Manager
- Trust Policy Module Manager

These service categories are considered basic because we believe that all applications using security services must use these services. Cryptographic services are the heart of security services and protocols. Identity, authentication, and integrity are embodied in digital credentials (such as certificates). A user's certificates must be persistently stored for use as long-term credentials. Policies will exist for how and when the credentials can be used. A security-aware application that does not use these services is unusual.

CSSM maintains these module managers in the system at all times and exports their respective APIs to all applications. Elective module managers export their APIs to applications on demand. When active in the CSSM environment, all modules managers are peers, all are managed uniformly by CSSM, and all may cooperate and coordinate with each other as required to perform their tasks.

2.5 Dispatching Application Calls for Security Services

Multiple add-in modules of each type may be concurrently active within the CSSM infrastructure. CSSM module managers use unique handles to identify and maintain logical connections between an application and attached service modules. The handle maintains the state of the connection, enabling add-in modules to be re-entrant. When an application invokes the CSSM API, the module manager who exports the invoked API dispatches the call to the appropriate module by invoking the corresponding Service Provider Interface (SPI) supported by the add-in module. Figure 2-3 shows how managers dispatch function calls to attached add-in modules.

In Figure 2-3, the application invokes func1 in the cryptographic module identified by the handle CSP1. A dispatcher forwards the function call to func1 in the CSP1 module. The application also invokes func7 in the trust policy module, identified by the handle TP2. A dispatcher forwards the function call to func7 in the TP2 module. The implementation of func7 in the TP2 module uses functions implemented by a certificate library module. The TP2 module must invoke the certificate library functions via the dispatching mechanism. To accomplish this, the TP2 module attaches the certificate library module, obtaining the handle CL1, and invokes func13 in the certificate library identified by the handle CL1. A dispatcher forwards the function call to func13 in the CL1 module.

Calls to the CSSM security API can originate in an application, in another add-in security module, or in CSSM itself. The dispatching mechanism forwards all calls uniformly, regardless of their origin. CSSM ensures access to CSSM internal structures is serialized through thread synchronization primitives. If CSSM is implemented as a shared library then process synchronization primitives are also employed. Add-in modules need not have multi-threaded implementations to interoperate with CSSM. Multi-threaded capabilities are registered with CSSM at module install time. Access to non-multithreaded add-ins is serialized by CSSM.

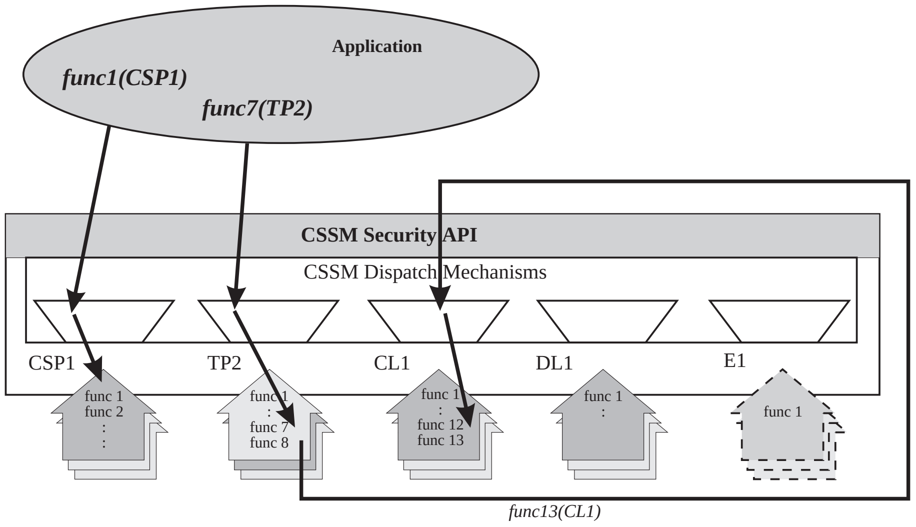

Figure 2-3 CSSM Dispatches Calls to Selected Add-In Security Modules

Modules must be loaded before they can receive function calls from a dispatcher. An error condition occurs if the invoked function is not implemented by the selected module.

2.6 Integrity Services

CSSM provides a set of integrity services used by CSSM, module managers, add-in modules, and applications to verify the integrity of themselves and other components in the CSSM environment. The dynamic, configurable environment defined by CDSA and supported by CSSM provides the level of service and flexibility that applications require. In balance with the benefits are the increased risk of introducing tampered components into the environment. To address this, CSSM provides a set of integrity verification and identity verification functions. CSSM also requires their use during each dynamic reconfiguration of the CDSA environment.

2.6.1 CSSM-Enforced Integrity Verification

CDSA checks the integrity of modules as they are dynamically attached to the system. A bilateral authentication procedure is designed for two entities to establish trust in the identity and integrity of each other. When attaching an add-in module or an elective module manager, CSSM requires that the attaching party participates in a bilateral procedure to verify the identity and the integrity of both parties. If authentication fails, the module is not attached and system execution is interrupted.

Both parties in the bilateral procedure must have three pieces of signed credentials:

- A certificate, signed with a valid, recognized manufacturer
- A manifest object that aggregates all of the sub-components and attributes describing the capabilities of the component, signed with the component's certificate
- A set of object code modules, signed with the component's certificate

These credentials are stored in the local file system and are associated with the component. CSSM's credentials are also placed on the system during installation. The credentials of a dynamic component are placed on the system when that component is installed with CSSM.

As part of attach processing, CSSM performs the first half of the bilateral protocol, which proceeds as follows:

- Locate the component's credentials in the local system
- Verify the component's certificate and manifest
- Load the component's object code and verify it signature
- Determine that the component's initial entry point is within the checked object code (ensuring secure linkage) and invoke the verified component

The component completes the authentication procedure as follows:

- Self-check the object code signature
- Locate CSSM's credentials in the local system
- Verify CSSM's certificate and manifest
- Verify the object code signature for the loaded CSSM
- Determine that your return address for CSSM is within the checked CSSM object code (ensuring secure linkage)
- Complete attach processing and return to CSSM

When the three credentials verify, it is still necessary to ensure secure linkage between the components. For the CSSM, this entails checking that the called address is in fact in the appropriate code module. For the attaching component, the return address must be verified to be within the CSSM calling module. (Even in the case of self-checking, one may require that the return address be within the module being checked.)

Linkage checks prevent attacks of the *stealth* class, where the object being verified is not the object that is being used. Also, the checks increase the difficulty of the *man-in-the-middle* attack, where a rogue component will insert itself between two communicating modules, masquerading itself as the other component to each component.

Bilateral authentication should also be performed between applications and CSSM. This requires a manufacturing, installation and start-up process in which applications can:

- Create credentials of the same form as add-in modules and elective module managers
- Voluntarily place their credentials on the local system during application installation
- Perform their half of the bilateral authentication process with CSSM

Applications that do not want to implement or cannot implement the entire bilateral procedure can still benefit from CSSM integrity services by invoking selected CSSM integrity functions to:

- Perform a self-integrity check
- Check the identity and integrity of CSSM
- Check the identity and integrity of add-in service modules

2.7 Creating Checkable Components

The integrity of a CDSA component is based on verification of a digital signature on that component. The identity of a CDSA component is based on verification of a certificate belonging to that component. To verify a certificate and the signature of an object module requires that these credentials be created as part of the manufacturing process.

The enhanced, off-line manufacturing process for all dynamic components of CDSA is as follows:

- Issue the component's certificate—this identifies the component, its author, its publisher and defines the components capabilities. This certificate must be signed with a CSSM-recognized certificate owned by the manufacturer.
- Uses the certificate to digitally sign all software routines comprising the component—this tightly binds what the component is (for example, the software that represents it) with the identity and authority defined in the certificate.

When manufactured in this manner, the identity and integrity of the component can be checked. Applications that wish to present credentials for privileged services or to be authenticated by CSSM must follow an analogous manufacturing process.

2.7.1 Verifying Components

CSSM provides signature verification functions to authenticate the manufacturer as the author and publisher of the binary object and determine whether or not the CSSM object was modified after it was signed. Signature verification requires the use of public keys. Public keys are public information stored in certificates. They are not secrets to be protected, but they must be protected from modification. If replaced with an impostor's public key, an unauthorized component could pass the integrity check and be erroneously added to the system.

CSSM must provide verification services without assuming any central authority as the universal base of trust. Software vendors can cross-license with other vendors using their *digital signature.* These root keys can be provided to CSSM integrity services. CSSM can perform authentication based on these additional roots of trust only if the keys are signed and that signature can be verified by CSSM based on previously known roots of trust. By using certificates to introduce new vendors, the number of verifiable vendors need not be limited.

The verification tests can be applied as a self-check or to check another component in the CSSM environment. Periodic, runtime re-checks can be performed to verify constancy of a component's integrity. If tampering is detected in any component, the verification function will interrupt system execution.

Verification services are available for use on demand by add-in modules, module managers, applications, and CSSM itself.

2.8 Security Context Services

Security Context Services creates, initializes, and maintains concurrent security contexts. A security context is a run-time structure containing security-related execution parameters, and potentially secrets of an application process or thread. The structure aggregates the numerous parameters an application must specify when requesting a cryptographic operation.

Once cryptographic contexts have been created the application may freely use those contexts without CSSM-imposed security checks. Security contexts may contain secrets, such as encryption keys, passphrases and passphrase functions. Applications are responsible for protecting these secrets. Applications desiring maximal protection should use passphrase callback functions that limit the duration in which the passphrase is present in the system.

Applications retain handles to each security context used during execution. The context handle is a required input parameter to many security service functions. Most applications instantiate and use multiple security contexts. Only one context may be passed to a function, but the application is free to switch among contexts at will, or as required (even per function call).

A knowledgeable CSP-aware application initializes the security context structure with values obtained by querying CSSM to obtain the capabilities of the Cryptographic Services Provider (CSP) from the CSSM Registry.

An application may create multiple contexts directly or indirectly. Indirect creation may occur when invoking layered services, system utilities, trust policy modules, certificate library modules, or data storage library modules, that create and use their own appropriate security context as part of the service they provide to the invoking application. <REFERENCE UNDEFINED>(fig05) shows an example of a hidden security context. An application creates a context specifying the use of sec_context1. The application invokes func1 in the certificate library using sec_context1 as a parameter. The certificate library performs two calls to the cryptographic service provider. For the call to func5, the hidden security context is used. For the call to func6, the application's security context is passed as a parameter to the CSP.

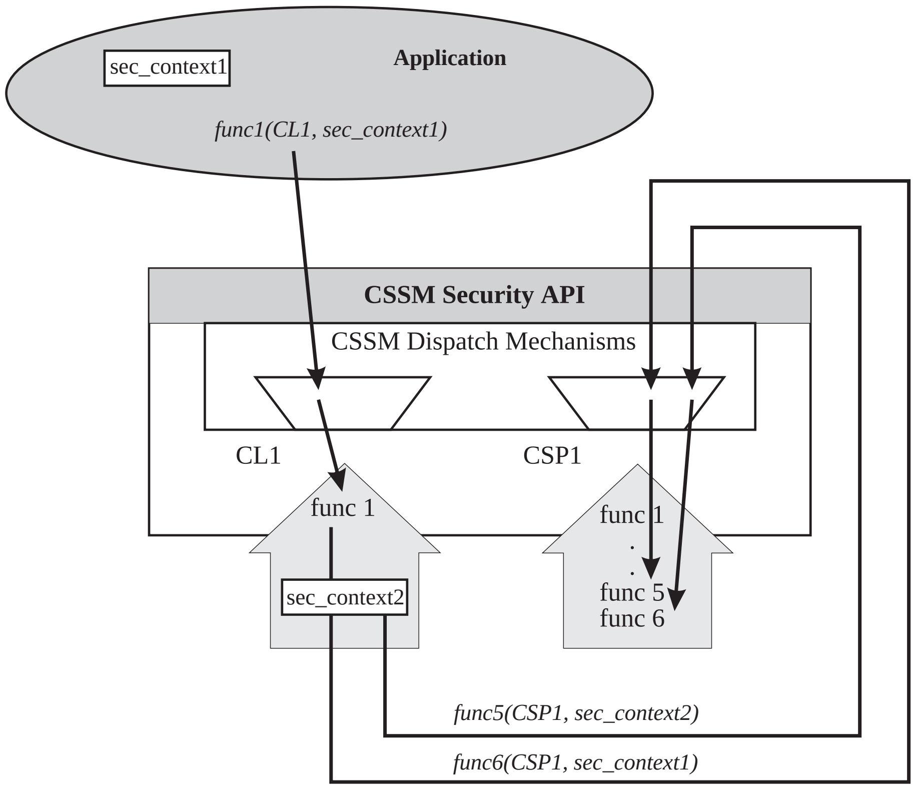

Figure 2-4 Indirect Creation of a Security Context

These transparent contexts do not concern the application developer, as they are managed entirely by the layered service or add-in module that creates them. Each process or thread that creates a security context is responsible for explicitly terminating that context.

Security context management provides mechanisms that:

- Allow an application to use multiple CSPs concurrently
- Allow an application to concurrently use different parameters for a single CSP algorithm
- Support layered implementations in their transparent use of multiple CSPs or different algorithm parameters for the same CSP
- Enable development of re-entrant CSPs
- Enable development of re-entrant layered services
- Enable development of re-entrant applications

Chapter 3

Cryptographic Service Provider Modules

The CSSM infrastructure doesn't implement general purpose cryptography. It has been termed "crypto with a hole." The Cryptographic Services Manager provides applications with access to cryptographic functions that are implemented by Cryptographic Service Provider (CSP) modules. This centralizes all the cryptography into exchangeable modules.

The nature of the cryptographic functions contained in any particular CSP depends on what task the CSP was designed to perform. For example, a VISA cryptographic hardware token would be able to digitally sign credit card transactions on behalf of the card's owner. A digital employee badge would be able to authenticate a user for physical or electronic access.

A CSP can perform one or more of these cryptographic functions and services:

- Bulk encryption and decryption
- Digital signing and verification
- Cryptographic hash
- Key-pair generations
- Random number generator
- Encrypted storage of private keys

Every CSP must provide secured storage of private keys. Applications may query the CSP to retrieve private keys stored within the CSP. The CSP is responsible for controlling access to the private keys it secures. The application must prove it is authorized to use the private key. A passphrase is used for this purpose. It can be provided by a callback function invoked by the CSP and implemented by the requester to identify and authorize the user or process requesting the private key. Most CSPs are capable of importing private keys created by other CSPs and providing secured storage for such keys.

Applications can create complex execution models for interacting with one or more CSPs, while a given CSP implementation can have a much simpler execution model. For example, an application could attach to the same CSP multiple times with different threads of execution each time. Each thread would get the appearance of having exclusive access to the CSP. Meanwhile the CSP may be implemented according to a single-threaded model. Additionally, the CSP may be managing multiple installed cards or multiple portable card slots on the system. An application may attach to the same CSP once for each card, as it looks like a different CSP, even though there is a single instance of the CSP attached to the CSSM.

Most applications use the CSSM CSP-APIs directly to request cryptographic operations. Applications also use CSP services indirectly through the certificate-based services of another add-in module (such as a trust policy).

3.1 CSP Form Factor

No particular form factor is assumed for a CSP. CSPs can be instantiated in hardware, software or both. Operationally, the distinction must be transparent. The two visible distinctions between hardware and software implementations are the degree of trust the application receives by using a given CSP, and the cost of developing that CSP. A hardware implementation should be more tamper-resistant than a software implementation. Hence a higher level of trust is achieved by the application.

Cryptographic service providers, whose capabilities may change after installation, may make dynamic requests to update CSSM registration information. The dynamic nature of removable and software-loadable cryptographic service providers is supported by CDSA.

Software CSPs are convenient and portable. Software CSPs can be carried as an executable file on common forms of removable media. The components that implement a CSP must be digitally signed, to authenticate their origin and integrity. This requirement extends to composite implementations involving both software and hardware. Multiple CSPs may be loaded and active within the CSSM at any time. A single application may use multiple CSPs concurrently. Interpreting the resulting level of trust and security is the responsibility of the application or the trust policy module used by the application.

3.2 Legacy CSPs

CSPs existed prior to the definition of the CSSM Cryptographic API. These legacy CSPs have defined their own APIs for cryptographic services. These interfaces are CSP-specific, nonstandard, and (in general) low-level, key-based interfaces. They present a considerable development effort to the application developer attempting to secure an application by using those services.

CSSM defines a high-level, certificate-based API for cryptographic services to support application development. The Cryptographic Services Module Manager defines a lower-level Service Provider Interface (SPI) that more closely resembles typical CSP APIs, and provides CSP developers with a single interface to support.

Embracing legacy CSPs, the CSSM architecture defines an optional adaptation layer between the Cryptographic Services Module Manager and a CSP. The adaptation layer allows the CSP vendor to implement a shim to map the CSSM SPI to the CSP's existing API, and to implement any additional management functions that are required for the CSP to function as an add-in module in the extensible CSSM architecture. New CSPs may support the CSSM SPI directly (without the aid of an adaptation layer).

A CSP may or may not support multi-threaded applications.

3.3 Cryptographic Service Provider Registration

Each CSP registers a description of its functions and services with CSSM. Applications query this information to select appropriate CSPs for their use. CSPs with dynamic capabilities will not register this information with CSSM. When no capability description is provided, CSSM will refresh information about the CSP whenever an application queries the CSPs registry. In this fashion an application can poll a CSP to become informed of a change in its status.

It is anticipated that some CSP add-in modules will span SPI functional boundaries. For example, a smart card may also register as a data storage module that contains certificates and credentials in tamper-resistant storage.

3.4 Cryptographic Services API

The security services API defined by the Cryptographic Service Providers Module Manager (CSPMM) is certificate-based. This contrasts with the approach taken by many CSPs, where low-level concepts such as key type, key size, hash functions, and byte ordering are the standard granularity of interface options. The CSPMM hides these behind high-level operations such as:

- SignData
- VerifyData
- DigestData
- EncryptData
- DecryptData
- GenerateKeyPair
- GenerateRandom
- WrapKey

Security-conscious applications use these high-level concepts to provide authentication, data integrity, data and communication privacy, and nonrepudiation of messages to the end-users.

The CSP may implement any algorithm. For example, CSPs may provide one or more of the following algorithms, in one or more modes:

- Bulk encryption algorithm: DES, Triple DES, IDEA, RC2, RC4, RC5, Blowfish, CAST
- Digital signature algorithm: RSA, DSS
- Key negotiation algorithm: Diffie / Hellman
- Cryptographic hash algorithm: MD4, MD5, SHA
- Unique identification number: hard-coded or random-generated
- Random number generator: attended and unattended
- Encrypted storage: symmetric-keys, private-keys

The application's associated security context defines parameter values for the low-level variables that control the details of cryptographic operations. Applications use CSPs that provide the services and features required by the application. For example, an application issuing a request to EncryptData may reference a security context that defines the following parameters:

- The algorithm to be used (such as RC5)

- Algorithm-specific parameters (such as key length)
- The object upon which the operation is conducted (such as filename)
- The cryptographic variables (such as the key)

Most applications will use default (predefined) contexts. Typically a distinct context will be used for encrypting, hashing, and signing. For a given application, once initialized, these contexts will change little (if at all) during the application's execution, or between executions. This allows the application developer to implement security by manipulating certificates, using previously-defined security contexts, and maintaining a high-level view of security operations.

Application developers who demand fine-grained control of cryptographic operations can achieve this by directly and repeatedly updating the security context to direct the CSP for each operation, and by using the Cryptographic Services API *pass-through* feature.

The pass-through feature allows a highly knowledgeable application to call low-level CSP functions that are not available through to the common Cryptographic API. The CSPMM will either reject the call or pass it through to the selected CSP. The CSPMM will not alter the result of the request, or generate other side effects based on the request. The philosophy of CDSA and the numerous services provided by CSSM is to reduce the need for applications to work at this low level.

3.5 Additional CSP Services

Unique services

Application processes may use the unique cryptographic services provided by a CSP via a pass-through capability in the Cryptographic Services API. The parameters to the pass-through interface include a security context, a CSP-specific function name, and the arguments to the function. After determining the authorization for the call (based on the invoking process, the CSP selected by the security context, and the specific function requested), the call is passed through to the specified CSP. The application process is responsible for the correctness of the arguments supplied to the call.

Key management

Every CSP is responsible for implementing its own secure, persistent storage and management of private keys. To support chains of trust across application domains, CSPs must support importing and exporting both public and private keys. This means transferring keys among remote and possibly foreign systems. The ability to transfer keys assumes the ability to convert one key format into any other key format, and to secure the transfer of private and symmetric keys.

Each CSP is responsible for securely storing the private keys it generates or imports from other sources. Additional storage-related operations include retrieving a private key when given its corresponding public key, and wrapping private keys as key blobs for secure exportation to other systems.

Note that each CSP will create and manage its own private-key database. If an application requires that more than one CSP perform operations using the same private key, then that key must be exported from some source and imported to all CSPs needing to use it. Wrapping keys as key blobs manages the problem of different key formats among different CSPs. This assumes that the key length is acceptable to all CSPs using the same key.

Each CSP defines and implements its own key-management functions. Recent CSP implementations, such as Microsoft's Crypto API, define internal storage formats and key-blob wrappers for exporting keys outside of the CSP. CSPs will exchange private keys through secured communication protocols (such as wrappers), rather than through access to a shared database for private keys.

The CSPMM API defines how private keys will be passed up and down through the layers of the CDSA, but it does not specify how private keys will be stored within the CSP.

Chapter 4

Trust Policy Modules

A digital certificate binds an identification in a particular domain to a public key. When a certificate is issued (created and signed) by the owner and authority of a domain, the binding between key and identity is attested by the digital signature on the certificate. The issuing authority also associates a level of trust with the certificate. The actions of the user, whose identity is bound to the certificate, are constrained by the trust policy governing the certificate's usage domain. A digital certificate is intended to be an unforgeable credential in cyberspace.

The use of digital certificates is the foundation on which the CDSA is designed. The CDSA assumes the concept of digital certificates in its broadest sense. Applications use the credential for:

- Identification
- Authentication
- Authorization

How applications interpret and manipulate the contents of certificates to achieve these ends is defined by the real-world trust model the application has chosen as its model for trust and security.

The primary purpose of a Trust Policy (TP) module is to answer the question "Is this certificate trusted for this action?" The CSSM Trust Policy API defines the generic operations that should be defined for certificate-based trust in every application domain. The semantics of each operation are defined by the:

- Application domain
- Policy statement for a domain
- Certificate type
- Real-world operation the user requests within the application domain

The trust model is expressed as an executable policy used / invoked by all applications ascribing to that policy and the trust model it represents.

As an infrastructure, CSSM is policy neutral; it does not incorporate any single policy. For example, the verification procedure for a credit card certificate should be defined and implemented by the credit company issuing the certificate. Employee access to a lab housing a critical project should be defined by the company whose intellectual property is at risk. Rather than defining policies, CSSM provides the infrastructure for installing and managing policy-specific modules. This ensures complete extensibility of certificate-based trust on every platform hosting CSSM.

Policy describing the intended use of security objects such as private keys, credentials and certificates requires flexible mechanisms. The CDSA trust policy module can support trust policy interpreters such as PolicyMaker. The implementation of a policy interpreter may rely heavily on the CL and DL modules for certificate parsing and policy object storage.

Trust policy statements can also be expressed concisely as action tags that assert actions a principal is authorized to perform. Fixed trust policy action assertions enable simpler expression of trust policy which makes policy inspections straightforward. These actions are primary operations on the basic objects common to almost all trust models. These include certificates, credentials and certificate revocation lists. The basic operations on certificates are sign, verify, and revoke.

Based on this analysis, CSSM defines two categories of API calls that should be implemented by TP modules. The first category allows the TP module to define and expose actions specific to the trust domain (such as requesting authorization to make a $200 charge on a credit card certificate, and requesting access to the locked project lab). The second category specifies basic operations (for example, sign, verify, and revoke) on certificates and certificate revocation lists.

Application developers and trust domain authorities benefit from the ability to define and implement policy-based modules. Application developers are freed from the burden of implementing a policy description and certifying that their implementation conforms. Instead, the application only needs to build in a list of the authorities and certificate issuers it uses.

Domain authorities also benefit from an infrastructure that supports add-in trust policy modules. Authorities are sure that applications using their module(s) will adhere to the policies of the domain. Also, dynamic download of trust modules (possibly from remote systems) ensures timely and accurate propagation of policy changes. Individual functions within the module may combine local and remote processing. This flexibility allows the module developer to implement policies based on the ability to communicate with a remote authority system. This also allows the policy implementation to be decomposed in any convenient distributed manner.

Implementing a trust policy module may or may not be tightly coupled with one or more certificate library modules and one or more data storage Library modules. The trust policy embodies the semantics of the domain. The certificate library and the data storage library embody the syntax of a certificate format and operations on that format. A trust policy can be completely independent of certificate format, or it may be defined to operate with one or a small number of certificate formats. A trust policy implementation may invoke a certificate library module and/or a data storage library module to manipulate certificates.

4.1 Trust Policy Services API

The CSSM TP-API defines three categories of API calls:

- Determine if a user is/was trusted to perform an application-specific operation on some object at a specified time
- Determine trusted access to CSSM objects such as certificates and CRLs
- Semantic manipulation of groups of related certificates

The following descriptions present general, recommended semantics. The specific semantic implemented by the TP module is defined by the specific trust model it represents.

Determining trust for an application-specific operation. Each trust policy is specific to an application domain. The policy should support all applications in that domain. This includes an understanding of all domain-specific operations and the authorizations required to perform the operations in that domain. Determination is based on the caller's certificate. Policy evaluation may require remote processing. If the caller is authorized to perform the operation, the TP module can be designed to perform the operation on the caller's behalf, or the TP module can return an affirmative response granting the caller permission to act.

Determining trust for accessing CSSM objects. There are operations that are global to all application domains. These operations involve the manipulation of CSSM-recognized objects, such as certificates and CRLs. Trust evaluation is required to manipulate these objects. For example, a caller can present the TP module with a newly-released CRL for its use. Should the TP module trust the caller who is providing this CRL? Accepting or rejecting CRL updates is common to all application domains, and as such, is not a natural part of the verification functions discussed above.

Manipulating groups of related certificates. All trust domains assign trust in one credential based on trust in another. This is a property of hierarchical trust models and introducer trust models. The TP module manipulates groups of semantically-related certificates. In the hierarchical trust model, a certificate is signed by one certificate / key and a group of certificates can be arranged in a rooted hierarchy based on this signature relationship. In the introducer trust model, certificates can include signatures from multiple parties. These certificates are be arranged in a graph based on this signature relationship. The primary operations to be performed are constructing a set of certificates related by a specific semantic, decomposing a set of related certificates, and verifying that groups of certificates are related based on a given semantic. The semantic is usually known to the TP module a priori. For example, if the application domain served by the trust policy supports only the hierarchical model, then verifying a group of certificates means to check the certificate-signatures to confirm that they form a signed, hierarchical certificate chain. If the trust policy supports an introducer model, then verifying a group of certificates could mean checking for a minimum number of recognized signatures on one or more certificates in the group.

Chapter 5

Certificate Library Modules

The primary purpose of a Certificate Library (CL) module is to perform memory-based, syntactic manipulations on the basic objects of trust: certificates and certificate revocation lists (CRLs). The data format of a certificate will influence (if not determine) the data format of CRLs used to track revoked certificates. For this reason, these objects should be manipulated by a single, cohesive library. Certificate library modules incorporate detailed knowledge of data formats. The Certificate Library Services Manager defines API calls to perform security operations, (such as signing, verifying, revoking, viewing, and so on) on memory-resident certificates and CRLs. The mechanics of performing these operations is tightly bound to the data format of a given certificate. One or more modules may support the same certificate format, such as X.509 DER-encoded certificates, SDSI certificates, and SPKI certificates.

As new standard formats are defined and accepted by the industry, certificate library modules will be defined and implemented by industry members and used directly and indirectly by many applications. Certificate library modules perform syntactic manipulations of certificate and CRL data objects. The semantic interpretation of certificate and revocation security objects are considered policy transformations and are implemented as trust policy modules.

Certificate library modules manipulate memory-based objects only. The persistence of these objects is an independent property. It is the responsibility of the application and/or the trust policy module to use data storage add-in modules to make these objects persistent (if appropriate). The storage mechanism used by a data storage module may be independent of other modules.

Application developers and trust policy module developers both benefit from the extensibility of add-in certificate library modules. Applications are free to use multiple certificate types, without requiring the application developer to write format-specific code to manipulate certificates and CRLs. Without increased development complexity, multiple certificate formats can be used on one system, within one application domain, or by one application. CAs who issue certificates also benefit. Dynamically downloading certificate libraries ensures timely and accurate propagation of data-format changes.

5.1 Certificate Library API

The Certificate Library Services API defines operations on memory-resident certificates and certificate revocation lists (CRLs) as required by every certificate type. These operations include:

- Creating new certificates and new CRLs
- Signing existing certificates and existing CRLs
- Viewing certificates
- Verifying certificates and CRLs
- Extracting values (such as public keys) from certificates
- Importing and exporting certificates of other data formats
- Revoking certificates
- Reinstating revoked certificates

- Transitioning through all phases of the certificate and key life cycle
- Searching certificate revocation lists
- Pass-through for unique, format-specific certificate and CRL operations

Every certificate library (CL) module should implement most if not all of these functions. A *pass-through* function is also defined by the certificate library API. This allows CLs to provide additional functionality if required to manipulate the certificate data format and CRL data format supported by the module.

Some of the functions can be implemented as remote services. For example, creating a certificate can be performed by a remote Certificate Authority (CA). Remote operations have an impact on the specification of the CL-API when the operation could require asynchronous completion.

The following is a brief description of the CSSM-recommended semantics of certificate library functions. The data format-dependent manipulation of certificates and CRLs is implemented by the CL module developer.

Creating new Certificates and new CRLs. The CL issues a signed, memory-resident certificate containing values specified by the caller and values specified by the issuing process. The CL also creates an initialized, but empty, memory-resident CRL. Revocation records can be added to the CRL using the CL module's revocation function.

Signing Certificates and CRLs. The signing function is used to add subsequent signature to a signed certificate. Subsequent signatures can represent a local endorsement of the certificate. For example, in the introducer model of trust, a local policy may define that once a certificate has been verified, it is re-signed using a locally trusted certificate. This can significantly reduce the effort required for future verifications of the certificate. This is akin to the function of a notary public. When signing certificates, the CL computes the digital signature over a certificate or a CRL, and includes the newly-generated signature in the memory-resident copy of the certificate or CRL. CL modules may forward signing requests to external signing authorities or perform them locally. The CL module may use the services of a CSP add-in module to calculate the signature. Many certificate formats define fields that must be excluded from the signature calculation.

For example, management fields whose state must change over time without invalidating the certificate cannot be included in the signature calculation. The CL module uses its knowledge of the certificate data format to include only those certificate fields that must not change during the life of the certificate. Signing memory-resident CRLs is performed in the same manner.

Verifying Certificates and CRLs. Mechanically verifying one or more signatures associated with a certificate or CRL depends on the format of the signed objects. The CL module embodies knowledge of which subset of the object's fields were included in the signing process. Regardless of data format, every CL module must test the integrity of the signature. This means that the object associated with that signature has not been modified since the signature was calculated.

Typically the CL module will invoke a CSP to recalculate the one-way hash of the certificate or CRL, decrypt the signed hash value, and compare it with the calculated hash complete the verification process. Depending on the fields stored in the certificate or the CRL, additional checks may be required to complete syntactic verification.

Extracting Values (such as Public Keys) from Certificates. Applications and trust policy modules may need selected values from a certificate. Extracting field values depends on the certificate data format. The CL module must extract and return, to the caller, any requested field value.

Importing and Exporting Certificates. When presenting a certificate to an application or sending a certificate from one system to another, a data format translation is often required. A full-service CL module should provide format translation functions that import certificates of foreign format to the library's particular format. Also the reverse translation should be provided.

These import and export functions allow applications to more easily accept certificates in one of several distinct formats by converting the foreign format to the format typically processed by the application. CRLs are typically stored only in the CL module's native format. It is assumed that CRLs are exchanged only among systems that support the same CRL format.

Revoking and Reinstating Certificates. A certificate may be permanently or temporarily revoked for a number of reasons. Some certificate formats include one or more revocation status fields. In this case, the CL module will mark the certificate as revoked. When revoking a certificate, the CL module will typically add a revocation record to the supplied CRL.

To ensure the integrity of the revocation record, it should be digitally signed, using the private key associated with the revoking agent's certificate. To reinstate a temporarily revoked certificate, the revocation record must be removed from the CRL. If fields in the certificate itself were modified to indicate the revoked state, these certificate values must also be updated and a renewed certificate is issued.

Searching Certificate Revocation Lists. Certificate revocations must be reported to all systems that may receive the certificate as a security credential. To avoid constant online revocation checks, CRLs are distributed periodically to all systems that need to verify certificates possibly contained in the revocation list. When a user presents a certificate to an application, the application must verify that certificate.

Certificate verification (by a CL module) includes a check to ensure the certificate in not revoked. This test requires a search of all CRLs that may contain the certificate in question. A CL module must support searches over a CRL, selecting revocation records based on selection criteria appropriate to the CRL data format.

Chapter 6

Data Storage Library Modules

The primary purpose of a Data Storage Library (DL) module is to provide secure, persistent storage and retrieval of security-relevant data objects such as certificates, certificate revocation lists (CRLs), keys and policy assertions. The persistence of these generic trust objects is independent of the memory-based manipulations performed by certificate library modules. DL modules may be invoked by applications, trust policy modules, or certificate library modules that make decisions about the persistence of these objects.

A single DL module may cooperate with a Certificate Library (CL) module or may independently manage the persistence of opaque objects. A data storage library that is coupled with a certificate library module may use information obtained from the CL in the implementation of a physical data storage and retrieval model. For example, a DL might interrogate a certificate to construct/use indexes to speed data retrieval.

Each DL module can manage any number of independent, physical data stores. Each data store must have a logical name used by callers to refer to the persistent data store. Implementation of the DL module may use local file system facilities, commercial database management systems, custom stable storage devices and remote storage facilities. A data storage module may execute and store its data locally or remotely.

Properties and capabilities of each storage device are managed by the DL Module Manager. Applications may hold multiple references to a single storage device. The DL manages open references as context handles to internal data structures and physical media. DL modules are not restricted from using caching or other performance optimization techniques. Processes and threads may access common physical storage devices, however, device handles unique, opaque values, which are not storage device-specific.

A DL module is responsible for the integrity of the objects it stores. If the DL module uses an underlying commercial database management system (DBMS), it may choose to further secure the data store by leveraging integrity services provided by the DBMS. DL module designers must choose which mechanisms best address the availability, integrity, privacy and performance needs of the perceived customer. For example tamper-resistant storage devices and encryption could be used to protect secret objects. Local and remote redundant storage devices could facilitate integrity and availability, while caching could improve performance.

The persistent objects managed by a DL module have semantic typing associated with them. This semantic information is used to describe the object’s intended use. For example certificate objects whose corresponding private key is local to the system would receive the semantic label of "owned". Other certificates may be trusted as a root of authority or as a cross-certified entity. These would receive the semantic label of "root". Applications and TP modules may use the semantic information when manipulating and evaluating objects that are semantically related.

6.1 Data Storage Library Registration

CSSM defines API calls for installing and registering of DL modules. The CSSM records each DL module name and capability description. This information enables applications to select a DL module appropriate for their needs. For example, a DL module built on top of an X.500 directory service may indicate different naming and usage semantics than for file system-based storage. Other capabilities such as structured query language support, removable media and latent operation, are also traits registered with the DL module.

DL modules may have capabilities that change after installation and registration. For example remote or removable data stores' capabilities may change during regular operation of the DL module. In this case, the DL module cannot know what functions it supports until the application requests capabilities of a particular data store. The same situation held for Cryptographic Service Providers. A DL module can be dynamically queried by CSSM to obtain these dynamic module capabilities.

6.2 Data Storage Library API

The data store management functions operate on a data store as a single unit. These operations include:

- Opening and closing data stores
- Creating and deleting data stores
- Importing and exporting data stores

The persistence operations on data stores include:

- Inserting
- Updating
- Deleting
- Retrieving
- Module-specific operations

A data store may contain a single object type or multiple object types. DL modules will register which object types the DL is capable of storing at installation time, or whenever an application polls for capabilities information.

Creating and Deleting Data Stores. The DL module creates a new, empty data store and opens it for future access by the caller. An existing data store may be deleted. Deletion discards all data contained in the data store. Deletion will not occur if there are outstanding open references to a data store. Data store creation also involves specifying an access schema and setting other configuration information. This includes describing indexed fields and fields requiring unique database keys.

Opening and Closing Data Stores. The DL module manages the mapping of logical data store names to physical storage mechanisms. The caller uses logical names to reference persistent data stores. The open operation initializes physical storage mechanisms and associates a context to the logical storage facility. The close operation terminates current access to the data store and cleans up any temporal state created during initialization and operation.

Importing and Exporting Data Stores. Local data stores may be moved from one system to another or from one storage medium to another storage medium. The import and export operations support the transfer of an entire data store. The export operation prepares a snapshot of a data store. (Export does not delete the data store it snapshots.)

The import operation accepts a snapshot (generated by the export operation) and creates a new data store or adds the data records to an existing data store managed by the DL module.

The following is a brief description of the CSSM-recommended semantics of data storage library functions. The persistence mechanisms are implemented by the DL module developer.

Inserting Objects. The DL module adds a persistent copy of the supplied object to an open data store. This operation may include updating index entries or other components of the physical data model. The mechanisms used to store and retrieve persistent objects is specific to the implementation of the DL module and transparent to applications.

Updating Objects. The DL module updates objects when the inserted object already exists in the data store and the meta-data indicates unique key space. In the case of non-unique key spaces, the inserted object is appended to the data store. It is anticipated that applications will store and retrieve multiple objects using the same key. For example an application may associate several identity certificates, several policy objects and possibly a key object with a user name. A query for the user name would result in all the user's objects being returned. Updates to these objects must be done by deleting and then re-adding to the data store.

Deleting Objects. The DL module removes the specified objects from the data store.

Retrieving Objects. Applications and add-in security modules need to search persistent data stores for objects specified in the query. The DL module must provide a search mechanism for retrieving a copy of selected persistent objects. A selection predicate controls the query. Selection predicates may be expressed as a string of structured language or as data structures of a query tree.

Chapter 7

Multi-Service Modules

CSSM APIs are logically partitioned into functional categories. The goal of this logical partitioning is to assist application developers in understanding and making effective use of the security APIs. To this end, the partitioning has been effective.

Vendors providing add-in security service modules are developing products that provide services in more than one functional category. Vendors may not want to partition their products in this manner. More pointedly, they can be unable to do so. Consider a class 2 PKCS#11 cryptographic device. This device performs cryptographic operations and provides persistent storage for keys, certificates, and other security-related objects. These services are logically partitioned between the CSP-APIs and the DLM-APIs. Implementing two separate add-in modules is not feasible. In order to provide correct service, the two modules must share execution state, such as PKCS#11 session identifiers. Additional examples exist, as shown in Figure 7-1.

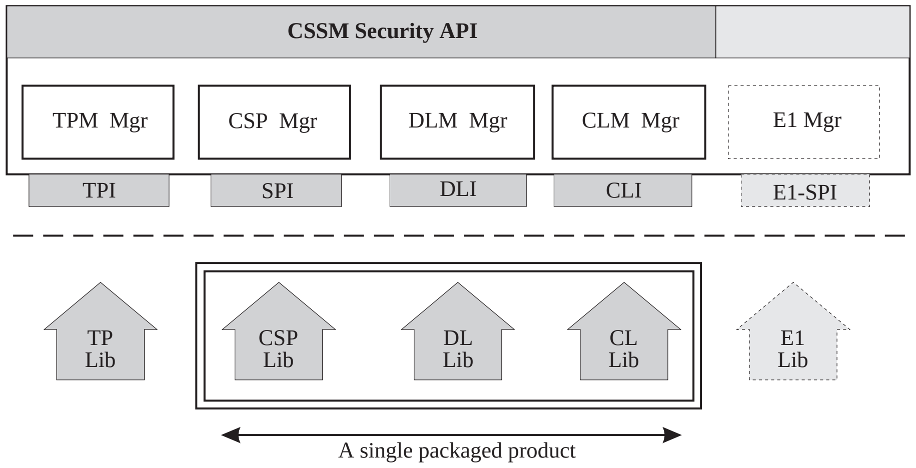

Figure 7-1 A Multi-Service, Add-In Module

Serving Three Logical, Functional Categories

Multi-service add-in modules separate module packaging from the application developers functional view of CSSM APIs. A multi-service module is a single, dynamic add-in module that implements CSSM functions from two or more functional categories of the CSSM APIs.

7.1 Application Developer's View of a Multi-Service Add-in Module

Application developers must have some (but limited) visibility into the organization of the service provider modules available through the CSSM framework. Knowledge of underlying implementations should be kept to a minimum.

Applications attach a multi-service module as they would any other module. The attach function returns a handle representing a unique pairing between the caller and the attached module. The caller uses this single handle to obtain any and all types of services implemented by the attached module. Figure 7-2 shows the handle for an attached PKCS#11 service provider that performs cryptographic operations and persistent storage of certificates.This single handle can be used as the CSPHandle in cryptographic operations and as the DLHandle in data storage operations.

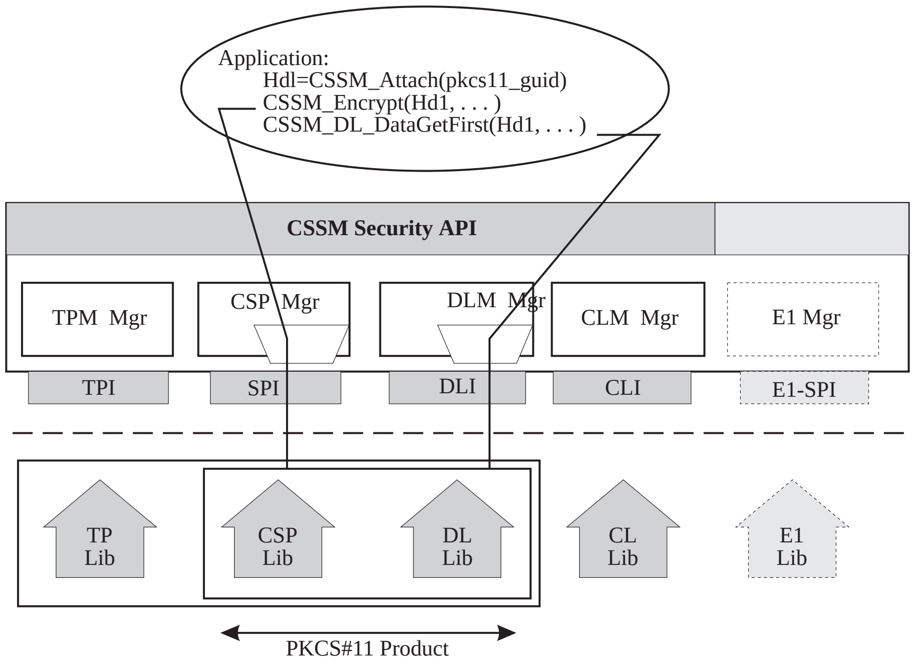

Figure 7-2 A Single Handle References a Multi-Service Add-In Module

Multiple calls to attach are viewed as independent requests. Each attach request returns separate, independent handles that do not share execution state.

Before attaching a service module, an application can query the CSSM registry to obtain information about that module. A multi-service module has exactly one CSSM registry entry containing multiple capability descriptions. There are one or more capability descriptions per functional category supported by the module. Each set of capabilities includes a type identifier to distinguish CSPinfo from Clinfo, and so on.

7.2 Service Provider's View of a Multi-Service Add-in Module

A Multi-Service Module is a single product. It has a single associated globally-unique identifier (GUID). It's implementation may consist of several libraries, forming a single service.

When an add-in module is installed on a CSSM system, the module registers its name, GUID, and capability descriptions with CSSM. CSSM securely records this information in the CSSM registry (making it available for application queries). A multi-service module will register capabilities for each of the service categories supported by the module.

A multi-service module is not required to implement all of the functions in any functional categories. The CSSM dispatching mechanism invokes only to those interfaces registered with the CSSM.

7.3 Companion Modules

It can also be useful for a set of separate modules that interoperate to declare their interoperability. Such modules are referred to as "companion modules". Each can be a multi-service or a single service module. Modules register a list of companion modules with CSSM during module installation. CSSM records this in the CSSM registry with other information about the module. Applications can query this information about companion modules. The list is optional and if supplied, it is strictly advisory. Applications may ignore or use this information to their advantage. For example, a trust policy module for SET applications may register a companion CL module that manipulates DER-encoded X.509 certificates. Similarly, a DLM that implements access to an X.500 directory service may register the same CL module as a companion.

Chapter 8

System Security Services

The System Security Services layer is the appropriate architectural layer for defining and implementing sophisticated security protocols, based on the security services of the CSSM and its add-in modules. These services and protocols may include:

- Secure and private file systems (such as PFP secured files)
- Protocols for secure electronic commerce (such as JEPI and SET)
- Protocols for private communication (such as SHTTP, SSL, PGP, and S / MIME)
- Multi-language access to the CSSM API (such as CSSM-Java API)
- CSSM management tools (such as a CSSM installation and configuration tool)

CAE Specification

Part 2:

Common Security Services Manager (CSSM)

The Open Group

Chapter 9

Introduction

This chapter provides:

- An overview of the Common Data Security Architecture
- An overview of the Common Security Services Manager Application Programming Interface specification

9.1 Common Data Security Architecture

The Common Data Security Architecture (CDSA) defines the infrastructure for a comprehensive set of security services to address the needs of individual users and the business enterprise. CDSA is an extensible architecture that provides mechanisms to manage add-in security service modules. These modules provide cryptographic services and certificate services for use in building secure applications. Figure 9-1 shows the four basic layers of the Common Data Security Architecture: Applications, System Security Services, the Common Security Services Manager, and Security Add-in Modules. The Common Security Services Manager (CSSM) is the core of CDSA. It provides a means for applications to directly access security services through the CSSM security API, or to indirectly access security services via layered security services and tools implemented over the CSSM API. CSSM manages the add-in security modules and re-directs application calls through the CSSM API to the selected add-in modules that will service the request.

This four layer architecture defines four categories of basic add-in module security services. Basic services are required to meet the security needs of all applications. CSSM also supports the dynamic inclusion of APIs for new categories of security services, required by selected applications. These elective services are dynamically, and transparently added to a running CSSM environment when required by an application. Elective services are required by only a subset of security aware applications. When an elective service is needed a module manager for that category of service can be transparently attached to the system followed by the requested add-in service module. Once attached to the system, the elective module manager is a peer with all other CSSM module managers. Applications interact uniformly with add-in modules of all types.

The four basic categories of security services modules are:

- Cryptographic Service Providers (CSP)
- Trust Policy Modules (TPM)
- Certificate Library Modules (CLM)
- Data Storage Library Modules (DLM)

Cryptographic Service Providers (CSPs) are add-in modules that perform cryptographic operations including encryption, decryption, digital signaturing, key pair generation, random number generation, and key exchange. Trust Policy (TP) modules implement policies defined by authorities, institutions, and applications, such as your Corporate Information Technology Group (as a certificate authority), MasterCard* (as an institution), or Secure Electronic Transfer (SET) applications. Each trust policy module embodies the semantics of a trust environment based on digital credentials. A certificate is a form of digital credential. Applications may use a digital certificate as an identity credential and/or an authorization credential. Certificate Library (CL) modules provide format-specific, syntactic manipulation of memory-resident digital certificates and certificate revocation lists. Data Storage Library (DL) modules provide persistent storage for certificates, certificate revocation lists, and other security-related objects.

Examples of elective security service categories are key recovery and audit logging.

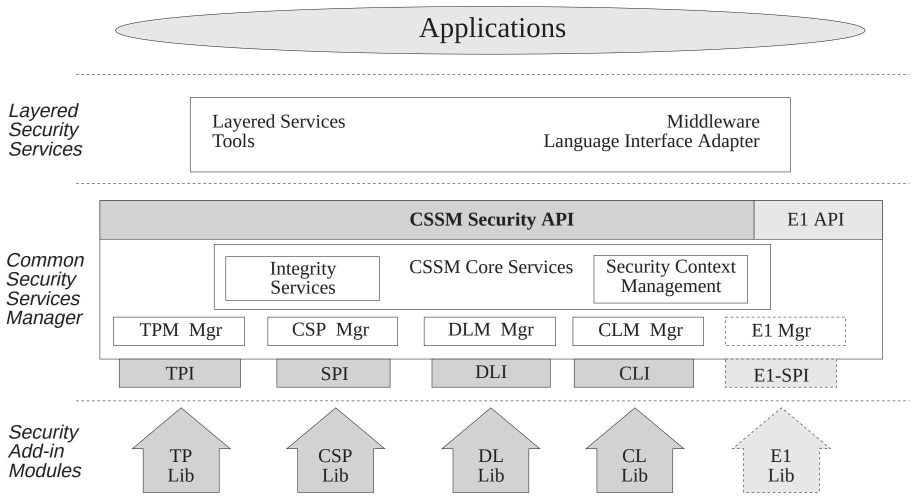

Figure 9-1 The Common Data Security Architecture for all Platforms

Applications dynamically select the modules used to provide security services. These add-in modules can be provided by independent software and hardware vendors. A single add-in module can provide services in multiple categories of service. These are called multi-service modules.

The majority of the CSSM APIs support service operations. Service operations are functions that perform a security operation, such as encrypting data, adding a certificate to a certificate revocation list, or verifying that a certificate is trusted and/or authorized to perform some action.

Modules can also provide services beyond those defined by the CSSM API. Module-specific operations are enabled in the API through pass-through functions whose behavior and use is defined by the add-in module developer. (For example, a CSP implementing signaturing with a fragmented private key can make this service available as a pass-through.) The PassThrough is viewed as a proving ground for potential additions to the CSSM APIs.

CSSM core services support:

- Module management
- Security context management
- System integrity services

The module management functions are used by applications and by add-in modules to support module installation, registration of module features and attributes, and queries to retrieve information on module availability and features.

Security context management provides runtime caching of user-specific, cryptographic state information. Multi-step cryptographic operations, such as staged hashing, require multiple calls to a CSP. Intermediate operation state must be managed. CSSM manages this state information for the CSP, enabling more CSPs to easily support multiple concurrent callers.

The CSSM Embedded Integrity Services Library (EISL) provides tamper resistant verification services. CSSM, add-in modules, and optionally applications use EISL to check the identity and integrity of components of CDSA. Checkable components include: add-in service modules, CSSM itself, and in the future applications that use CSSM. The EISL services focus on detecting impostors or unauthorized components in the system and tampering of authorized components.

In summary, the direct services provided by CSSM through its API calls include:

- Comprehensive, extensible SPIs for each of four categories of security services
- Registration and management of all add-in security service modules available to applications
- Registration and management of elective module managers providing other security services
- Caching of runtime state for cryptographic operations
- Call-back functions used by add-in modules and CSSM to interact with an application process
- Notification services to inform add-in modules of selected actions taken by an application
- An Integrity Services Library providing tamper resistant test-and-check services for CDSA components
- Management support for concurrent security operations

Chapter 10

Core Services API

10.1 Overview

The CSSM provides a set of core service APIs for:

- Core Services for CSSM Management
- Module Management
- Memory Management Support (described in more detail in Appendix B)
- Security Context Management (described in Chapter 11)
- Integrity Verification Services

These APIs are implemented by the CSSM, not by add-in modules.

10.2 Core Services for CSSM Management

CSSM provides functions for managing multiple instances of CSSM. These instances can be distinct versions of CSSM or multiple copies of the same instance of CSSM. Applications can select which instance of CSSM to use at runtime. Three pieces of information help to identify each instance of a CSSM executable:

- A unique identification GUID, which distinguishes the CSSM executable itself and its manufacturer
- An interface-GUID, which distinguishes the APIs and architectural features supported by that instance of CSSM
- Major and minor version numbers, which further distinguishes the supported APIs, feature set, and bug fixes

Applications can use this descriptive information to assist in selecting the appropriate CSSM instance.

Every CSSM instance must specify its unique identification GUID. Specification of the interface GUID and the version numbers is optional, but believed to be of value as an augmentation to the distinguished name for an executable instance of CSSM. Using these three pieces of information can determine interoperability and compatibility with an instance of CSSM.

Applications use the CSSM_GetInfo interface to obtain this identification information for any instance of CSSM that has been installed on a local system. Once an instance has been selected, the application must load that instance using CSSM_Load. Following the dynamic load operation, the application must perform all required initialization steps before using other CSSM services. Initialization includes invoking CSSM_Init and the optional exchange of application credentials for purposes of authentication between CSSM and the application.

10.2.1 Module Management Services

The CSSM module management functions support module installation, dynamic selection and loading of modules, and querying of module features and status.

System administration utilities use CSSM install and uninstall functions to maintain add-in modules on a local system.

The CSSM registry records information about each installed add-in module and elective module manager for the local system. The registry is CSSM's critical information base. CSSM must support the following services and features with respect to the CSSM registry:

- Persistently store values identifying and describing each dynamic component installed with CSSM
- Retrieve of information from the registry upon request
- Ensure the integrity of the stored/retrieved values
- Control write access to stored values

The registry entries are queried by applications, add-in modules, and components of CSSM.

Applications select the particular security services they will use by selectively attaching add-in modules. These modules are provided by independent vendors. Each has an assigned, Globally Unique ID (GUID), and a set of descriptive attributes to assist applications in selecting appropriate modules for their use. A module can implement a range of services across the CSSM APIs (such as, cryptographic functions and data storage functions) or a module can restrict its services to a single CSSM category of service (such as, certificate library services only). Modules that span service categories are called Multi-Service modules.

Applications use a module's GUID to specify the module to be attached. The attach function returns a handle representing a unique pairing between the caller and the attached module. This handle must be provided as in input parameter when requesting services from the attached module. CSSM uses the handle to match the caller with the appropriate service module.

The calling application uses the handle to obtain any and all types of services implemented by the attached module. Figure 10-1 shows how the handle for an attached PKCS#11 service provider is used to perform cryptographic operations and persistent storage of certificates. The single handle value can be used as the CSPHandle in cryptographic operations and as the DLHandle in data storage operations.

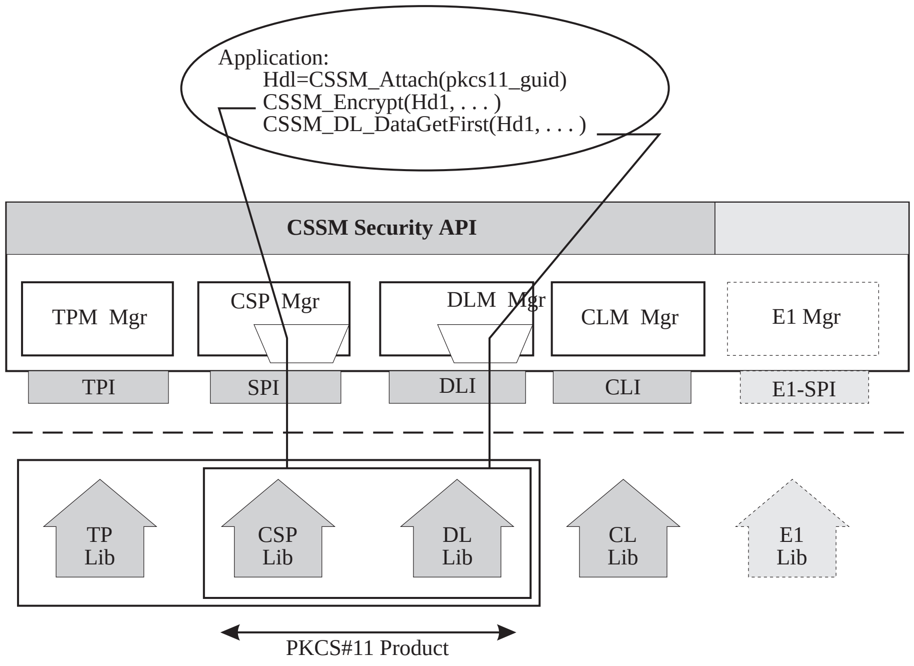

Figure 10-1 Application Using Cryptographic Services and Persistent Storage Services of a Class 2, PKCS#11 device

Multiple calls to attach are viewed as independent requests. Each attach request returns separate, independent handles that do not share execution state.

Before attaching a service module, an application can query the CSSM registry to obtain information on:

- The modules installed on the system
- The capabilities (and functions) implemented by those modules
- The GUID associated with a given module

Applications use this information to select a module for use. A multi-service module has multiple capability descriptions associated with it, at least one per functional area supported by the module. Some areas (such as CSP and TP) may have multiple independent capability descriptions for a single functional area. There is one CSSM registry entry for a multi-service module. That entry records all service types for the module. CSSM returns all information about a module’s capabilities when queried by the application. Each set of capabilities includes a type identifier to distinguish CSPinfo from Clinfo, and so on.

Applications can query about CSSM itself. Different versions of CSSM and add-in modules will exist. CSSM provides several functions to assist applications in selecting a version that meets the application’s needs. One function returns version information about the running CSSM. Another function verifies whether the application’s expected CSSM version is compatible with the currently-running CSSM version. The general function to query add-in module information also returns the module’s version information.

10.2.2 Memory Management Support

The CSSM memory management functions are a class of routines for reclaiming memory allocated by CSSM on behalf of an application from the CSSM memory heap. When CSSM allocates objects from its own heap and returns them to an application, the application must inform CSSM when it no longer requires the use of that object. Applications use specific APIs to free CSSM-allocated memory. When an application invokes a free function, CSSM can choose to retain or free the indicated object, depending on other conditions known only to CSSM. In this way CSSM and applications work together to manage these objects in the CSSM memory heap.

10.2.3 Integrity of the CSSM Environment

As a security framework, CSSM provides each application with additional assurance of the integrity of the CSSM execution environment With dynamic link-loading of add-in service modules, viruses and other forms of impersonation are common threats. CSSM defines and enforces an umbrella integrity policy that reduces the risk of these threats.

At module attach time, CSSM requires successful certificate-based trust verification for:

- All add-in service modules
- All elective module managers

All verifications performed to enforce CSSM-defined policy are based on CSSM-selected public root keys as points of trust.

When CSSM performs a verification check on any component in the CSSM environment, the verification process has three aspects:

- Verification of identity using a certificate chain naming the component's creator or manufacturer
- Verification of object code integrity based on a signed hash of the object code
- Tightly binding the verified identity with the verified object code

These steps are implemented by CSSM' s Integrity Services. Integrity Services are packaged as a static library called the Embedded Integrity Services Library (EISL). CDSA defines a bilateral authentication procedure by which CSSM and a component interacting with CSSM authenticate each other to achieve a mutual trust.

As part of bilateral authentication, CSSM calls EISL to verify and load a module or a module manager. If EISL returns a failure condition, then the module or the module manager has not been linked and loaded. CSSM must detect this failure and must return the value CSSM_ATTACH_ERROR to the caller of the CSSM_ModuleAttach operation.

EISL services support unilateral authentication, identity verification, and object code integrity checks. EISL facilities are documented in the *CSSM Embedded Integrity Services Library API Spec*.

10.2.4 Module-Defined Usage Policies

Service module vendors may wish to provide enhanced services to selected applications or classes of applications. A module-defined policy is in addition to the CSSM's general integrity policy.

Module-defined policies are enforced by one of the following authentication checks:

- CSSM authenticates the application that is requesting the module attach, based on CSSM trust points.
- CSSM authenticates the application that is requesting the module attach, based on module-specified trust points.
- The add-in module authenticates the attached application, based on module-specified trust points.

The module specifies its policy by selecting one of these authentication checks. Options one and two use CSSM to enforce the module-defined policy during attach processing. Option three is carried out independently by the add-in module, using EISL services. The add-in module requests CSSM enforcement by setting MODULE_FLAGS corresponding to options one and two in the MODULE_INFO structure. When option two is selected, the MODULE_INFO structure should also contain a set of module-specific, public root keys corresponding to the module's points of trust.

The MODULE_INFO structure is presented to CSSM during module installation in two forms:

- As an attribute value in the service module's signed credentials
- As information for the CSSM registry

The policy is securely stored in the signed credentials. These credentials are authenticated by CSSM each time the module is attached. CSSM uses the signed policy description as the authoritative representation of the policy. The MODULE_INFO structure is also stored in the CSSM registry allowing applications to read the policy description by calling CSSM_GetModuleInfo.

Add-in modules can independently authenticate applications based on module-defined points of trust. The application must incorporate a verifiable certificate in its credentials. To authenticate the application directly, the add-in module:

- Locates the application's credential files using information passed to the add-in module during attach processing
- Invokes EISL facilities to verify the application credentials based on module-defined roots of trust

An application's verifiable credentials must be created during application manufacturing. The application vendor must obtain a manufacturing/signing certificate from all service module vendors and CSSM vendors who will provide it with privileged status. The application vendor uses the manufacturing certificates to create the certificate chains shown in Figure 10-2. The application must carry all of these certificate chains in the signature block for its persistent credentials. When the application calls CSSM_ModuleAttach on a service module for which it has been granted special privileges, CSSM or the service module can verify at least one of the certificate chains in the application's manifest signature block based on CSSM-defined or module-defined roots of trust.

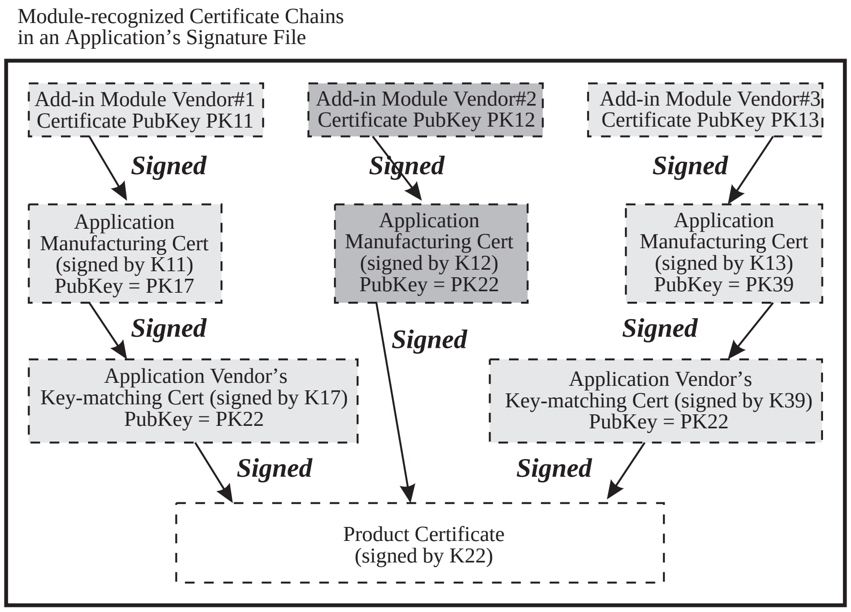
Module-recognized Certificate Chains
in an Application's Signature File
Add-in Module Vendor#1 Certificate PubKey PK11
Add-in Module Vendor#2 Certificate PubKey PK12
Add-in Module Vendor#3 Certificate PubKey PK13
Signed
Signed
Signed
Application Manufacturing Cert (signed by K11) PubKey = PK17
Application Manufacturing Cert (signed by K12) PubKey = PK22
Application Manufacturing Cert (signed by K13) PubKey = PK39
Signed
Signed
Signed
Application Vendor's Key-matching Cert (signed by K17) PubKey = PK22
Application Vendor's Key-matching Cert (signed by K39) PubKey = PK22
Signed
Signed
Product Certificate (signed by K22)

Figure 10-2 Three Module-Specific Certificate Chains

representing the application's module-specific credentials
for three distinct modules or module vendors

10.2.5 Application-Authenticated Add-In Modules

An application vendor or an application installer can define and enforce a policy that precludes the end-user from using non-authorized add-in service modules. This policy is in addition to CSSM's general integrity policy.

Application-defined policies are enforced by one of the following authentication checks:

- CSSM authenticates the attach-target add-in module based on CSSM trust points.
- CSSM authenticates the attach-target add-in module based on application-specified trust points.
- The application authenticates the attached add-in module, based on application-specified trust points.

The application specifies its policy by selecting one of these authentication checks. Options one and two use CSSM to enforce the application-defined policy during attach processing. Option one is part of the umbrella integrity policy defined and enforced by CSSM. This check is always performed by CSSM. Options two and three are checks performed in addition to the CSSM check. Option three is carried out independently by the application, using EISL services. The application requests CSSM enforcement by setting APP_SERVICE_FLAGS corresponding to option two in the APP_SERVICE_INFO structure. The APP_SERVICE_INFO structure should also contain a set of application-specific, public root keys corresponding to the application's points of trust.

The APP_SERVICE_INFO structure must be flattened and stored as an attribute value in the application's signed credentials. These credentials are used during module attach and provide verifiable information to CSSM for application-directed authentication of the attached add-in module.

Alternatively, applications can independently authenticate selected add-in service modules based on application-defined points of trust. This authentication procedure is in addition to procedures automatically performed by CSSM based on CSSM-defined roots of trust. By performing a second authentication, an application vendor or an application installer can preclude the use of non-authorized add-in modules. The policy defining which add-in modules an application is authorized to use is specified and maintained outside of the CSSM.

The add-in service module must incorporate a verifiable certificate in its credentials. The application must locate the module's credential files and verify them directly by invoking EISL facilities. A module's credentials can be located using file system path information published in the module's CSSM registry entry. (CSSM registry information can be retrieved using the CSSM_GetModuleInfo function.) Verification using EISL facilities must be based on application-defined roots of trust.

An application defines its roots of trust for authenticating add-in modules by one of two methods:

- Application adopts existing module credentials—the add-in module vendor creates a certificate, adds the certificate to the signature file of the signed manifest associated with the add-in module, and publishes the certificate with a directory service or in product documentation. This certificate should be part of a verifiable chain with the product signing certificate. The application adopts the published certificate as a root of trust.
- Application issues signing certificates to the module vendor—the application vendor creates and issues a signing certificate to the add-in module vendor. The application adopts the issued certificate as a root of trust. The add-in module vendor adds the application-issued certificate as part of a verifiable certificate chain in the signature file of the module's signed manifest.

In either case, the application defines a root of trust that can be used to verify the add-in module. The add-in module incorporates a verifiable certificate in its credentials and the application can use CSSM's EISL facilities to authenticate the add-in module based on the application's (adopted or issued) root of trust. The add-in module's credentials will appear as shown in Figure 10-3.

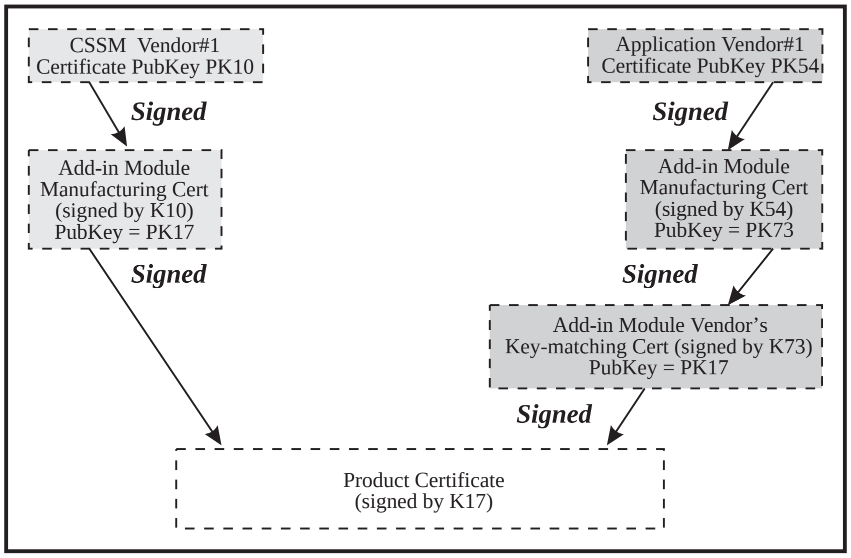

Figure 10-3 Module credentials with app-specific chain

To authenticate the add-in module using EISL, the application proceeds as follows:

1. Call CSSM_GetModuleInfo to obtain the pathname and filename for the target module.
2. Construct the name of the module's associated credentials (using the module's pathname and filename).
3. For each trusted public root key that could authenticate the target add-in module, call ISL_VerifyLoadedModuleAndCredentials specifying the name of the module's credentials and the trusted public root key, until you find one that verified or all keys fail to verify.

It is important to note that the application-defined roots of trust for authenticating add-in service modules are independent of the module-defined roots of trust for authenticating applications.

A service module's verifiable credentials must be created during module manufacturing. The module vendor must obtain a manufacturing/signing certificate from all application vendors who wish to enforce exclusive use of the service module by their application. The module vendor uses the manufacturing certificates to create the certificate chains shown in Figure 10-3. The module must carry all of these certificate chains in the signature block for its persistent credentials. The application or CSSM can successfully verify at least one of the certificate chains in the module's credentials based on CSSM-defined or application-defined roots of trust.

10.2.6 Application Exemptions

CSSM and the CSSM module managers implement a small number of built-in checks for normal controlled functioning of security services. Applications must be able to request exemption from these built-in checks. Exemption is granted if the caller provides credentials that:

- Are successfully authenticated by CSSM
- Carry implied authorization for the requested exemptions

Exemptions can be granted per application thread, if threads are supported in the operating environment. Exemption privileges can not be inherited by spawned processes or spawned threads. Each process or thread must present credentials and obtain its own exemption status.

The CSSM_RequestCssmExemption function is used to request exemptions. Applications can invoke this function at any time after invoking the CSSM_Init function. This allows applications to change exemption status as appropriate during execution. Authentication and implied authorization are checked by CSSM at each request.

A bit mask represents the set of requested exemptions.

New elective module managers can define and implement additional built-in checks. Exemption categories, with corresponding bit mask values, should be defined by the elective module manager. This allows authorized applications to be exempt from these additional built-in checks.

10.3 Data Structures for Core Services

10.3.1 CSSM_BOOL

This data type is used to indicate a true or false condition.

```
typedef uint32 CSSM_BOOL;
#define CSSM_TRUE 1
#define CSSM_FALSE 0
```

Definition

CSSM_TRUE
Indicates a true result or a true value.

CSSM_FALSE
Indicates a false result or a false value.

10.3.2 CSSM_RETURN

This data type is used to indicate whether a function was successful.

```
typedef enum cssm_return {
    CSSM_OK = 0,
    CSSM_FAIL = -1
} CSSM_RETURN
```

Definition

CSSM_OK
Indicates operation was successful.

CSSM_FAIL
Indicates operation was unsuccessful.

10.3.3 CSSM_STRING

This is used by CSSM data structures to represent a character string inside of a fixed-length buffer. The character string is expected to be NULL-terminated. The string size was chosen to accommodate current security standards, such as PKCS #11.

```
#define CSSM_MODULE_STRING_SIZE 64
typedef char CSSM_STRING [CSSM_MODULE_STRING_SIZE + 4];
```

10.3.4 CSSM_DATA

The CSSM_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory. This memory must be allocated and freed using the memory management routines provided by the calling application via CSSM. Trust policy modules and certificate libraries use this structure to hold certificates and CRLs. Other add-in service modules, such as CSPs, use this same structure to hold general data buffers, and DLMs use this structure to hold persistent security-related objects.

```
typedef struct cssm_data{
    uint32 Length; /* in bytes */
    uint8 *Data;
} CSSM_DATA, *CSSM_DATA_PTR
```

Definition

Length
Length of the data buffer in bytes.

Data
Points to the start of an arbitrary length data buffer.

10.3.5 CSSM_GUID

This structure designates a global unique identifier (GUID) that distinguishes one add-in module from another. All GUID values should be computer-generated to guarantee uniqueness (the GUID generator in Microsoft Developer Studio* and the RPC UUIDGEN/uuid_gen program on a number of UNIX* platforms can be used).

```
typedef struct cssm_guid{
    uint32 Data1;
    uint16 Data2;
    uint16 Data3;
    uint8 Data4[8];
} CSSM_GUID, *CSSM_GUID_PTR
```

Definition

Data1
Specifies the first eight hexadecimal digits of the GUID.

Data2
Specifies the first group of four hexadecimal digits of the GUID.

Data3
Specifies the second group of four hexadecimal digits of the GUID.

Data4
Specifies an array of eight elements that contains the third and final group of eight hexadecimal digits of the GUID in elements 0 and 1, and the final 12 hexadecimal digits of the GUID in elements 2 through 7.

10.3.6 CSSM_VERSION

This structure is used to represent the version of CDSA components.

```
typedef struct cssm_version {
    uint32 Major;
    uint32 Minor;
} CSSM_VERSION, *CSSM_VERSION_PTR
```

Definition

Major
The major version number of the component.

Minor
The minor version number of the component.

10.3.7 CSSM_SUBSERVICE_UID

This structure uniquely identifies a set of behaviors within a subservice within a CSSM add-in module.

```
typedef struct cssm_subservice_uid {
    CSSM_GUID Guid;
    CSSM_VERSION Version;
    uint32 SubserviceId;
    uint32 SubserviceFlags;
} CSSM_SUBSERVICE_UID, *CSSM_SUBSERVICE_UID_PTR;
```

Definition

Guid
A unique identifier for a CSSM add-in module.

Version
The version of the add-in module.

SubserviceId
An identifier for the subservice within the add-in module.

SubserviceFlags
An identifier for a set of behaviors provided by this subservice.

10.3.8 CSSM_HANDLE

A unique identifier for an object managed by CSSM or by an add-in module.

```
typedef uint32 CSSM_HANDLE, *CSSM_HANDLE_PTR
```

10.3.9 CSSM_MODULE_HANDLE

A unique identifier for an attached service provider module.

```
typedef uint32 CSSM_MODULE_HANDLE
```

10.3.10 CSSM_LIST_ITEM

This structure is used to encapsulate the name and GUID of an add-in module.

```
typedef struct cssm_list_item{
    CSSM_SUBSERVICE_UID SubserviceUid;
    char *Name;
} CSSM_LIST_ITEM, *CSSM_LIST_ITEM_PTR
```

Definition

SubserviceUid
The global, persistent, unique identifier of the module.

Name
The name of the module.

10.3.11 CSSM_LIST

This structure is used to encapsulate an array of CSSM_LIST_ITEMs, where the array length is given by the NumberItems variable.

```
typedef struct cssm_list{
    uint32 NumberItems;
    CSSM_LIST_ITEM_PTR Items;
} CSSM_LIST, *CSSM_LIST_PTR
```

Definition

NumberItems
The number of entries in the Items array.

Items
An array of name and GUID pairs.

10.3.12 CSSM_CSSMINFO

This structure describes attributes of the CSSM infrastructure itself.

```
typedef struct cssm_cssminfo {
    CSSM_VERSION Version;
    CSSM_STRING Description; /* Description of CSSM */
    CSSM_STRING Vendor; /* Vendor of CSSM */
    CSSM_BOOL ThreadSafe;
    CSSM_STRING Location;
    CSSM_GUID GUID CssmGUID;
    CSSM_GUID InterfaceGUID; /* opt GUID defining supported
                                                 interface */
}CSSM_CSSMINFO, *CSSM_CSSMINFO_PTR
```

Definition

Version
The major and minor version numbers of the CSSM Core component.

Description
A text description of the CSSM Core.

Vendor
The name and description of the CSSM Core vendor.

ThreadSafe
An indicator of whether or not this CSSM Core implementation is thread safe.

Location
The path to the CSSM Core library.

CssmGUID
The unique identifier of the CSSM Core library.

InterfaceGUID
the unique identifier of the interface implemented by the CSSM core library.

10.3.13 CSSM_EVENT_TYPE

Events occur when an application calls a CSSM core service function. CSSM informs the attached module of this event using the EventNotify call to the Service provider module. Six types of events are defined:

```
typedef uint32 CSSM_EVENT_TYPE, *CSSM_EVENT_TYPE_PTR;
#define CSSM_EVENT_ATTACH (0)
/* application has requested an attach operation */
#define CSSM_EVENT_DETACH (1)
/* application has requested an detach operation */
#define CSSM_EVENT_INFOATTACH (2)
/* application has requested module info for dynamic module
                                            capabilities */
#define CSSM_EVENT_INFODETACH (3)
/* CSSM has completed obtaining dynamic module capabilities */
#define CSSM_EVENT_CREATE_CONTEXT (4)
/* application has performed a create context operation */
#define CSSM_EVENT_DELETE_CONTEXT (5)
/* application has performed a delete context operation */
```

10.3.14 CSSM_SERVICE_MASK

This defines a bit mask of all the types of CSSM services a single module can implement.

```
typedef uint32 CSSM_SERVICE_MASK;
#define CSSM_SERVICE_CSSM 0x1
#define CSSM_SERVICE_CSP 0x2
#define CSSM_SERVICE_DL 0x4
#define CSSM_SERVICE_CL 0x8
#define CSSM_SERVICE_TP 0x10
#define CSSM_SERVICE_LAST CSSM_SERVICE_TP
```

10.3.15 CSSM_SERVICE_TYPE

This data type is used to identify a single service from the CSSM_SERVICE_MASK options defined above.

```
typedef CSSM_SERVICE_MASK CSSM_SERVICE_TYPE
```

10.3.16 CSSM_SERVICE_FLAGS

This bitmask is used to identify characteristics of the service, such as whether it contains any embedded products.

```
typedef uint32 CSSM_SERVICE_FLAGS
#define CSSM_SERVICE_ISWRAPPEDPRODUCT 0x1
    /* On = Contains one or more embedded products
    Off = Contains no embedded products */
```

10.3.17 CSSM_SERVICE_INFO

This structure holds a description of a module service. The service described is of the CSSM service type specified by the module usage type.

```
typedef struct cssm_serviceinfo {
    CSSM_STRING Description; /* Service description */
    CSSM_SERVICE_TYPE Type; /* Service type */
    CSSM_SERVICE_FLAGS Flags; /* Service flags */
    uint32 NumberOfSubServices; /* Number of sub services in
                                    SubService List */
    union cssm_subservice_list { /* List of sub services */
        void *SubServiceList;
        CSSM_CSPSUBSERVICE_PTR CspSubServiceList;
        CSSM_DLSUBSERVICE_PTR DlSubServiceList;
        CSSM_CLSUBSERVICE_PTR ClSubServiceList;
        CSSM_TPSUBSERVICE_PTR TpSubServiceList;
    } SubserviceList;
    void *Reserved;
} CSSM_SERVICE_INFO, *CSSM_SERVICE_INFO_PTR;
```

Definition

Description
A text description of the service.

Type
Specifies exactly one type of service structure, such as CSSM_SERVICE_CSP, CSSM_SERVICE_CL, and so on.

Flags
Characteristics of this service, such as whether it contains any embedded products.

NumberOfSubServices
The number of elements in the module SubServiceList.

SubServiceList
A list of descriptions of the encapsulated SubServices which are not of the basic service types.

CspSubServiceList
A list of descriptions of the encapsulated CSP SubServices.

DlSubServiceList
A list of descriptions of the encapsulated DL SubServices.

ClSubServiceList
A list of descriptions of the encapsulated CL SubServices.

TpSubServiceList
A list of descriptions of the encapsulated TP SubServices.

Reserved
This field is reserved for future use. It should always be set to NULL.

10.3.18 CSSM_MODULE_FLAGS

This bitmask is used to identify characteristics of the module, such as whether it is threadsafe, exportable, an so on. The flags also describe if and how CSSM must perform additional authentication checks on behalf of the add-in service module during module attach. The service module can select one of the following authentication checks:

- The attaching application must be successfully authenticated by CSSM, based on CSSM's roots of trust.
- The attaching application must be successfully authenticated by CSSM, based on module-specified roots of trust.

```
typedef uint32 CSSM_MODULE_FLAGS;

#define CSSM_MODULE_THREADSAFE 0x1 /* Module is threadsafe */
#define CSSM_MODULE_EXPORTABLE 0x2 /* Module can be exported
                                        outside the USA */
#define CSSM_MODULE_CALLER_AUTHENTOCSSM 0x04
    /* CSSM authenticates the caller based on CSSM-known points
                                                of trust */
#define CSSM_MODULE_CALLER_AUTHENTOMODULE 0x08
    /* CSSM authenticates the caller based on module-supplied
                                        points of trust */
```

10.3.19 CSSM_MODULE_INFO

This structure aggregates all service descriptions about all service types of a module implementation.

```
typedef struct cssm_moduleinfo {
    CSSM_VERSION Version; /* Module version */
    CSSM_VERSION CompatibleCSSMVersion; /* CSSM version the
                                        module is written for*/
    CSSM_GUID_PTR InterfaceGUID; /* opt GUID defining supported
                                           interface */
    CSSM_STRING Description; /* Module description */
    CSSM_STRING Vendor; /* Vendor name */
    CSSM_MODULE_FLAGS Flags; /* Flags to describe and control
                                           module use */
    CSSM_KEY_PTR AppAuthenRootKeys; /* Module-specific keys to
                                        authenticate apps */
    uint32 NumberOfAppAuthenRootKeys; /* Number of module-
                                        specific root keys */
    CSSM_SERVICE_MASK ServiceMask; /* Bit mask of supported
                                        services */
    uint32 NumberOfServices; /* Number of services in ServiceList */
    CSSM_SERVICE_INFO_PTR ServiceList; /* A list of service
                                        info structures */
    void *Reserved;
} CSSM_MODULE_INFO, *CSSM_MODULE_INFO_PTR;
```

Definition

Version
The major and minor version numbers of this add-in module.

CompatibleCSSMVersion
The version of CSSM that this module was written to.

InterfaceGUID
GUID describing the interface supported by the version of CSSM that this module was written to

Description
A text description of this module and its functionality.

Vendor
The name and description of the module vendor.

Flags
Characteristics of this module, such as whether or not it is threadsafe.

AppAuthenRootKeys
Public root keys used by CSSM to verify an application's credentials when the service module has requested authentication based on module-specified root keys by setting the CSSM_MODULE_CALLER_AUTHENTOMODULE bit to true in its CSSM_MODULE_FLAGS mask. These keys should successfully authenticate only those applications that the service module wishes to recognize to receive the services the module has registered with CSSM during module installation.

NumberOfAppAuthenRootKeys
The number of public root keys in the AppAuthenRoot Keys list.

ServiceMask
A bit mask identifying the types of services available in this module.

NumberOfServices
The number of services for which information is provided. Multiple descriptions (as sub-services) can be provided for a single service category.

ServiceList
An array of pointers to the service information structures. This array contains NumberOfServices entries.

Reserved
This field is reserved for future use. It should always be set to NULL.

10.3.20 CSSM_ALL_SUBSERVICES

This data type is used to identify that information on all of the sub-services is being requested or returned.

10.3.21 CSSM_INFO_LEVEL

This enumerated list defines the levels of information detail that can be retrieved about the services and capabilities implemented by a particular module. Modules can implement multiple CSSM service types. Each service may provide one or more sub-services, and can also be have dynamically available services and features.

```
typedef enum cssm_info_level {
    CSSM_INFO_LEVEL_MODULE = 0,
    /* values from CSSM_SERVICE_INFO struct */
    CSSM_INFO_LEVEL_SUBSERVICE = 1,
    /* values from CSSM_SERVICE_INFO and XXsubservice struct */
    CSSM_INFO_LEVEL_STATIC_ATTR = 2,
    /* values from CSSM_SERVICE_INFO and XXsubservice and
    all static-valued attributes of a subservice */
    CSSM_INFO_LEVEL_ALL_ATTR = 3,
    /* values from CSSM_SERVICE_INFO and XXsubservice and
    all attributes, static and dynamic, of a subservice */
} CSSM_INFO_LEVEL;
```

10.3.22 CSSM_NET_ADDRESS_TYPE

This enumerated type defines representations for specifying the location of a service.

```
typedef enum cssm_net_address_type {
    CSSM_ADDR_NONE = 0,
    CSSM_ADDR_CUSTOM = 1,
    CSSM_ADDR_URL = 2, /* char* */
    CSSM_ADDR_SOCKADDR = 3,
    CSSM_ADDR_NAME = 4 /* char* - qualified by access method */
} CSSM_NET_ADDRESS_TYPE;
```

10.3.23 CSSM_NET_ADDRESS

This structure holds the address of a service. Typically the service is remote, but the value of the address field may resolve to the local system. The AddressType field defines how the Address field should be interpreted.

```
typedef struct cssm_net_address {
    CSSM_NET_ADDRESS_TYPE AddressType;
    CSSM_DATA Address;
} CSSM_NET_ADDRESS, *CSSM_NET_ADDRESS_PTR;
```

10.3.24 CSSM_NET_PROTOCOL

This enumerated list defines the application-level protocols that could be supported by a Certificate Library Module that communicates with Certification Authorities, Registration Authorities and other services, or by a Data Storage Library Module that communicates with service-based storage and directory services.

```
typedef enum cssm_net_protocol {
    CSSM_NET_PROTO_NONE = 0, /* local */
    CSSM_NET_PROTO_CUSTOM = 1, /* proprietary implementation */
    CSSM_NET_PROTO_UNSPECIFIED = 2, /* implementation default */
    CSSM_NET_PROTO_LDAP = 3, /* light weight directory access
                                            protocol */
    CSSM_NET_PROTO_LDAPS = 4, /* ldap/ssl where SSL initiates
                                    the connection */
    CSSM_NET_PROTO_LDAPNS = 5, /* ldap where ldap negotiates an
                                    SSL session */
    CSSM_NET_PROTO_X500DAP = 6, /* x.500 Directory access
                                    protocol */
    CSSM_NET_PROTO_FTPDAP = 7, /* file transfer protocol for
                                    cert/crl fetch */
    CSSM_NET_PROTO_FTPDAPS = 8, /* ftp/ssl where SSL initiates
                                    the connection */
    CSSM_NET_PROTO_NDS = 9, /* Novell directory services */
    CSSM_NET_PROTO_OCSP = 10, /* online certificate status
                                    protocol */
    CSSM_NET_PROTO_PKIX3 = 11, /* the cert request protocol
                                    in PKIX3 */
    CSSM_NET_PROTO_PKIX3S = 12, /* The ssl/tls derivative
                                    of PKIX3 */
    CSSM_NET_PROTO_PKCS_HTTP = 13, /* PKCS client <=> CA protocol
                                    over HTTP */
    CSSM_NET_PROTO_PKCS_HTTPS = 14, /* PKCS client <=> CA protocol
                                    over HTTPS */
} CSSM_NET_PROTOCOL;
```

10.3.25 CSSM_APP_SERVICE_FLAGS

As part of module-attach processing, CSSM authenticates every attached service module based on CSSM-defined roots of trust. Applications can elect to have CSSM perform an additional authentication check on behalf of the application. This additional verification is performed during module attach and is based on application-specified roots of trust. An application service flag is used to request this additional service by CSSM.

```
typedef uint32 CSSM_APP_SERVICE_FLAGS

#define CSSM_APP_SERVICE_AUTHENTOAPP  0x1
      /* CSSM authenticates the service module based on
         application-supplied points of trust */
```

10.3.26 CSSM_APP_KEYS

This structure aggregates the roots of trust for authenticating a particular add-in service module during module attach.

```
typedef struct cssm_app_keys {
    CSSM_KEY_PTR ModuleAuthenRootKeys,
          /* Application-specified keys to authen service modules*/
    uint32 NumberOfModuleAuthenRootKeys,
          /* Number of application-specified root keys */
} CSSM_APP_KEYS, *CSSM_APP_KEYS_PTR;
```

Definition

ModuleAuthenRootKeys
Public root keys used by CSSM to verify an service module's credentials when the application has requested authentication based on application-specified root keys by setting the CSSM_APP_SERVICE_AUTHENTOAPP bit to true in the flags mask in the CSSM_APP_SERVICE_INFO structure. These keys should successfully authenticate only those service modules that the application wishes to recognize.

NumberOfModuleAuthenRootKeys
The number of public root keys in the ModuleAuthenRoot Keys list.

10.3.27 CSSM_APP_SERVICE_INFO

This structure aggregates all information required by CSSM to perform additional authentication of add-in service modules on behalf of an application during module attach processing.

```
typedef struct cssm_app_service_info {
    CSSM_SUBSERVICE_UID_PTR ModuleList;  /* List of module
                                         service ID structs */
    uint32 NumberOfModules;              /* Number of modules to
                                         authenticate */
    CSSM_APP_SERVICE_FLAGS Flags;
/* Flags selecting CSSM or app-specified roots of trust */
    CSSM_APP_KEYS_PTR *Keys,
/* Application-specified keys to authenticate modules */
    void *Reserved;
} CSSM_APP_SERVICE_INFO, *CSSM_APP_SERVICE_INFO_PTR;
```

Definition

ModuleList
The unique identifier for each module that CSSM must authenticate on behalf of the application.

NumberOfModules
The number of module identification structure in ModuleList.

Flags
Specify whether CSSM a second verification of the service module using application-specified roots of trust.

Keys
A pointer to a list of sets of application-specified root keys, one for each module in ModuleList. These keys are used by CSSM to verify the module during module attach.

Reserved
This field is reserved for future use. It should always be set to NULL.

10.3.28 CSSM_EXEMPTION_MASK

This bitmask represents the exemptions requested by the calling application process or thread. Exemptions are defined corresponding to built-in checks performed by CSSM and the CSSM Module Managers. Elective Module Managers can define additional categories of exemption for built-in checks performed by those elective managers. The caller must possess the necessary credentials to be granted the exemptions.

```
typedef uint32 CSSM_EXEMPTION_MASK

#define CSSM_EXEMPT_NONE 0x00000001
#define CSSM_EXEMPT_MULTI_ENCRYPT_CHECK 0x00000002
#define CSSM_EXEMPT_ALL 0xFFFFFFFF
```

10.3.29 CSSM_USER_AUTHENTICATION_MECHANISM

This enumerated list defines different methods an add-in module can require when authenticating a caller. The module specifies which mechanism the caller must use for each sub-service type provided by the module. CSSM-defined authentication methods include password-based authentication, a login sequence, or a certificate and passphrase. It is anticipated that new mechanisms will be added to this list as required.

```
typedef enum cssm_user_authentication_mechanism {
    CSSM_AUTHENTICATION_NONE = 0,
    CSSM_AUTHENTICATION_CUSTOM = 1,
    CSSM_AUTHENTICATION_PASSWORD = 2,
    CSSM_AUTHENTICATION_USERID_AND_PASSWORD = 3,
    CSSM_AUTHENTICATION_CERTIFICATE_AND_PASSPHRASE = 4,
    CSSM_AUTHENTICATION_LOGIN_AND_WRAP = 5,
} CSSM_USER_AUTHENTICATION_MECHANISM;
```

10.3.30 CSSM_CALLBACK

An application uses this data type to request that an add-in module call back into the application for certain cryptographic information.

```
typedef CSSM_DATA_PTR (CSSMAPI *CSSM_CALLBACK) (void *allocRef, uint32 ID);
```

Definition

allocRef
Memory heap reference specifying which heap to use for memory allocation.

ID
Input data to identify the callback.

10.3.31 CSSM_CRYPTO_DATA

This data structure is used to encapsulate cryptographic information, such as the passphrase to use when accessing a private key.

```
typedef struct cssm_crypto_data {
    CSSM_DATA_PTR Param;
    CSSM_CALLBACK Callback;
    uint32 ID;
}CSSM_CRYPTO_DATA, *CSSM_CRYPTO_DATA_PTR
```

Definition

Param
A pointer to the parameter data and its size in bytes.

Callback
An optional callback routine for the add-in modules to obtain the parameter.

ID
A tag that identifies the callback.

10.3.32 CSSM_USER_AUTHENTICATION

This structure holds the user's credentials for authenticating to a module. The type of credentials required is defined by the module and specified as a CSSM_USER_AUTHENTICATION_MECHANISM.

```
typedef struct cssm_user_authentication {
    CSSM_DATA_PTR Credential; /* a cert, a shared secret, other */
    CSSM_CRYPTO_DATA_PTR MoreAuthenticationData;
} CSSM_USER_AUTHENTICATION, *CSSM_USER_AUTHENTICATION_PTR;
```

Definition

Credential
A certificate, a shared secret, a magic token or whatever is required by an add-in service modules for user authentication. The required credential type is specified as a CSSM_USER_AUTHENTICATION_MECHANISM.

MoreAuthenticationData
A passphrase or other data that can be provided as immediate data within this structure or via a callback function to the user/caller.

10.3.33 CSSM_NOTIFY_CALLBACK

An application uses this data type to request that an add-in module call back into the application to notify it of certain events.

```
typedef CSSM_RETURN (CSSMAPI *CSSM_NOTIFY_CALLBACK)
    (CSSM_MODULE_HANDLE ModuleHandle,
    uint32 Application,
    uint32 Reason,
    void* Param);
```

Definition

ModuleHandle
The handle of the attached add-in module.

Application
Input data to identify the callback.

Reason
The reason for the notification.

Reason	Description
CSSM_NOTIFY_SURRENDER	The add-in module is temporarily surrendering control of the process
CSSM_NOTIFY_COMPLETE	An asynchronous operation has completed
CSSM_NOTIFY_DEVICE_REMOVED	A device, such as a token or storage device, has been removed
CSSM_NOTIFY_DEVICE_INSERTED	A device, such as a token or storage device, has been inserted

Param
Any additional information about the event.

10.3.34 CSSM_MEMORY_FUNCS and CSSM_API_MEMORY_FUNCS

This structure is used by applications to supply memory functions for the CSSM and the add-in modules. The functions are used when memory needs to be allocated by the CSSM or add-ins for returning data structures to the applications

```
typedef struct cssm_memory_funcs {
  void * (*malloc_func) (uint32 Size, void *AllocRef);
  void (*free_func) (void *MemPtr, void *AllocRef);
  void * (*realloc_func) (void *MemPtr, uint32 Size, void *AllocRef);
  void * (*calloc_func) (uint32 Num, uint32 Size, void *AllocRef);
  void *AllocRef;
} CSSM_MEMORY_FUNCS, *CSSM_MEMORY_FUNCS_PTR;

typedef CSSM_MEMORY_FUNCS CSSM_API_MEMORY_FUNCS;
typedef CSSM_API_MEMORY_FUNCS *CSSM_API_MEMORY_FUNCS_PTR;
```

Definition

malloc_func
Pointer to a function that returns a void pointer to the allocated memory block of at least Size bytes from heap AllocRef.

free_func
Pointer to a function that deallocates a previously-allocated memory block (MemPtr) from heap AllocRef.

realloc_func
Pointer to a function that returns a void pointer to the reallocated memory block (MemPtr) of at least Size bytes from heap AllocRef.

calloc_func
Pointer to a function that returns a void pointer to an array of Num elements of length Size initialized to zero from heap AllocRef.

AllocRef
Indicates which memory heap the function operates on.

See Appendix B for details about the application memory functions.

10.4 Core Functions

The manpages for Core Functions follow on the next page.

NAME

CSSM_Init

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_Init
    (const CSSM_VERSION_PTR Version,
    const void * Reserved)
```

DESCRIPTION

This function initializes CSSM and verifies that the version of CSSM expected by the application is compatible with the version of CSSM on the system. This function should be called once by each application.

PARAMETERS

Version (input)
The major and minor version number of the CSSM release the application is compatible with.

Reserved (input)
A reserved input.

RETURN VALUE

A CSSM_OK return value signifies the initialization operation was successful. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_INCOMPATIBLE_VERSION
Incompatible version.

NAME

CSSM_GetInfo

SYNOPSIS

```
CSSM_CSSMINFO_PTR CSSMAPI CSSM_GetInfo
    ( const CSSM_MEMORY_FUNCS_PTR MemoryFunctions,
      uint32 *NumCssmInfos)
```

DESCRIPTION

This function returns the version information of all the CSSM instances installed/registered on the local system. Memory to hold the info structure is obtaining using the memory allocation functions. The list the list is no loner needed it can be de-allocated by invoking the CSSM_FreeInfo function or the caller can de-allocate the list directly.

PARAMETERS

MemoryFunctions (input)
A table of API_MEMORY_FUNCTION pointers for use by the CSSM representative. The representative uses the memory allocation function to obtain memory to hold a CSSM information structure for each CSSM installed on the local system.

NumCssmInfos (output)
The number of CSSM instances installed on the local system and the number of information structures returned by this function.

RETURN VALUE

A pointer to an array of CSSM_CSSMINFO structures. If the pointer is NULL, an error occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_MEMORY_ERROR
Error in allocating memory.

CSSM_NOT_INSTALLED
No CSSM as not been installed.

SEE ALSO

CSSM_FreeInfo, *CSSM_Load*

NAME

CSSM_FreeInfo

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_FreeInfo
    (const CSSM_CSSMINFO_PTR CssmInfo,
     const CSSM_MEMORY_FUNCS_PTR MemoryFunctions,
     uint32 NumCssmInfos)
```

DESCRIPTION

This function frees the memory containing the version information of all the CSSM instances installed/registered on the local system. Memory to de-allocated using the memory deallocation function.

PARAMETERS

CssmInfo (input)
A CSSM_INFO_PTR referencing a list of Info structures allocated by the CSSM_GetInfo function.

MemoryFunctions (input)
A table of API_MEMORY_FUNCTION pointers for use by CSSM. CSSM uses the memory de-allocation function to release memory holding CSSM information structures.

NumCSSMInfos (input)
The number of CSSM information structures contained in the list.

RETURN VALUE

A CSSM_OK return value signifies the CSSM_INFO structures have been successfully de-allocated. When CSSM_FAIL is returned, an error occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_MEMORY_ERROR
Error in de-allocating memory.

CSSM_INVALID_POINTER
Invalid pointer.

SEE ALSO

CSSM_FreeInfo, *CSSM_Load*

NAME

CSSM_Load

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_Load
    (CSSM_CSSMINFO_PTR CssmInfo)
```

DESCRIPTION

This function loads the CSSM instance specified by the CSSM_CSSMINFO structure.

PARAMETERS

CssmInfo (input)
A pointer to the CSSM_CSSMINFO structure specifying the CSSM instance to be loaded.

RETURN VALUE

A CSSM_OK return value signifies the CSSM instance has been successfully loaded. When CSSM_FAIL is returned, an error occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_LOAD_FAIL
Load operation failed.

CSSM_NOT_INSTALLED
Specified CSSM has not been installed.

SEE ALSO

CSSM_GetInfo, *CSSM_FreeInfo*

NAME

CSSM_RequestCssmExemption

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_RequestCssmExemption
    (CSSM_EXEMPTION_MASK ExemptionRequests,
    const char *AppFileName,
    const char *AppPathName,
    const void * Reserved)
```

DESCRIPTION

This function authenticates the application and verifies whether it is authorized to receive the requested CSSM exemptions. Authentication is based on successful verification of the application's signed manifest credentials. Implied authorization can require credential verification based on specific roots of trust.

The exemption mask defines the requested exemptions. The application file name and application pathname specify the location of the application's credentials.

Applications may invoke this function multiple times. Each successful verification replaces the previously granted exemptions. Exemptions are not inherited by spawned processes or spawned threads.

CSSM and CSSM elective module managers are the authorization entities that define the roots of trust for authenticating applications and granting exemptions from built-in checks. The set of trusted roots can grow during execution. This makes an application's request for exemption dependent on execution order. If an application performs all module attach operations before calling CSSM_RequestCssmExemption, then all points of trust/authorization that could be effected by that request are known and the request can be accurately processed. If an application's authentication (and implied authorization) is dependent on roots of trust that are not yet known, then the application cannot be authenticated and the request for exemption will be denied.

PARAMETERS

ExemptionRequest (input)
A bitmask of all exemptions being requested by the caller.

AppFileName (input)
The name of the file that implements the application (containing its main entry point). This file name is used to locate the application's credentials for purposes of application authentication by CSSM.

AppPathName (input)
The path to the file that implements the application (containing its main entry point). This path name is used to locate the application's credentials for purposes of application authentication by CSSM.

Reserved (input/optional)
A reserved input.

RETURN VALUE

A CSSM_OK return value signifies the verification operation was successful and the exemption has been granted. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_CREDENTIALS
Malformed or missing credentials.

CSSM_NOT_AUTHORIZED
Credentials do not verify for requested exemptions.

NAME

CSSM_VerifyComponents

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_VerifyComponents
    (void)
```

DESCRIPTION

This function performs an integrity check on all the components of CSSM to insure no tampering has occurred since installation.

PARAMETERS

None

RETURN VALUE

A CSSM_TRUE return value signifies that all components verified successfully. When CSSM_FALSE is returned, either the verification failed or an error occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_VERIFY_COMPONENTS_FAILED
Unable to verify components.

CSSM_INTEGRITY_COMPROMISED
Integrity check failed.

10.5 Module Management Functions

The manpages for Module Management Functions follow on the next page.

NAME

CSSM_ModuleInstall

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleInstall
    (const char *ModuleName,
    const char *ModuleFileName,
    const char *ModulePathName,
    const CSSM_GUID_PTR GUID,
    const CSSM_MODULE_INFO_PTR ModuleDescription,
    const void * Reserved1,
    const CSSM_DATA_PTR Reserved2)
```

DESCRIPTION

This function registers the module with CSSM. CSSM adds the module's descriptive information to its persistent registry. This makes the service module available for use on the local system. The function accepts as input the name and unique identifier for the module, the location executable code for the module, and a digitally-signed list of capabilities supported by the module. The capabilities list includes flags describing the module's attach time policy. The module's attach time procedure requirements are defined by its MODULE_FLAGS that control authentication. In addition to the module-declared policy, CSSM always enforces its internal policy requiring authentication. CSSM evaluates its policy based on CSSM-selected public root keys as points of trust. The service module policy can require application authentication based on a set of module-selected public root keys as point of trust. A copy of these module-selected keys are included in the CSSM_MODULE_INFO structure. The effective module policy definition must be included in the module's signed credentials. The registry copy is only informational. The installation process records the module name and module info in the CSSM Registry, making the module available for use by applications.

PARAMETERS

ModuleName (input)
The name of the module.

ModuleFileName (input)
The name of the file that implements the module.

ModulePathName (input)
The path to the file that implements the module.

GUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the module.

ModuleDescription (input)
A pointer to the CSSM_MODULE_INFO structure containing a description of the module.

Reserved1 (input)
Reserve data for the function.

Reserved2 (input)
Reserve data for the function.

RETURN VALUE

A CSSM_OK return value signifies that information has been updated. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_REGISTRY_ERROR
Error in the registry.

SEE ALSO

CSSM_ModuleUninstall

NAME

CSSM_ModuleUninstall

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleUninstall
    (const CSSM_GUID_PTR GUID)
```

DESCRIPTION

This function deletes the persistent CSSM internal information about the module, removing it from the name space of available modules in the CSSM system.

PARAMETERS

GUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the module.

RETURN VALUE

A CSSM_OK return value means the module has been successfully uninstalled. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_INVALID_GUID
CSP module was not installed.

CSSM_REGISTRY_ERROR
Unable to delete information.

SEE ALSO

CSSM_ModuleInstall

NAME

CSSM_ModuleAttach

SYNOPSIS

```
CSSM_CSP_HANDLE CSSMAPI CSSM_ModuleAttach
    (const CSSM_GUID_PTR GUID,
    const CSSM_VERSION_PTR Version,
    const CSSM_API_MEMORY_FUNCS_PTR MemoryFuncs,
    uint32 SubserviceID,
    uint32 SubserviceFlags,
    uint32 Application,
    const CSSM_NOTIFY_CALLBACK Notification,
    const char *AppFileName,
    const char *AppPathName,
    const void * Reserved)
```

DESCRIPTION

This function attaches the service provider module and verifies that the version of the module expected by the application is compatible with the version on the system. The module can implement sub-services (as described in the service provider's documentation). The caller can specify a specific sub-service provided by the module. Sub-service flags may be required to set parameters for the service.

PARAMETERS

GUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the CSP module.

Version (input)
The major and minor version number of the service provider module that the application is compatible with.

MemoryFuncs (input)
A structure containing pointers to the memory routines.

SubserviceID (input)
The number of a sub-service provided by the module. This value should always be taken from the CSSM_MODULE_INFO structure to insure that a compatible identifier is used. (Service provider modules that implement only one service can use zero as the sub-service identifier.)

SubserviceFlags (input)
Bitmask of service options defined by a particular sub-service of the module. Legal values are described in module-specific documentation. A default set of flags is specified in the CSSM_MODULE_INFO structure for use by the caller.

Application (input/optional)
Nonce passed to the application when its callback is invoked allowing the application to determine the proper context of operation.

Notification (input/optional)
Callback provided by the application that is used by the add-in module to notify the application of certain events. For example, a CSP may use this callback in the following situations: a parallel operation completes, a token running in serial mode surrenders control to the application or the token is removed (hardware-specific).

AppFileName (input/optional)

The name of the file that implements the application (containing its main entry point). This file name is used to locate the application's credentials for purposes of application authentication by CSSM or by CSSM on behalf of the target add-in module. This input must be provided if the target add-in module defines a usage policy that requires authentication of the application's credentials. The add-in module's declared policy is recorded by the MODULE_FLAGS contained in module's MODULE_INFO structure and in the module's signed credentials. If application authentication is not required by the target add-in module, this parameter should be NULL.

AppPathName (input/optional)

The path to the file that implements the application (containing its main entry point). This path name is used to locate the application's credentials for purposes of application authentication by CSSM or by CSSM on behalf of the target add-in module. This input must be provided if the target add-in module defines a usage policy that requires authentication of the application's credentials. The add-in module's declared policy is recorded by the MODULE_FLAGS contained in the module's MODULE_INFO structure and in the module's signed credentials. If application authentication is not required by the target add-in module, this parameter should be NULL.

Reserved (input)

A reserved input.

RETURN VALUE

A handle is returned for the attached service provider module. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_INCOMPATIBLE_VERSION
Incompatible version.

CSSM_EXPIRE
Add-in module has expired.

CSSM_NOT_INITIALIZE
CSSM has not been invoked.

CSSM_ATTACH_FAIL
Unable to load service provider module.

SEE ALSO

CSSM_ModuleDetach

NAME

CSSM_ModuleDetach

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleDetach
    (CSSM_MODULE_HANDLE ModuleHandle)
```

DESCRIPTION

This function detaches the application from the service provider module.

PARAMETERS

ModuleHandle (input)
The handle that describes the service provider module.

RETURN VALUE

A CSSM_OK return value signifies that the application has been detached from the module. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_ADDIN_HANDLE
Invalid module handle.

SEE ALSO

CSSM_ModuleAttach

NAME

CSSM_ListModules

SYNOPSIS

```
CSSM_LIST_PTR CSSMAPI CSSM_ListModules
    (CSSM_SERVICE_MASK ServiceMask,
    CSSM_BOOL MatchAll)
```

DESCRIPTION

This function returns a list containing the GUID/name pair for each of the currently-installed service provider modules that provide services in any of the CSSM functional categories selected in the service mask.

PARAMETERS

ServiceMask (input)
A bit mask selecting the CSSM functional categories of interest for selecting information about potential service provider modules.

MatchAll (input)
A boolean value defining how the multiple bits in the service mask are interpreted. TRUE means the service modules selected must match all service areas specified by the service mask. FALSE means the service module selected must specify one or more of the service areas specified by the service mask.

RETURN VALUE

A pointer to the CSSM_LIST structure containing the GUID/name pair for each of the modules. If the pointer is NULL, an error has occurred; use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_NO_ADDIN
No add-ins found.

CSSM_MEMORY_ERROR
Error in memory allocation.

CSSM_NOT_INITIALIZE
CSSM_Init has not been invoked.

CSSM_REGISTRY_ERROR
Registry error.

SEE ALSO

CSSM_GetModuleInfo, *CSSM_FreeModuleInfo*

NAME

CSSM_GetModuleInfo

SYNOPSIS

```
CSSM_MODULE_INFO_PTR CSSMAPI CSSM_GetModuleInfo
    (const CSSM_GUID_PTR ModuleGUID,
    CSSM_SERVICE_MASK ServiceMask,
    sint32 SubserviceID,
    CSSM_INFO_LEVEL InfoLevel);
```

DESCRIPTION

This function returns descriptive information about the module identified by the GUID. The information returned can include all of the capability information, for each subservices for each of the service types implemented by the selected module. The request for information can be limited to a particular set of services, as specified by the service bit mask. The request may be further limited to one or all of the sub-services implemented in one or all of the service categories. Finally the detail level of the information returned can be controlled by the InfoLevel input parameter. This is particularly important for modules with dynamic capabilities. InfoLevel can be used to request static attribute values only or dynamic values.

PARAMETERS

ModuleGUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the service provider module.

ServiceMask (input)
A bit mask specifying the module usage types used to restrict the capabilities information returned by this function. An input value of zero specifies all usages for the specified module.

SubserviceID (input)
A single sub-service ID or the value CSSM_ALL_SUBSERVICES must be provided. If a sub-service ID is provided the get operation is limited to the specified sub-service. Note that the operation may already be limited by a service mask. If so, the sub-service ID applies to all service categories selected by the service mask. If CSSM_ALL_SUBSERVICES is specified, information for all sub-services (as limited by the service mask) are returned by this function.

InfoLevel (input)
Indicates the level of detail returned by this function. Information retrieval can be restricted as follows:

- CSSM_INFO_LEVEL_MODULE—returns only the information contained in the CSSM_SERVICE_INFO structure.
- CSSM_INFO_LEVEL_SUBSERVICE—returns the information returned by CSSM_INFO_LEVEL_MODULE and the information contained in the XXsubservice structure, where XX corresponds to the module type, such as tpsubservice, clsubservice, dlsubservice, cpsubservice.
- CSSM_INFO_LEVEL_STATIC_ATTR—returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically defined for the module.
- CSSM_INFO_LEVEL_ALL_ATTR—returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically or dynamically defined for the module. Dynamic modules, whose capabilities

change over time, support a query function used by CSSM to interrogate the module's current capability status.

RETURN VALUE

A pointer to a module info structure containing a pointer to an array of zero or more service information structures. Each structure contains type information identifying the service description as representing certificate library services, data storage library services, and so on. The service descriptions are sub-classed into sub-service descriptions which describe the attributes and capabilities of a sub-service.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_INVALID_USAGE_MASK
Invalid bit mask.

CSSM_INVALID_SUBSERVICEID
Invalid sub-service ID.

CSSM_INVALID_INFO_LEVEL
Invalid info level indicator.

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_INVALID_GUID
Unknown GUID.

CSSM_NOT_INITIALIZE
CSSM_Init has not been invoked.

CSSM_MEMORY_ERROR
Internal Memory Error.

CSSM_REGISTRY_ERROR
A registry error occurred.

SEE ALSO

CSSM_SetModuleInfo, *CSSM_FreeModuleInfo*

NAME

CSSM_SetModuleInfo

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SetModuleInfo
    (const CSSM_GUID_PTR ModuleGUID,
    const CSSM_MODULE_INFO_PTR ModuleInfo);
```

DESCRIPTION

This function replaces all of the currently registered descriptive information about the module identified by the ModuleGUID with the newly specified information. The operation is a total replacement of all information for all service categories and all subservices.

If the caller wishes to retain any of the information registered prior to execution of this call, the caller must use the *CSSM_GetModuleInfo* function to retrieve the current information, update their private copy, and then use the *CSSM_SetModuleInfo* function to place the updated copy back into the CSSM registry.

This function should be used to incrementally update descriptive information that is unspecified at installation time.

PARAMETERS

ModuleGUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the service provider module.

ModuleInfo (input)
A pointer to the complete structured set of descriptive information about the module.

RETURN VALUE

A CSSM_RETURN value indicating pass or fail. CSSM_OK indicates success, otherwise use *CSSM_GetError* to determine the type of error that has occurred.

ERRORS

CSSM_INVALID_GUID
Unknown GUID.

CSSM_INVALID_MODULE_INFO
Invalid module info structure.

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_NOT_INITIALIZE
CSSM_Init has not been invoked.

CSSM_REGISTRY_ERROR
Registry error.

CSSM_INVALID_POINTER
Invalid input pointer.

SEE ALSO

CSSM_GetModuleInfo, *CSSM_FreeModuleInfo*

NAME

CSSM_FreeModuleInfo

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_FreeModuleInfo
    (CSSM_MODULE_INFO_PTR ModuleInfo)
```

DESCRIPTION

This function frees the memory allocated to hold all of the info structures returned by *CSSM_GetModuleInfo*. All sub-structures within the info structure are freed by this function.

PARAMETERS

ModuleInfo (input)
A pointer to the CSSM_MODULE_INFO structures to be freed.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_NOT_INITIALIZE
CSSM_Init has not been invoked.

CSSM_INVALID_POINTER
Invalid input pointer.

SEE ALSO

CSSM_GetModuleInfo, *CSSM_SetModuleInfo*

NAME

CSSM_GetGUIDUsage

SYNOPSIS

```
CSSM_SERVICE_MASK CSSMAPI CSSM_GetGUIDUsage
    (const CSSM_GUID_PTR ModuleGUID)
```

DESCRIPTION

Returns a bit mask describing the CSSM function categories of service provided by the module identified by the specified GUID.

PARAMETERS

ModuleGUID (input)
Globally unique identifier for the module of interest.

RETURN VALUE

A CSSM_SERVICE_MASK from the info structure describing the services provided by the module referenced by the GUID.

ERRORS

CSSM_INVALID_GUID
Invalid GUID.

CSSM_INVALID_POINTER
Invalid input pointer.

SEE ALSO

CSSM_GetHandleUsage

NAME

CSSM_GetHandleUsage

SYNOPSIS

```
CSSM_SERVICE_MASK CSSMAPI CSSM_GetHandleUsage
    (CSSM_HANDLE ModuleHandle)
```

DESCRIPTION

Returns a bit mask describing the CSSM function categories of service provided by the module identified by the specified handle for an attached module.

PARAMETERS

ModuleHandle (input)
Handle of the module for which information should be returned.

RETURN VALUE

A CSSM_SERVICE_MASK from the info structure describing the services provided by the module referenced by the handle.

ERRORS

CSSM_INVALID_MODULE_HANDLE
Invalid add-in handle.

SEE ALSO

CSSM_GetGUIDUsage

NAME

CSSM_GetModuleGUIDFromHandle

SYNOPSIS

```
CSSM_GUID_PTR CSSMAPI CSSM_GetModuleGUIDFromHandle
    (CSSM_HANDLE ModuleHandle)
```

DESCRIPTION

Returns the GUID of the attached module identified by the specified handle.

PARAMETERS

ModuleHandle (input)
Handle of the module for which the GUID should be returned.

RETURN VALUE

Non-NULL if the function was successful. NULL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_MODULE_HANDLE
Invalid add-in handle.

SEE ALSO

CSSM_GetHandleUsage, *CSSM_GetSubserviceUIDFromHandle*

NAME

CSSM_GetSubserviceUIDFromHandle

SYNOPSIS

```
CSSM_SUBSERVICE_UID_PTR CSSMAPI CSSM_GetSubserviceUIDFromHandle
    (CSSM_HANDLE ModuleHandle,
    const CSSM_MEMORY_FUNCS_PTR MemoryFuncs)
```

DESCRIPTION

This function returns the unique identifier of the attached module subservice, as identified by the input handle. If provided, the MemoryFuncs override the CSSM's default memory functions which were set by the most recent call to CSSM_Init.

PARAMETERS

ModuleHandle (input)
Handle of the module subservice for which the subservice unique identifier should be returned.

MemoryFunctions (input/optional) A table of API_MEMORY_FUNCS pointers. CSSM uses these functions to perform memory management operations on behalf of the caller during the service of this function call. If no table of functions is specified, CSSM uses the default memory management functions set by the caller's most recent call to the CSSM_Init function.

RETURN VALUE

Non-NULL if the function was successful. NULL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_MODULE_HANDLE
Invalid add-in handle.

SEE ALSO

CSSM_GetGUIDFromHandle

10.6 Utility Functions

The manpages for Utility Functions follow on the next page.

NAME

CSSM_FreeList

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_FreeList
    (CSSM_LIST_PTR CSSMList)
```

DESCRIPTION

This function frees the memory allocated to hold a list of strings.

PARAMETERS

CSSMList (input)
A pointer to the CSSM_LIST structure containing the GUID, name pair of add-ins.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer input.

NAME

CSSM_Free

SYNOPSIS

```
void CSSMAPI CSSM_Free
    (void *MemPtr,
     CSSM_HANDLE AddInHandle)
```

DESCRIPTION

This function frees the memory allocated by the specified add-in.

PARAMETERS

MemPtr (input)
A pointer to the memory to be freed.

AddInHandle (input)
The handle to add-in module that needs to free memory

NAME

CSSM_GetAPIMemoryFunctions

SYNOPSIS

```
CSSM_API_MEMORY_FUNCS_PTR CSSMAPI CSSM_GetAPIMemoryFunctions
    (CSSM_HANDLE AddInHandle)
```

DESCRIPTION

This function retrieves the memory function table associated with the add-in module.

PARAMETERS

AddInHandle (input)
The handle to add-in module that is associated to memory function table.

RETURN VALUE

Non NULL if the function was successful. NULL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_ADDIN_HANDLE
Invalid add-in handle.

CSSM_MEMORY_ERROR
Internal memory error.

Chapter 11

Cryptographic Services API

11.1 Overview

Cryptographic Service Providers (CSPs) are add-in modules which perform cryptographic operations including encryption, decryption, digital signaturing, key and key pair generation, random number generation, message digest, key wrapping, key unwrapping, and key exchange. Cryptographic services can be implemented by a hardware-software combination or by software only. Besides the traditional cryptographic functions, CSPs may provide other vendor-specific services. The set of services provided can be dynamic even after the CSP has been attached for service by a caller. This means the capabilities registered when the CSP was installed can change during execution based on changes internal or external to the system.

The CSP is always responsible for the secure storage of private keys. Optionally the CSP may assume responsibility for the secure storage of other object types, such as symmetric keys and certificates. The implementation of secured persistent storage for keys can use the services of a Data Storage Library module within the CSSM framework (if that module provides secured storage) or some approach internal to the CSP. Accessing persistent objects managed by the CSP, other than keys, is performed using CSSM's Data Storage Library APIs.

CSPs optionally support a password-based login sequence. When login is supported, the caller is allowed to change passwords as deemed necessary. This is part of a standard user-initiated maintenance procedure. Some CSPs support operations for privileged, CSP administrators. The model for CSP administration varies widely among CSP implementations. For this reason, CSSM does not define APIs for vendor-specific CSP administration operations. CSP vendors can make these services available to CSP administration tools using the CSSM_Passthrough function.

The range and types of cryptographic services a CSP supports is at the discretion of the vendor. A registry and query mechanism is available through the CSSM for CSPs to disclose the services and details about the services. As an example, a CSP may register with the CSSM: Encryption is supported, the algorithms present are DES with cipher block chaining for key sizes 40 and 56 bits, triple DES with 3 keys for key size 168 bits.

All cryptographic services requested by applications will be channeled to one of the CSPs via the CSSM. CSP vendors only need target their modules to CSSM for all security-conscious applications to have access to their product.

Calls made to a Cryptographic Service Provider (CSP) to perform cryptographic operations occur within a framework called a session, which is established and terminated by the application. The session context (simply referred to as the context) is created prior to starting CSP operations and is deleted as soon as possible upon completion of the operation. Context information is not persistent; it is not saved permanently in a file or database.

Before an application calls a CSP to perform a cryptographic operation, the application uses the query services function to determine what CSPs are installed, and what services they provide. Based on this information, the application then can determine which CSP to use for subsequent operations; the application creates a session with this CSP and performs the operation.

Depending on the class of cryptographic operations, individualized attributes are available for the cryptographic context. Besides specifying an algorithm when creating the context, the application may also initialize a session key, pass an initialization vector and/or pass padding information to complete the description of the session. A successful return value from the create function indicates the desired CSP is available. Functions are also provided to manage the created context.

When a context is no longer required, the application calls CSSMDeleteContext. Resources that were allocated for that context can be reclaimed by the operating system.

Cryptographic operations come in two types—a single call to perform an operation and a staged method of performing the operation. For the single call method, only one call is needed to obtain the result. For the staged method, there is an initialization call followed by one or more update calls, and ending with a completion (final) call. The result is available after the final function completes its execution for most cryptographic operations—staged encryption/decryption are an exception in that each update call generates a portion of the result.

11.1.1 Key Formats for Public Key-Based Algorithms

To ensure interoperability among cryptographic service providers and portability for application developers, CSSM must mandate standard key formats for public key based cryptographic algorithms. Standard key formats have not been defined for many of the algorithms identified by CSSM because these algorithms are not yet in wide spread use. For those algorithms in wide spread use, CDSA adopts existing standard formats or defines a format when no standard exists.

The two PKI-based algorithms with wide spread usage are:

- RSA-based algorithms
- DSA-based algorithms

For RSA-based algorithms, CDSA adopts the PKCS#1 standard for key representation.

For DSA-based algorithms, no organization has published a standard and no *de facto* standard seems to exists. CDSA defines a standard representation for DSA key based on the DSA algorithm definitions in the FIPS 186 and FIPS 186a standards. Complete documentation on these standards can be found at *http://csrc.ncsl.nist.gov/fips/fips186.txt* and at *http://csrc.ncsl.nist.gov/fips/fips186a.txt* respectively.

A DSA public key is represented as a BER-encoded, ordered sequence containing the prime modulus, the prime divisor, the order modulo the prime modulus, and the public key value. A DSA private key is represented as a BER-encoded, ordered sequence containing the prime modulus, the prime divisor, the order modulo the prime modulus, and the private key value. Additional information is provided in the specification titled CSSM Cryptographic Service Providers Interface.

11.2 Data Structures

11.2.1 CSSM_CC_HANDLE

```
typedef uint32 CSSM_CC_HANDLE /* Cryptographic Context Handle */
```

11.2.2 CSSM_CSP_HANDLE

```
typedef uint32 CSSM_CSP_HANDLE /* Cryptographic Service Provider
                                        Handle */
```

11.2.3 CSSM_DATE

```
typedef struct cssm_date {
    uint8 Year[4];
    uint8 Month[2];
    uint8 Day[2];
} CSSM_DATE, *CSSM_DATE_PTR;
```

Definition

Year
Four-digit ASCII representation of the year.

Month
Two-digit ASCII representation of the month.

Day
Two-digit ASCII representation of the day.

11.2.4 CSSM_RANGE

```
typedef struct cssm_range {
    uint32 Min; /* inclusive minimum value */
    uint32 Max; /* inclusive maximum value */
} CSSM_RANGE, *CSSM_RANGE_PTR;
```

Definition

Min
Minimum value in the range.

Max
Maximum value in the range.

11.2.5 CSSM_QUERY_SIZE_DATA

```
typedef struct cssm_query_size_data {
    uint32 SizeInputBlock; /* size of input data block */
    uint32 SizeOutputBlock; /* size of resulting output
                                        data block */
} CSSM_QUERY_SIZE_DATA, *CSSM_QUERY_SIZE_DATA_PTR;
```

Definition

SizeInputBlock
Size of the data block to be input for processing.

SizeOutputBlock
Size of the output data block that results from processing.

11.2.6 CSSM_HEADERVERSION

```
typedef uint32 CSSM_HEADERVERSION;

#define CSSM_KEYHEADER_VERSION (2)
```

Definition

Represents the version number of a key header structure. This version number is an integer that increments with each format revision. The current revision number is represented by the defined constant CSSM_KEYHEADER_VERSION.

11.2.7 CSSM_KEY_SIZE

This structure holds the key size and the effective key size for a given key. The metric used is bits. The number of effective bits is the number of key bits that can be used in a cryptographic operation compared with the number of bits that may be present in the key. When the number of effective bits is less than the number of actual bits, this is known as "dumbing down".

```
typedef struct cssm_key_size {
   uint32 KeySizeInBits; /* Key size in bits */
   uint32 EffectiveKeySizeInBits; /* Effective key size in bits */
} CSSM_KEYSIZE, *CSSM_KEYSIZE_PTR;
```

Definition

KeySizeInBits
The actual number of bits in a key.

EffectiveKeySizeInBits
The number of key bits that can be used for cryptographic operations.

11.2.8 CSSM_KEYHEADER

The key header contains meta-data about a key. It contains the GUID of the CSP that owns the data. Attributes of the key are defined by the CSP and the application when the key is created. Most of these attributes describe both the CSP-stored copy of the key and the application's local copy of the key or the key reference. A subset of the attributes describe only the application-resident copy of the key or the key reference. A table at the end of this section summarizes the scope of each key header attribute.

```
typedef struct cssm_keyheader {
    CSSM_HEADERVERSION HeaderVersion; /* Key header version */
    CSSM_GUID CspId; /* GUID of CSP generating the key */
    uint32 BlobType; /* See BlobType #define's */
    uint32 Format; /* Raw or Reference format */
    uint32 AlgorithmId; /* Algorithm ID of key */
    uint32 KeyClass; /* Public/Private/Secret, etc. */
    uint32 EffectiveKeySizeInBits; /* Size of logical
                                      key/modulus/prime in bits */
    uint32 KeySizeInBits; /* Size of actual key/modulus/prime
                                        in bits */
    uint32 KeyAttr; /* Attribute flags */
    uint32 KeyUsage; /* Key use flags */
    CSSM_DATE StartDate; /* Effective date of key */
    CSSM_DATE EndDate; /* Expiration date of key */
    uint32 WrapAlgorithmId; /* == CSSM_ALGID_NONE if clear key */
    uint32 WrapMode; /* if alg supports multiple wrapping modes */
    uint32 Reserved;
} CSSM_KEYHEADER, *CSSM_KEYHEADER_PTR;
```

Definition

HeaderVersion
This is the version of the keyheader structure. The current version is represented by the defined constant CSSM_KEYHEADER_VERSION.

CspId
If known, the GUID of the CSP that generated the key. This value will not be known if a key is received from a third party, or extracted from a certificate.

BlobType
Describes the basic format of the key data. It can be any one of the following values:

Keyblob Type Identifier	**Description**
CSSM_KEYBLOB_RAW	The blob is a clear, raw key
CSSM_KEYBLOB_RAW_BERDER	The blob is a clear key, DER encoded
CSSM_KEYBLOB_REFERENCE	The blob is a reference to a key
CSSM_KEYBLOB_WRAPPED	The blob is a wrapped RAW key
CSSM_KEYBLOB_WRAPPED_BERDER	The blob is a wrapped DER encoded key
CSSM_KEYBLOB_OTHER	The blob is a wrapped DER encoded key

Format
Describes the detailed format of the key data based on the value of the BlobType field. If the blob type has a non-reference basic type, then a CSSM_KEYBLOB_RAW_FORMAT identifier must be used, otherwise a CSSM_KEYBLOB_REF_FORMAT identifier is used. Any of the following values are valid as format identifiers.

Keyblob Format Identifier	**Description**
CSSM_KEYBLOB_RAW_FORMAT_NONE	Raw format is unknown
CSSM_KEYBLOB_RAW_FORMAT_PKCS1	RSA PKCS1 V1.5 See "RSA Encryption Standard", an RSA Laboratories publication *http://www.rsa.com/rsalabs/pubs/PKCS/*
CSSM_KEYBLOB_RAW_FORMAT_PKCS3	RSA PKCS3 V1.5 See"Diffie-Hellman Key-Agreement Standard", an RSA Laboratories publication *http://www.rsa.com/rsalabs/pubs/PKCS/*
CSSM_KEYBLOB_RAW_FORMAT_MSCAPI	Microsoft CAPI V2.0
CSSM_KEYBLOB_RAW_FORMAT_PGP	PGP See "PGP Cryptographic Software Development Kit (PGP sdk)", a PGP Publication
CSSM_KEYBLOB_RAW_FORMAT_FIPS186	US Gov. FIPS 186: DSS V
CSSM_KEYBLOB_RAW_FORMAT_BSAFE	RSA Bsafe V3.0 See "BSAFE, A Cryptographic Toolkit, Library Reference Manual", an RSA Data Security Inc. publication
CSSM_KEYBLOB_RAW_FORMAT_PKCS8	RSA PKCS8 V1.2 See "Private-Key Information Syntax Standard", an RSA Laboratories publication *http://www.rsa.com/rsalabs/pubs/PKCS/"*
CSSM_KEYBLOB_RAW_FORMAT_PKCS11	RSA PKCS11 V2.0 See "Cryptographic Token Interface Standard", an RSA Laboratories publication *http://www.rsa.com/rsalabs/pubs/PKCS/*
CSSM_KEYBLOB_RAW_FORMAT_CDSA	CDSA format See this specifications and *CSSM Cryptographic Service Provider Interface Specification*
CSSM_KEYBLOB_RAW_FORMAT_OTHER	Other, CSP defined
CSSM_KEYBLOB_REF_FORMAT_INTEGER	Reference is a number or handle
CSSM_KEYBLOB_REF_FORMAT_STRING	Reference is a string or name
CSSM_KEYBLOB_REF_FORMAT_OTHER	Reference is a CSP-defined format

AlgorithmId

The algorithm for which the key was generated. This value does not change when the key is wrapped. Any of the defined CSSM algorithm IDs may be used.

KeyClass

Class of key contained in the key blob. Valid key classes are as follows:

Key Class Identifier	**Description**
CSSM_KEYCLASS_PUBLIC_KEY	Key is a public key
CSSM_KEYCLASS_PRIVATE_KEY	Key is a private key
CSSM_KEYCLASS_SESSION_KEY	Key is a session or symmetric key
CSSM_KEYCLASS_SECRET_PART	Key is part of secret key
CSSM_KEYCLASS_OTHER	Other

EffectiveKeySizeInBits

This is the logical size of the key in bits. The logical size is the value referred to when describing the length of the key. For instance, an RSA key would be described by the size of its modulus and a DSA key would be represented by the size of its prime. Symmetric key sizes describe the actual number of bits in the key. For example, DES keys would be 64 bits and an RC4 key could range from 1 to 128 bits.

KeyAttr

Attributes of the key represented by the data. These attributes are used by CSPs and applications to convey information about stored or referenced keys. Some of the attribute values are used only as input or output values for CSP functions, can appear in a keyheader, and some can be used only by the CSP. The attributes are represented by a bitmask. The attribute name, its description, and its usage constraints are summarized in the following:

Attribute values valid only as inputs to functions and will never appear in a key header:	
Attribute	**Description**
CSSM_KEYATTR_RETURN_DEFAULT	Key is returned in CSP's default form.
CSSM_KEYATTR_RETURN_DATA	Key is returned with key bits present. The format of the returned key can be raw or wrapped.
CSSM_KEYATTR_RETURN_REF	Key is returned as a reference.
CSSM_KEYATTR_RETURN_NONE	Key is not returned.
Attribute values valid as inputs to functions and retained values in a key header:	
Attribute	**Description**
CSSM_KEYATTR_PERMANENT	Key is stored persistently in the CSP, such asa PKCS11 token object.
CSSM_KEYATTR_PRIVATE	Key is a private object and protected by either a user login, a password, or both.
CSSM_KEYATTR_MODIFIABLE	The key or its attributes can be modified.
CSSM_KEYATTR_SENSITIVE	Key is sensitive. It may only be extracted from the CSP in a wrapped state.
CSSM_KEYATTR_EXTRACTABLE	Key is extractable from the CSP. If this bit is not set, either the key is not stored in the CSP, or it cannot be extracted under any circumstances.
Attribute values valid in a key header when set by a CSP:	

Attribute	**Description**
CSSM_KEYATTR_ALWAYS_SENSITIVE	Key has always been sensitive.
CSSM_KEYATTR_NEVER_EXTRACTABLE	Key has never been extractable.

Key Usage

A bitmask representing the valid uses of the key. Any of the following values are valid:

Usage Mask	**Description**
CSSM_KEYUSE_ANY	Key may be used for any purpose supported by the algorithm.
CSSM_KEYUSE_ENCRYPT	Key may be used for encryption.
CSSM_KEYUSE_DECRYPT	Key may be used for decryption.
CSSM_KEYUSE_SIGN	Key can be used to generate signatures. For symmetric keys this represents the ability to generate MACs.
CSSM_KEYUSE_VERIFY	Key can be used to verify signatures. For symmetric keys this represents the ability to verify MACs.
CSSM_KEYUSE_SIGN_RECOVER	Key can be used to perform signatures with message recovery. This form of a signature is generated using the *CSSM_EncryptData* API with the algorithm mode set to
CSSM_ALGMODE_PUBLIC_KEY	This attribute is only valid for asymmetric algorithms.
CSSM_KEYUSE_VERIFY_RECOVER	Key can be used to verify signatures with message recovery. This form of a signature verified using the *CSSM_DecryptData* API with the algorithm mode set to
CSSM_ALGMODE_PUBLIC_KEY	This attribute is only valid for asymmetric algorithms.
CSSM_KEYUSE_WRAP	Key can be used to wrap another key.
CSSM_KEYUSE_UNWRAP	Key can be used to unwrap a key.
CSSM_KEYUSE_DERIVE	Key can be used as the source for deriving other keys.

StartDate

Date from which the corresponding key is valid. All fields of the CSSM_DATA structure will be set to zero if the date is unspecified or unknown.

EndDate
Data that the key expires and can no longer be used. All fields of the CSSM_DATA structure will be set to zero is the date is unspecified or unknown.

WrapAlgorithmId
If the key data contains a wrapped key, this field contains the algorithm used to create the wrapped blob. This field will be set to CSSM_ALGID_NONE if the key is not wrapped.

WrapMode
If the wrapping algorithm supports multiple wrapping modes, this field contains the mode used to wrap the key. This field is ignored if the WrapAlgorithmId is CSSM_ALGID_NONE.

Reserved
This field is reserved for future use. It should always be set to zero.

The scope of the key header attributes is summarized as follows:

Attribute Name	**Pertains to the Application's local copy of the key**	**Pertains to the CSP-stored copy of the key**
BlobType	X	
Format	X	
AlgorithmId	X	X
KeyClass	X	X
EffectiveKeySizeInBits	X	X
KeyAttr	Only the flag bits RETURN_XXX	All the flag bits except RETURN_XXX
KeyUsage	X	X
StartDate	X	X
EndDate	X	X
WrapAlgorithmId	X	
WrapMode	X	

11.2.9 CSSM_KEY

This structure is used to represent keys in CSSM.

```
typedef struct cssm_key {
    CSSM_KEYHEADER KeyHeader; /* Fixed length key header */
    CSSM_DATA KeyData; /* Variable length key data */
} CSSM_KEY, *CSSM_KEY_PTR;
```

Definition

KeyHeader
Header describing the key.

KeyData
Data representation of the key.

11.2.10 CSSM_WRAP_KEY

This type is used to reference keys that are known to be in wrapped form.

```
typedef CSSM_KEY CSSM_WRAP_KEY, *CSSM_WRAP_KEY_PTR;
```

11.2.11 CSSM_CSP_TYPE

```
typedef enum cssm_csptype {
    CSSM_CSP_SOFTWARE = 1,
    CSSM_CSP_HARDWARE = CSSM_CSP_SOFTWARE+1,
    CSSM_CSP_HYBRID = CSSM_CSP_SOFTWARE+2,
}CSSM_CSPTYPE;
```

11.2.12 CSSM_CSP_SESSION_TYPE

A session type flags is used as an input parameter to the *CSSM_ModuleAttach* function to declare the type of session requested by the caller.

```
#define CSSM_CSP_SESSION_EXCLUSIVE 0x0001
       /* single user CSP */
#define CSSM_CSP_SESSION_READWRITE 0x0002
       /* caller can read and write objects such as keys in
                                    the CSP */
#define CSSM_CSP_SESSION_SERIAL 0x0004
       /* multi-user, re-entrant CSP that requires serial
                                    access */
```

11.2.13 CSSM_PADDING

Enumerates the padding options that can be provided by a CSP.

```
typedef enum cssm_padding {
    CSSM_PADDING_NONE = 0,
    CSSM_PADDING_CUSTOM = CSSM_PADDING_NONE+1,
    CSSM_PADDING_ZERO = CSSM_PADDING_NONE+2,
    CSSM_PADDING_ONE = CSSM_PADDING_NONE+3,
    CSSM_PADDING_ALTERNATE = CSSM_PADDING_NONE+4,
    CSSM_PADDING_FF = CSSM_PADDING_NONE+5,
    CSSM_PADDING_PKCS5 = CSSM_PADDING_NONE+6,
    CSSM_PADDING_PKCS7 = CSSM_PADDING_NONE+7,
    CSSM_PADDING_CipherStealing = CSSM_PADDING_NONE+8,
    CSSM_PADDING_RANDOM = CSSM_PADDING_NONE+9,
} CSSM_PADDING.
```

11.2.14 CSSM_CONTEXT_ATTRIBUTE

```
typedef struct cssm_context_attribute{
    uint32 AttributeType;
    uint32 AttributeLength;
    union cssm_context_attribute_value{
        char *String;
        uint32 Uint32;
        CSSM_CRYPTO_DATA_PTR Crypto;
        CSSM_KEY_PTR Key;
        CSSM_DATA_PTR Data;
        CSSM_DATE_PTR Date;
        CSSM_RANGE_PTR Range;
        CSSM_VERSION_PTR Version;
    } Attribute;
} CSSM_CONTEXT_ATTRIBUTE, *CSSM_CONTEXT_ATTRIBUTE_PTR;
```

Definition

AttributeType

An identifier describing the type of attribute. Valid attribute types are as follows:

Value	Description	Data Type
CSSM_ATTRIBUTE_NONE	No attribute	None
CSSM_ATTRIBUTE_CUSTOM	Custom data	Opaque pointer
CSSM_ATTRIBUTE_DESCRIPTION	Description of attribute	String
CSSM_ATTRIBUTE_KEY	Key Data	CSSM_KEY
CSSM_ATTRIBUTE_INIT_VECTOR	Initialization vector	CSSM_DATA
CSSM_ATTRIBUTE_SALT	Salt	CSSM_DATA
CSSM_ATTRIBUTE_PADDING	Padding information	CSSM_PADDING
CSSM_ATTRIBUTE_RANDOM	Random data	CSSM_DATA
CSSM_ATTRIBUTE_SEED	Seed	CSSM_CRYPTO_DATA
CSSM_ATTRIBUTE_PASSPHRASE	Pass phrase	CSSM_CRYPTO_DATA
CSSM_ATTRIBUTE_KEY_LENGTH	Key length specified in bits	uint32
CSSM_ATTRIBUTE_KEY_LENGTH_RANGE	Key length range specified in bits	CSSM_RANGE
CSSM_ATTRIBUTE_BLOCK_SIZE	Block size	uint32
CSSM_ATTRIBUTE_OUTPUT_SIZE	Output size	uint32
CSSM_ATTRIBUTE_ROUNDS	Number of runs or rounds	uint32
CSSM_ATTRIBUTE_IV_SIZE	Size of initialization vector	uint32
CSSM_ATTRIBUTE_ALG_PARAMS	Algorithm parameters	CSSM_DATA
CSSM_ATTRIBUTE_LABEL	Label placed on an object when it is created	CSSM_DATA
CSSM_ATTRIBUTE_KEY_TYPE	Type of key to generate or derive	uint32

CSSM_ATTRIBUTE_MODE	Algorithm mode to use for encryption	uint32
CSSM_ATTRIBUTE_EFFECTIVE_BITS	Number of effective bits used in the RC2 cipher	uint32
CSSM_ATTRIBUTE_START_DATE	Starting date for an object's validity	CSSM_DATE
CSSM_ATTRIBUTE_END_DATE	Ending date for an object's validity	CSSM_DATE
CSSM_ATTRIBUTE_KEYUSAGE	Usage restriction on the key	uint32
CSSM_ATTRIBUTE_KEYATTR	Key attribute	uint32
CSSM_ATTRIBUTE_VERSION	Version number	CSSM_VERSION
CSSM_ATTRIBUTE_PRIME	Prime value	CSSM_DATA
CSSM_ATTRIBUTE_BASE	Base Value	CSSM_DATA
CSSM_ATTRIBUTE_SUBPRIME	Subprime Value	CSSM_DATA
CSSM_ATTRIBUTE_ALG_ID	Algorithm identifier	uint32
CSSM_ATTRIBUTE_ITERATION_COUNT	Algorithm iterations	uint32
CSSM_ATTRIBUTE_ROUNDS_RANGE	Range of number of rounds possible	CSSM_RANGE

The data referenced by a CSSM_ATTRIBUTE_CUSTOM attribute must be a single continuous memory block. This allows the CSSM to appropriately release all dynamically allocated memory resources.

AttributeLength
Length of the attribute data.

Attribute
Union representing the attribute data. The union member used is named after the type of data contained in the attribute. See the attribute types table for the data types associated with each attribute type

11.2.15 CSSM_CONTEXT

```
typedef uint32 CSSM_CC_HANDLE /* Cryptographic Context Handle */

typedef struct cssm_context {
    uint32 ContextType; /* context type */
    uint32 AlgorithmType; /* algorithm type of context */
    uint32 Reserve; /* reserved for future use */
    uint32 NumberOfAttributes; /* number of attributes associated
                                    with context */
    CSSM_CONTEXT_ATTRIBUTE_PTR ContextAttributes; /* pointer to
                                    attributes */
} CSSM_CONTEXT, *CSSM_CONTEXT_PTR
```

Definition

ContextType
An identifier describing the type of services for this context.

Value	**Description**
CSSM_ALGCLASS_NONE	Null Context type
CSSM_ALGCLASS_CUSTOM	Custom Algorithms
CSSM_ALGCLASS_KEYXCH	Key Exchange Algorithms
CSSM_ALGCLASS_SIGNATURE	Signature Algorithms
CSSM_ALGCLASS_SYMMETRIC	Symmetric Encryption Algorithms
CSSM_ALGCLASS_DIGEST	Message Digest Algorithms
CSSM_ALGCLASS_RANDOMGEN	Random Number Generation Algorithms
CSSM_ALGCLASS_UNIQUEGEN	Unique ID Generation Algorithms
CSSM_ALGCLASS_MAC	Message Authentication Code Algorithms
CSSM_ALGCLASS_ASYMMETRIC	Asymmetric Encryption Algorithms
CSSM_ALGCLASS_KEYGEN	Key Generation Algorithms
CSSM_ALGCLASS_DERIVEKEY	Key Derivation Algorithms

AlgorithmType
An ID number describing the algorithm to be used.

Value	**Description**
CSSM_ALGID_NONE	Null algorithm
CSSM_ALGID_CUSTOM	Custom algorithm
CSSM_ALGID_DH	Diffie Hellman key exchange algorithm
CSSM_ALGID_PH	Pohlig Hellman key exchange algorithm
CSSM_ALGID_KEA	Key Exchange Algorithm
CSSM_ALGID_MD2	MD2 hash algorithm
CSSM_ALGID_MD4	MD4 hash algorithm
CSSM_ALGID_MD5	MD5 hash algorithm
CSSM_ALGID_SHA1	Secure Hash Algorithm
CSSM_ALGID_NHASH	N-Hash algorithm
CSSM_ALGID_HAVAL	HAVAL hash algorithm (MD5 variant)
CSSM_ALGID_RIPEMD	RIPE-MD hash algorithm (MD4 variant developed for the European Community's RIPE project)
CSSM_ALGID_IBCHASH	IBC-Hash (keyed hash algorithm or MAC)
CSSM_ALGID_RIPEMAC	RIPE-MAC
CSSM_ALGID_HASHwithHitachi	Hitachi hash algorithm
CSSM_ALGID_DES	Data Encryption Standard block cipher
CSSM_ALGID_DESX	DESX block cipher (DES variant from RSA)
CSSM_ALGID_RDES	RDES block cipher (DES variant)
CSSM_ALGID_3DES_3KEY	Triple-DES block cipher (with 3 keys)
CSSM_ALGID_3DES_2KEY	Triple-DES block cipher (with 2 keys)
CSSM_ALGID_3DES_1KEY	Triple-DES block cipher (with 1 key)
CSSM_ALGID_IDEA	IDEA block cipher
CSSM_ALGID_RC2	RC2 block cipher
CSSM_ALGID_RC5	RC5 block cipher
CSSM_ALGID_RC4	RC4 stream cipher
CSSM_ALGID_SEAL	SEAL stream cipher
CSSM_ALGID_CAST	CAST block cipher
CSSM_ALGID_BLOWFISH	BLOWFISH block cipher
CSSM_ALGID_SKIPJACK	Skipjack block cipher
CSSM_ALGID_LUCIFER	Lucifer block cipher
CSSM_ALGID_MADRYGA	Madryga block cipher
CSSM_ALGID_FEAL	FEAL block cipher
CSSM_ALGID_REDOC	REDOC 2 block cipher
CSSM_ALGID_REDOC3	REDOC 3 block cipher
CSSM_ALGID_LOKI	LOKI block cipher
CSSM_ALGID_KHUFU	KHUFU block cipher
CSSM_ALGID_KHAFRE	KHAFRE block cipher
CSSM_ALGID_MMB	MMB block cipher (IDEA variant)

CSSM_ALGID_GOST	GOST block cipher
CSSM_ALGID_SAFER	SAFER K-40, K-64, K-128 block cipher
CSSM_ALGID_CRAB	CRAB block cipher
CSSM_ALGID_MULTI2	MULTI2 block cipher algorithm(MULTI variant from Hitachi)
CSSM_ALGID_RSA	RSA public key cipher
CSSM_ALGID_CIPHERwithHitachiECCS	Hitachi's public key cipher algorithm with Elliptic Curve Cryptosystems
CSSM_ALGID_DSA	Digital Signature Algorithm
CSSM_ALGID_MD5WithRSA	MD5/RSA signature algorithm
CSSM_ALGID_MD2WithRSA	MD2/RSA signature algorithm
CSSM_ALGID_SIGwithHitachiECCS	Hitachi's signature algorithm with Elliptic Curve Cryptosystems
CSSM_ALGID_ElGamal	ElGamal signature algorithm
CSSM_ALGID_MD2Random	MD2-based random numbers
CSSM_ALGID_MD5Random	MD5-based random numbers
CSSM_ALGID_SHARandom	SHA-based random numbers
CSSM_ALGID_DESRandom	DES-based random numbers
CSSM_ALGID_MULTI2Random	MULTI2-based random numbers
CSSM_ALGID_SHA1WithRSA	SHA-1/RSA signature algorithm
CSSM_ALGID_RSA_PKCS	RSA as specified in PKCS #1
CSSM_ALGID_RSA_ISO9796	RSA as specified in ISO 9796
CSSM_ALGID_RSA_RAW	Raw RSA as assumed in X.509
CSSM_ALGID_CDMF	CDMF block cipher
CSSM_ALGID_CAST3	Entrust's CAST3 block cipher
CSSM_ALGID_CAST5	Entrust's CAST5 block cipher
CSSM_ALGID_GenericSecret	Generic secret operations
CSSM_ALGID_ConcatBaseAndKey	Concatenate two keys, base key first
CSSM_ALGID_ConcatKeyAndBase	Concatenate two keys, base key last
CSSM_ALGID_ConcatBaseAndData	Concatenate base key and random data, key first
CSSM_ALGID_ConcatDataAndBase	Concatenate base key and data, data first
CSSM_ALGID_XORBaseAndData	XOR a byte string with the base key
CSSM_ALGID_ExtractFromKey	Extract a key from base key, starting at arbitrary bit position
CSSM_ALGID_SSL3PreMasterGen	Generate a 48 byte SSL 3 pre-master key
CSSM_ALGID_SSL3MasterDerive	Derive an SSL 3 key from a pre-master key
CSSM_ALGID_SSL3KeyAndMacDerive	Derive the keys and MACing keys for the SSL cipher suite
CSSM_ALGID_SSL3MD5_MAC	Performs SSL 3 MD5 MACing
CSSM_ALGID_SSL3SHA1_MAC	Performs SSL 3 SHA-1 MACing

CSSM_ALGID_MD5_PBE	Generate key by MD5 hashing a base key
CSSM_ALGID_MD2_PBE	Generate key by MD2 hashing a base key
CSSM_ALGID_SHA1_PBE	Generate key by SHA-1 hashing a base key
CSSM_ALGID_WrapLynks	Spyrus LYNKS DES based wrapping scheme w/checksum
CSSM_ALGID_WrapSET_OAEP	SET key wrapping
CSSM_ALGID_BATON	Fortezza BATON cipher
CSSM_ALGID_ECDSA	Elliptic Curve DSA
CSSM_ALGID_MAYFLY	Fortezza MAYFLY cipher
CSSM_ALGID_JUNIPER	Fortezza JUNIPER cipher
CSSM_ALGID_FASTHASH	Fortezza FASTHASH
CSSM_ALGID_3DES	Generix 3DES
CSSM_ALGID_SSL3MD5	SSL3 with MD5
CSSM_ALGID_SSL3SHA1	SSL3 with SHA1
CSSM_ALGID_FortezzaTimestamp	Fortezza with Timestamp
CSSM_ALGID_SHA1WithDSA	SHA1 with DSA
CSSM_ALGID_SHA1WithECDSA	SHA1 with Elliptic Curve DSA
CSSM_ALGID_DSA_BSAFE	DSA with BSAFE Key format
CSSM_ALGID_Bcrypt	BSI algorithm
CSSM_ALGID_LUCpkcds	LUC Public key crypto and Dig Sig Alg
CSSM_ALGID_BARAS	
CSSM_ALGID_SxalMbal	Substitution Xor Alg / Multi Block Alg
CSSM_ALGID_MISTY1	Block Cipher
CSSM_ALGID_ENCRIP	

Some of the above algorithms operate in a variety of modes. The desired mode is specified using an attribute of type CSSM_ATTRIBUTE_MODE. The valid values for the mode attribute are as follows:

Value	**Description**
CSSM_ALGMODE_NONE	Null Algorithm mode
CSSM_ALGMODE_CUSTOM	Custom mode
CSSM_ALGMODE_ECB	Electronic Code Book
CSSM_ALGMODE_ECBPad	ECB with padding
CSSM_ALGMODE_CBC	Cipher Block Chaining
CSSM_ALGMODE_CBC_IV8	CBC with Initialization Vector of 8 bytes
CSSM_ALGMODE_CBCPadIV8	CBC with padding and Initialization Vector of 8 bytes
CSSM_ALGMODE_CFB	Cipher FeedBack
CSSM_ALGMODE_CFB_IV8	CFB with Initialization Vector of 8 bytes
CSSM_ALGMODE_CFBPadIV8	CFB with Initialization Vector of 8 bytes and padding
CSSM_ALGMODE_OFB	Output FeedBack
CSSM_ALGMODE_OFB_IV8	OFB with Initialization Vector of 8 bytes
CSSM_ALGMODE_OFBPadIV8	OFB with Initialization Vector of 8 bytes and padding
CSSM_ALGMODE_COUNTER	Counter
CSSM_ALGMODE_BC	Block Chaining
CSSM_ALGMODE_PCBC	Propagating CBC
CSSM_ALGMODE_CBCC	CBC with Checksum
CSSM_ALGMODE_OFBNLF	OFB with NonLinear Function
CSSM_ALGMODE_PBC	Plaintext Block Chaining
CSSM_ALGMODE_PFB	Plaintext FeedBack
CSSM_ALGMODE_CBCPD	CBC of Plaintext Difference
CSSM_ALGMODE_PUBLIC_KEY	Use the public key
CSSM_ALGMODE_PRIVATE_KEY	Use the private key
CSSM_ALGMODE_SHUFFLE	Fortezza shuffle mode
CSSM_ALGMODE_ECB64	Electronic Code Book 64 bytes
CSSM_ALGMODE_CBC64	Cipher Block Chaining 64 bytes
CSSM_ALGMODE_OFB64	Output Feedback 64 bytes
CSSM_ALGMODE_CFB64	Cipher Feedback 64 bytes
CSSM_ALGMODE_CFB32	Cipher Feedback 32 bytes
CSSM_ALGMODE_CFB16	Cipher Feedback 16 bytes
CSSM_ALGMODE_CFB8	Cipher Feedback 8 bytes
CSSM_ALGMODE_WRAP	
CSSM_ALGMODE_PRIVATE_WRAP	
CSSM_ALGMODE_RELAYX	
CSSM_ALGMODE_ECB128	Electronic Code Book 128 bytes
CSSM_ALGMODE_ECB96	Electronic Code Book 96 bytes
CSSM_ALGMODE_CBC128	Cipher Block Chaining 128 bytes
CSSM_ALGMODE_OAEP_HASH	Algorithm mode for SET key wrapping

NumberOfAttributes
Number of attributes associated with this service.

ContextAttributes
Pointer to data that describes the attributes. To retrieve the next attribute, advance the attribute pointer.

11.2.16 CSSM_CSP_CAPABILITY

```
typedef CSSM_CONTEXT CSSM_CSP_CAPABILITY, *CSSM_CSP_CAPABILITY_PTR;
```

11.2.17 CSSM_SOFTWARE_CSPSUBSERVICE_INFO

```
typedef struct cssm_software_cspsubservice_info {
    uint32 NumberOfCapabilities;
    CSSM_CSP_CAPABILITY_PTR CapabilityList;
    VOID* Reserved;
} CSSM_SOFTWARE_CSPSUBSERVICE_INFO, *CSSM_SOFTWARE_CSPSUBSERVICE_INFO_PTR;
```

Definition

NumberOfCapabilities
Number of capabilities available from the CSP.

CapabilityList
Pointer to an array of CSSM_CSP_CAPABILITY structures that represent the capabilities available from the CSP.

Reserved
This field is reserved for future use and must always be set to zero.

11.2.18 CSSM_HARDWARE_CSPSUBSERVICE_INFO

```
typedef struct cssm_hardware_cspsubservice_info {
    uint32 NumberOfCapabilities;
    CSSM_CSP_CAPABILITY_PTR CapabilityList;
    void* Reserved;

    /* Reader/Slot Info */
    char *ReaderDescription;
    char *ReaderVendor;
    char *ReaderSerialNumber;
    CSSM_VERSION ReaderHardwareVersion;
    CSSM_VERSION ReaderFirmwareVersion;
    uint32 ReaderFlags;
    uint32 ReaderCustomFlags;

    char *TokenDescription;
    char *TokenVendor;
    char *TokenSerialNumber;
    CSSM_VERSION TokenHardwareVersion;
    CSSM_VERSION TokenFirmwareVersion;

    uint32 TokenFlags;
    uint32 TokenCustomFlags;
    uint32 TokenMaxSessionCount;
    uint32 TokenOpenedSessionCount;
    uint32 TokenMaxRWSessionCount;
    uint32 TokenOpenedRWSessionCount;
    uint32 TokenTotalPublicMem;
    uint32 TokenFreePublicMem;
    uint32 TokenTotalPrivateMem;
    uint32 TokenFreePrivateMem;
    uint32 TokenMaxPinLen;
    uint32 TokenMinPinLen;
    char TokenUTCTime[16];

    CSSM_STRING UserLabel;
    CSSM_DATA UserCACertificate;
} CSSM_HARDWARE_CSPSUBSERVICE_INFO, *CSSM_HARDWARE_CSPSUBSERVICE_INFO_PTR;
```

Definition

NumberOfCapabilities
Number of capabilities available from the CSP.

CapabilityList
A context list that specifies the capabilities of the CSP.

Reserved
This field is reserved for future use and must always be set to zero.

ReaderDescription
A NULL-terminated character string that contains a text description of the device reader.

ReaderVendor
A NULL-terminated string that contains the name of the reader vendor.

ReaderSerialNumber
A NULL-terminated string that contains the serial number of the reader.

ReaderHardwareVersion
Hardware version of the reader.

ReaderFirmwareVersion
Firmware version of the reader.

ReaderFlags
Bit mask containing information about the reader. The flags specified in the mask are as follows:

Reader Flag	**Description**
CSSM_CSP_RDR_TOKENPRESENT	Token is present in the reader
CSSM_CSP_RDR_TOKENREMOVABLE	Reader supports removable tokens
CSSM_CSP_RDR_HW	Reader is a hardware device

ReaderCustomFlags
Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

The following fields may not be valid if the CSSM_CSP_RDR_TOKENPRESENT flag is not set in the ReaderFlags field. Unknown string and CSSM_DATA fields will be set to NULL, integer and date fields will be set to zero and flag fields will have all flags set to false.

TokenDescription
A NULL-terminated character string that contains a text description of the token. This value may be NULL or equal to ReaderDescription if the token is not removable.

TokenVendor
A NULL-terminated string that contains the name of the token vendor. This value may be NULL or equal to ReaderVendor if the token is not removable.

TokenSerialNumber
A NULL-terminated string that contains the serial number of the token. This value may be NULL or equal to ReaderSerialNumber if the token is not removable.

TokenHardwareVersion
Hardware version of the token.

TokenFirmwareVersion
Firmware version of the token.

TokenFlags
Bit mask containing information about the token. The flags specified in the mask are as follows:

Token Flags	**Description**
CSSM_CSP_TOK_RNG	Token has random number generator
CSSM_CSP_TOK_WRITE_PROTECTED	Token is write protected
CSSM_CSP_TOK_LOGIN_REQUIRED	User must login to access private objects
CSSM_CSP_TOK_USER_PIN_INITIALIZED	User’s PIN has been initialized
CSSM_CSP_TOK_EXCLUSIVE_SESSION	An exclusive session currently exists
CSSM_CSP_TOK_CLOCK_EXISTS	Token has built in clock
CSSM_CSP_TOK_ASYNC_SESSION	Token supports asynchronous operations
CSSM_CSP_TOK_PROT_AUTHENTICATION	Token has protected authentication path
CSSM_CSP_TOK_DUAL_CRYPTO_OPS	Token supports dual cryptographic operations

TokenCustomFlags
Flags defined by the vendor. Consult the individual CSP user’s guide for the list of valid flags.

TokenMaxSessionCount
Maximum number of CSP handles referencing the token that may exist simultaneously.

TokenOpenedSessionCount
Number of existing CSP handles referencing the token.

TokenTotalPublicMem
Amount of public storage space in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE if the CSP does not wish to expose this information.

TokenFreePublicMem
Amount of public storage space available for use in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE if the CSP does not wish to expose this information.

TokenTotalPrivateMem
Amount of private storage space in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE if the CSP does not wish to expose this information.

TokenFreePrivateMem
Amount of private storage space available for use in the CSP. This value will be set to

CSSM_VALUE_NOT_AVAILABLE if the CSP does not wish to expose this information.

TokenMaxPinLen
Maximum length of passwords that can be used for authentication to the CSP.

TokenMinPinLen
Minimum length of passwords that can be used for authentication to the CSP.

TokenUTCTime
Character array containing the current UTC time value in the CSP. The value is valid if the CSSM_CSP_TOK_CLOCK_EXISTS flag is true. The time is represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second; and 2 additional reserved '0' characters).

UserLabel
A NULL-terminated string containing the label of the token.

UserCACertificate
Certificate of the CA.

11.2.19 CSSM_HYBRID_CSPSUBSERVICE_INFO

```
typedef CSSM_HYBRID_CSPSUBSERVICE_INFO
                    CSSM_HARDWARE_CSPSUBSERVICE_INFO
```

11.2.20 CSSM_CSP_WRAPPEDPRODUCTINFO

```
typedef struct cssm_csp_wrappedproductinfo {
    CSSM_VERSION StandardVersion;
    CSSM_STRING StandardDescription;
    CSSM_VERSION ProductVersion;
    CSSM_STRING ProductDescription;
    CSSM_STRING ProductVendor;
    uint32 ProductFlags;
    uint32 ProductCustomFlags;
} CSSM_CSP_WRAPPEDPRODUCTINFO, *CSSM_CSP_WRAPPEDPRODUCTINFO_PTR;
```

Definition

StandardVersion
Version of the standard to which the wrapped product complies.

StandardDescription
A CSSM character string containing a text description of the standard to which the wrapped product complies.

ProductVersion
Version of the product wrapped by the CSP.

ProductDescription
A CSSM character string containing a text description of the product wrapped by the CSP.

ProductVendor
A CSSM character string containing the name of the wrapped product's vendor.

ProductFlags
This version of CSSM has no flags defined. This field must be set to zero.

ProductCustomFlags
Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

11.2.21 CSSM_CSP_FLAGS

A bit mask containing information about the CSP. The mask may be a combination of any of the following:

```
typedef uint32 CSSM_CSP_FLAGS;

#define CSSM_CSP_STORES_PRIVATE_KEYS
#define CSSM_CSP_STORES_PUBLIC_KEYS
#define CSSM_CSP_STORES_SESSION_KEYS
```

11.2.22 CSSM_CSPSUBSERVICE

```
typedef struct cssm_cspsubservice {
    uint32 SubServiceId;
    CSSM_STRING Description;
    CSSM_CSP_FLAGS CspFlags;
    uint32 CspCustomFlags;
    uint32 AccessFlags;
    CSSM_CSPTYPE CspType;
    union cssm_subservice_info{
        CSSM_SOFTWARE_CSPSUBSERVICE_INFO SoftwareCspSubService;
        CSSM_HARDWARE_CSPSUBSERVICE_INFO HardwareCspSubService;
        CSSM_HYBRID_CSPSUBSERVICE_INFO HybridCspSubService;
    } SubServiceInfo;
    CSSM_CSP_WRAPPEDPRODUCTINFO WrappedProduct;
} CSSM_CSPSUBSERVICE, *CSSM_CSPSUBSERVICE_PTR;
```

Definition

SubServiceId
The sub-service ID required for an attach call to connect a CSP to an individual sub-service within a CSP.

Description
A CSSM character string containing a text description of the sub-service.

CspFlags
CSSM-defined flags indicating the key storage services provided by the CSP.

CspCustomFlags
Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

AccessFlags
Flags that are required to be provided by the application during an attach call when specifying the sub-service ID given in SubServiceId.

CspType
Identifier that determines the type of CSP information structure referenced by CspInfo. The following values and their corresponding CSP information structures are currently defined.

CSP Information Structure Identifier	**Structure Type**
CSSM_CSP_SOFTWARE	CSSM_SOFTWARE_CSPSUBSERVICE_INFO
CSSM_CSP_HARDWARE	CSSM_HARDWARE_CSPSUBSERVICE_INFO

SubServiceInfo
A CSP sub-service information structure of the type specified by CspType.

WrappedProduct
A CSSM_CSP_WRAPPEDPRODUCTINFO structure describing a product that is wrapped by the CSP.

11.3 Cryptographic Context Operations

The manpages for Cryptographic Context Operations follow on the next page.

NAME

CSSM_CSP_CreateSignatureContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateSignatureContext
    (CSSM_CSP_HANDLE CSPHandle,
    uint32 AlgorithmID,
    const CSSM_CRYPTO_DATA_PTR PassPhrase,
    const CSSM_KEY_PTR Key)
```

DESCRIPTION

This function creates a signature cryptographic context for sign and verify given a handle of a CSP, an algorithm identification number, a key, and a passphrase structure. The passphrase will be used to unlock the private key when this context is used to perform a signing operation. The cryptographic context handle is returned. The cryptographic context handle can be used to call sign and verify cryptographic functions.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

AlgorithmID (input)
The algorithm identification number for a signature/verification algorithm.

PassPhrase (input)
The passphrase is required to unlock the private key. The passphrase structure accepts an immediate value for the passphrase or the caller can specify a callback function the CSP can use to obtain the passphrase. The passphrase is needed only for signature operations, not verify operations.

Key (input)
The key used to sign. The caller passes in a pointer to a CSSM_KEY structure containing the key and the key length.

RETURN VALUE

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__INVALID_CSP_HANDLE
Invalid provider handle.

CSSM__MEMORY_ERROR
Internal memory error.

SEE ALSO

CSSM_SignData, *CSSM_SignDataInit*, *CSSM_SignDataUpdate*, *CSSM_SignDataFinal*, *CSSM_VerifyData*, *CSSM_VerifyDataInit*, *CSSM_VerifyDataUpdate*, *CSSM_VerifyDataFinal*, *CSSM_GetContext*, *CSSM_SetContext*, *CSSM_DeleteContext*, *CSSM_GetContextAttribute*, *CSSM_UpdateContextAttributes*

NAME

CSSM_CSP_CreateSymmetricContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateSymmetricContext
    (CSSM_CSP_HANDLE CSPHandle,
    uint32 AlgorithmID,
    uint32 Mode,
    const CSSM_KEY_PTR Key,
    const CSSM_DATA_PTR InitVector,
    CSSM_PADDING Padding,
    uint32 Params)
```

DESCRIPTION

This function creates a symmetric encryption cryptographic context given a handle of a CSP, an algorithm identification number, a key, an initial vector, padding, and the number of encryption rounds. The cryptographic context handle is returned. The cryptographic context handle can be used to call symmetric encryption functions and the cryptographic wrap/unwrap functions.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

AlgorithmID (input)
The algorithm identification number for symmetric encryption.

Mode (input)
The mode of the specified algorithm ID.

Key (input)
The key used for symmetric encryption. The caller passes in a pointer to a CSSM_KEY structure containing the key. This key can be used directly for wrap and unwrap operations.

InitVector (input/optional)
The initial vector for symmetric encryption; typically specified for block ciphers.

Padding (input/optional)
The method for padding; typically specified for ciphers that pad.

Params (input/optional)
Specifiesany additional parameters required to perform encryption using the specified algorithm.

RETURN VALUE

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__INVALID_CSP_HANDLE
Invalid provider handle.

CSSM__MEMORY_ERROR
Internal memory error.

SEE ALSO

CSSM_EncryptData, *CSSM_QuerySize*, *CSSM_EncryptDataInit*, *CSSM_EncryptDataUpdate*, *CSSM_EncryptDataFinal*, *CSSM_DecryptData*, *CSSM_DecryptDataInit*, *CSSM_DecryptDataUpdate*, *CSSM_DecryptDataFinal*, *CSSM_GetContext*, *CSSM_SetContext*, *CSSM_DeleteContext*, *CSSM_GetContextAttribute, CSSM_UpdateContextAttributes*

NAME

CSSM_CSP_CreateDigestContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateDigestContext
    (CSSM_CSP_HANDLE CSPHandle,
     uint32 AlgorithmID)
```

DESCRIPTION

This function creates a digest cryptographic context, given a handle of a CSP and an algorithm identification number. The cryptographic context handle is returned. The cryptographic context handle can be used to call digest cryptographic functions.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

AlgorithmID (input)
The algorithm identification number for message digests.

RETURN VALUE

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__INVALID_CSP_HANDLE
Invalid crypto services provider handle.

CSSM__MEMORY_ERROR
Internal memory error.

SEE ALSO

CSSM_DigestData, *CSSM_DigestDataInit*, *CSSM_DigestDataUpdate*, *CSSM_DigestDataFinal*, *CSSM_GetContext*, *CSSM_SetContext*, *CSSM_DeleteContext*, *CSSM_GetContextAttribute*, *CSSM_UpdateContextAttributes*

NAME

CSSM_CSP_CreateMacContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateMacContext
    (CSSM_CSP_HANDLE CSPHandle,
    uint32 AlgorithmID,
    const CSSM_KEY_PTR Key)
```

DESCRIPTION

This function creates a message authentication code cryptographic context, given a handle of a CSP, algorithm identification number, key, and the length of the key in bits. The cryptographic context handle is returned. The cryptographic context handle can be used to call message authentication code functions.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

AlgorithmID (input)
The algorithm identification number for the MAC algorithm.

Key (input)
The key used to generate a message authentication code. Caller passes in a pointer to a CSSM_KEY structure containing the key.

RETURN VALUE

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__INVALID_CSP_HANDLE
Invalid crypto services provider handle.

CSSM__MEMORY_ERROR
Internal memory error.

SEE ALSO

CSSM_GenerateMac, *CSSM_GenerateMacInit*, *CSSM_GenerateMacUpdate*, *CSSM_GenerateMacFinal*, CSSM_VerifyMac, CSSM_VerifyMacInit, CSSM_VerifyMacUpdate, CSSM_VerifyMacFinal, *CSSM_GetContext*, *CSSM_SetContext*, *CSSM_DeleteContext*, *CSSM_GetContextAttribute*, *CSSM_UpdateContextAttributes*

NAME

CSSM_CSP_CreateRandomGenContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateRandomGenContext
    (CSSM_CSP_HANDLE CSPHandle,
    uint32 AlgorithmID,
    const CSSM_CRYPTO_DATA_PTR Seed,
    uint32 Length)
```

DESCRIPTION

This function creates a random number generation cryptographic context, given a handle of a CSP, an algorithm identification number, a seed, and the length of the random number in bytes. The cryptographic context handle is returned, and can be used for the random number generation function.

PARAMETERS

CSPHandle (input)
: The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

AlgorithmID (input)
: The algorithm identification number for random number generation.

Seed (input/optional)
: A seed used to generate random number. The caller can either pass a seed and seed length in bytes or pass in a callback function. If NULL is passed, the cryptographic service provider will use its default seed handling mechanism.

Length (input)
: The length of the random number to be generated.

RETURN VALUE

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__INVALID_CSP_HANDLE
: Invalid provider handle.

CSSM__MEMORY_ERROR
: Internal memory error.

SEE ALSO

CSSM_GenerateRandom, *CSSM_GetContext*, *CSSM_SetContext*, *CSSM_DeleteContext*, *CSSM_GetContextAttribute*, *CSSM_UpdateContextAttributes*

NAME

CSSM_CSP_CreateAsymmetricContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateAsymmetricContext
    (CSSM_CSP_HANDLE CSPHandle,
    uint32 AlgorithmID,
    const CSSM_CRYPTO_DATA_PTR PassPhrase,
    const CSSM_KEY_PTR Key,
    uint32 Padding)
```

DESCRIPTION

This function creates an asymmetric encryption cryptographic context, given a handle of a CSP, an algorithm identification number, a key, padding, and the key mode (CSSM_ALGMODE_PRIVATE_KEY or CSSM_ALGMODE_PUBLIC_KEY). The cryptographic context handle is returned. The cryptographic context handle can be used to call asymmetric encryption functions and cryptographic wrap/unwrap functions.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns an error.

AlgorithmID (input)
The algorithm identification number for the algorithm used for asymmetric encryption.

PassPhrase (input)
The passphrase is required to unlock the private key. The passphrase structure accepts an immediate value for the passphrase or the caller can specify a callback function the CSP can use to obtain the passphrase. The passphrase is needed only for signature operations, not verify operations. When the context is used for a wrap or unwrap operation, the passphrase can be used to generate a symmetric key for wrapping or unwrapping.

Key (input)
The key used for asymmetric encryption. The caller passes a pointer to a CSSM_KEY structure containing the key. When the context is used for a sign operation, the passphrase is required to access the private key used for signing. When the context is used for a verify operation, the public key is used to verify the signature. When the context is used for a wrapkey operation, the public key can be used as the wrapping key. When the context is used for an unwrap operation, the passphrase is required to access the private key used to perform the unwrapping.

Padding (input/optional)
The method for padding. Typically specified for ciphers that pad.

RETURN VALUE

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__INVALID_CSP_HANDLE
Invalid provider handle.

CSSM__MEMORY_ERROR
Internal memory error.

SEE ALSO

CSSM_EncryptData, *CSSM_QuerySize*, *CSSM_EncryptDataInit*, *CSSM_EncryptDataUpdate*, *CSSM_EncryptDataFinal*, *CSSM_DecryptData*, *CSSM_DecryptDataInit*, *CSSM_DecryptDataUpdate*, *CSSM_DecryptDataFinal*, *CSSM_GetContext*, *CSSM_SetContext*, *CSSM_DeleteContext*, *CSSM_GetContextAttribute, CSSM_UpdateContextAttributes*

NAME

CSSM_CSP_CreateDeriveKeyContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateDeriveKeyContext
    (CSSM_CSP_HANDLE CSPHandle,
    uint32 AlgorithmID,
    CSSM_KEY_TYPE DeriveKeyType,
    uint32 DeriveKeyLengthInBits,
    uint32 IterationCount,
    const CSSM_DATA_PTR Salt,
    const CSSM_CRYPTO_DATA_PTR Seed,
    const CSSM_CRYPTO_DATA_PTR PassPhrase)
```

DESCRIPTION

This function creates a cryptographic context to derive a symmetric key given a handle of a CSP, an algorithm, the type of symmetric key to derive, the length of the derived key, and an optional seed or an optional passphrase from which to derive a new key. The cryptographic context handle is returned. The cryptographic context handle can be used for calling the cryptographic derive key function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns an error.

AlgorithmID (input)
The algorithm identification number for a derived key algorithm.

DeriveKeyType (input)
The type of symmetric key to derive.

DeriveKeyLengthInBits (input)
The length of the key to derive in bits.

InterationCount (input/optional)
The number of iterations to be performed during the derivation process. Used heavily by password-based derivation methods.

Salt (input/optional)
A Salt used in deriving the key.

Seed (input/optional)
A seed used to generate a random number. The caller can either pass a seed and seed length in bytes or pass in a callback function. If NULL is passed, the cryptographic service provider will use its default seed handling mechanism.

PassPhrase (input/optional)
The passphrase is required to unlock the private key. The passphrase structure accepts an immediate value for the passphrase or the caller can specify a callback function the CSP can use to obtain the passphrase. The passphrase is needed only for signature operations, not verify operations.

RETURN VALUE

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__INVALID_CSP_HANDLE
Invalid provider handle.

CSSM__MEMORY_ERROR
Internal memory error.

SEE ALSO
CSSM_DeriveKey

NAME

CSSM_CSP_CreateKeyGenContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreateKeyGenContext
    (CSSM_CSP_HANDLE CSPHandle,
    uint32 AlgorithmID,
    const CSSM_CRYPTO_DATA_PTR PassPhrase,
    uint32 KeySizeInBits,
    const CSSM_CRYPTO_DATA_PTR Seed,
    const CSSM_DATA_PTR Salt,
    const CSSM_DATA_PTR StartDate,
    const CSSM_DATA_PTR EndDate,
    const CSSM_DATA_PTR Params)
```

DESCRIPTION

This function creates a key generation cryptographic context, given a handle of a CSP, an algorithm identification number, a pass phrase, a modulus size (for public/private keypair generation), a key size (for symmetric key generation), a seed, salt, and a label. The cryptographic context handle is returned. The cryptographic context handle can be used to call key / keypair generation functions.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

AlgorithmID (input)
The algorithm identification number of the algorithm used for key generation.

PassPhrase (input)
The passphrase is required to unlock the private key. The passphrase structure accepts an immediate value for the passphrase or the caller can specify a callback function the CSP can use to obtain the passphrase. The passphrase is needed only for signature operations, not verify operations. Once the new key is created, the passphrase or nickname must be provided in all future references to access the private or symmetric key.

KeySizeInBits (input)
The logical size of the key (specified in bits). This refers to either the actual key size (for symmetric key generation) or the modulus size (for asymmetric key pair generation). This is the effective key size.

Seed (input/optional)
A seed used to generate the key. The caller can either pass a seed and seed length in bytes or pass in a callback function. If NULL is passed, the cryptographic service provider will use its default seed handling mechanism.

Salt (input/optional)
A Salt used to generate the key.

StartDate (input/optional)
A start date for the validity period of the key or key pair being generated.

EndDate (input/optional)
An end date for the validity period of the key or key pair being generated.

Params (input/optional)
A data buffer containing parameters required to generate a key pair for a specific algorithm.

RETURN VALUE

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__INVALID_CSP_HANDLE
Invalid provider handle.

CSSM__MEMORY_ERROR
Internal memory error.

SEE ALSO

CSSM_GenerateKey, *CSSM_GenerateKey*Pair, *CSSM_GetContext*, *CSSM_SetContext*, *CSSM_DeleteContext*, *CSSM_GetContextAttribute*, *CSSM_UpdateContextAttributes*

NAME

CSSM_CSP_CreatePassThroughContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_CSP_CreatePassThroughContext
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_KEY_PTR Key,
    const CSSM_DATA_PTR ParamBufs,
    uint32 ParamBufCount)
```

DESCRIPTION

This function creates a custom cryptographic context, given a handle of a CSP and pointer to a custom input data structure. The cryptographic context handle is returned. The cryptographic context handle can be used to call the CSSM pass-through function for the CSP.

PARAMETERS

CSPHandle (input)
: The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

Key (input)
: The key to be used for the context. The caller passes in a pointer to a CSSM_KEY structure containing the key.

ParamBufs (input)
: Array of input buffers to the pass-through call.

ParamBufCount (input)
: The number of input buffers pointed to by ParamBufs.

RETURN VALUE

Returns a cryptographic context handle. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__INVALID_CSP_HANDLE
: Invalid provider handle.

CSSM__MEMORY_ERROR
: Internal memory error.

Comments

A CSP can create its own set of custom functions. The context information can be passed through its own data structure. The *CSSM_CSP_PassThrough* function should be used along with the function ID to call the desired custom function.

SEE ALSO

CSSM_CSP_PassThrough, *CSSM_GetContext*, *CSSM_SetContext*, *CSSM_DeleteContext*, *CSSM_GetContextAttribute*, *CSSM_UpdateContextAttributes*

NAME

CSSM_GetContext

SYNOPSIS

```
CSSM_CONTEXT_PTR CSSMAPI CSSM_GetContext
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function retrieves the context information when provided with a context handle.

PARAMETERS

CCHandle (input)
The handle to the context information.

RETURN VALUE

The pointer to the CSSM_CONTEXT structure that describes the context associated with the handle CCHandle. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code. Call *CSSM_FreeContext* to free the memory allocated by the CSSM.

ERRORS

CSSM__INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__MEMORY_ERROR
Unable to allocate memory.

CSSM__MEMORY_ERROR
Internal Memory Error.

SEE ALSO

CSSM_SetContext, *CSSM_FreeContext*

NAME

CSSM_FreeContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_FreeContext
    (CSSM_CONTEXT_PTR Context)
```

DESCRIPTION

This function frees the memory associated with the context structure.

PARAMETERS

Context (input)
The pointer to the memory that describes the context structure.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__INVALID_CONTEXT_POINTER
Invalid context pointer.

SEE ALSO

CSSM_GetContext

NAME

CSSM_SetContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SetContext
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context)
```

DESCRIPTION

This function replaces the context information associated with an existing context handle with the new context information supplied in Context. Before replacing the context, this function queries the provider associated with the context, to make sure the services requested from it are available in the provider.

PARAMETERS

CCHandle (input)
The handle to the context.

Context (input)
The context data describing the service to replace the current service associated with context handle CCHandle.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__INVALID_CONTEXT_POINTER
Invalid context pointer.

CSSM__MEMORY_ERROR
Internal Memory Error.

SEE ALSO

CSSM_GetContext

NAME

CSSM_DeleteContext

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DeleteContext
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function frees the context structure allocated by any of the CSSM_CreateXXXXXContext functions.

PARAMETERS

CCHandle (input)
The handle that describes a context to be deleted.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__INVALID_CONTEXT_HANDLE
Invalid context handle.

SEE ALSO

CSSM_CSP_CreateSymmetricContext, *CSSM_CSP_CreateAsymmetricContext*, *CSSM_CSP_CreateKeyGenContext*, *CSSM_CSP_CreateDigestContext*, *CSSM_CSP_CreateSignatureContext*, and others

NAME

CSSM_GetContextAttribute

SYNOPSIS

```
CSSM_CONTEXT_ATTRIBUTE_PTR CSSMAPI CSSM_GetContextAttribute
    (const CSSM_CONTEXT_PTR Context,
    uint32 AttributeType)
```

DESCRIPTION

This function retrieves the context attributes information for the given context and attribute type.

PARAMETERS

Context (input)
A pointer to the context.

AttributeType (input)
The attribute type of the desired attribute value.

RETURN VALUE

The pointer to the CSSM_ATTRIBUTE structure that describes the context attributes associated with the handle CCHandle and the attribute type. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code. Call the *CSSM_DeleteContextAttributes* to free memory allocated by the CSSM.

ERRORS

CSSM__INVALID_CONTEXT_HANDLE
Invalid context handle.

SEE ALSO

CSSM_DeleteContextAttributes, *CSSM_GetContext*

NAME

CSSM_UpdateContextAttributes

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_UpdateContextAttributes
    (CSSM_CC_HANDLE CCHandle,
    uint32 NumberAttributes,
    const CSSM_CONTEXT_ATTRIBUTE_PTR ContextAttributes)
```

DESCRIPTION

This function updates the security context. When an attribute is already present in the context, this update operation replaces the previously-defined attribute with the current attribute.

PARAMETERS

CCHandle (input)
The handle to the context.

NumberAttributes (input)
The number of CSSM_CONTEXT_ATTRIBUTE structures to allocate.

ContextAttributes (input)
Pointer to data that describes the attributes to be associated with this context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__INVALID_POINTER
Invalid pointer to attributes.

CSSM__MEMORY_ERROR
Internal Memory Error.

SEE ALSO

CSSM_GetContextAttribute, *CSSM_DeleteContextAttributes*

NAME

CSSM_DeleteContextAttributes

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DeleteContextAttributes
    (CSSM_CC_HANDLE CCHandle,
    uint32 NumberOfAttributes,
    CSSM_CONTEXT_ATTRIBUTE_PTR ContextAttributes)
```

DESCRIPTION

This function deletes internal data associated with given attribute type of the context handle.

PARAMETERS

CCHandle (input)
The handle that describes a context that is to be deleted.

NumberOfAttributes (input)
The number of attributes to be deleted as specified in the array of context attributes.

ContextAttributes (input)
The attribute to be deleted from the context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__INVALID_POINTER
Invalid pointer to attributes.

SEE ALSO

CSSM_GetContextAttributes, *CSSM_UpdateContextAttributes*

11.4 Cryptographic Sessions and Logon

The manpages for Cryptographic Sessions and Logon follow on the next page.

NAME

CSSM_CSP_Login

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_Login
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_CRYPTO_DATA_PTR Password,
    const CSSM_DATA_PTR pReserved)
```

DESCRIPTION

Logs the user into the CSP, allowing for multiple login types and parallel operation notification.

PARAMETERS

CSPHandle (input)
Handle of the CSP to log into.

Password (input)
Password used to log into the token.

PReserved (input)
This field is reserved for future use. The value NULL should always be given. (May be used for multiple user support in the future.)

RETURN VALUE

CSSM_OK if login is successful, CSSM_FAIL is login fails. Use *CSSM_GetError* to determine the exact error.

ERRORS

CSSM__CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_INVALID_PASSWORD
Invalid password.

CSSM__CSP_ALREADY_LOGGED_IN
User attempted to log in more than once.

SEE ALSO

CSSM_CSP_ChangeLoginPassword, *CSSM_CSP_Logout*

NAME

CSSM_CSP_Logout

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_Logout
    (CSSM_CSP_HANDLE CSPHandle)
```

DESCRIPTION

Terminates the login session associated with the specified CSP Handle.

PARAMETERS

CSPHandle (input)
Handle for the target CSP.

RETURN VALUE

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use *CSSM_GetError* to determine the exact error.

ERRORS

CSSM__CSP_INVALID_CSP
Invalid CSP handle.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_NOT_LOGGED_IN
No login session existed.

SEE ALSO

CSSM_CSP_Login, *CSSM_CSP_ChangeLoginPassword*

NAME

CSSM_CSP_ChangeLoginPassword

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSP_ChangeLoginPassword
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_CRYPTO_DATA_PTR OldPassword,
    const CSSM_CRYPTO_DATA_PTR NewPassword)
```

DESCRIPTION

Changes the login password of the current login session from the old password to the new password. The requesting user must have a login session in process.

PARAMETERS

CSPHandle (input)
Handle of the CSP supporting the current login session.

OldPassword (input)
Current password used to log into the token.

NewPassword (input)
New password to be used for future logins by this user to this token.

RETURN VALUE

CSSM_OK if login is successful, CSSM_FAIL is login fails. Use *CSSM_GetError* to determine the exact error.

ERRORS

CSSM__CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_INVALID_PASSWORD
Old password is invalid.

SEE ALSO

CSSM_CSP_Login, *CSSM_CSP_Logout*

11.5 Cryptographic Operations

The manpages for Cryptographic Operations follow on the next page.

NAME

CSSM_SignData

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SignData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Signature)
```

DESCRIPTION

This function signs data using the private key associated with the public key specified in the context.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs to be signed.

Signature (output)
A pointer to the CSSM_DATA structure for the signature.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__MANIFEST_ATTRIBUTES_NOT_FOUND
No capability attribute found in the Manifest.

CSSM__CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_OPERATION_UNSUPPORTED
Sign service not supported.

CSSM__CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM__CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the context.

CSSM__CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM__CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM__CSP_INVALID_KEYCLASS
Key class is not private key class.

CSSM__CSP_KEY_USAGE_INCORRECT
Key usage does not allow signature.

CSSM__CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM__CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM__CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM__CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

CSSM__CSP_CALLBACK_FAILED
Passphrase callback function failed.

CSSM__CSP_PRIKEY_NOT_FOUND
Cannot find the corresponding private key.

CSSM__CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed.

CSSM__CSP_PASSPHRASE_INCORRECT
Passphrase incorrect.

CSSM__CSP_PRIKEY_ERROR
Error in getting the raw private key or private key storage error.

CSSM__CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSSM_VerifyData, *CSSM_SignDataInit*, *CSSM_SignDataUpdate*, *CSSM_SignDataFinal*

NAME

CSSM_SignDataInit

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SignDataInit
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function initializes the staged sign data function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__MANIFEST_ATTRIBUTES_NOT_FOUND
No capability attribute found in the Manifest.

CSSM__CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_INVALID_ATTR_PASSPHRASE
Invalid passphrase attribute in the asymmetric context.

CSSM__CSP_INVALID_ATTR_KEY
Invalid key attribute in the context.

CSSM__CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM__CSP_INVALID_KEYCLASS
Key class is not private key class.

CSSM__CSP_KEY_USAGE_INCORRECT
Key usage does not allow signature.

CSSM__CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM__CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM__CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM__CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSSM_SignData, *CSSM_SignDataUpdate*, *CSSM_SignDataFinal*

NAME

CSSM_SignDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SignDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the data for the staged sign data function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs to be signed.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__MANIFEST_ATTRIBUTES_NOT_FOUND
No capability attribute found in the Manifest.

CSSM__CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM__CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

SEE ALSO

CSSM_SignData, *CSSM_SignDataInit*, *CSSM_SignDataFinal*

NAME

CSSM_SignDataFinal

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SignDataFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Signature)
```

DESCRIPTION

This function completes the final stage of the sign data function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Signature (output)
A pointer to the CSSM_DATA structure for the signature.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__MANIFEST_ATTRIBUTES_NOT_FOUND
No capability attribute found in the Manifest.

CSSM__CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM__NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_CALLBACK_FAILED
Passphrase callback function failed.

CSSM__CSP_PRIKEY_NOT_FOUND
Cannot find the corresponding private key.

CSSM__CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed.

CSSM__CSP_PASSPHRASE_INCORRECT
Passphrase incorrect.

CSSM__CSP_PRIKEY_ERROR
Error in getting the raw private key or private key storage error.

CSSM__CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM__CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSSM_SignData, *CSSM_SignDataInit*, *CSSM_SignDataUpdate*

NAME

CSSM_VerifyData

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_VerifyData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    const CSSM_DATA_PTR Signature)
```

DESCRIPTION

This function verifies the input data against the provided signature.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs to be verified.

Signature (input)
A pointer to a CSSM_DATA structure which contains the signature and the size of the signature.

RETURN VALUE

A CSSM_TRUE return value signifies the signature was successfully verified. When CSSM_FALSE is returned, either the signature was not successfully verified or an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_OPERATION_UNSUPPORTED
Verify service not supported.

CSSM__CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM__CSP_INVALID_SIGNATURE
Invalid or missing signature.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM__CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM__CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM__CSP_INVALID_KEYCLASS
Key class is not public key class.

CSSM__CSP_KEY_USAGE_INCORRECT
Key usage does not allow verify.

CSSM__CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM__CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM__CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM__CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSSM_SignData, *CSSM_VerifyDataInit*, *CSSM_VerifyDataUpdate*, *CSSM_VerifyDataFinal*

NAME

CSSM_VerifyDataInit

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_VerifyDataInit
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function initializes the staged verify data function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM__CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM__CSP_INVALID_KEYCLASS
Key class is not public key class.

CSSM__CSP_KEY_USAGE_INCORRECT
Key usage does not allow verify.

CSSM__CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM__CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM__CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM__CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSSM_VerifyDataUpdate, *CSSM_VerifyDataFinal*, *CSSM_VerifyData*

NAME

CSSM_VerifyDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_VerifyDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the data to the staged verify data function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs to be verified.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

SEE ALSO

CSSM_VerifyData, *CSSM_VerifyDataInit*, *CSSM_VerifyDataFinal*

NAME

CSSM_VerifyDataFinal

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_VerifyDataFinal
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR Signature)
```

DESCRIPTION

This function finalizes the staged verify data function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Signature (input)
A pointer to a CSSM_DATA structure which contains the starting address for the signature to verify against and the length of the signature in bytes.

RETURN VALUE

A CSSM_TRUE return value signifies the signature successfully verified. When CSSM_FALSE is returned, either the signature was not successfully verified or an error has occurred; use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_INVALID_SIGNATURE
Invalid or missing signature.

CSSM__CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

SEE ALSO

CSSM_VerifyData, CSSM_VerifyDataInit, CSSM_VerifyDataUpdate

NAME

CSSM_DigestData

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DigestData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Digest)
```

DESCRIPTION

This function computes a message digest for the supplied data.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs.

Digest (output)
A pointer to the CSSM_DATA structure for the message digest.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM__CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_OPERATION_UNSUPPORTED
Digest service not supported.

CSSM__CSP_OPERATION_FAILED
 Cryptographic operation failed.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
 Supports only a single buffer of input.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSSM_DigestDataInit, *CSSM_DigestDataUpdate*, *CSSM_DigestDataFinal*

NAME

CSSM_DigestDataInit

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DigestDataInit
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function initializes the staged message digest function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

SEE ALSO

CSSM_DigestData, *CSSM_DigestDataUpdate*, *CSSM_DigestDataClone*, *CSSM_DigestDataFinal*

NAME

CSSM_DigestDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DigestDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the staged message digest function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM__CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

SEE ALSO

CSSM_DigestData, *CSSM_DigestDataInit*, *CSSM_DigestDataClone*, *CSSM_DigestDataFinal*

NAME

CSSM__DigestDataClone

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_DigestDataClone
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function clones a given staged message digest context with its cryptographic attributes and intermediate result.

PARAMETERS

CCHandle (input)
The handle that describes the context of a staged message digest operation.

RETURN VALUE

The pointer to a user-allocated CSSM_CC_HANDLE for holding the cloned context handle return from CSSM. If the pointer is NULL, an error has occurred; use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

Comments

When a digest context is cloned, a new context is created with data associated with the parent context. Changes made to the parent context after calling this function will not be reflected in the cloned context. The cloned context could be used with the *CSSM_DigestDataUpdate* and *CSSM_DigestDataFinal* functions.

SEE ALSO

CSSM_DigestData, *CSSM_DigestDataInit*, *CSSM_DigestDataUpdate*, *CSSM_DigestDataFinal*

NAME

CSSM_DigestDataFinal

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DigestDataFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Digest)
```

DESCRIPTION

This function finalizes the staged message digest function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Digest (output)
A pointer to the CSSM_DATA structure for the message digest.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM__CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSSM_DigestData, *CSSM_DigestDataInit*, *CSSM_DigestDataUpdate*, *CSSM_DigestDataClone*

NAME

CSSM_GenerateMac

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateMac
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function generates a Message Authentication Code (MAC) for the supplied data.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs.

Mac (output)
A pointer to the CSSM_TATA structure for the Message Authentication Code.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM__CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_OPERATION_UNSUPPORTED
Generate MACs Service not supported.

CSSM__CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM__CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM__CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM__CSP_INVALID_KEYCLASS
Key class is not session key class.

CSSM__CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM__CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM__CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM__CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application’s declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSSM_GenerateMacInit, *CSSM_GenerateMacUpdate*, *CSSM_GenerateMacFinal*

NAME

CSSM_GenerateMacInit

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateMacInit
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function initializes the staged message authentication code function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM__CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM__CSP_INVALID_KEYCLASS
Key class is not session key class.

CSSM__CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM__CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM__CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM__CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSSM_GenerateMac, *CSSM_GenerateMacUpdate*, *CSSM_GenerateMacFinal*

NAME

CSSM_GenerateMacUpdate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateMacUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the staged message authentication code function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

SEE ALSO

CSSM_GenerateMac, *CSSM_GenerateMacInit*, *CSSM_GenerateMacFinal*

NAME

CSSM_GenerateMacFinal

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateMacFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function finalizes the staged message authentication code function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Mac (output)
A pointer to the CSSM_DATA structure for the message authentication code.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM__CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application’s declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSSM_GenerateMac, *CSSM_GenerateMacInit*, *CSSM_GenerateMacUpdate*

NAME

CSSM_VerifyMac

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_VerifyMac
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    const CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function verifies a message authentication code for the supplied data.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs.

Mac (input)
A pointer to the CSSM_DATA structure containing the MAC to verify.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_OPERATION_UNSUPPORTED
Verify MACs Service not supported.

CSSM__CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM__CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM__CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM__CSP_INVALID_KEYCLASS
Key class is not session key class.

CSSM__CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM__CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM__CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM__CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSSM_VerifyMacInit, *CSSM_VerifyMacUpdate*, *CSSM_VerifyMacFinal*

NAME

CSSM_VerifyMacInit

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_VerifyMacInit
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function initializes the staged message authentication code verification function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM__CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM__CSP_INVALID_KEYCLASS
Key class is not session key class.

CSSM__CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM__CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM__CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM__CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSSM_VerifyMac, *CSSM_VerifyMacUpdate*, *CSSM_VerifyMacFinal*

NAME

CSSM_VerifyMacUpdate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateMacUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the staged message authentication code verification function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

SEE ALSO

CSSM_VerifyMac, *CSSM_VerifyMacInit*, *CSSM_VerifyMacFinal*

NAME

CSSM_VerifyMacFinal

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_VerifyMacFinal
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function finalizes the staged message authentication code verification function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Mac (input)
A pointer to the CSSM_DATA structure containing the MAC to verify.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if the MAC verifies correctly, CSSM_FAIL otherwise.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM__CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

SEE ALSO

CSSM_VerifyMac, *CSSM_VerifyMacInit*, *CSSM_VerifyMacUpdate*

NAME

CSSM_QuerySize

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_QuerySize
    (CSSM_CC_HANDLE CCHandle,
    CSSM_BOOL Encrypt,
    uint32 QuerySizeCount,
    CSSM_QUERY_SIZE_DATA_PTR DataBlockSizes)
```

DESCRIPTION

This function queries for the size of the output data for encryption and decryption context types. This function can also be used to query the output size requirements for the intermediate steps of a staged cryptographic operation. There may be algorithm-specific and token-specific rules restricting the lengths of data following data update calls.

PARAMETERS

CCHandle (input)
The handle for an encryption and decryption context.

Encrypt (input)
A boolean indicating whether encryption is the operation for which the output data size should be calculated. If CSSM_TRUE, the operation is encryption. If CSM_FALSE the operation is decryption.

QuerySizeCount (input)
The number of entries in the array of DataBlockSizes.

DataBlockSizes (input/output)
An array of data block input sizes and corresponding entries for the data block output sizes that are returned by this function.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_POINTER
Invalid output query size data pointer.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_OPERATION_UNSUPPORTED
Query size service not supported.

CSSM__CSP_OPERATION_FAILED
Query size operation failed.

CSSM__CSP_INVALID_PADDING
Unknown padding.

CSSM__CSP_INVALID_MODE
Unknown algorithm mode for symmetric context.

CSSM__CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the asymmetric context.

CSSM__CSP_QUERY_SIZE_UNKNOWN
Cannot determine size of output data blocks.

SEE ALSO

CSSM_EncryptData, *CSSM_EncryptDataUpdate*, *CSSM_DecryptData*, *CSSM_DecryptDataUpdate*, *CSSM_SignData*, *CSSM_VerifyData*, *CSSM_DigestData*, *CSSM_GenerateMac*

NAME

CSSM_EncryptData

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_EncryptData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    uint32 *bytesEncrypted,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function encrypts the supplied data using information in the context. The *CSSM_QuerySize* function can be used to estimate the output buffer size required.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ClearBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

ClearBufCount (input)
The number of ClearBufs.

CipherBufs (output)
A pointer to a vector of CSSM_DATA structures that contain the results of the operation on the data.

CipherBufCount (input)
The number of CipherBufs.

bytesEncrypted (output)
A pointer to uint32 for the size of the encrypted data in bytes.

RemData (output)
A pointer to the CSSM_DATA structure for the last encrypted block containing padded data.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM__CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM__CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM__CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM__CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM__CSP_INVALID_DATA_COUNT
Invalid data count; data count cannot be 0.

CSSM__CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM__CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM__CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM__CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM__CSP_OPERATION_UNSUPPORTED
Encrypt data service not supported.

CSSM__CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM__CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM__CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM__CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM__CSP_INVALID_KEYCLASS
Key class is not private or public key class for asymmetric context or is not session class for symmetric context.

CSSM__CSP_KEY_USAGE_INCORRECT
Key usage does not allow encryption.

CSSM__CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM__CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM__CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM__CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

CSSM__CSP_INVALID_PADDING
Unknown padding.

CSSM__CSP_INVALID_MODE
Unknown algorithm mode for symmetric context.

CSSM__CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the asymmetric context.

CSSM__CSP_PRIKEY_NOT_FOUND
Cannot find the corresponding private key.

CSSM__CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed for asymmetric context.

CSSM__CSP_PASSPHRASE_INCORRECT
Passphrase incorrect for asymmetric context.

CSSM__CSP_PRIKEY_ERROR
Error in getting the raw private key or private key storage error for asymmetric context.

CSSM__CSP_INVALID_ATTR_INIT_VECTOR
Init vector attribute data or length error for symmetric context.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM__DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM__CSP_INVALID_DATA_POINTER is returned. In-place encryption can be done by supplying the same input and output buffers.

SEE ALSO

CSSM_QuerySize, *CSSM_DecryptData*, *CSSM_EncryptDataInit*, *CSSM_EncryptDataUpdate*, *CSSM_EncryptDataFinal*

NAME

CSSM_EncryptDataInit

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_EncryptDataInit
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function initializes the staged encrypt function. There may be algorithm-specific and token-specific rules restricting the lengths of data following data update calls making use of these parameters.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_MANIFEST_ATTRIBUTES_NOT_FOUND
No capability attribute found in the manifest.

CSSM_CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
Key class is not private or public key class for asymmetric context or is not session class for symmetric context.

CSSM_CSP_KEY_USAGE_INCORRECT
Key usage does not allow encryption.

CSSM_CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

CSSM_CSP_INVALID_PADDING
Unknown padding.

CSSM_CSP_INVALID_MODE
Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
Cannot find the corresponding private key for asymmetric context.

CSSM_CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed for asymmetric context.

CSSM_CSP_PASSPHRASE_INCORRECT
Passphrase incorrect for asymmetric context.

CSSM_CSP_PRIKEY_ERROR
Error in getting the raw private key or private key storage error for asymmetric context.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
Init vector attribute data or length error for symmetric context.

SEE ALSO

CSSM_EncryptData, *CSSM_EncryptDataUpdate*, *CSSM_EncryptDataFinal*

NAME

CSSM_EncryptDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_EncryptDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    uint32 *bytesEncrypted)
```

DESCRIPTION

This function updates the staged encrypt function. The *CSSM_QuerySize* function can be used to estimate the output buffer size required for each update call. There may be algorithm-specific and token-specific rules restricting the lengths of data in CSSM_EncryptUpdate calls.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ClearBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

ClearBufCount (input)
The number of ClearBufs.

CipherBufs (output)
A pointer to a vector of CSSM_DATA structures that contain the encrypted data resulting from the encryption operation.

CipherBufCount (input)
The number of CipherBufs.

bytesEncrypted (output)
A pointer to uint32 for the size of the encrypted data in bytes.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_MANIFEST_ATTRIBUTES_NOT_FOUND
No capability attribute found in the manifest.

CSSM_CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input or output data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place encryption can be done by supplying the same input and output buffers.

SEE ALSO

CSSM_EncryptData, *CSSM_EncryptDataInit*, *CSSM_EncryptDataFinal*, *CSSM_QuerySize*

NAME

CSSM_EncryptDataFinal

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_EncryptDataFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function finalizes the staged encrypt function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RemData (output)
A pointer to the CSSM_DATA structure for the last encrypted block containing padded data.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_MANIFEST_ATTRIBUTES_NOT_FOUND
No capability attribute found in the manifest.

CSSM_CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place encryption can be done by supplying the same input and output buffers.

SEE ALSO

CSSM_EncryptData, *CSSM_EncryptDataInit*, *CSSM_EncryptDataUpdate*

NAME

CSSM_DecryptData

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DecryptData
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    uint32 *bytesDecrypted,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function decrypts the supplied encrypted data. The *CSSM_QuerySize* function can be used to estimate the output buffer size required.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

CipherBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

CipherBufCount (input)
The number of CipherBufs.

ClearBufs (output)
A pointer to a vector of CSSM_DATA structures that contain the decrypted data resulting from the decryption operation.

ClearBufCount (input)
The number of ClearBufs.

BytesDecrypted (output)
A pointer to uint32 for the size of the decrypted data in bytes.

RemData (output)
A pointer to the CSSM_DATA structure for the last decrypted block.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
: Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
: The output buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
: Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
: Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
: Decrypt data service not supported.

CSSM_CSP_OPERATION_FAILED
: Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
: Supports only a single buffer of input.

CSSM_CSP_INVALID_ATTR_KEY
: Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
: Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
: Key class is not private or public key class for asymmetric context or is not session class for symmetric context.

CSSM_CSP_KEY_USAGE_INCORRECT
: Key usage does not allow decryption.

CSSM_CSP_KEY_ALGID_MISMATCH
: The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
: Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
: Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
: Key size in bits unsupported.

CSSM_CSP_INVALID_PADDING
: Unknown padding.

CSSM_CSP_INVALID_MODE
: Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
: Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
: Cannot find the corresponding private key for asymmetric context.

CSSM_CSP_PASSPHRASE_INVALID
: Passphrase length error or passphrase badly formed for asymmetric context.

CSSM_CSP_PASSPHRASE_INCORRECT
Passphrase incorrect for asymmetric context.

CSSM_CSP_PRIKEY_ERROR
Error in getting the raw private key or private key storage error for asymmetric context.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
Init vector attribute data or length error for symmetric context.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place decryption can be done by supplying the same input and output buffers

SEE ALSO

CSSM_QuerySize, CSSM_EncryptData, CSSM_DecryptDataInit, CSSM_DecryptDataUpdate, CSSM_DecryptDataFinal

NAME

CSSM_DecryptDataInit

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CSSM_DecryptDataInit
    (CSSM_CC_HANDLE CCHandle)
```

DESCRIPTION

This function initializes the staged decrypt function.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
Key class is not private or public key class for asymmetric context or is not session class for symmetric context.

CSSM_CSP_KEY_USAGE_INCORRECT
Key usage does not allow decryption.

CSSM_CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

CSSM_CSP_INVALID_PADDING
Unknown padding.

CSSM_CSP_INVALID_MODE
Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
Cannot find the corresponding private key.

CSSM_CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed.

CSSM_CSP_PASSPHRASE_INCORRECT
Passphrase incorrect.

CSSM_CSP_PRIKEY_ERROR
Error in getting the raw private key or private key storage error.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
Init vector attribute data or length error for symmetric context.

SEE ALSO

CSSM_DecryptData, *CSSM_DecryptDataUpdate*, *CSSM_DecryptDataFinal*

NAME

CSSM_DecryptDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DecryptDataUpdate
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    uint32 *bytesDecrypted)
```

DESCRIPTION

This function updates the staged decrypt function. The *CSSM_QuerySize* function can be used to estimate the output buffer size required for each update call. There may be algorithm-specific and token-specific rules restricting the lengths of data in CSSM_DecryptUpdate calls.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

CipherBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

CipherBufCount (input)
The number of CipherBufs.

ClearBufs (output)
A pointer to a vector of CSSM_DATA structures that contain the decrypted data resulting from the decryption operation.

ClearBufCount (input)
The number of ClearBufs.

bytesDecrypted (output)
A pointer to uint32 for the size of the decrypted data in bytes.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input or output data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application’s declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place decryption can be done by supplying the same input and output buffers.

SEE ALSO

CSSM_DecryptData, *CSSM_DecryptDataInit*, *CSSM_DecryptDataFinal*, *CSSM_QuerySize*

NAME

CSSM_DecryptDataFinal

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DecryptDataFinal
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function finalizes the staged decrypt function.

PARAMETERS

CCHandle (input)
: The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RemData (output)
: A pointer to the CSSM_DATA structure for the last decrypted block.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
: Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
: Invalid output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
: Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
: The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
: Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
: Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
: Staged Cryptographic operation failed.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place decryption can be done by supplying the same input and output buffers.

SEE ALSO

CSSM_DecryptData, *CSSM_DecryptDataInit*, *CSSM_DecryptDataUpdate*

NAME

CSSM_QueryKeySizeInBits

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_QueryKeySizeInBits
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_KEY_PTR Key,
    CSSM_KEY_SIZE_PTR KeySize)
```

DESCRIPTION

This function queries a crypto service provider for the effective and real size of a key in bits.

The key can be specified alone or in the context of a cryptographic context. If specified alone, the CSP determines the effective bit size of the key based on the real bit size and any known constraints on the usage of that key. If a cryptographic context is provided, the effective bit size of the key is determined based on the assumption that the key would be used to perform the operation described by that cryptographic context.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function.

CCHandle (input/optional)
A handle to the cryptographic context describing the operation for which the effective bit size of the key should be determined. If the context is specified, it must contain the key whose effective bit size is being queried. If the cryptographic context is not specified, then the key must be provided in the optional Key input parameter.

Key (input/optional)
A pointer to a CSSM_KEY structure containing the key for which size is to be determined. If the specific cryptographic context in which the key is to be used is not known the key must be specified alone in this parameter and the cryptographic context input parameter must be NULL. If the context is known and is specified by the CCHandle input parameter, then the key must be contained in the context structure and the Key input parameter must be NULL.

KeySize (output)
Pointer to a CSSM_KEY_SIZE data structure returns the actual size and the effective size of the key in bits.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_KEY_POINTER
Key pointer is missing or invalid.

CSSM_CSP_INVALID_KEY
Invalid key buffer.

CSSM_CSP_INVALID_POINTER
Invalid output CSSM_KEY_SIZE pointer.

CSSM_CSP_OPERATION_UNSUPPORTED
Query key size in bits service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

SEE ALSO

CSSM_GenerateRandom, *CSSM_GenerateKey*Pair

NAME

CSSM_GenerateKey

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateKey
    (CSSM_CC_HANDLE CCHandle,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA_PTR KeyLabel,
    CSSM_KEY_PTR Key)
```

DESCRIPTION

This function generates a symmetric key. The CSP may cache keying material associated with the new symmetric key. When the symmetric key is no longer in active use, the application can invoke the CSSM_FreeKey interface to allow cached keying material associated with the symmetric key to be removed.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

KeyUsage (input)
A bit mask indicating all permitted uses for the new key.

KeyAttr (input)
A bit mask defining attribute values for the new key.

KeyLabel (input)
A key label value to be associated with the new key.

Key (output)
Pointer to CSSM_KEY structure used to hold the new key. The CSSM_KEY structure should be empty upon input to this function. The CSP will ignore any values residing in this structure at function invocation. Input values should be supplied in the cryptographic context, KeyUsage, KeyAttr, and KeyLabel input parameters.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_MANIFEST_ATTRIBUTES_NOT_FOUND
No capability attribute found in the manifest.

CSSM_CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid CSSM_DATA pointer for KeyLabel.

CSSM_CSP_INVALID_DATA
Invalid CSSM_DATA buffer for KeyLabel.

CSSM_CSP_INVALID_KEY_POINTER
Invalid or missing CSSM_KEY pointer.

CSSM_CSP_INVALID_KEY
Invalid CSSM_KEY buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output key buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Generate key service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_KEYUSAGE_MASK
Specified key usage mask is invalid.

CSSM_CSP_KEYUSAGE_MASK_UNSUPPORTED
Requested key usage mask unsupported.

CSSM_CSP_INVALID_KEYATTR_MASK
Specified key attribute mask is invalid.

CSSM_CSP_KEYATTR_MASK_UNSUPPORTED
Requested key attribute mask unsupported.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

CSSM_CSP_INVALID_ATTR_SEED
Invalid seed attribute in the context if caller provides the seed crypto data structure.

CSSM_CSP_CALLBACK_FAILED
Seed callback function failed if caller provides a seed callback function.

CSSM_CSP_INVALID_ATTR_SALT
Invalid salt attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_ALG_PARAMS
Invalid param attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_START_DATE
Invalid start date attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_END_DATE
Invalid end date if caller provides one.

Comments

The KeyData field of the CSSM_KEY structure is not required to be allocated. In this case the memory required to represent the key is allocated by the CSP. The application is required to free this memory. The CSP will only allocate memory if the Data field of KeyData is NULL and the Length field is zero.

SEE ALSO

CSSM_GenerateRandom, *CSSM_GenerateKey*Pair

NAME

CSSM_GenerateKeyPair

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateKeyPair
    (CSSM_CC_HANDLE CCHandle,
    uint32 PublicKeyUsage,
    uint32 PublicKeyAttr,
    const CSSM_DATA_PTR PublicKeyLabel,
    CSSM_KEY_PTR PublicKey,
    uint32 PrivateKeyUsage,
    uint32 PrivateKeyAttr,
    const CSSM_DATA_PTR PrivateKeyLabel,
    CSSM_KEY_PTR PrivateKey)
```

DESCRIPTION

This function generates an asymmetric key pair. The CSP may cache keying material associated with the new asymmetric keypair. When one or both of the keys are no longer in active use, the application can invoke the CSSM_FreeKey interface to allow cached keying material associated with the key to be removed.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

PublicKeyUsage (input)
A bit mask indicating all permitted uses for the new public key.

PublicKeyAttr (input)
A bit mask defining attribute values for the new public key.

PublicKeyLabel (input)
A key label value to be associated with the new public key.

PublicKey (output)
Pointer to CSSM_KEY structure used to hold the new public key. The CSSM_KEY structure should be empty upon input to this function. The CSP will ignore any values residing in this structure at function invocation. Input values should be supplied in the cryptographic context, PublicKeyUsage, PublicKeyAttr, and PublicKeyLabel input parameters.

PrivateKeyUsage (input)
A bit mask indicating all permitted uses for the new private key. The CSSM_KEY structure should be empty upon input to this function. The CSP will ignore any values residing in this structure at function invocation. Input values should be supplied in the cryptographic context, PublicKeyUsage, PublicKeyAttr, and PublicKeyLabel input parameters.

PrivateKeyAttr (input)
A bit mask defining attribute values for the new private key.

PrivateKeyLabel (input)
A key label value to be associated with the new private key.

PrivateKey (output)
Pointer to CSSM_KEY structure used to hold the new private key.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_MANIFEST_ATTRIBUTES_NOT_FOUND
No capability attribute found in the manifest.

CSSM_CONTEXT_FILTER_FAILED
Requested context was not in the manifest capability attribute.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid CSSM_DATA pointer for PublicKeyLabel or PrivateKeyLabel.

CSSM_CSP_INVALID_DATA
Invalid CSSM_DATA buffer for PublicKeyLabel or PrivateKeyLabel.

CSSM_CSP_INVALID_KEY_POINTER
Invalid or missing CSSM_KEY pointer.

CSSM_CSP_INVALID_KEY
Invalid CSSM_KEY buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output key buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Generate key pair service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the context.

CSSM_CSP_CALLBACK_FAILED
Passphrase callback function failed.

CSSM_CSP_INVALID_KEYUSAGE_MASK
Specified key usage mask is invalid.

CSSM_CSP_KEYUSAGE_MASK_UNSUPPORTED
Requested key usage mask unsupported.

CSSM_CSP_INVALID_KEYATTR_MASK
Specified key attribute mask is invalid.

CSSM_CSP_KEYATTR_MASK_UNSUPPORTED
Requested key attribute mask unsupported.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

CSSM_CSP_INVALID_ATTR_ALG_PARAMS
Invalid param attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_START_DATE
Invalid start date attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_END_DATE
Invalid end date attribute if caller provides one.

Comments

The KeyData field of the CSSM_KEY structures are not required to be allocated. In this case the memory required to represent the key is allocated by the CSP. The application is required to free this memory. The CSP will only allocate memory if the Data field of KeyData is NULL and the Length field is zero.

SEE ALSO

CSSM_GenerateRandom

NAME

CSSM_GenerateRandom

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateRandom
    (CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RandomNumber)
```

DESCRIPTION

This function generates random data.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RandomNumber (output)
Pointer to CSSM_DATA structure used to obtain the random number and the size of the random number in bytes.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid or missing output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Generate random service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_SEED
Invalid seed attribute in the context if caller provides the seed crypto data structure.

CSSM_CSP_CALLBACK_FAILED
Seed callback function failed if caller provides a seed callback function.

CSSM_CSP_INVALID_ATTR_OUTPUT_SIZE
Invalid or missing output length attribute.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned.

NAME

CSSM_ObtainPrivateKeyFromPublicKey

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ObtainPrivateKeyFromPublicKey (
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_KEY_PTR PublicKey,
    CSSM_KEY_PTR Private_Key);
```

DESCRIPTION

Given a public key this function returns a reference to the private key. The private key and its associated passphrase can be used as an input to any function requiring a private key value.

PARAMETERS

CSPHandle (input)
The handle that describes the module to perform this operation.

PublicKey (input)
The public key corresponding to the private key being sought.

PrivateKey (output)
A reference to the private key corresponding to the public key.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CSP_PRIKEY_NOT_FOUND
Corresponding private key not found.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

NAME

CSSM_WrapKey

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_WrapKey
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_CRYPTO_DATA_PTR PassPhrase,
    CSSM_KEY_PTR Key,
    CSSM_DATA_PTR DescriptiveData,
    CSSM_WRAP_KEY_PTR WrappedKey)
```

DESCRIPTION

This function wraps the supplied key using the context. The key can be a symmetric key or a reference to a private key. If the key is a symmetric key, then a symmetric context must be provided describing the wrapping algorithm. If the key is a private key, then an asymmetric context describing the wrapping algorithm, and a passphrase to unlock the referenced private key must be provided. If the specified wrapping algorithm is NULL, then the key is returned in raw format, if permitted and supported by the CSP. All significant key attributes are incorporated into the wrapped key, such that the state of the key can be fully restored by the unwrap process.

PARAMETERS

CCHandle (input)
The handle to the context that describes this cryptographic operation.

PassPhrase (input)
The passphrase or a callback function to be used to obtain the passphrase that can be used by the CSP to unlock the private key before it is wrapped. This input is ignored when wrapping a symmetric, secret key.

Key (input)
A pointer to the target key to be wrapped. If a private key is to be wrapped, this is a reference to the private key. If a symmetric key is to be wrapped, the target key is that symmetric key.

DescriptiveData (input/optional)
A pointer to a CSSM_DATA structure containing additional descriptive data to be associated and included with the key during the wrapping operation. The caller and the wrapping algorithm incorporate knowledge of the structure of the descriptive data. If the wrapping algorithm does not accept additional descriptive data, then this parameter must be NULL. If the wrapping algorithm accepts descriptive data, the corresponding unwrapping algorithm can be used to extract the descriptive data and the key.

WrappedKey (output)
A pointer to a CSSM_WRAP_KEY structure that returns the wrapped key.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match. The context has to be either symmetric context

or asymmetric context.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Wrap key service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_KEY_POINTER
Invalid CSSM_KEY or CSSM_WRAP_KEYpointers.

CSSM_INVALID_SUBJECT_KEY
Invalid wrapping subject key (key to be wrapped).

CSSM_CSP_INVALID_CRYPTO_DATA_POINTER
Invalid or missing passphrase (parameter required if the subject key is a private key).

CSSM_CSP_CALLBACK_FAILED
Passphrase callback function failed for subject private key or for wrapping key in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
Cannot find the subject private key.

CSSM_CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed for either the passphrase parameter or passphrase in the asymmetric context.

CSSM_CSP_PRIKEY_ERROR
Error in getting the subject private key or subject private key storage error.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
Key class is not private or public key class for asymmetric context or is not session key class for symmetric context.

CSSM_CSP_KEY_ALGID_MISMATCH
The key in the context (key to be used for wrapping) does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data (for the wrapping key) is inconsistent.

CSSM_CSP_KEY_USAGE_INCORRECT
Key usage mask (for the wrapping key) does not allow wrap.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown key format (for the wrapping key).

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported (for the wrapping key).

CSSM_CSP_INVALID_PADDING
Unknown padding.

CSSM_CSP_INVALID_MODE
Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
Init vector attribute data or length error for symmetric context.

SEE ALSO

CSSM_UnwrapKey

NAME

CSSM_UnwrapKey

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_UnwrapKey
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_CRYPTO_DATA_PTR NewPassPhrase,
    const CSSM_KEY_PTR PublicKey
    const CSSM_WRAP_KEY_PTR WrappedKey,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA_PTR KeyLabel,
    CSSM_KEY_PTR UnwrappedKey,
    CSSM_DATA_PTR DescriptiveData)
```

DESCRIPTION

This function unwraps the wrapped key using the context. The wrapped key can be a symmetric key or a private key. If the key is a symmetric key, then a symmetric context must be provide describing the unwrapping algorithm. If the key is a private key, then an asymmetric context must be provide describing the unwrapping algorithm. Depending on the persistent object mode of the CSP and the storage mode specified by the key attribute value in the wrapped key header, the unwrapped key can be securely stored by the CSP and locked by the new passphrase. If the unwrapping algorithm is NULL and the wrapped key is actually a raw key (as indicated by its key attributes), then the key is imported into the CSP. Support for a NULL unwrapping algorithm, is at the option of the CSP. The unwrapped key is restored to its original pre-wrap state based on the key attributes recorded by the wrapped key during the wrap operation. These attributes must not be modified by the caller.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation.

PassPhrase (input/optional)
The passphrase or a callback function to be used to obtain the passphrase. If the unwrapped key is a private key and the persistent object mode is true, then the private key is unwrapped and securely stored by the CSP. The PassPhrase is used to control access to the private key after it is unwrapped. If a symmetric key is being unwrapped, then this parameter is optional.

PublicKey (input/optional)
The public key corresponding to the private key being unwrapped. If a symmetric key is being unwrapped, then this parameter must be NULL.

WrappedKey (input)
A pointer to the wrapped key. The wrapped key may be a symmetric key or the private key of a public/private key pair. The unwrapping method is specified as meta data within the wrapped key and is not specified outside of the wrapped key.

KeyUsage (input/optional)
A bit mask indicating all permitted uses for the imported key. If no value is specified, the CSP defines the usage mask for the imported key.

KeyAttr (input)
A bit mask defining attribute values to be associated with the unwrapped key.

KeyLabel (input/optional)
Pointer to a byte string that will be used as the label for the unwrapped key.

UnwrappedKey (output)
A pointer to a CSSM_KEY structure that returns the unwrapped key.

DescriptiveData (output)
A pointer to a CSSM_DATA structure that returns any additional descriptive data that was associated with the key during the wrapping operation. It is assumed that the caller incorporated knowledge of the structure of this data. If no additional data is associated with the imported key, this output value is NULL.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_OPERATION_UNSUPPORTED
Unwrap key service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_KEYATTR
Specified key attribute is incorrect or unsupported.

CSSM_CSP_INVALID_KEY_POINTER
Invalid CSSM_KEY or CSSM_WRAP_KEYpointers.

CSSM_INVALID_SUBJECT_KEY
Invalid subject key (key to be unwrapped).

CSSM_CSP_INVALID_CRYPTO_DATA_POINTER
Invalid or missing passphrase (parameter required if the subject key is a private key).

CSSM_CSP_CALLBACK_FAILED
Passphrase callback function failed for subject private key or for private key in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
Cannot find the corresponding private key for either the subject private key or the private key in the asymmetric context.

CSSM_CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed for either the passphrase parameter or

passphrase in the asymmetric context.

CSSM_CSP_PRIKEY_ERROR
: Error in getting the raw private key or private key storage error for either the subject private key or the private key in the asymmetric context.

CSSM_CSP_INVALID_KEY
: Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
: Key class is not private or public key class for asymmetric context or is not session key class for symmetric context.

CSSM_CSP_KEY_ALGID_MISMATCH
: The key in the context (key to be used for unwrapping) does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
: Key header and key data (for the unwrapping key) is inconsistent.

CSSM_CSP_KEY_USAGE_INCORRECT
: Key usage mask (for the unwrapping key) does not allow unwrap.

CSSM_CSP_KEY_FORMAT_INCORRECT
: Unknown key format (for the unwrapping key).

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
: Key size in bits unsupported (for the unwrapping key).

CSSM_CSP_INVALID_PADDING
: Unknown padding.

CSSM_CSP_INVALID_MODE
: Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
: Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
: Init vector attribute data or length error for symmetric context.

CSSM_CSP_INVALID_KEYATTR
: Specified key attribute is incorrect or unsupported.

Comments

The KeyData field of the CSSM_KEY structure is not required to be allocated. In this case the memory required to represent the key is allocated by the CSP. The application is required to free this memory. The CSP will only allocate memory if the Data field of KeyData is NULL and the Length field is zero.

SEE ALSO

CSSM_WrapKey

NAME

CSSM_DeriveKey

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DeriveKey
    (CSSM_CC_HANDLE CCHandle,
    const CSSM_KEY_PTR BaseKey,
    CSSM_DATA_PTR Param,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA_PTR KeyLabel,
    CSSM_KEY_PTR DerivedKey)
```

DESCRIPTION

This function derives a new symmetric key using the context and information from the base key.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation.

BaseKey (input)
The base key used to derive the new key. The base key may be a public key, a private key, or a symmetric key.

Param (input/output)
This parameter varies depending on the derivation algorithm.

KeyUsage (input/optional)
A bit mask indicating all permitted uses for the new derived key.

KeyAttr (input/optional)
A bit mask defining attribute values for the new derived key.

KeyLabel (input/optional)
Pointer to a byte string that will be used as the label for the derived key.

DerivedKey (output)
A pointer to a CSSM_KEY structure that returns the derived key.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Derive key service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_SUBJECT_KEY
Invalid or missing BaseKey.

CSSM_CSP_INVALID_KEYUSAGE_MASK
Specified usage mask for the key being derived is invalid.

CSSM_CSP_KEYUSAGE_MASK_UNSUPPORTED
Requested usage mask for the key being derived is unsupported.

CSSM_CSP_INVALID_KEYATTR_MASK
Specified attribute mask for the key being derived is invalid.

CSSM_CSP_KEYATTR_MASK_UNSUPPORTED
Requested attribute mask for the key being derived is unsupported.

CSSM_CSP_KEY_USAGE_INCORRECT
Usage mask on BaseKey does not allow key derivation.

CSSM_CSP_INVALID_KEY
Invalid buffer specified for the DerivedKey parameter.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output DerivedKey buffer is not big enough.

CSSM_CSP_KEY_ALGID_MISMATCH
The BaseKey does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
BaseKey header and BaseKey data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown BaseKey format.

CSSM_CSP_INVALID_ATTR_SEED
Invalid seed attribute in the context if caller provides the seed crypto data structure.

CSSM_CSP_CALLBACK_FAILED
Seed callback function failed if caller provides a seed callback function.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed.

CSSM_CSP_INVALID_ATTR_SALT
Invalid salt attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_INTERATION_COUNT
Invalid iteration count attribute or value.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
The key size in bits for BaseKey or DerivedKey is unsupported.

Comments

The KeyData field of the CSSM_KEY structure is not required to be allocated. In this case the memory required to represent the key is allocated by the CSP. The application is required to free this memory. The CSP will only allocate memory if the Data field of KeyData is NULL and the Length field is zero.

SEE ALSO

CSSM_CSP_CreateDeriveKeyContext

NAME

CSSM_FreeKey

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_FreeKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_KEY_PTR KeyPtr)
```

DESCRIPTION

This function requests the cryptographic service provider to clean up any key material associated with the key. This function also releases the internal storage referenced by the KeyData field of the key structure, which can hold the actual key value. The key reference by KeyPtr can be a persistent key or a transient key. This function clears the cached copy of the key and has no effect on the long term persistence or transience of the key.

PARAMETERS

CSPHandle (input)
: The handle that describes the module to perform this operation.

KeyPtr (input)
: The key whose associated keying material can be discarded at this time.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_KEY
: Key not recognized by this CSP.

CSSM_CSP_MEMORY_ERROR
: Internal memory error.

CSSM_FUNCTION_NOT_IMPLEMENTED
: Function not implemented.

NAME

CSSM_GenerateAlgorithmParams

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_GenerateAlgorithmParams
    (CSSM_CC_HANDLE CCHandle,
    uint32 ParamBits,
    CSSM_DATA_PTR Param)
```

DESCRIPTION

This function generates algorithm parameters for the specified context. These parameters include Diffie-Hellman key agreement parameters and DSA key generation parameters.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ParamBits (input)
Used to generate parameters for the algorithm (for example, Diffie-Hellman).

Param (output)
Pointer to CSSM_DATA structure used to obtain the key exchange parameter and the size of the key exchange parameter in bytes.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_OPERATION_UNSUPPORTED
Generate algorithm params not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

Comments

The output is returned to the caller either by filling the caller-specified buffer or by using the application's declared memory allocation functions to allocate buffer space. To specify a specific, pre-allocated output buffer, the caller must provide an array of one or more CSSM_DATA structures each one containing a Length field value greater than zero and a non-NULL Data pointer field value. To specify automatic output buffer allocation by the CSP, the caller must provide an array of one or more CSSM_DATA structures each containing a Length field value equal to zero and a NULL Data pointer field value. The application is always responsible for de-allocating the memory when it is no longer needed. If the CSSM_DATA_PTR parameter is NULL (that is, does not point to an array of CSSM_DATA structures) or the number of CSSM_DATA structures is specified as zero, the error code CSSM_CSP_INVALID_DATA_POINTER is returned

11.6 Miscellaneous Functions

The manpages for Miscellaneous Functions follow on the next page.

NAME

CSSM_RetrieveUniqueId

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_RetrieveUniqueId
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_DATA_PTR UniqueID)
```

DESCRIPTION

This function returns an identifier that could be used to uniquely differentiate the cryptographic device from all other devices from the same vendor or different vendors.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

UniqueID (output)
Pointer to CSSM_DATA structure that contains data that uniquely identifies the cryptographic device.

RETURN VALUE

A CSSM_OK return value signifies that the identifier is retrieved. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_CSP_HANDLE
Invalid provider handle.

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

NAME

CSSM_RetrieveCounter

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_RetrieveCounter
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_DATA_PTR Counter)
```

DESCRIPTION

This function returns the value of a tamper resistant clock/counter of the cryptographic device.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

Counter (output)
Pointer to CSSM_DATA structure that contains data of the tamper resistant clock/counter of the cryptographic device.

RETURN VALUE

A CSSM_OK return value signifies that the identifier was retrieved. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_CSP_HANDLE
Invalid provider handle.

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

NAME

CSSM_VerifyDevice

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_VerifyDevice
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DATA_PTR DeviceCert)
```

DESCRIPTION

This function triggers the cryptographic module to perform self verification and integrity checking.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

DeviceCert (input)
Pointer to CSSM_DATA structure that contains data that identifies the cryptographic device.

RETURN VALUE

A CSSM_OK return value signifies that the verification was successful. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_CSP_HANDLE
Invalid provider handle.

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_VERIFICATION_FAIL
Device unable to verify itself.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

11.7 Extensibility Functions

The *CSSM_CSP_PassThrough* function is provided to allow CSP developers to extend the crypto functionality of the CSSM API. Because it is only exposed to CSSM as a function pointer, its name, internal to the CSP, can be assigned at the discretion of the CSP module developer. However, its parameter list and return value must match what is shown below. The error codes given in this chapter constitute the generic error codes which may be used by all CSPs to describe common error conditions.

NAME

CSSM_CSP_PassThrough

SYNOPSIS

```
void * CSSMAPI CSSM_CSP_PassThrough
    (CSSM_CC_HANDLE CCHandle,
    uint32 PassThroughId,
    const void *InData)
```

DESCRIPTION

The *CSSM_CSP_PassThrough* function is provided to allow CSP developers to extend the crypto functionality of the CSSM API.

PARAMETERS

CCHandle (input)
The handle that describes the context of this cryptographic operation.

PassThroughId (input)
An identifier specifying the custom function to be performed.

InData (input)
A pointer to a module, implementation-specific structure containing the input data.

RETURN VALUE

A pointer to a module, implementation-specific structure containing the output data. If successful, this function returns a non-NULL value. A NULL value indicates an error has occurred. Use *CSSM_GetError* to obtain a specific error code.

ERRORS

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Derive key service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

Chapter 12

Trust Policy Services API

12.1 Overview

The primary purpose of a Trust Policy (TP) module is to answer the question, *Is this certificate authorized for this action in this trust domain?* Applications are executed within a trust domain. For example, executing an installation program at the office takes place within the corporate information technology trust domain. Executing an installation program on a system at home takes place within the user's personal system trust domain. The trust policy that allows or blocks the installation action is different for the two domains. The corporate domain may require extensive credentials and accept only credentials signed by selected parties. The personal system domain may require only a credential that establishes the bearer as a known user on the local system.

The general CSSM trust model defines a set of basic trust objects that most (if not all) trust policies use to model their trust domain and the policies over that domain. These basic trust objects include:

- Policies
- Certificates
- Defined sources of trust (called anchors)
- Certificate revocation lists
- Application-specific actions
- Evidence

Policies define the credentials required for authorization to perform an action on another object. (For example, a system administrator policy controls creating new user accounts on a computer system.) Certificates are the basic credentials representing a trust relationship among a set of two or more parties. When an organization issues certificates it defines its issuing procedure in a Certification Practice Statement (CPS). The statement identifies existing policies with which it is consistent The statement can also be the source of new policy definitions if the action and target object domains are not covered by an existing, published policy. An application domain can recognize multiple policies. A given policy can be recognized by multiple application domains.

Evaluation of trust depends on relationships among certificates. Certificate chains represent hierarchical trust, where a root authority is the source of trust. Entities attain a level of trust based on their relationship to the root authority. Certificate graphs represent an introducer model of trust, where the number and strength of endorsers (represented by immediate links in the trust graph) increases the level of trust attained by an entity. In both models, the trust domain can define accepted sources of trust, called anchors. Anchors can be mandated by fiat or can be computed by some other means. In contrast to the sources of trust, certificate revocation lists represent sources of distrust. Trust policies may consult these lists during the verification process.

Trust evaluation can be performed with respect to a specific action the bearer wishes to perform, or with respect to a policy, or with respect to the application domain in general. In the latter case, the action is understood to be either one specific action, or any and all actions in the domain.

When verifying trust, a Trust Policy Module (TPM) processes a group of certificates. The first certificate in the group is the target of the verification process. The other certificates in the group are used in the verification process to connect the target certificate with one or more anchors of trust. Supporting certificates can also be provided from a data store accessed by the TPM. It is also possible to provide a data store of anchor certificates. This case is less common. Typically the points of trust are few in number and are embedded in the caller or in the TPM during software manufacturing or at runtime.

The result of verification is a list of evidence, which forms an audit trail of the process. The evidence may be a list of verified attribute values that were contained in the certificates, or the entire set of verified certificates, or some other information that serves as evidence of the verification. In the end, the trust and authorizations asserted are based on the authority implied by a set of assumed or otherwise-specified public keys.

Many applications are hard-coded to select a specific trust policy. The CSSM registry and query mechanisms provide applications access to TP module descriptions. This information is provided by the TP module during installation and can assist the application in selecting the appropriate TP module for a given application domain.

12.2 Data Structures

12.2.1 CSSM_TP_HANDLE

This data structure represents the trust policy module handle. The handle value is a unique pairing between a trust policy module and an application that has attached that module. TP handles can be returned to an application as a result of the *CSSM_ModuleAttach* function.

```
typedef uint32 CSSM_TP_HANDLE /* Trust Policy Handle */
```

12.2.2 CSSM_TP_ACTION

This data structure represents a descriptive value defined by the trust policy module. A trust policy can define application-specific actions for the application domains over which the trust policy applies. Given a set of credentials, the trust policy module verifies authorizations to perform these actions.

```
typedef uint32 CSSM_TP_ACTION
```

12.2.3 CSSM_REVOKE_REASON

This data structure represents the reason a certificate is being revoked.

```
typedef enum cssm_revoke_reason {
    CSSM_REVOKE_CUSTOM,
    CSSM_REVOKE_UNSPECIFIC,
    CSSM_REVOKE_KEYCOMPROMISE,
    CSSM_REVOKE_CACOMPROMISE,
    CSSM_REVOKE_AFFILIATIONCHANGED,
    CSSM_REVOKE_SUPERCEDED,
    CSSM_REVOKE_CESSATIONOFOPERATION,
    CSSM_REVOKE_CERTIFICATEHOLD,
    CSSM_REVOKE_CERTIFICATEHOLDRELEASE,
    CSSM_REVOKE_REMOVEFROMCRL
} CSSM_REVOKE_REASON
```

12.2.4 CSSM_TP_STOP_ON

This enumerated list defines the conditions controlling termination of the verification process by the trust policy module when a set of policies/conditions must be tested.

```
typedef enum cssm_tp_stop_on {
    CSSM_TP_STOP_ON_POLICY = 0, /* use the pre-defined stopping
                                   criteria */
    CSSM_TP_STOP_ON_NONE = 1, /* evaluate all condition
                                   whether T or F */
    CSSM_TP_STOP_ON_FIRST_PASS = 2, /* stop evaluation at
                                   first TRUE */
    CSSM_TP_STOP_ON_FIRST_FAIL = 3 /* stop evaluation at
                                   first FALSE */
} CSSM_TP_STOP_ON;
```

12.2.5 CSSM_CERTGROUP

This structure contains a set of certificates. It is assumed that the certificates are related based on co-signaturing. The certificate group is a syntactic representation of a trust model. All certificates in the group must be of the same type.

```
typedef struct {
    CSSM_CERT_TYPE CertType; /* Certificate domain/type
                                   identifier */
    CSSM_CERT_ENCODING CertEncoding; /* certificate encoding */
    uint32 NumCerts; /* number of elements in CertList array */
    CSSM_DATA_PTR CertList; /* List of opaque certificates */
    void *reserved;
} CSSM_CERTGROUP, *CSSM_CERTGROUP_PTR;
```

Definition

CertType
An identifier indicating how the certificate is formatted and the domain of interpretation.

CertEncoding
An indicator of the encoding applied to the certificates in the cert group.

NumCerts
Number of certificates in the group.

CertList
List of certificates.

reserved
Reserved for future use.

12.2.6 CSSM_EVIDENCE_FORM

This structure contains certificates, CRLs and other information used as audit trail evidence.

```
#define CSSM_EVIDENCE_FORM_UNSPECIFIC 0x0
#define CSSM_EVIDENCE_FORM_CERT 0x1
#define CSSM_EVIDENCE_FORM_CRL 0x2
```

```
typedef struct cssm_evidence {
    uint32 EvidenceForm; /* CSSM_EVIDENCE_FORM_CERT,
                            CSSM_EVIDENCE_FORM_CRL */
    union cssm_format_type {
        CSSM_CERT_TYPE CertType;
        CSSM_CRL_TYPE CrlType
    } FormatType ;
    union cssm_format_encoding {
        CSSM_CERT_ENCODING CertEncoding;
        CSSM_CRL_ENCODING CrlEncoding
    } FormatEncoding ;
    CSSM_DATA_PTR Evidence; /* Evidence content */
} CSSM_EVIDENCE, *CSSM_EVIDENCE_PTR;
```

Definition

EvidenceForm
An identifier directing how to interpret the evidence format.

FormatType
Identifies the certificate type or the CRL type contained in the Evidence buffer.

FormatEncoding
Identifies the certificate encoding or the CRL encoding contained in the Evidence buffer.

Evidence
Buffer containing audit trail components.

12.2.7 CSSM_VERIFYCONTEXT

This data structure contains parameters useful in verifying certificate groups, certificate revocation lists and other forms of signed document

```
Typedef struct cssm_verify_context {
    CSSM_FIELD_PTR PolicyIdentifiers,
    uint32 NumberofPolicyIdentifiers,
    CSSM_TP_STOP_ON VerificationAbortOn,
    CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    CSSM_DATA_PTR AnchorCerts,
    uint32 NumberofAnchorCerts,
    CSSM_FIELD_PTR VerifyScope,
    uint32 ScopeSize,
    CSSM_TP_ACTION Action,
    CSSM_NOTIFY_CALLBACK CallbackWithVerifiedCert,
    CSSM_DATA_PTR ActionData,
    CSSM_EVIDENCE_PTR *Evidence,
    uint32 *NumberOfEvidences;
} CSSM_VERIFYCONTEXT, *CSSM_VERIFYCONTEXT_PTR;
```

PolicyIdentifiers
The policy identifier is an OID-value pair. The CSSM_OID structure contains the name of the policy and the value is an optional, caller-specified input value for the TP module to use when applying the policy. The name space for policy identifiers is defined externally by the application domains served by the trust policy module.

NumberofPolicyIdentifiers
The number of Policy Identifiers provided in the PolicyIdentifiers parameter.

AnchorCerts
A pointer to the CSSM_DATA structure containing one or more Certificates to be used in order to validate the Subject Certificate. These certificates can be root certificates, cross-certified certificates, and certificates belonging to locally-designated sources of trust.

NumberofAnchorCerts
The number of anchor certificates provided in the AnchorCerts parameter.

VerificationAbortOn
When a TP module verifies multiple conditions or multiple policies, the TP module can allow the caller to specify when to abort the verification process. If supported by the TP module, this selection can effect the evidence returned by the TP module to the caller. The default stopping condition is to stop evaluation according to the policy defined in the TP Module. The specify-able stopping conditions and their meaning are defined as follows:

CSSM_TP_STOP_ON	Definition
CSSM_STOP_ON_POLICY	Stop verification whenever the policy dictates it
CSSM_STOP_ON_NONE	Stop verification only after all conditions have been tested (ignoring the pass-fail status of each condition)
CSSM_STOP_ON_FIRST_PASS	Stop verification on the first condition that passes
CSSM_STOP_ON_FIRST_FAIL	Stop verification on the first condition that fails

The TP module may ignore the caller's specified stopping condition and revert to the default of stopping according to the policy embedded in the module.

UserAuthentication
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on—depending on the context of the request. The required format for this credential is defined by the TP and recorded in the TPSubservice structure describing this module. If the supplied credential is insufficient, additional information can be obtained from the substructure field named MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If additional information is not required, this value can be NULL.

VerifyScope
A pointer to the CSSM_FIELD array containing the OID / Value pairs that are to be used to qualify the validity of the Certificate. The context of the validity checks will be evident from each OID / Value pairing. If VerifyScope is not specified, the TP Module must assume a

default scope (portions of the Subject certificate) when performing the verification process.

ScopeSize
The number of entries in the verify scope list. If the verification scope is not specified, the input scope size must be zero.

Action
An application-specific and application-defined action to be performed under the authority of the input certificate. If no action is specified, the TP module defines a default action and performs verification assuming that action is being requested.

Note: It is also possible that a TP module verifies certificates for only one action.

CallbackWithVerifiedCert
A caller-defined function to be invoked by the TP module once for each certificate examined in the verification process. The verified certificate is passed back to the caller via this function. The module invokes the callback with four input parameters. 1) module handle, 2) application specific handle, 3) reason code and 4) pointer to returned data parameter. The reason code will be CSSM_NOTIFY_CERT_VERIFIED and the data value will be a pointer to CSSM_DATA. Contained in the CSSM_DATA will be an opaque certificate. The callback function must free the CSSM_DATA structure and its contents. If the verification process completes in a single verify step, then no callbacks are made. If the callback function pointer is NULL, no callbacks are performed.

ActionData
A pointer to the CSSM_DATA structure containing the action-specific data or a reference to the action-specific data upon which the requested action should be performed. If no data is specified, and the specified action requires action data then the TP module defines one or more default data objects upon which the action or default action would be performed.

Evidence
A pointer to a list of CSSM_EVIDENCE objects containing an audit trail of evidence constructed by the TP module during the verification process. Typically this contains Certificates and CRLs that were used to establish the validity of the Subject Certificate, but other objects may be appropriate for other types of trust policies.

NumberOfEvidences
The number of entries in the Evidence list. The returned value is zero if no evidence is produced. Evidence may be produced even when verification fails. This evidence can describe why and how the operation failed to verify the subject certificate.

12.2.8 CSSM_TP_WRAPPEDPRODUCTINFO

This structure holds information describing any backend products used by the TP module to implement its services. This descriptive information is stored in the CSSM registry when the TP module is installed with CSSM. CSSM checks the integrity of the TP module description before using the information.

The descriptive information stored in this structures can be queried using the function *CSSM_GetModuleInfo*( ) and specifying the trust policy module GUID, service mask, subservice identifier, and level of information disclosure.

```
typedef struct cssm_tp_wrappedproductinfo {
    CSSM_VERSION StandardVersion;
    CSSM_STRING StandardDescription; /* Descrip of standard
                                        product */
    CSSM_STRING_ProductVendor; /* Vendor of wrapped product */
    uint32 ProductFlags;
} CSSM_TP_WRAPPEDPRODUCTINFO, *CSSM_TP_WRAPPEDPRODUCTINFO_PTR;
```

Definition

StandardVersion
Version number of the product behind this module.

StandardDescription
A string containing a descriptive name or title for this wrapped product.

ProductVendor
Name of the vendor who developed (and markets) the wrapped product.

ProductFlags
A bit mask describing attributes of the wrapped product.

12.2.9 CSSM_TPSUBSERVICE

Four structures are used to contain the attributes that describe a trust policy add-in module: the moduleinfo, the serviceinfo, the tp_wrappedproductinfo, and the tpsubservice structure. The first two structures are general and the attributes contained in them are applicable to all types of service modules. The last two structures are trust policy module-specific. This descriptive information is stored in the CSSM registry when the TP module is installed with CSSM. CSSM checks the integrity of the TP module description before using the information.

A trust policy module may implement multiple types of services and organize them as sub-services.

The descriptive information stored in these structures can be queried using the function *CSSM_GetModuleInfo*( ) and specifying the trust policy module GUID, service mask, subservice identifier, and level of information disclosure.

```
typedef struct cssm_tpsubservice {
    uint32 SubServiceId;
    char *Description; /* Description of this subservice */
    CSSM_CERT_TYPE CertType; /* cert types accepted by
                                    this module */
    CSSM_CERT_ENCODING CertEncoding; /* Encoding of cert accepted
                                    by TP */
    CSSM_CALLER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
    uint32 NumberOfPolicyIdentifiers;
    CSSM_FIELD_PTR PolicyIdentifiers;
    CSSM_TP_WRAPPEDPRODUCTINFO WrappedProduct;
} CSSM_TPSUBSERVICE, *CSSM_TPSUBSERVICE_PTR;
```

Definition

SubServiceId
A unique, identifying number for the sub-service described in this structure.

Description
A string containing a descriptive name or title for this sub-service.

CertType
A bitmask of the certificate types processed by the trust policy.

CertEncoding
A bitmask of the certificate encodings processed by the trust policy.

AuthenticationMechanism
An enumerated value defining the credential format accepted by the TP module. An authentication credential is required for some TP functions. Presented credentials must be of the required format.

NumberOfPolicyIdentifiers
The number of policies supported by this TP module.

PolicyIdentifiers
A list of the policies (represented by their identifiers) supported by this TP module. There must be NumberOfPolicyIdentifiers entries in this list.

WrappedProduct
A pointer to the wrapped product description.

12.3 Trust Policy Operations

The manpages for Trust Policy Operations follow on the next page.

NAME

CSSM_TP_CertRequest

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_TP_CertRequest
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_SUBSERVICE_UID CSPSubserviceUid,
    const CSSM_FIELD_PTR CertFields,
    uint32 NumberOfFields,
    const CSSM_FIELD_PTR PolicyIdentifier,
    uint32 NumberOfPolicyIdentifiers,
    CSSM_CA_SERVICES MoreServiceRequests,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

If the caller is authorized to create a new certificate, this function creates a template for a new certificate and requests certificate creation from a certification authority process. The certificate template is determined by the policies defined by the policy identifiers. The template is initialized with values from the input OID/value pairs and any default values determined by the selected policies. The template is forwarded to a certification authority for processing.

The CSPSubserviceUid uniquely identifies the cryptographic service provider that must store the private key associated with the new certificate.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimate time defines the expected certificate creation time. This time may be substantial when certificate issuance requires offline authentication procedures by the CA process. In contrast, the estimated time can be zero, meaning the certificate can be obtained immediately. After the specified time has elapsed, the caller must use the CL module interface CSSM_CL_CertRetrieve, with the reference identifier, to obtain the signed certificate.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CSPSubserviceUid (input)
The persistent ID identifying the add-in CSP module where the private key is to be stored. Optionally the CL module can use this CSP to perform additional cryptographic operations or may use another default CSP for that purpose.

CertFields (input)
A pointer to an array of OID/value pairs that identify the field values as initial values in the new certificate.

NumberOfFields (input)
The number of certificate field values being input. This number specifies the number of entries in the CertFields array.

PolicyIdentifier (input/optional)
The policy identifier to be enforced when creating the Certificate template. This identifies

which certificate template should be initialized and controls initialization, including the specification of required fields, and default field values. If no policy identifier is provided as input, the TP module assumes a default policy and initializes the certificate template associated with that policy.

NumberOfPolicyIdentifiers (input)
The number of policy domains in which generated certificate template should be valid. This number specifies the number of entries in the PolicyIdentifier array.

MoreServiceRequests (input/optional)
A bit mask requesting additional certificate-creation-related services from the Certificate Authority issuing the certificate. CSSM-defined bit masks allow the caller to request backup or archive of the certificate's private key, publication of the certificate in a certificate directory service, and request out-of-band notification of the need to renew this certificate.

UserAuthentication (input/optional)
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on— depending on the context of the request. The required format for this credential is defined by the TP and recorded in the TPSubservice structure describing this module. If the information provided is insufficient, additional information can be obtained from the substructure field named MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If additional information is not required, this parameter must be NULL.

EstimatedTime (output)
The number of seconds estimated before the signed certificate will be ready to be retrieved. A (default) value of zero indicates that the signed certificate can be retrieved immediately via the corresponding *CL_CertRetrieve* function call. When the certification process cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceIdentifier (output)
A reference identifier which uniquely identifies this specific request. The handle persists across application executions until it is terminated by the successful or failed completion of the *CSSM_TP_CertRetrieve* function.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the unsigned certificate template. If the return pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid Trust Policy Library Handle.

CSSM_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_TP_INVALID_FIELD_POINTER
Invalid pointer input.

CSSM_TP_INVALID_OID
Invalid attribute OID for this cert type.

CSSM_TP_MEMORY_ERROR
Not enough memory.

CSSM_TP_AUTHENTICATION_FAIL
Caller is not authorized for operation.

SEE ALSO

CSSM_TP_CertRetrieve, *CSSM_CL_CertRequest*, *CSSM_CL_CertRetrieve*

NAME

CSSM_TP_CertRetrieve

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_TP_CertRetrieve
    (CSSM_TP_HANDLE TPHandle,
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the certificate created in response to the TP_CertRequest function call. The reference identifier denotes the corresponding CertRequest call. The signing operation, performed by the Certificate Authority (CA) process, may have been performed locally or remotely. In either case, the private key associated with the certificate has been stored in the local CSP specified in the call to TP_CertRequest. The TP module, CL module, and the CA process provide secure handling (via key wrapping) of the private key until it is securely stored in the local CSP.

The caller may be required to provide additional authentication information to retrieve the certificate. The format of these credentials is defined by the Policy identifiers specified in the corresponding TP_CertRequest call and the CL module used to create the certificate. The CL module cert format is recorded in the CLSubservice structure, which can be queried by the caller.

It is possible that the certificate is not ready to be retrieved when this call is made. In that case, an EstimatedTime to complete certificate creation is returned with the reference identifier and a NULL certificate pointer. The caller must attempt to retrieve the certificate again after the estimated time to completion has elapsed.

PARAMETERS

TPHandle (input)

The handle that describes the add-in trust policy library module used to perform this function.

CLHandle (input)

The handle that describes the add-in certificate library module used to perform this function.

ReferenceIdentifier (input)

A reference identifier which uniquely identifies the *CSSM_TP_CertRequest* call that initiated creation of the certificate returned by this function. The identifier persists across application executions until the *CSSM_CL_CertRetrieve* function completes (in success or failure).

UserAuthentication (input/optional)

A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on— depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information provided is insufficient, additional information can be provided by the substructure field names MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If additional information is not required, this parameter must be NULL.

EstimatedTime (output)

The number of seconds estimated before the signed Certificate will be returned. A (default) value of zero indicates that the signed Certificate has been returned as a result of this call. When the certification process cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the signed certificate. If the pointer is NULL, the calling application is expected to call back after the specified EstimatedTime. If the pointer is NULL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_TP_INVALID_REFERENCE
Invalid reference identifier.

SEE ALSO

CSSM_TP_CertRequest, *CSSM_CL_CertRequest*, *CSSM_CL_CertUnsign*, *CSSM_CL_CertVerify*

NAME

CSSM_TP_CertGroupVerify

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_TP_CertGroupVerify
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_CERTGROUP_PTR CertGroupToBeVerified
    const CSSM_VERIFYCONTEXT_PTR VerifyContext);
```

DESCRIPTION

This functions verifies that the subject certificate is authorized to perform an action on some data. The action and the target data are specified in the verifycontext structure along with many other input and output parameters for this operation. Anchor certificates are also specified. These are implicitly trusted certificates including root certificates, cross-certified certificates, and locally-defined sources of trust. These certificates form the basis to determine trust in the subject certificate.

The verifycontext includes a set of policy identifiers. Each policy identifier specifies an additional set of conditions that must be satisfied by the subject certificate in order to meet the trust criteria. A stopping condition for evaluating that set of conditions can also be specified.

Typically certificate verification involves the verification of multiple certificates. These certificates can be contained in the provided certificate group or supporting certificates can be stored in the data stores specified in the DBList. This allows the trust policy module to construct a certificate group and perform verification in one operation. The data stores specified by DBList can also contain certificate revocation lists used in the verification process. The caller can select to be notified incrementally as each certificate is verified. The CallbackWithVerifiedCert parameter (in the verifycontext) can specify a caller function to be invoked at the end of each certificate verification, returning the verified certificate for use by the caller.

The evaluation and verification process can produce a list of evidence. The evidence can be selected values from the certificates examined in the verification process, complete certificates from the verification process, or other pertinent information that forms an audit trail of the verification process. This evidence is returned to the caller after all steps in the verification process have been completed. The location for this output is specified in the verifycontext.

If verification succeeds, the trust policy module may carry out the action on the specified data or may return approval for the action requiring the caller to perform the action. The caller must consult TP module documentation outside of this specification to determine all module-specific side effects of this operation.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the subject certificate and anchor certificates. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

CSPHandle (input/optional)
The handle that describes the add-in cryptographic service provider module that can be used to perform the cryptographic operations required to carry out the verification. If no

CSP module is specified, the TP module uses an assumed CSP module.

DBList (input/optional)
The structure is a list of data storage library handles and data store handles. These handles should be used to store or retrieve objects (such as certificates and CRLs) related to the subject certificate and anchor certificates. If no data store is specified, the TP module uses an assumed data store module and assumed data store, if required.

CertGroupToBeVerified (input)
A group of one or more certificates to be verified. The first certificate in the group is the primary target certificate for verification. Use of the subsequent certificates during the verification process is specific to the trust domain.

VerifyContext (input)
A pointer to the CSSM_VERIFYCONTEXT structure containing a set of input and output parameters. The input parameters describe how the verification process should be performed. Most of the input parameters are optional. If not specified, the TP module can use default values for unspecified inputs.

RETURN VALUE

A CSSM_TRUE return value signifies that the certificate can be trusted. It can also indicate that the action has been performed as a side effect of the operation. When CSSM_FALSE is returned, either the certificate cannot be trusted or an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_INVALID_CSP_HANDLE
Invalid handle.

CSSM_TP_INVALID_CERT_GROUP
Invalid certificate group structure.

CSSM_TP_NOT_SIGNER
Signer certificate is not signer of subject.

CSSM_TP_NOT_TRUSTED
Signature can't be trusted.

CSSM_TP_CERT_VERIFY_FAIL
Unable to verify certificate.

CSSM_TP_INVALID_ACTION_DATA
Invalid action data specified for action.

CSSM_TP_VERIFY_ACTION_FAIL
Unable to determine trust for action.

CSSM_TP_INVALID_ANCHOR
An anchor certificate could not be identified.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

NAME

CSSM_TP_CertSign

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_TP_CertSign
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    CSSM_DATA_PTR CertToBeSigned,
    const CSSM_CERTGROUP_PTR SignerCertGroup,
    const CSSM_VERIFYCONTEXT_PTR SignerVerifyContext,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize);
```

DESCRIPTION

This functions co-signs or notorizes the certificate if the signer is authorized to perform the signing operation. The verification context provides the input parameters required to verify the signer's certificate. Once verified, the signer's private key is used to perform the operation, hence the passphrase associated with the signer's key must be provided. The SignScope is used to control the signing process.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the subject certificate and anchor certificates. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

CCHandle (input/optional)
The handle that describes the cryptographic context for signing the certificate. This context also identifies the cryptographic service provider to be used to perform the signing operation. If this handle is not provided by the caller, the trust policy module can assume a default signing algorithm and a default CSP, but the trust policy module may be unable to unlock the caller's private key without the caller's passphrase. If the trust policy module does not assume defaults or the default CSP, is not available on the local system an error occurs.

DBList (input/optional)
The structure is a list of data storage library handles and data store handles. These handles can be used to store or retrieve objects (such as certificate and CRLs) related to the signer's certificate and anchor certificates. If no data store is specified, the TP module uses an assumed data storage library module and one or more assumed data stores, if required.

CertToBeSigned (input)
A pointer to the CSSM_DATA structure containing the certificate to be co-signed.

SignerCertGroup (input)
A pointer to the CSSM_CERTGROUP containing a set of certificates of or related to the signer.

SignerVerifyContext (input)
A pointer to the CSSM_VERIFYCONTEXT structure containing a set of input and output parameters for the signature process. The input parameters describe how the verification

process should be performed. Most of the input parameters are optional. If not specified, the TP module can use default values for unspecified inputs.

SignScope (input/optional)
A pointer to the CSSM_FIELD structures specifying OIDs for the certificate fields to be included in the signature. If no signing scope is specified, a default scope is assumed.

ScopeSize (input)
A count of the number of OIDs specified in the SignScope. If no scope is specified, this value must be zero.

RETURN VALUE

A pointer to the CSSM_DATA containing the signed certificate. When NULL is returned, either the certificate template cannot be signed or an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_TP_INVALID_CERT_GROUP
Invalid certificate group structure.

CSSM_TP_CERTIFICATE_CANT_OPERATE
Signer certificate can't sign subject.

CSSM_TP_MEMORY_ERROR
Error in allocating memory.

CSSM_TP_CERT_VERIFY_FAIL
Unable to verify signer's certificate.

SEE ALSO

CSSM_TP_CertVerify, *CSSM_CL_CertRequest*, *CSSM_CL_CertRetrieve*

NAME

CSSM_TP_CertRevoke

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_TP_CertRevoke
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_DATA_PTR OldCrl,
    CSSM_CERTGROUP_PTR CertToBeRevoked,
    CSSM_CERTGROUP_PTR RevokerCertGroup,
    const CSSM_VERIFYCONTEXT_PTR RevokerVerifyContext,
    CSSM_REVOKE_REASON Reason);
```

DESCRIPTION

This function updates a certificate revocation list. The TP module determines whether the revoking certificate can revoke the target certificates. If authorized, one or more records are added to the CRL and returned to the caller.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the certificates targeted for revocation and the revoker's certificates. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

CCHandle (input/optional)
The handle that describes the cryptographic context for signing the CRL record. This context also identifies the cryptographic service provider to be used to perform the signing operation. If this handle is not provided by the caller, the trust policy module can assume a default signing algorithm and a default CSP. If the trust policy module does not assume defaults or the default CSP is not available on the local system an error occurs.

DBList (input/optional)
A list of handle pairs specifying a data storage library module and a data store managed by that module. These data stores can be used to store or retrieve objects (such as certificate and CRLs) related to the subject certificate and revoker's certificate. If no DL and DB handle pairs are specified, the TP module can use an assumed DL module and an assumed data store, if required.

OldCrl (input/optional)
A pointer to the CSSM_DATA structure containing an existing certificate revocation list. If this input is NULL, a new list is created.

CertGroupToBeRevoked (input)
A pointer to the CSSM_CERTGROUP structure containing one or more related certificates to be revoked.

RevokerCertGroup (input)
A pointer to the CSSM_CERTGROUP structure containing the certificate used to revoke the target certificates.

RevokerVerifyContext (input)
A structure containing policy elements useful in verifying certificates and their use with respect to a security policy. Optional elements in the verify context left unspecified will cause the internal default values to be used. Default values are specified in the TP module vendor release documents. This context is used to verify the revoker certificate group.

Reason (input/optional)
The reason for revoking the target certificates.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the updated certificate revocation list. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_TP_INVALID_CRL
Invalid CRL.

CSSM_TP_INVALID_CERTIFICATE
Invalid certificate.

CSSM_TP_CERTIFICATE_CANT_OPERATE
Revoker certificate can't revoke subject.

CSSM_TP_MEMORY_ERROR
Error in allocating memory.

CSSM_TP_CERT_REVOKE_FAIL
Unable to revoke certificate.

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_INVALID_CSP_HANDLE
Invalid handle.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

CSSM_CL_CrlAddCert

NAME

CSSM_TP_CrlVerify

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_TP_CrlVerify
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_DATA_PTR CrlToBeVerified,
    CSSM_CRL_TYPE CrlType,
    CSSM_CRL_ENCODING CrlEncoding,
    const CSSM_CERTGROUP_PTR SignerCertGroup,
    const CSSM_VERIFYCONTEXT_PTR VerifyContext);
```

DESCRIPTION

This function verifies the integrity of the certificate revocation list and determines whether it is trusted. The conditions for trust are part of the trust policy module. It can include conditions such as validity of the signer's certificate, verification of the signature on the CRL, the identity of the signer, the identity of the sender of the CRL, date the CRL was issued, the effective dates on the CRL, and so on.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the certificates to be verified. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

CSPHandle (input/optional)
The handle referencing a Cryptographic Service Provider to be used to verify signatures on the signer's certificate and on the CRL. The TP module is responsible for creating the cryptographic context structure required to perform the verification operation. If no CSP is specified, the TP module uses an assumed CSP to perform the operations.

DBList (input/optional)
A list of handle pairs specifying a data storage library module and a data store managed by that module. These data stores can be used to store or retrieve objects (such as certificates and CRLs) related to the signer's certificate. If no DL and DB handle pairs are specified, the TP module can use an assumed DL module and an assumed data store, if required.

CrlToBeVerified (input)
A pointer to the CSSM_DATA structure containing a signed certificate revocation list to be verified.

CrlType (input)
An indicator of the type of CRL contained in the CrlToBeVerified.

CrlEncoding (input)
An indicator of the encoding of CRL contained in the CrlToBeVerified.

SignerCertGroup (input)
A pointer to the CSSM_CERTGROUP structure containing one or more related certificates used to sign the CRL.

VerifyContext (input)
A pointer to the CSSM_VERIFYCONTEXT structure containing input and output parameters to control verification of the CRL and the signer's certificate group. Many parameters in the context structure are optional. Default values are used for each optional, unspecified value.

RETURN VALUE

A CSSM_TRUE return value signifies that the certificate revocation list can be trusted. When CSSM_FALSE is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_TP_INVALID_CERTIFICATE
Invalid certificate.

CSSM_TP_NOT_SIGNER
Signer certificate is not signer of CRL.

CSSM_TP_NOT_TRUSTED
Certificate revocation list can't be trusted.

CSSM_TP_CRL_VERIFY_FAIL
Unable to verify certificate.

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

CSSM_CL_CrlVerify

NAME

CSSM_TP_CrlSign

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_TP_CrlSign
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_DATA_PTR CrlToBeSigned,
    CSSM_CRL_TYPE CrlType,
    CSSM_CRL_ENCODING CrlEncoding,
    const CSSM_CERTGROUP_PTR SignerCertGroup,
    const CSSM_VERIFYCONTEXT_PTR SignerVerifyContext,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize);
```

DESCRIPTION

This function signs an entire certificate revocation list. The TP module determines whether the signer's certificate is trusted to sign the certificate revocation list. If trust is satisfied, then the TP module signs the revocation list using the signer's private key. Individual records in the CRL were signed when they were added to the CRL. Once the entire CRL is signed, revocation records can no longer be added to that CRL. To do so, would break the integrity of the signature resulting in a non-verifiable, rejected CRL.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the certificates to be verified. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

CCHandle (input/optional)
The handle that describes the cryptographic context for signing the CRL. This context also identifies the cryptographic service provider to be used to perform the signing operation. If this handle is not provided by the caller, the trust policy module can assume a default signing algorithm and a default CSP. If the trust policy module does not assume defaults or the default CSP is not available on the local system an error occurs.

DBList (input/optional)
A list of handle pairs specifying a data storage library module and a data store managed by that module. These data stores can be used to store or retrieve objects (such as certificate and CRLs) related to the signer's certificate or a data store for storing a resulting signed CRL. If no DL and DB handle pairs are specified, the TP module can use an assumed DL module and an assumed data store, if required.

CrlToBeSigned (input)
A pointer to the CSSM_DATA structure containing a certificate revocation list to be signed.

CrlType (input)
An indicator of the type of CRL contained in the CrlToBeSigned.

CrlEncoding (input)
An indicator of the encoding of CRL contained in the CrlToBeSigned.

SignerCertGroup (input)
A pointer to the CSSM_CERTGROUP structure containing one or more related certificates used to sign the CRL.

SignerVerifyContext (input)
A pointer to the CSSM_VERIFYCONTEXT structure containing input and output parameters to control verification of the signer's certificate group. Many parameters in the context structure are optional. Default values are used for each optional, unspecified value.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the OIDs of the CRL fields to be included in the signing process. If the signing scope is null, the TP Module must assume a default scope (portions of the CRL to be hashed) when performing the signing process.

ScopeSize (input)
The number of entries in the sign scope list. If the signing scope is not specified, the input parameter value for scope size must be zero.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the signed certificate revocation list. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_TP_INVALID_CERTIFICATE
Invalid certificate.

CSSM_TP_CERTIFICATE_CANT_OPERATE
Signer certificate can't sign certificate revocation list.

CSSM_TP_MEMORY_ERROR
Error in allocating memory.

CSSM_TP_CRL_SIGN_FAIL
Unable to sign certificate revocation list.

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

CSSM_CL_CrlSign

NAME

CSSM_TP_ApplyCrlToDb

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_TP_ApplyCrlToDb
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_DATA_PTR CrlToBeApplied,
    CSSM_CRL_TYPE CrlType,
    CSSM_CRL_ENCODING CrlEncoding,
    const CSSM_CERTGROUP_PTR SignerCert,
    const CSSM_VERIFYCONTEXT_PTR SignerVerifyContext)
```

DESCRIPTION

This function updates persistent storage to reflect entries in the certificate revocation list. The TP module determines whether the memory-resident CRL is trusted, and if it should be applied to one or more of the persistent databases. Side effects of this function can include saving a persistent copy of the CRL in a data store, or removing certificate records from a data store.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the CRL as it is applied to the data store and to manipulate the certificates effected by the CRL, if required. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

CSPHandle (input/optional)
The handle referencing a Cryptographic Service Provider to be used to verify signatures on the CRL determining whether to trust the CRL and apply it to the data store. The TP module is responsible for creating the cryptographic context structures required to perform the verification operation. If no CSP is specified, the TP module uses an assumed CSP to perform these operations.

DBList (input/optional)
A list of handle pairs specifying a data storage library module and a data store managed by that module. These data stores can contain certificates that might be effected by the CRL, they may contain CRLs, or both. If no DL and DB handle pairs are specified, the TP module must use an assumed DL module and an assumed data store for this operation.

CrlToBeApplied (input)
A pointer to the CSSM_DATA structure containing a certificate revocation list to be applied to the data store.

CrlType (input)
An indicator of the type of CRL contained in the CrlToBeApplied.

CrlEncoding (input)
An indicator of the encoding of CRL contained in the CrlToBeApplied.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate that was used to sign the CRL.

SignerVerifyContext (input)
A pointer to the CSSM_VERIFYCONTEXT structure containing input and output parameters to control verification of the signer's certificate and the CRL. Many parameters in the context structure are optional. Default values are used for each optional, unspecified value.

RETURN VALUE

A CSSM_OK return value signifies that the revocations contained in the certificate revocation list have been appropriately applied to the specified database. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_TP_INVALID_CRL
Invalid certificate revocation list.

CSSM_TP_NOT_TRUSTED
Certificate revocation list can't be trusted.

CSSM_TP_APPLY_CRL_TO_DB_FAIL
Unable to apply certificate revocation list on database.

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

CSSM_CL_CrlGetFirstItem, CSSM_CL_CrlGetNextItem, CSSM_DL_CertRevoke

12.4 Group Functions

The manpages for Group Functions follow on the next page.

NAME

CSSM_TP_CertGroupConstruct

SYNOPSIS

```
CSSM_CERTGROUP_PTR CSSMAPI CSSM_TP_CertGroupConstruct
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    CSSM_CERTGROUP_PTR CertGroupFrag);
```

DESCRIPTION

This function constructs an ordered certificate group using the certificates in *CertGroupFrag* as a starting point. There is no implied ordering for the certificates in *CertGroupFrag* except that the certificate in position 0 of the certificate group is assumed to be the starting point for constructing the remaining certificate group. An ordering relationship may be defined and recorded in the certificates themselves or assumed by the trust policy model.

The certificate group is augmented by adding semantically-related certificates obtained by searching the certificate data stores specified in *DBList*. In a hierarchical model of certificate chains, the leaf certificate in the chain is a CertGroup fragment and the complete certificate chain including the root certificate is the anticipated result of the construction operation.

PARAMETERS

TPHandle (input)
The handle to the trust policy module to perform this operation.

CLHandle (input/optional)
The handle to the certificate library module that can be used to manipulate and parse values in stored in the certgroup certificates. If no certificate library module is specified, the TP module uses an assumed CL module.

CSPHandle (input./optional)
A handle specifying the Cryptographic Service Provider to be used to verify certificates as the certificate group is constructed. If the a CSP handle is not specified, the trust policy module can assume a default CSP. If the module cannot assume a default, or the default CSP is not available on the local system, an error occurs.

DBList (input)
A list of handle pairs specifying a data storage library module and a data store managed by that module. These data stores should contain certificates (and possibly other security objects). The data stores should be searched to complete construction of a semantically-related certificate group.

CertGroupFrag (input)
A list of certificates that form a possibly incomplete set of certificates. The first certificate in the group represents the target certificate for which a group of semantically related certificates will be assembled

RETURN VALUE

A CSSM_CERTGROUP_PTR to a list of certificates that form a complete certificate group based on the original subset of certificates and the certificate data stores. A NULL list indicates an error. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid trust policy handle.

CSSM_INVALID_CL_HANDLE
Invalid certificate library handle.

CSSM_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_INVALID_DB_HANDLE
Bad database handle.

CSSM_CL_INVALID_CERTIFICATE
Invalid certificate.

CSSM_TP_CERTGROUP_NOT_FOUND
Unable to construct meaningful cert group.

CSSM_MEMORY_ERROR
Not enough memory to allocate.

SEE ALSO

CSSM_TP_CertGroupPrune, *CSSM_TP_CertVerify*

NAME

CSSM_TP_CertGroupPrune

SYNOPSIS

```
CSSM_CERTGROUP_PTR CSSMAPI CSSM_TP_CertGroupPrune
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    CSSM_CERTGROUP_PTR OrderedCertGroup);
```

DESCRIPTION

This function removes certificates from a certificate group. The prune operation can remove those certificates that have been signed by any local certificate authority, as it is possible that these certificates will not be meaningful on other systems.

This operation can also remove additional certificates that can be added to the certificate group again using the CertGroupConstruct operation. The pruned certificate group should be suitable for transmission to external hosts, which can in turn reconstruct and verify the certificate group.

The DBList parameter specifies a set of data stores containing certificates that should be pruned from the group.

PARAMETERS

TPHandle (input)
The handle to the trust policy module to perform this operation.

CLHandle (input/optional)
The handle to the certificate library module that can be used to manipulate and parse the certgroup certificates and the certificates in the specified data stores. If no certificate library module is specified, the TP module uses an assumed CL module.

DBList (input)
A list of handle pairs specifying a data storage library module and a data store managed by that module. These data stores must contain certificates (and possibly other security objects). The data stores are searched for anchor certificates restricted to have local scope. These certificates are candidates for removal from the subject certificate group.

OrderedCertGroup (input)
The initial, complete set of certificates from which certificates will be selectively removed.

RETURN VALUE

Returns a certificate group containing those certificates which are verifiable credentials outside of the local system. If the list is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid trust policy handle.

CSSM_INVALID_CL_HANDLE
Invalid certificate library handle.

CSSM_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_INVALID_DB_HANDLE
Invalid data store handle.

CSSM_TP_INVALID_CERTIFICATE
: Invalid certificate.

CSSM_TP_INVALID_CERT_GROUP
: Invalid certificate group.

CSSM_MEMORY_ERROR
: Internal memory error.

CSSM_FUNCTION_NOT_IMPLEMENTED
: Function not implemented.

SEE ALSO

CSSM_TP_CertGroupConstruct, *CSSM_TP_CertVerify*

12.5 Extensibility Functions

The manpages for Extensibility Functions follow on the next page.

NAME

CSSM_TP_PassThrough

SYNOPSIS

```
void * CSSMAPI CSSM_TP_PassThrough
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    uint32 PassThroughId,
    const void *InputParams)
```

DESCRIPTION

This function allows applications to call trust policy module-specific operations that have been exported. Such operations may include queries or services specific to the domain represented by the TP module.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the subject certificate and anchor certificates. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

CSPHandle (input/optional)
The handle that describes the add-in cryptographic service provider module that can be used to perform cryptographic operations as required to perform the requested operation. If no CSP module is specified, the TP module uses an assumed CSP module, if required.

DBList (input/optional)
A list of handle pairs specifying a data storage library module and a data store managed by that module. These data stores can contain certificates that might be effected by the CRL, they may contain CRLs, or both. If no DL and DB handle pairs are specified, the TP module can use an assumed DL module and an assumed data store for this operation.

PassThroughId (input)
An identifier assigned by the TP module to indicate the exported function to perform.

InputParams (input)
A pointer to a module, implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested TP module. If the passthrough function requires access to a private key located in the CSP referenced by CSPHandle, then the InputParams should contain a passphrase, or a callback or cryptographic context that can be used to obtain the passphrase.

RETURN VALUE

A pointer to an implementation-specific structure defined by the trust policy module provider. The structure contains the output from the pass-through function. The output data must be interpreted by the calling application based on externally available information. If the pointer is NULL, an error has occurred.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_TP_INVALID_DATA_POINTER
Invalid pointer for input data.

CSSM_TP_INVALID_ID
Invalid pass through ID.

CSSM_TP_MEMORY_ERROR
Error in allocating memory.

CSSM_TP_PASS_THROUGH_FAIL
Unable to perform pass through.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

Chapter 13

Certificate Library Services API

13.1 Overview

The primary purpose of a Certificate Library (CL) module is to perform syntactic manipulations on a specific certificate format, and its associated certificate revocation list (CRL) format. These manipulations include the complete life cycle of a certificate and the keypair associated with that certificate. Certificates and CRLs are related by the life cycle model and by the data formats used to represent them. For this reason, these objects should be manipulated by a single, cohesive library.

Certificate libraries manipulate memory-based objects only. The persistence of certificates, CRLs, and other security-related objects is an independent property of these objects. It is the responsibility of the application and/or the trust policy module to use data storage add-in modules to make objects persistent (if appropriate). The particular storage mechanism used by a data storage module can often be selected, independent of the trust policy and the application.

13.1.1 Certificate Life Cycle

The Certificate Library provides life cycle support and format-specific manipulation which an application can access via CSSM. These libraries allow applications and add-in modules to create, sign, verify, revoke, renew, and recover certificates without requiring knowledge of certificate and CRL formats and encodings.

A certificate is a form of credential. Under current certificate models, such as X.509, SDSI, SPKI, and so on, a single certificate represents the identity of an entity and optionally associates authorizations with that entity. When a certificate is issued, the issuer includes a digital signature of the certificate. Verification of this signature is the mechanism used to establish trust in the identity and authorizations recorded in the certificate. Certificates are signed by one or more other certificates. Root certificates are self-signed. The syntactic process of signing corresponds to a trust relationship between the entities identified by the certificates.

The certificate life cycle is presented in Figure 13-1. It begins with the registration process. During registration, the authenticity of a user's identity is verified. This can be a two-part process beginning with manual procedures requiring physical presence followed by backoffice procedures to entire status and results for use by the automated system. The level of verification associated with the identity of the individual will depend on the Security Policy and Certificate Management Practice Statements that apply to the individual who will receive a certificate, and the domain in which that certificate will be used.

After registration, keying material is generated and certificates are created. Once the private key material and public key certificate are issued to a user and backed up if appropriate, the active phase of the certificate management life cycle begins.

The active phase includes:

- Retrieval—retrieving a certificate from a remote repository such as an X.500 directory
- Verification—verifying the validity dates, signatures on a certificate and revocation status
- Revocation—asserting that a previously-legitimate certificate is no longer a valid certificate

- Recovery—when an end-user has forgotten the passphrase required to use the certificate for signing or for decryption
- Update—issuing a new public/private key pair when a legitimate pair has or will expire soon.

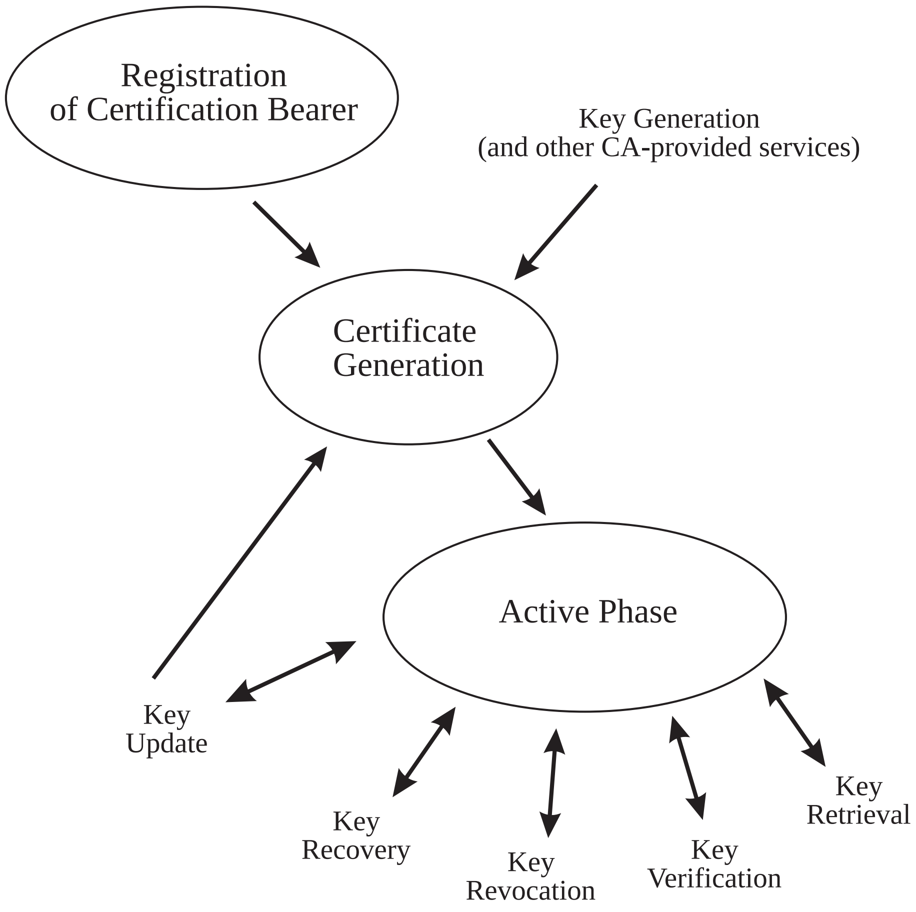

Figure 13-1 Certificate Life Cycle States and Actions

The CSSM Certificate Library APIs define four functions supporting certificate creation and update, two functions supporting certificate verification, seven functions supporting certificate parsing, ten functions supporting certificate revocation and CRL manipulation, and four functions supporting certificate recovery. The certificate library passthrough function is defined so library implementors can extend the library with additional services (as appropriate).

13.1.2 Application and Certificate Library Interaction

An application determines the availability and basic capabilities of a Certificate Library by querying the CSSM Registry. When a new CL is installed on a system, the certificate types and certificate fields that it supports are registered with CSSM. An application uses registry information to find an appropriate CL and to request that CSSM attach to the CL. When CSSM attaches to the CL, it returns a CL handle to the application which uniquely identifies the pairing of the application thread to the CL module instance. This handle is used by the application to identify the CL in future function calls.

CSSM passes CL function calls from an application to the application-selected Certificate Library.

The application is responsible for the allocation and de-allocation of all memory which is passed into or out of the Certificate Library module. The application must register memory allocation and de-allocation upcalls with CSSM when it attaches any add-in service module. These upcalls and the handle identifying the application and module pairing are passed to the CL module at that time. The Certificate Library Module uses these functions to allocate and de-allocate memory which belongs to or will belong to the application.

13.1.3 Operations on Certificates

CSSM defines the general security API that all certificate libraries should provide to manipulate certificates and certificate revocation lists. The basic areas of functionality include:

- Certificate operations
- Certificate revocation list operations
- Extensibility functions

Each certificate library may implement some or all of these functions. The available functions are registered with CSSM when the module is attached. Each certificate library should be accompanied with documentation specifying supported functions, non-supported functions, and module-specific passthrough functions. It is the responsibility of the application developer to obtain and use this information when developing applications using a selected certificate library.

13.2 Data Structures

This chapter describes the data structures which may be passed to or returned from a Certificate Library function. They will be used by applications to prepare data to be passed as input parameters into CSSM API function calls which will be passed without modification to the appropriate CL. The CL is then responsible for interpreting them and returning the appropriate data structure to the calling application via CSSM. These data structures are defined in the header file **<cssmtype.h>**, distributed with CSSM.

13.2.1 CSSM_CL_HANDLE

The CSSM_CL_HANDLE is used to identify the association between an application thread and an instance of a CL module. It is assigned when an application causes CSSM to attach to a Certificate Library. It is freed when an application causes CSSM to detach from a Certificate Library. The application uses the CSSM_CL_HANDLE with every CL function call to identify the targeted CL. The CL module uses the CSSM_CL_HANDLE to identify the appropriate application's memory management routines when allocating memory on the application's behalf.

```
typedef uint32 CSSM_CL_HANDLE
```

13.2.2 CSSM_CERT_TYPE

This variable specifies the type of certificate format supported by a certificate library and the types of certificates understood for import and export. They are expected to define such well-known certificate formats as X.509 Version 3 and SDSI, as well as custom certificate formats. The list of enumerated values can be extended for new types by defining a label with an associated value greater than CSSM_CL_CUSTOM_CERT_TYPE.

```
typedef enum cssm_cert_type {
    CSSM_CERT_UNKNOWN = 0x00,
    CSSM_CERT_X_509v1 = 0x01,
    CSSM_CERT_X_509v2 = 0x02,
    CSSM_CERT_X_509v3 = 0x03,
    CSSM_CERT_PGP = 0x04,
    CSSM_CERT_SPKI = 0x05,
    CSSM_CERT_SDSIv1 = 0x06,
    CSSM_CERT_Intel = 0x08,
    CSSM_CERT_X_509_ATTRIBUTE = 0x09, /* X.509
                                        attribute cert */
    CSSM_CERT_X9_ATTRIBUTE = 0x0A, /* X9 attribute cert */
    CSSM_CERT_LAST = 0x7FFF,
} CSSM_CERT_TYPE, *CSSM_CERT_TYPE_PTR;

/* Applications wishing to define their own custom certificate
 * type should create a random uint32 whose value is greater than
 * the CSSM_CL_CUSTOM_CERT_TYPE */
#define CSSM_CL_CUSTOM_CERT_TYPE 0x08000
```

13.2.3 CSSM_CERT_ENCODING

This variable specifies the certificate encoding format supported by a certificate library.

```
typedef enum cssm_cert_encoding {
    CSSM_CERT_ENCODING_UNKNOWN = 0x00,
    CSSM_CERT_ENCODING_CUSTOM = 0x01,
    CSSM_CERT_ENCODING_BER = 0x02,
    CSSM_CERT_ENCODING_DER = 0x03,
    CSSM_CERT_ENCODING_NDR = 0x04,
} CSSM_CERT_ENCODING, *CSSM_CERT_ENCODING_PTR;
```

13.2.4 CSSM_CERT_BUNDLE_TYPE

This enumerated type lists the signed certificate aggregates that are considered to be certificate bundles.

```
typedef enum cssm_cert_bundle_type {
    CSSM_CERT_BUNDLE_UNKNOWN = 0x00,
    CSSM_CERT_BUNDLE_CUSTOM = 0x01,
    CSSM_CERT_BUNDLE_PKCS7_SIGNED_DATA = 0x02,
    CSSM_CERT_BUNDLE_PKCS7_SIGNED_ENVELOPED_DATA = 0x03,
    CSSM_CERT_BUNDLE_PKCS12 = 0x04,
    CSSM_CERT_BUNDLE_PFX = 0x05,
    CSSM_CERT_BUNDLE_LAST = 0x7FFF
} CSSM_CERT_BUNDLE_TYPE;

/* Applications wishing to define their own custom certificate
 * BUNDLE type should create a random uint32 whose value
 * is greater than the CSSM_CL_CUSTOM_CERT_BUNDLE_TYPE */

#define CSSM_CL_CUSTOM_CERT_BUNDLE_TYPE 0x8000
```

13.2.5 CSSM_CERT_BUNDLE_ENCODING

This enumerated type lists the encoding methods applied to the signed certificate aggregates that are considered to be certificate bundles.

```
typedef enum cssm_cert_bundle_encoding {
    CSSM_CERT_BUNDLE_ENCODING_UNKNOWN = 0x00,
    CSSM_CERT_BUNDLE_ENCODING_CUSTOM = 0x01,
    CSSM_CERT_BUNDLE_ENCODING_BER = 0x02,
    CSSM_CERT_BUNDLE_ENCODING_DER = 0x03
} CSSM_CERT_BUNDLE_ENCODING;
```

13.2.6 CSSM_CERT_BUNDLE_HEADER

This structure defines a bundle header, which describes the type and encoding of a certificate bundle.

```
typedef struct cssm_cert_bundle_header {
    CSSM_CERT_BUNDLE_TYPE BundleType;
    CSSM_CERT_BUNDLE_ENCODING BundleEncoding;
} CSSM_CERT_BUNDLE_HEADER, *CSSM_CERT_BUNDLE_HEADER_PTR;
```

Definition

BundleType
A descriptor which identifies the format of the certificate aggregate.

BundleEncoding
A descriptor which identifies the encoding of the certificate aggregate.

13.2.7 CSSM_CERT_BUNDLE

This structure defines a certificate bundle, which consists of a descriptive header and a pointer to the opaque bundle. The bundle itself is a signed opaque aggregate of certificates.

```
typedef struct cssm_cert_bundle {
    CSSM_CERT_BUNDLE_HEADER BundleHeader;
    CSSM_DATA Bundle;
} CSSM_CERT_BUNDLE, *CSSM_CERT_BUNDLE_PTR;
```

Definition

BundleHeader
Information describing the format and encoding of the bundle contents.

Bundle
A signed opaque aggregate of certificates.

13.2.8 CSSM_OID

The object identifier (OID) structure is used to hold a unique identifier for the atomic data fields and the compound substructure that comprise the fields of a certificate or CRL. CSSM_OIDs exist outside of a certificate or a CRL. Typically, they are not stored within a certificate or CRL. A certificate library module implements a particular representation for certificates and CRLs. This representation is specified by the pair [certificate_type, certificate_encoding]. The underlying representation of a CSSM_OID is outside of the representation for a certificate or a CRL. Possible representations for a CSSM_OID include:

- A character string in a character set native to the platform
- A portable character string that can be exchanged across platforms
- A DER-encoded, X.509-like OID that is parsed when used as a reference
- A variable-length sequence of integers
- An S-expression that must be evaluated when used as a reference
- An enumerated value that is defined in header files supplied by group representing one or more CLMs

At most one representation and interpretation for a CSSM_OID should be defined for each unique cert-CRL representation. This provides interoperability among certificate library modules that manipulate the same certificate and CRL representations. Also the selected representation for CSSM_OIDs should be consist with the cert-CRL representation. For example, CLMs supporting BER/DER encoded X.509 certificates and CRL could use DER-encoded X.509-like OIDs as the representation for CSSM_OIDs. In contrast, CLMs supporting SDSI certificates could use S-expressions as the representation for CSSM_OIDs.

```
typedef CSSM_DATA CSSM_OID, *CSSM_OID_PTR
```

13.2.9 CSSM_CRL_TYPE

This structure represents the type of format used for revocation lists.

```
typedef enum cssm_crl_type {
    CSSM_CRLTYPE_UNKNOWN,
    CSSM_CRLTYPE_X_509v1,
    CSSM_CRLTYPE_X_509v2,
} CSSM_CRL_TYPE, *CSSM_CRL_TYPE_PTR;
```

13.2.10 CSSM_CRL_ENCODING

This structure represents the encoding format used for revocation lists.

```
typedef enum cssm_crl_encoding {
    CSSM_CRL_ENCODING_UNKNOWN,
    CSSM_CRL_ENCODING_CUSTOM,
    CSSM_CRL_ENCODING_BER,
    CSSM_CRL_ENCODING_DER,
    CSSM_CRL_ENCODING_BLOOM
} CSSM_CRL_ENCODING, *CSSM_CRL_ENCODING_PTR;
```

13.2.11 CSSM_FIELD

This structure contains the OID/value pair for any item that can be identified by an OID. A certificate library module uses this structure to hold an OID/value pair for fields in a certificate or CRL.

```
typedef struct cssm_field {
    CSSM_OID FieldOid;
    CSSM_DATA FieldValue;
}CSSM_FIELD, *CSSM_FIELD_PTR
```

Definition

FieldOid
The object identifier which identifies the certificate or CRL data type or data structure.

FieldValue
A CSSM_DATA type which contains the value of the specified OID in a contiguous block of memory.

13.2.12 CSSM_ESTIMATED_TIME_UNKNOWN

The value used by an authority or process to indicate that an estimated completion time cannot be determined.

```
#define CSSM_ESTIMATED_TIME_UNKNOWN -1
```

13.2.13 CSSM_CA_SERVICES

This bit mask defines the additional certificate-creation-related services that an issuing Certificate Authority (CA) can offer. Such services include (but are not limited to) archiving the certificate and keypair, publishing the certificate to one or more certificate directory services, and sending automatic, out-of-band notifications of the need to renew a certificate. A CA may offer any subset of these services. Additional services can be defined over time.

```
typedef uint32 CSSM_CA_SERVICES;
        /* bit masks for additional CA services at cert enroll */
#define CSSM_CA_KEY_ARCHIVE 0x0001 /* archive cert and keys */
#define CSSM_CA_CERT_PUBLISH 0x0002 /* cert in directory
                                        service */
#define CSSM_CA_CERT_NOTIFY_RENEW 0x0004 /* notify at renewal
                                        time */
#define CSSM_CA_CERT_DIR_UPDATE 0x0008 /* multi-signed cert to
                                        dir svc */
#define CSSM_CA_CRL_DISTRIBUTE 0x0010 /* push CRL to everyone */
```

13.2.14 CSSM_CL_CA_CERT_CLASSINFO

This structure describes a class of certificates issued by a given CA.

```
typedef struct cssm_cl_ca_cert_classinfo {
    CSSM_STRING CertClassName; /* Name of the class of
                                        certificate */
    CSSM_DATA CACert; /* CA cert used to sign this cert class */
} CSSM_CL_CA_CERT_CLASSINFO, *CSSM_CL_CA_CERT_CLASSINFO_PTR;
```

Definition

CertClassName
The CA's description of the certificate class, including its name.

CACert
The CA's cert used to sign issued certificates of this cert class.

13.2.15 CSSM_CL_CA_PRODUCTINFO

This structure holds product information about a backend Certificate Authority (CA) that is accessible to the CL module. The CL module vendor is not required to provide this information, but may choose to do so.

```
typedef struct cssm_cl_ca_productinfo {
    CSSM_VERSION StandardVersion; /* Ver of standard this product
                                        conforms to */
    CSSM_STRING StandardDescription; /* Desc of standard this
                                        product conforms to */
    CSSM_VERSION ProductVersion; /* Version of wrapped
                                        product/library */
    CSSM_STRING ProductDescription; /* Description of wrapped
                                        product/library */
    CSSM_STRING ProductVendor; /* Vendor of wrapped product
                                        library */
    CSSM_NET_PROTOCOL NetworkProtocol; /* The network protocol
                                   supported by the CA service */
    CSSM_CERT_TYPE CertType; /* Type of certs supported by CA */

    CSSM_CERT_ENCODING CertEncoding;  /* Cert encoding supported
                                         by CA */
    CSSM_CRL_TYPE CrlType;  /* CRL type supported by CA */

    CSSM_CRL_ENCODING CrlEncoding;  /* CRL encoding supported
                                       by CA */
    CSSM_CA_SERVICES AdditionalServiceFlags; /* Mask of additional
                                 services a caller can request */
    uint32 NumberOfCertClasses; /* Number of different cert types
                                   or classes the CA can issue */
    CSSM_CL_CA_CERT_CLASSINFO_PTR CertClasses /* Information about
                             the cert classes issued by this CA */
} CSSM_CL_CA_PRODUCTINFO, *CSSM_CL_CA_PRODUCTINFO_PTR;
```

Definition

StandardVersion
If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription
If this product conforms to an industry standard, this is a description of that standard.

ProductVersion
Version number information for the actual product version used in this version of the DL module.

ProductDescription
A string describing the product.

ProductVendor
The name of the product vendor.

NetworkProtocol
The name of the network protocol supported by the CA service.

CertType
An enumerated value specifying the certificate type that the CA manages.

CertEncoding
An enumerated value specifying the certificate encoding that the CA manages

CrlType
An enumerated value specifying the CRL type that the CA manages

CrlEncoding
An enumerated value specifying the CRL encoding that the CA manages

AdditionalServiceFlags
A bit mask indicating the additional services a caller can request from a CA (as side effects and in conjunction with other service requests.

NumberOfCertClasses
The number of classes or levels of Certificates managed by this CA.

CertClasses
An array of information about the classes of certificates supported by this CA.

13.2.16 CSSM_CL_ENCODER_PRODUCTINFO

This structure holds product information about embedded products that a CL module uses to provide its services. The CL module vendor is not required to provide this information, but may choose to do so.

```
typedef struct cssm_cl_encoder_productinfo {
    CSSM_VERSION StandardVersion; /* Ver of standard the product
                                    conforms to */
    CSSM_STRING StandardDescription; /* Desc of standard this
                                    product conforms to */
    CSSM_VERSION ProductVersion; /* Version of wrapped product or
                                    library */
    CSSM_STRING ProductDescription; /* Description of wrapped
                                    product or library */
    CSSM_STRING ProductVendor; /* Vendor of wrapped product or
                                    library */
    CSSM_CERT_TYPE CertType; /* Type of certs supported by
                                    encoder */
    CSSM_CRL_TYPE CrlType: /* Type of CRLs supported by
                                    encoder */
    uint32 ProductFlags; /* Mask of selectable encoder features
                                   actually used by the CL */
} CSSM_CL_ENCODER_PRODUCTINFO, *CSSM_CL_ENCODER_PRODUCTINFO_PTR;
```

Definition

StandardVersion
If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription
If this product conforms to an industry standard, this is a description of that standard.

ProductVersion
Version number information for the actual product version used in this version of the DL module.

ProductDescription
A string describing the product.

ProductVendor
The name of the product vendor.

CertType
An enumerated value specifying the certificate type that the encoder processes (if limited to one type).

CrlType
An enumerated value specifying the CRL type that the encoder processes (if limited to one type).

ProductFlags
A bit mask indicating any selectable features of the embedded product that the CL module

selected to use.

13.2.17 CSSM_CL_WRAPPEDPRODUCTINFO

This structure lists the set of embedded products and the CA service used by the CL module to implement its services. The CL module is not required to provide any of this information, but may choose to do so.

```
typedef struct cssm_cl_wrappedproductinfo {
      /* List of encode/decode/parse libraries embedded in
                                        the CL module */
    CSSM_CL_ENCODER_PRODUCTINFO_PTR EmbeddedEncoderProducts;
      /* library product description */
    uint32 NumberOfEncoderProducts;
      /* number of encode/decode/parse libraries used in CL */
      /* List of CAs accessible to the CL module */
    CSSM_CL_CA_PRODUCTINFO_PTR AccessibleCAProducts;
      /* CA product description*/
    uint32 NumberOfCAProducts;
      /* Number of accessible CAs */
} CSSM_CL_WRAPPEDPRODUCTINFO, *CSSM_CL_WRAPPEDPRODUCTINFO_PTR;
```

Definition

EmbeddedEncoderProducts
An array of structures that describe each embedded encoder product used in this CL module implementation.

NumberOfEncoderProducts
A count of the number of distinct embedded certificate encoder products used in the CL module implementation.

AccessibleCAProducts
An array of structures that describe each type of Certificate Authority accessible through this CL module implementation.

NumberOfCAProducts
A count of the number of distinct CA products described in the array AccessibleCAProducts.

13.2.18 CSSM_CLSUBSERVICE

This structure contains the static information that describes a certificate library sub-service. This information is stored in the CSSM registry when the CL module is installed with CSSM. CSSM checks the integrity of the CL module description before using the information. A certificate library module may implement multiple types of services and organize them as sub-services.

The descriptive information stored in these structures can be queried using the function *CSSM_GetModuleInfo*( ) and specifying the certificate library module GUID

```
typedef struct cssm_clsubservice {
    uint32 SubServiceId;
    CSSM_STRING Description;
    CSSM_CERT_TYPE CertType;
    CSSM_CERT_ENCODING CertEncoding;
    uint32 NumberOfBundleInfos;
    CSSM_CERT_BUNDLE_HEADER_PTR BundleInfo; /* first is default
                                    value */
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
    uint32 NumberOfTemplateFields;
    CSSM_OID_PTR CertTemplate;
    uint32 NumberOfTranslationTypes;
    CSSM_CERT_TYPE_PTR CertTranslationTypes;
    CSSM_CL_WRAPPEDPRODUCTINFO WrappedProduct;
} CSSM_CLSUBSERVICE, *CSSM_CLSUBSERVICE_PTR;
```

Definition

SubServiceId
A unique, identifying number for the sub-service described in this structure.

Description
A string containing a description name or title for this sub-service.

CertType
An identifier for the type of certificate.

CertEncoding
An identifier for the certificate encoding format.

NumberOfBundleInfos
The number of distinct bundle type/encoding pairs supported by the certificate library module.

BundleInfo
A pointer to a list of bundle header structures. Each structure defines a bundle type and encoding supported by the certificate library module. The first bundle header is the default for the library.

AuthenticationMechanism
An enumerated value defining the credential format accepted by the CL module. Authentication credentials may be required when requesting certificate creation or other CL functions. Presented credentials must be of the required format.

NumberOfTemplateFields
The number of certificate template fields. This number also indicates the length of the CertTemplate array.

CertTemplate
A pointer to an array of tag/value pairs which identify the field values of a certificate.

NumberOfTranslationTypes
The number of certificate types that this certificate library add-in module can import and export. This number also indicates the length of the CertTranslationTypes array.

CertTranslationTypes
A pointer to an array of certificate types. This array indicates the certificate types that can be imported into and exported from this certificate library module's native certificate type.

WrappedProduct
Descriptions of the set of embedded products used by this module and the CA services available via this module.

13.3 Certificate Operations

This chapter describes the function prototypes and error codes supported by a Certificate Library module for operations on certificates. The error codes given in this chapter constitute the generic error codes which are defined by CSSM for use by all certificate libraries in describing common error conditions. A certificate library may also define and return vendor-specific error codes. Applications must consult vendor-supplied documentation for the specification and description of any error codes defined outside of this specification.

NAME

CSSM_CL_CertRequest

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CL_CertRequest
    (CSSM_CL_HANDLE CLHandle,
    CSSM_SUBSERVICE_UID CSPSubserviceUid, /* a unique id for the
                                        CSP subservice */
    const uint32 SubServiceId, /* sub service Id for the CSP */
    const CSSM_FIELD_PTR SubjectCertTemplate,
    uint32 NumberOfFields,
    const CSSM_DATA_PTR CACert,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize,
    CSSM_NET_ADDRESS_PTR CALocation,
    CSSM_CA_SERVICES MoreServiceRequests,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

This function submits a certificate creation request to a Certificate Authority (CA) process. The CA process is identified by the SignerCert. The caller can obtain the certificate for all of the certification authorities supported by the CL by querying the CSSM registry for the CL's CA information.The CA process may be local or remote. The certificate fields provides the initial values for the certificate. The CA can add other default values known only to the CA.

As the certificate issuer, the CA process signs the new certificate. If the signer's certificate is not specified in this function, the CA assumes a default signing certificate it uses to issue certificates. The SignScope defines the set of certificate fields to be included in the signing process. The signing operation may be performed locally or remotely. The caller may specify the CSP to be used for cryptographic operations. The CL module is responsible for creating and destroying all cryptographic contexts required to perform these operations.

The caller can request additional certificate-creation-related services from the CA. These requests are designated by the MoreServiceRequests bit mask. CSSM-defined bit masks allow the caller to request certificate and key archival, certificate registration with a directory service, certificate renewal notification, and so on. CAs are not required to provide such services. The CL module works with the CA process to provide the requested services.

The caller is required to provide authentication information so the CA process can determine whether the caller is authorized to request a certificate. The specific format of the credential is specified by the CL module. The caller can query the CL Module Info structure to obtain this information.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimate time defines the expected certificate creation time. This time may be substantial when certificate issuance requires offline authentication procedures by the CA process. In contrast, the estimated time can be zero, meaning the certificate can be obtained immediately. After the specified time has elapsed, the caller must use the CL module interface CSSM_CL_CertRetrieve, with the reference identifier, to obtain the signed certificate.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CSPSubserviceUid (input)
The identifier which uniquely describes the add-in CSP module subservice where the private key is to be stored. Optionally, the CL module can use this CSP to perform additional cryptographic operations or may use another default CSP for that purpose.

SubServiceId (input/optional)
The sub-service number identifying the CSP sub-service to use when storing the private key associate with the certificate in the local CSP. If the CSP supports only one sub-service or the CL module assumes a default sub-service of a CSP, then the sub-service identifier can be omitted.

SubjectCertTemplate (input)
A pointer to an array of OID / Value pairs providing the initial values for the certificate.

NumberOfFields (input)
The number of certificate field values being input. This number specifies the number of entries in the SubjectCertTemplate array.

CACert (input/optional)
A pointer to the CSSM_DATA structure containing the desired Certification Authority's signing certificate. If the CACert is NULL, the CL module or the CA process can provide a default signing certificate.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the OID/value pairs specifying the certificate fields to be signed. When the input value is NULL, the CL assumes and includes a default set of certificate fields in the signing process.

ScopeSize (input)
The number of entries in the sign scope list. If no signing scope is specified, then the scope size must be zero.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on the CACert input parameter or can assume a default CA process location. If a CACert is not specified and a default cannot be assumed, the request cannot be initiated and the operation fails.

MoreServiceRequests (input/optional)
A bit mask requesting additional certificate-creation-related services from the Certificate Authority issuing the certificate. CSSM-defined bit masks allow the caller to request backup or archive of the certificate's private key, publication of the certificate in a certificate directory service, request out-of-band notification of the need to renew this certificate, and so on.

UserAuthentication (input/optional)
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on, depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this

module. If the supplied information is insufficient, additional information can be provided by the substructure field named MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If other information is not required, this parameter must be NULL.

EstimatedTime (output)
The number of seconds estimated before the signed certificate will be ready to be retrieved. A (default) value of zero indicates that the signed certificate can be retrieved immediately via the corresponding *CL_CertRetrieve* function call. When the certification process cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceIdentifier (output)
A reference identifier which uniquely identifies this specific request. The handle persists across application executions until it is terminated by the successful or failed completion of the CSSM_CL_CertRetrieve *function.*

RETURN VALUE

A CSSM_OK return value signifies the requested operation has proceeded and that *CL_CertRetrieve* should be called (after the specified amount of time) in order to retrieve the results. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_INVALID_SIGNER_CERTIFICATE
Revoked or expired signer certificate.

CSSM_CL_INVALID_SCOPE
Invalid scope.

CSSM_AUTHENTICATION_FAIL
Invalid/unauthorized credential.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_CERT_REQUEST_FAIL
Unable to submit certificate creation request.

SEE ALSO

CSSM_CL_CertRetrieve, CSSM_CL_CertVerify

NAME

CSSM_CL_CertRetrieve

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CertRetrieve
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_NET_ADDRESS_PTR CALocation,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the certificate created in response to the *CL_CertRequest* function call. The reference identifier denotes the corresponding CertRequest call. The signing operation, performed by the Certificate Authority (CA) process, may have been performed locally or remotely. In either case, the private key associated with the certificate is stored in the local CSP specified by the caller. The CL module and the CA process provide secure handling (via key wrapping) of the private key until it is securely stored in the local CSP.

The caller may be required to provide additional authentication information to retrieve the certificate. The format of these credentials is defined by the CL module and recorded in the CLSubservice structure, which can be queried by the caller.

This function returns the signed certificate and stores the associated private key in the CSP specified in CSSM_CL_CertRequest. *It is possible that the certificate is not ready to be retrieved when this call is made. In that case, an EstimatedTime to complete certificate creation is returned with a NULL certificate pointer. The caller must attempt to retrieve the certificate again after the estimated time to completion has elapsed.*

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ReferenceIdentifier (input)
A reference identifier which uniquely identifies the *CSSM_CL_CertRequest* call that initiated creation of the certificate returned by this function. The identifier persists across application executions until the *CSSM_CL_CertRetrieve* function completes (in success or failure).

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on state information associated with the ReferenceIdentifier or can assume a default CA process location. If insufficient state is associated with the ReferenceIdentifier and a default cannot be assumed, the retrieval cannot be completed and the operation fails.

UserAuthentication (input/optional)
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on, depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information is insufficient, additional information can be provided by the substructure field named MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If other information is not required, this parameter must be NULL.

EstimatedTime (output)
: The number of seconds estimated before the signed Certificate will be returned. A (default) value of zero indicates that the signed Certificate has been returned as a result of this call. When the certification process cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the signed certificate. If the pointer is NULL, the calling application is expected to call back after the specified Estimated Time. If the pointer is NULL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
: Invalid Certificate Library Handle.

CSSM_CL_INVALID_IDENTIFIER
: Invalid reference identifier.

CSSM_AUTHENTICATION_FAIL
: Invalid/unauthorized credential for operation.

CSSM_CL_CERT_SIGN_FAIL
: Unable to sign certificate.

CSSM_CL_EXTRA_SERVICE_FAIL
: Unable to perform additional certificate-creation-related services.

CSSM_CL_PRIVATE_KEY_STORE_FAIL
: Unable to store private key in CSP.

CSSM_CL_MEMORY_ERROR
: Not enough memory.

SEE ALSO

CSSM_CL_CertRequest, *CSSM_CL_CertVerify*

NAME

CSSM_CL_RegistrationFormRequest

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_RegistrationFormRequest
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_NET_ADDRESS_PTR RALocation)
```

DESCRIPTION

This function returns a blank registration form from a Registration Authority (RA) process. The RA process can be local or remote. The CL module incorporates knowledge of the name, location, and interface protocol for communicating with the RA.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

RALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the RA process. If the input is NULL, the module can assume a default RA process location. If a default cannot be assumed, the request cannot be initiated and the operation fails.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the blank registration form. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_UNABLE_TO_RETRIEVE_FORM
Unable to retrieve the registration form.

SEE ALSO

CSSM_CL_CertRequest

NAME

CSSM_CL_RegistrationFormSubmit

SYNOPSIS

```
CSSM_USER_AUTHENTICATION_PTR CSSMAPI
    CSSM_CL_RegistrationFormSubmit
    (CSSM_CL_Handle CLHandle,
    const CSSM_DATA_PTR RegistrationForm,
    const CSSM_NET_ADDRESS_ADDR RALocation,
    const CSSM_NET_ADDRESS_ADDR CALocation)
```

DESCRIPTION

The completed registration form is submitted to a Registration Authority requesting approval for certificate generation by a Certification Authority. An authentication credential is returned. This credential can be used as the input authentication credential in a certificate request call.

PARAMETERS

CLHandle (input)
A handle for the module that will perform the operation.

RegistrationForm (input)
A pointer to the CSSM_DATA structure containing the completed registration form to be submitted to the Registration Authority and Certification Authority.

RALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the RA process. If the input is NULL, the module can assume a default RA process location. If a default cannot be assumed, the request cannot be initiated and the operation fails.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module or the Registration Authority can assume a default CA process location. If a default cannot be assumed, the request cannot be initiated and the operation fails.

RETURN VALUE

A pointer to a CSSM_USER_AUTHENTICATION credential. When NULL is returned, an error occurred or the registration form was rejected by the RA or the CA. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_RA
Unknown or unreachable Registration Authority.

CSSM_CL_NO_DEFAULT_RA
No default Registration Authority.

CSSM_CL_RA_REJECTED_FORM
RA rejected the registration form.

CSSM_CL_CA_REJECTED_FORM
CA rejected the registration form.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_FORM_SUBMIT_FAIL
Unable to submit the registration form.

SEE ALSO
CSSM_CL_RegistrationFormRequest

NAME

CSSM_CL_CertMultiSignRequest

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CL_CertMultiSignRequest
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR SubjectCert,
    const CSSM_DATA_PTR CACerts,
    uint32 NumberOfCACerts,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize,
    const CSSM_NET_ADDRESS_PTR CALocation,
    CSSM_CA_SERVICES MoreServiceRequests,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

This function submits a request to a Certificate Authority (CA) process to add one or more signatures to an existing certificate. This could be a notary public service or a simple multiple signature facility. The CA process may be local or remote.

The CA process performs the signaturing operation once for each specified signer certificate. The signing operation may be performed locally or remotely. The CA must have access to the private keys associated with the signer certificates. If no signer's certificate is specified, the CA can assume one or more default signing certificates it uses for a multi-signing service. If no defaults are defined, the CA can reject the request.

The CL module selects and uses a default CSP to perform any required cryptographic operations. The CL module is responsible for creating and destroying all cryptographic contexts required to perform these operations.

The SignScope defines the set of certificate fields in the Subject Cert that are to be included in the signing process.

The caller can request additional signing-related services from the CA. These requests are designated by the MoreServiceRequests bit mask. CSSM-defined bit masks allow the caller to request full notary public services, and re-publishing the new multiply-signed certificate with all directory services holding a copy of the old certificate. CAs are not required to provide such services. The CL module works with the CA process to provide the requested services.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimate time defines the expected signing time. This time may be substantial when the multiple signature model requires off-line procedures (such as a notary public). In contrast, the estimated time can be zero, meaning the multiply-signed certificate can be obtained immediately. After the specified time has elapsed, the caller must use the CL module interface CSSM_CL_CertMultiSignRetrieve, with the reference identifier, to obtain the multiply-signed certificate.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

SubjectCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be signed multiple

times.

CACerts (input/optional)
A pointer to an array of one or more CSSM_DATA structures containing the signing certificates of the desired Certification Authorities. If CACerts is NULL, the CL module or the CA process can provide a default set of signing certificates.

NumberOfCACerts (input)
The number of CA signing certificates presented in the CACerts array. If no CA certificates are specified, the value of this parameter must be zero.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the OID/value pairs specifying the certificate fields to be included in the signature calculation. When the input value is NULL, the CL assumes and includes a default set of certificate fields in the signing process.

ScopeSize (input)
The number of entries in the sign scope list. If no signing scope is specified, then the scope size must be zero.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on the CACert input parameter or can assume a default CA process location. If a CACert is not specified and a default cannot be assumed, the request cannot be initiated and the operation fails.

MoreServiceRequests (input/optional)
A bit mask requesting additional signing-related services from the Certificate Authority performing this function.

UserAuthentication (input/optional)
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on—depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information is insufficient, additional information can be provided by the substructure field named MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If other information is not required, this parameter must be NULL.

EstimatedTime (output)
The number of seconds estimated before the multiply-signed certificate will be ready to be retrieved. A (default) value of zero indicates that the certificate can be retrieved immediately via the corresponding *CL_CertRetrieve* function call. When the signing authority cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceIdentifier (output)
A reference identifier which uniquely identifies this specific request. The identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_MultiSignRetrieve function.

RETURN VALUE

A CSSM_OK return value signifies the requested operation has proceeded and that *CL_CertMultiSignRetrieve* should be called (after the specified amount of time) in order to retrieve the results. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_INVALID_SIGNER_CERTIFICATE
Revoked or expired signer certificate.

CSSM_CL_INVALID_SCOPE
Invalid scope.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_SIGN_REQUEST_FAIL
Unable to submit certificate signing request.

SEE ALSO

CSSM_CL_CertMultiSignRetrieve

NAME

CSSM_CL_CertMultiSignRetrieve

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CertMultiSignRetrieve
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_NET_ADDRESS_PTR CALocation,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the multiply-signed certificate created in response to the *CL_CertMultiSignRequest* function call. The reference identifier denotes the corresponding call.

It is possible that the certificate is not ready to be retrieved when this call is made. In that case, an EstimatedTime to complete the signing process is returned with the reference identifier and a NULL certificate pointer. The caller must attempt to retrieve the certificate again after the estimated time to completion has elapsed.

PARAMETERS

CLHandle (input)
: The handle that describes the add-in certificate library module used to perform this function.

ReferenceIdentifier (input)
: A reference identifier which uniquely identifies the CSSM_CL_CertMultiSignRequest call that initiated the multiple signing request. This identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_MultiSignRetrieve function.

CALocation (input/optional)
: A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on state information associated with the ReferenceIdentifier or can assume a default CA process location. If insufficient state is associated with the ReferenceIdentifier and a default cannot be assumed, the retrieval cannot be completed and the operation fails.

EstimatedTime (output)
: The number of seconds estimated before the multiply-signed Certificate will be returned. A (default) value of zero indicates that the certificate has been returned as a result of this call. When the signing authority cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the multiply-signed certificate. If the pointer is NULL, the calling application is expected to call back after the specified EstimatedTime. If the pointer is NULL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
: Invalid Certificate Library Handle.

CSSM_CL_INVALID_IDENTIFIER
: Invalid reference identifier.

CSSM_CL_CERT_SIGN_FAIL
Unable to sign certificate.

CSSM_CL_EXTRA_SERVICE_FAIL
Unable to perform additional signing-related services.

CSSM_CL_MEMORY_ERROR
Not enough memory.

SEE ALSO

CSSM_CL_CertMultiSignRequest, *CSSM_CL_CertVerify*

NAME

CSSM_CL_CertRecoveryRequest

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CL_CertRecoveryRequest
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR CACert,
    const CSSM_NET_ADDRESS_PTR CALocation,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    const CSSM_FIELD_PTR SelectedCertFieldValues,
    const uint32 NumberOfFieldValues,
    sint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

This function submits a certificate recovery request to a Certificate Authority (CA) process (or other trusted backup facility) to prepare for the recovery of a set of certificates and their associated private keys. The caller can specify one or more certificate field values to limit the set of certificates selected for potential recovery. The recovery facility process may be local or remote.

The caller is required to provide authentication information so the CA process can determine whether the caller is authorized to recover a certificate. The specific format of the credential is specified by the CL module. The caller can query the CL Module Info structure to obtain this information. Additional authentication information may also be required. It can be provided in the substructure field named MoreAuthenticationData.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimated time defines the expected certificate recovery time. This time may be substantial when many certificates are being recovered or manual procedures are required. In contrast, the estimated time can be zero, meaning the set of recovered certificates can be obtained immediately. After the specified time has elapsed, the caller must use the CL module interface CSSM_CL_CertRecoveryRetrieve**, with the reference identifier, to obtain the set of recovered certificates from the CA process.**

PARAMETERS

CLHandle (input)

The handle that describes the add-in certificate library module used to perform this function.

CACert (input/optional)

The certificate of the certification authority that must perform the recovery operation. The caller can obtain the certificate for all of the certification authorities supported by the CL by querying the CSSM registry for the CL's CA information.

CALocation (input/optional)

A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on the CACert input parameter or can assume a default CA process location. If a CACert is not specified and a default cannot be assumed, the request cannot be initiated and the operation fails.

UserAuthentication (input/optional)

A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on, depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information is insufficient, additional information can be provided by the substructure field named MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If other information is not required, this parameter must be NULL.

SelectedCertFieldValues (input/optional)
An array of one or more field values that must be matched as part of the process of selecting certificates for recovery. If no certificate field values are specified, then the all of the caller's certificates (known to this CL module) will be selected for possible recovery.

NumberOfFieldValues (input)
The number of selected certificate field values listed in the array SelectedCertFieldValues. If no certificate field values are specified, then this value must be zero.

EstimatedTime (output)
The number of seconds estimated before the set of recovered certificates will be ready to be retrieved. A (default) value of zero indicates that the recovered certificates can be retrieved immediately via the corresponding *CL_CertRecoveryRetrieve* function call. When the recovery process cannot estimate the time required to prepare the recovered certificates, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceIdentifier (output)
A reference identifier which uniquely identifies this specific request. The handle must be used in all subsequent calls to retrieve the set of recovered certificates. The identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_CertRecoveryRetrieve *function.*

RETURN VALUE

A CSSM_OK return value signifies the requested operation has proceeded and that *CL_CertRecoveryRetrieve* should be called (after the specified amount of time) in order to retrieve the results. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_AUTHENTICATION_FAIL
Invalid/unauthorized credential.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_CERT_REQUEST_FAIL
Unable to submit certificate recovery request.

SEE ALSO

CSSM_CL_CertRecoveryRetrieve, CSSM_CL_CertRecover, CSSM_CL_CertKeyRecover, CSSM_CL_CertAbortRecovery

NAME

CSSM_CL_CertRecoveryRetrieve

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CL_CertRecoveryRetrieve
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_NET_ADDRESS_PTR CALocation,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    CSSM_HANDLE CacheHandle,
    uint32 *NumberOfRetrievedCerts,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the set of certificates recovered in response to the *CL_CertRecoveryRequest* function call. The reference identifier denotes the corresponding CertRecoveryRequest call.

The caller may be required to provide additional authentication information to recover the certificates. The format of these credentials is defined by the CL module and recorded in the CLSubservice structure, which can be queried by the caller.

The CL module selects and uses a default CSP to perform cryptographic operations, as required. Also the CL module creates and destroys all cryptographic contexts required to perform this operation.

This function obtains the set of recovered certificates and their associated private keys. It returns a cache handle to reference the returned set. The cache handle is used when retrieving individual certificates and keys using the CSSM_CL_CertRecover function.

It is possible that the recovered certificates are not ready to be retrieved when CSSM_CL_CertRecoveryRetrieve *is called. In that case, an EstimatedTime to complete certificate recovery is returned with the reference identifier and a NULL cache handle. The caller must attempt to retrieve the recovered certificates again after the estimated time to completion has elapsed.*

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ReferenceIdentifier (input)
A reference identifier which uniquely identifies the CSSM_CL_CertRecoveryRequest call that initiated recovery of the set of certificates obtained by this function. The identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_CertRecoveryRetrieve function.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on state information associated with the ReferenceIdentifier or can assume a default CA process location. If insufficient state is associated with the ReferenceIdentifier and a default cannot be assumed, the retrieval cannot be completed and the operation fails.

UserAuthentication (input/optional)
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a passphrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on, depending on the context of the request. The required format for this credential is defined by

the CL and recorded in the CLSubservice structure describing this module. If the supplied information is insufficient, additional information can be provided by the substructure field named MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If other information is not required, this parameter must be NULL.

CacheHandle (output)

A reference handle that uniquely identifies the cache of recovered certificates and their associated private keys. If the certificate retrieval process has not been completed, the returned cache handle is zero. A non-zero cache handle can be used in the CSSM_CL_CertRecover and CSSM_CL_CertKeyRecover functions to complete the recovery of an individual certificate and its private key. The handle is not persistent. It used is terminated by calling CSSM-CL_CertAbortRecovery or by termination of the caller process.

NumberOfRetrievedCerts (output)

The number of certificates in the cache.

EstimatedTime (output)

The number of seconds estimated before the set of recovered certificates will be returned. A (default) value of zero indicates that the set has been returned as a result of this call. When the recovery process cannot estimate the time required to prepare the recovered certificates, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A CSSM_RETURN value indicating whether the operation obtained a set of recovered certificates. If the result is CSSM_FAIL, and a NULL cache handle and a positive EstimatedTime are returned, then the calling application is expected to call this function again after the specified EstimatedTime. If the result is CSSM_FAIL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE

Invalid Certificate Library Handle.

CSSM_CL_INVALID_CSP_HANDLE

Invalid CSP Handle.

CSSM_CL_INVALID_IDENTIFIER

Invalid reference identifier.

CSSM_AUTHENTICATION_FAIL

Invalid/unauthorized credential for operation.

CSSM_CL_MEMORY_ERROR

Not enough memory.

SEE ALSO

CL_CertRecoveryRequest, CSSM_CL_CertRecover, CSSM_CL_CertKeyRecover, CSSM_CL_CertAbortRecovery

NAME

CSSM_CL_CertRecover

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CertRecover
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CacheHandle,
    const uint32 CacheIndex)
```

DESCRIPTION

This function returns a certificate from a cache of certificates retrieved by the CSSM_CL_CertRecoveryRetrieve function. The cache contains a set of certificates in unspecified order. The certificate to be retrieved is specified by the CacheIndex parameter, which is a simple counter from one to the number of certificates in the cache. The selected certificate is returned as a result of the function call.

This function has no effect on the private key associated with the recovered certificate. Recovery of the private key can be performed using the function CSSM_CL_CertKeyRecover**.**

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CacheHandle (input)
A reference handle that uniquely identifies the cache of retrieved, recovered certificates and their associated private keys.

CacheIndex (input)
An index value that selects a certificate from the ordered cache of retrieved, recovered certificates and associated keys. The value must be less than or equal to the number of certificates in the cache.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the recovered certificate. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_HANDLE
Invalid cache handle.

CSSM_CL_INVALID_INDEX
Cache index value is out of range.

CSSM_CL_MEMORY_ERROR
Not enough memory.

SEE ALSO

CL_CertRecoveryRequest, CSSM_CL_CertRecoveryRetrieve, CSSM_CL_CertKeyRecover, CSSM_CL_CertAbortRecovery

NAME

CSSM_CL_CertKeyRecover

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CL_CertKeyRecover
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CacheHandle,
    const uint32 CacheIndex,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_CRYPTO_DATA_PTR PassPhrase)
```

DESCRIPTION

This function recovers the private key associated with a certificate and securely stores that key in the specified cryptographic service provider. The key (and its associated certificate) are among a set of certificates and private keys contained in the cache specified by the CacheHandle.

Cache entries are in unspecified order. The private key to be retrieved is specified by the CacheIndex parameter, which is a simple counter from one to the number of certificates in the cache.

The recovery process associates the private key with the public key contained in the certificate, securely stores the private key in the specified cryptographic service provider, and associates the new PassPhrase with the recovered, stored, private key.

To selectively recover private keys from the cache, the function *CSSM_CL_CertRecover* can be used to review recovered certificates and determine the appropriate CacheIndex to use when recovering the associated private key.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CacheHandle (input)
A reference handle which uniquely identifies the cache of retrieved, recovered certificates and their associated private keys.

CacheIndex (input)
An index value that selects a certificate from the cache of retrieved, recovered certificates and associated keys. The value must be less than or equal to the number of certificates in the cache.

CSPHandle (input)
The handle that describes the add-in CSP module where the private key is to be stored. Optionally, the CL module can use this CSP to perform additional cryptographic operations or may use another default CSP for that purpose.

PassPhrase (input)
A pointer to the CSSM_CRYPTO_DATA structure containing the new passphrase to be associated with the recovered certificate and private key. The passphrase can be specified by immediate data in this parameter or a callback function to request a passphrase from the caller's process.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
: Invalid Certificate Library Handle.

CSSM_CL_INVALID_CSP_HANDLE
: Invalid CSP Handle.

CSSM_CL_INVALID_HANDLE
: Invalid cache handle.

CSSM_CL_INVALID_INDEX
: Cache index value is out of range.

CSSM_CL_PRIVATE_KEY_STORE_FAIL
: Unable to store private key in CSP.

CSSM_CL_MEMORY_ERROR
: Not enough memory.

SEE ALSO

CL_CertRecoveryRequest, *CSSM_CL_CertRecoveryRetrieve*, *CSSM_CL_CertRecover*, *CSSM_CL_CertAbortRecovery*

NAME

CSSM_CL_CertAbortRecovery

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CL_CertAbortRecovery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CacheHandle)
```

DESCRIPTION

This function terminates the iterative process of recovering certificates and their associated private keys from a cache of certificates. This function must be called even if all certificates and their associated private keys are recovered from the cache. This function destroys all intermediate state and secret information used during the certificate and key recovery process. At completion of this function, the specified cache handle is invalid and the operations CSSM_CL_CertRecover and CSSM_CL_CertKeyRecover cannot be invoked using this handle.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CacheHandle (input)
A handle which identifies the cache of retrieved, recovered certificates and their associated private keys.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_HANDLE
Invalid cache handle.

CSSM_CL_ABORT_RECOVERY_FAIL
Unable to abort the recovery process.

SEE ALSO

CL_CertRecoveryRequest, CSSM_CL_CertRecoveryRetrieve, CSSM_CL_CertRecover, CSM_CL_CertKeyRecover

NAME

CSSM_CL_CertVerify

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_CL_CertVerify
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR CertToBeVerified,
    const CSSM_DATA_PTR SignerCert,
    const CSSM_FIELD_PTR VerifyScope,
    uint32 ScopeSize)
```

DESCRIPTION

This function verifies that the signed certificate has not been altered since it was signed by the designated signer. Only one signature is verified by this function. If the certificate to be verified includes multiple signatures, this function must be applied once for each signature to be verified. This function verifies a digital signature over the certificate fields specified by VerifyScope. If the verification scope fields are not specified, the function performs verification using a pre-selected set of fields in the certificate.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

CertToBeVerified (input)
A pointer to the CSSM_DATA structure containing a certificate containing at least one signature for verification. An unsigned certificate template cannot be verified.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate used to sign the subject certificate. This certificate provides the public key to use in the verification process and if the certificate being verified contains multiple signatures, the signer's certificate indicates which signature is to be verified.

VerifyScope (input/optional)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be used in verifying the signature. (This should include all of the fields that were used to calculate the signature.) If the verify scope is null, the certificate library module assumes that its default set of certificate fields were used to calculate the signature, and those same fields are used in the verification process.

ScopeSize (input)
The number of entries in the verify scope list. If the verification scope is not specified, the input value for scope size must be zero.

RETURN VALUE

CSSM_TRUE if the certificate signature verified. CSSM_FALSE if the certificate signature did not verify or an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Cryptographic Context handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_INVALID_CONTEXT
Invalid context for the requested operation.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_INVALID_SCOPE
Invalid scope.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CERT_VERIFY_FAIL
Unable to verify certificate.

SEE ALSO
CSSM_CL_CertSign

NAME

CSSM_CL_CertGetFirstFieldValue

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CertGetFirstFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Cert,
    const CSSM_OID_PTR CertField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields)
```

DESCRIPTION

This function returns the value of the designated certificate field. If more than one field matches the CertField OID, the first matching field will be returned. The number of matching fields is an output parameter, as is the ResultsHandle to be used to retrieve the remaining matching fields.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate.

CertField (input)
A pointer to an object identifier which identifies the field value to be extracted from the Cert.

ResultsHandle (output)
A pointer to the CSSM_HANDLE which should be used to obtain any additional matching fields.

NumberOfMatchedFields (output)
The number of fields which match the CertField OID.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the value of the requested field. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_TAG
Unknown field tag in OID.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CERT_GET_FIELD_VALUE_FAIL
Unable to get field value.

SEE ALSO

CSSM_CL_CertGetNextFieldValue, *CSSM_CL_CertAbortQuery*, *CSSM_CL_CertGetAllFields*

NAME

CSSM_CL_CertGetNextFieldValue

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CertGetNextFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function returns the value of a certificate field, when that field occurs multiple times in a certificate. Certificates with repeated fields (such as multiple signatures) have multiple field values corresponding to a single OID. A call to the function *CSSM_CL_CertGetFirstFieldValue* initiates the process and returns a results handle identifying the certificate from which values are being obtained and the OID corresponding to those values. The *CSSM_CL_CertGetNextFieldValue* function can be called repeatedly to obtain these values, one at a time.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ResultsHandle (input)
The handle which identifies the results of a certificate query.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the value of the requested field. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_RESULTS_HANDLE
Invalid Results handle.

CSSM_CL_NO_FIELD_VALUES
No more field values for the input handle.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CERT_GET_FIELD_VALUE_FAIL
Unable to get field value.

SEE ALSO

CSSM_CL_CertGetFirstFieldValue, CSSM_CL_CertAbortQuery

NAME

CSSM_CL_CertAbortQuery

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CL_CertAbortQuery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function terminates the get operation initiated by *CSSM_CL_CertGetFirstFieldValue* and allows the CL to release all intermediate state information associated with the query. This function should be called even if all values retrieved by the call to *CSSM_CL_CertGetFirstFieldValue* are obtained by repeated calls to *CSSM_CL_CertGetNextFieldValue*.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ResultsHandle (input)
A pointer to the handle which identifies the results of a CSSM_CL_GetFieldValue request.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_RESULTS_HANDLE
Invalid Results handle.

CSSM_CL_CERT_ABORT_QUERY_FAIL
Unable to abort the certificate query.

SEE ALSO

CSSM_CL_CertGetFirstFieldValue, *CSSM_CL_CertGetNextFieldValue*

NAME

CSSM_CL_CertGetKeyInfo

SYNOPSIS

```
CSSM_KEY_PTR CSSMAPI CSSM_CL_CertGetKeyInfo
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Cert)
```

DESCRIPTION

This function returns the public key and integral information about the key from the specified certificate. The key structure returned is a compound object. It can be used in any function requiring a key, such as creating a cryptographic context.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate from which to extract the public key information.

RETURN VALUE

A pointer to the CSSM_KEY structure containing the public key and possibly other key information. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_UNKNOWN_TAG
Unknown field tag in OID.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CERT_GET_KEY_INFO_FAIL
Unable to get key information.

SEE ALSO

CSSM_CL_CertGetFirstFieldValue

NAME

CSSM_CL_CertGetAllFields

SYNOPSIS

```
CSSM_FIELD_PTR CSSMAPI CSSM_CL_CertGetAllFields
    (CSSM_CL_HANDLE CLHandle,
    CSSM_DATA_PTR Cert,
    uint32 *NumberOfFields)
```

DESCRIPTION

This function returns a list of the values stored in the input certificate.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate whose fields will be returned.

NumberOfFields (output)
The length of the returned array of fields.

RETURN VALUE

A pointer to an array of CSSM_FIELD structures which contain the values of all of the fields of the input certificate. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid DATA pointer.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_CERT_GET_FIELD_VALUE_FAIL
Unable to return the list of fields.

SEE ALSO

CSSM_CL_CertGetFirstFieldValue, *CSSM_CL_CertDescribeFormat*

NAME

CSSM_CL_CertGroupToSignedBundle

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CertGroupToSignedBundle
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CERTGROUP_PTR CertGroupToBundle,
    const CSSM_DATA_PTR SignerCert,
    const CSSM_CERT_BUNDLE_HEADER_PTR BundleInfo);
```

DESCRIPTION

This function accepts as input a certificate group (as an array of individual certificates) and returns a certificate bundle (a codified and signed aggregation of the certificates in the group). The certificate group will first be encoded according to the BundleInfo input by the user. If BundleInfo is NULL, the library will perform a default encoding for its default bundle type. If possible, the certificate group ordering will be maintained in this certificate aggregate encoding. After encoding, the certificate aggregate will be signed using the input context and signer certificate. The CL module embeds knowledge of the signing scope for the bundle types it supports. The signature is then associated with the certificate aggregate according to the bundle type and encoding rules and is returned as a bundle to the calling application.

PARAMETERS

CLHandle (input)
The handle of the add-in module to perform this operation.

CCHandle (input)
The handle of the cryptographic context to control the signing operation. The operation will fail if a signature is required for this type of bundle and the cryptographic context is not valid.

CertGroupToBundle (input)
An array of individual, encoded certificates. All of the certificates in this list will be included in the resulting certificate bundle.

SignerCert (input/optional)
If signing is required for this type of certificate bundle, this is the certificate to be used to sign the bundle. If a signing certificate is required but not specified, then the module will assume a default certificate. If a signature is not required for this certificate bundle type, this parameter will be ignored.

BundleInfo (input/optional)
A structure containing the type and encoding of the bundle to be created. If the type and the encoding are not specified, then the module will assume a default bundle type and bundle encoding.

RETURN VALUE

The function returns a pointer to a signed certificate bundle containing all of the certificates in the certificate group. The bundle is of the type and encoding requested by the caller or is the default defined by the library module if the BundleInfo was not specified by the caller. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid context handle.

CSSM_CL_INVALID_BUNDLE_INFO
Unknown bundle type or encoding.

CSSM_CL_INVALID_SIGNER_CERTIFICATE
Revoked or expired signer certificate.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_CERGROUPTOBUNDLE_FAIL
Unable to create the signed bundle.

SEE ALSO

CSSM_CL_CertGroupFromVerifiedBundle

NAME

CSSM_CL_CertGroupFromVerifiedBundle

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_CL_CertGroupFromVerifiedBundle
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CERT_BUNDLE_PTR CertBundle,
    const CSSM_DATA_PTR SignerCert,
    CSSM_CERTGROUP_PTR *CertGroup);
```

DESCRIPTION

This function accepts as input a certificate bundle (a codified and signed aggregation of the certificates in the group), verifies the signature of the bundle (if a signature is present) and returns a certificate group (as an array of individual certificates) including every certificate contained in the bundle. The signature on the certificate aggregate is verified using the cryptographic context and possibly using the input signer certificate. The CL module embeds the knowledge of the verification scope for the bundle types that it supports. A CL module's supported bundle types and encodings are available to applications by querying the CSSM registry. The type and encoding of the certificate bundle must be specified with the input bundle. If signature verification is successful, the certificate aggregate will be parsed into a certificate group whose order corresponds to the certificate aggregate ordering. This certificate group will then be returned to the calling application.

PARAMETERS

CLHandle (input)
The handle of the add-in module to perform this operation.

CCHandle (input)
The handle of the cryptographic context to control the verification operation.

CertBundle (input)
A structure containing a reference to a signed, encoded bundle of certificates, and to descriptors of the type and encoding of the bundle. The bundled certificates are to be separated into a certificate group (list of individual encoded certificates). If the bundle type and bundle encoding are not specified, the add-in module may either attempt to decode the bundle assuming a default type and encoding or may immediately fail.

SignerCert (input/optional)
The certificate to be used to verify the signature on the certificate bundle. If the bundle is signed but this field is not specified, then the module will assume a default certificate for verification.

CertGroup (output)
A pointer to the certificate group, represented as an array of individual, encoded certificates. The group contains all of the certificates contained in the certificate bundle.

RETURN VALUE

A CSSM_BOOL value corresponding to the result of the verification process. If a signature is required for this type of bundle and signature verification fails, the function returns CSSM_FALSE. If signature verification is required and succeeds, the function returns CSSM_TRUE and attempts to create a certificate group containing all certificates in the bundle. If the group cannot be created, the CertGroup is set to NULL and an error code is set. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid context handle.

CSSM_CL_INVALID_BUNDLE_INFO
Unknown bundle type or encoding.

CSSM_CL_INVALID_SIGNER_CERTIFICATE
Revoked or expired signer certificate.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_CERGROUPFROMBUNDLE_FAIL
Unable to create the cert group.

SEE ALSO

CSSM_CL_CertGroupToSignedBundle

NAME

CSSM_CL_CertImport

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CertImport
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CERT_TYPE ForeignCertType,
    CSSM_CERT_ENCODING ForeignCertEncoding,
    const CSSM_DATA_PTR ForeignCert)
```

DESCRIPTION

This function imports a certificate from the specified foreign format into the native format of the specified certificate library. The set of ForeignCertTypes supported for import is at the discretion of the certificate library and documented for each module as part of the CSSM_CLSUBSERVICE structure available from the CSSM Registry.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ForeignCertType (input)
A unique value that identifies the type of the certificate being imported.

ForeignCertEncoding (input)
A unique value that identifies the encoding of the certificate being imported.

ForeignCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be imported into the certificate library modules native certificate type.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the native-type certificate imported from the foreign certificate. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CERT_IMPORT_FAIL
Unable to import certificate.

SEE ALSO

CSSM_CL_CertExport

NAME

CSSM_CL_CertExport

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CertExport
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CERT_TYPE TargetCertType,
    CSSM_CERT_ENCODING TargetCertEncoding,
    const CSSM_DATA_PTR NativeCert)
```

DESCRIPTION

This function exports a certificate from the native format of the specified certificate library into the specified target certificate format. The set of TargetCertTypes supported for export is at the discretion of the certificate library and is documented for each module as part of the CSSM_CLSUBSERVICE structure available from the CSSM Registry.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

TargetCertType (input)
A unique value which identifies the target type of the certificate being exported.

TargetCertEncoding (input)
A unique value which identifies the encoding of the certificate being exported.

NativeCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be exported.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the target-type certificate exported from the native certificate. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CERT_EXPORT_FAIL
Unable to export certificate.

SEE ALSO

CSSM_CL_CertImport

NAME

CSSM_CL_CertDescribeFormat

SYNOPSIS

```
CSSM_OID_PTR CSSMAPI CSSM_CL_CertDescribeFormat
    (CSSM_CL_HANDLE CLHandle,
    uint32 *NumberOfFields)
```

DESCRIPTION

This function returns a list of the object identifiers used to describe the certificate format supported by the specified CL.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

NumberOfFields (output)
The length of the returned array of OIDs.

RETURN VALUE

A pointer to the array of CSSM_OIDs which represent the supported certificate format. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid handle.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_CERT_DESCRIBE_FORMAT_FAIL
Unable to return the list of fields.

SEE ALSO

CSSM_CL_CertGetAllFields, *CSSM_CL_CertGetFirstFieldValue*, *CSSM_CL_CertGetNextFieldValue*, *CSSM_CL_CertAbortQuery*, *CSSM_CL_CertGetKeyInfo*

13.4 Certificate Revocation List Operations

This chapter describes the function prototypes and error codes supported by a Certificate Library module for operations on certificate revocation lists (CRLs). The error codes given in this chapter constitute the generic error codes which are defined by CSSM for use by all certificate libraries in describing common error conditions. A certificate library may also define and return vendor-specific error codes. The error codes defined by CSSM are considered to be comprehensive and few if any vendor-specific codes should be required. Applications must consult vendor-supplied documentation for the specification and description of any error codes defined outside of this specification.

NAME

CSSM_CL_CrlCreateTemplate

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlCreateTemplate
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_FIELD_PTR CrlTemplate,
    uint32 NumberOfFields);
```

DESCRIPTION

This function creates an unsigned, memory-resident CRL. Fields in the CRL are initialized with the descriptive data specified by the OID/value input pairs. The specified OID/value pairs can initialize all or a subset of the general attribute fields in the new CRL. Subsequent values may be set using the *CSSM_CL_CrlSetFields* operation. The new CRL contains no revocation records.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CrlTemplate (input)
An array of OID/value pairs specifying the initial values for descriptive data fields of the new CRL.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlTemplate input parameter.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the new CRL. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_FIELD_POINTER
Invalid pointer input.

CSSM_CL_INVALID_TEMPLATE
Invalid template for this CRL type.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CRL_CREATE_FAIL
Unable to create CRL.

SEE ALSO

CSSM_CL_CrlSetFields, *CSSM_CL_CrlAddCert*, *CSSM_CL_CrlSign*, *CSSM_CL_CertGetFirstFieldValue*

NAME

CSSM_CL_CrlSetFields

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlSetFields
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_FIELD_PTR CrlTemplate,
    uint32 NumberOfFields,
    const CSSM_DATA_PTR OldCrl);
```

DESCRIPTION

This function will set the fields of the input CRL to the new values, specified by the input OID/value pairs. If there is more than one possible instance of an OID (for example, as in an extension or CRL record) then a NEW field with the specified value is added to the CRL.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CrlTemplate (input)
Any array of field OID/value pairs containing the values to initialize the CRL attribute fields.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlTemplate input parameter.

OldCrl (input)
The CRL to be updated with the new attribute values. The CRL must be unsigned and available for update.

RETURN VALUE

A pointer to the modified, unsigned CRL. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_FIELD_POINTER
Invalid pointer input.

CSSM_CL_INVALID_TEMPLATE
Invalid template for this CRL type.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CRL_SET_FAIL
Unable to set CRL field values.

SEE ALSO

CSSM_CL_CrlCreateTemplate, *CSSM_CL_CrlAddCert*, *CSSM_CL_CrlSign*, *CSSM_CL_CertGetFirstFieldValue*

NAME

CSSM_CL_CrlRequest

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CL_CrlRequest
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_FIELD_PTR CrlIdentifier,
    const CSSM_DATA_PTR CACert,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize,
    const CSSM_NET_ADDRESS_PTR CALocation,
    CSSM_CA_SERVICES MoreServiceRequests,
    sint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

This function submits a request to a Certificate Authority (CA) process to issue the most current version of a CRL of a specified name. The SignerCert input parameter indicates which CA process should receive the request. The selected CA process may be local or remote.

When all prerequisite conditions have been satisfied, such as some minimum time has elapsed since the last version of the requested CRL was issued, the CA process closes out the CRL, signs it and can distribute it to all interested and requesting parties. The CA must have access to the private keys associated with the signer's certificate to sign the CRL. If no signer's certificate is specified, the CL module can assume a default CA process from which it always acquires CRLs. If no defaults are known to the CL module, the CL module can reject the request.

The CL module selects and uses a default CSP for any required cryptographic operations. The CL module and the CA process are responsible for creating and destroying all cryptographic contexts required to perform this service.

The SignScope defines the set of CRL fields that are to be included in the signing process.

The caller can request additional CRL-related services from the CA. These requests are designated by the MoreServiceRequests bit mask. CSSM-defined bit masks allow the caller to request immediate distribution of the latest CRL to any and all interested parties. CAs are not required to provide these additional services. The CL module works with the CA process to provide the requested CRL.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimate time defines the expected closing, signing and distribution time. This time may be substantial when closing a CRL requires off-line procedures or the service model mandates a minimum time between distributions. In contrast, the estimated time can be zero, meaning the CRL can be obtained immediately. After the specified time has elapsed, the caller must use the CL module interface CSSM_CL_CrlRetrieve, with the reference identifier, to obtain the CRL.

PARAMETERS

CLHandle (input)
: The handle that describes the add-in certificate library module used to perform this function.

CrlIdentifier (input)
: A pointer to an OID-value pair that uniquely identifies (names) the CRL being requested from the CA.

CACert (input/optional)
: A pointer to the CSSM_DATA structure containing the desired Certification Authority's

signing certificate to be used when issuing the CRL. If the CACert is NULL, the CL module or the CA process can provide a default signing certificate for issuing the CRL.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the OID/value pairs specifying the CRL fields to be included in the signature calculation. When the input value is NULL, the CA assumes and includes a default set of CRL fields in the signing process.

ScopeSize (input)
The number of entries in the sign scope list. If no signing scope is specified, then the scope size must be zero.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on the CACert input parameter or can assume a default CA process location. If a CACert is not specified and a default cannot be assumed, the request cannot be initiated and the operation fails.

MoreServiceRequests (input/optional)
A bit mask requesting additional CRL-related services from the Certificate Authority performing this function.

EstimatedTime (output)
The number of seconds estimated before the CRL will be ready to be retrieved. A (default) value of zero indicates that the CRL can be retrieved immediately via the corresponding *CL_CrlRetrieve* function call. When the certification process cannot estimate the time required to prepare the CRL, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceIdentifier (output)
A reference identifier which uniquely identifies this specific request. The identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_CrlRetrieve *function.*

RETURN VALUE

A CSSM_OK return value signifies the requested operation has proceeded and that *CL_CrlRetrieve* should be called (after the specified amount of time) in order to retrieve the results. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library handle.

CSSM_CL_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_INVALID_SIGNER_CERTIFICATE
Revoked or expired signer certificate.

CSSM_CL_INVALID_SCOPE
Invalid scope.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_SIGN_REQUEST_FAIL
Unable to submit certificate signing request.

SEE ALSO

CL_CrlRetrieve

NAME

CSSM_CL_CrlRetrieve

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlRetrieve
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_NET_ADDRESS_PTR CALocation,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the CRL closed and issued in response to the *CL_CrlRequest* function call. The reference identifier identifies the corresponding call.

It is possible that the CRL is not ready to be retrieved when this call is made. In that case, an EstimatedTime to complete the CRL issuing process is returned with the reference identifier and a NULL certificate pointer. The caller must attempt to retrieve the CRL again after the estimated time to completion has elapsed.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ReferenceIdentifier (input)
A reference identifier which uniquely identifies the *CSSM_CL_CrlRequest* call that initiated the CRL issuing request. The identifier persists across application executions until it is terminated by successful or failed completion of the *CSSM_CL_CrlRetrieve* function.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on state information associated with the ReferenceIdentifier or can assume a default CA process location. If insufficient state is associated with the ReferenceIdentifier and a default cannot be assumed, the retrieval cannot be completed and the operation fails.

EstimatedTime (output)
The number of seconds estimated before the CRL will be returned. A (default) value of zero indicates that the CRL has been returned as a result of this call. When the certification process cannot estimate the time required to prepare the CRL, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the CRL. If the pointer is NULL, the calling application is expected to call back after the specified EstimatedTime. If the pointer is NULL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_IDENTIFIER
Invalid reference identifier.

CSSM_CL_CERT_SIGN_FAIL
Unable to sign CRL.

CSSM_CL_EXTRA_SERVICE_FAIL
Unable to perform additional CRL-related services.

CSSM_CL_MEMORY_ERROR
Not enough memory.

SEE ALSO

CSSM_CL_CrlRequest

NAME

CSSM_CL_CrlAddCert

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlAddCert
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR Cert,
    const CSSM_DATA_PTR RevokerCert,
    const CSSM_FIELD_PTR CrlEntryFields,
    uint32 NumberOfFields,
    const CSSM_DATA_PTR OldCrl)
```

DESCRIPTION

This function revokes the input certificate by adding a record representing the certificate to the CRL. The values for the new entry in the CRL are specified by the a list of OID/value input pairs. The reason for revocation is a typical value specified in the list. The revoker's certificate is used to sign the new CRL entry. The operation is valid only if the CRL has not been closed by the process of signing the CRL (by executing the CSSM_CL_CrlSign function). Once the CRL has been signed, entries cannot be added or removed.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be revoked.

RevokerCert (input)
A pointer to the CSSM_DATA structure containing the revoker's certificate.

CrlEntryFields (input)
An array of OID/value pairs specifying the initial values for descriptive data fields of the new CRL entry.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlEntryFields input parameter.

OldCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to which the newly-revoked certificate will be added.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the updated CRL. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Context Handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
Invalid Certificate.

CSSM_CL_INVALID_CRL
Invalid CRL.

CSSM_CL_INVALID_FIELD_POINTER
Invalid pointer input.

CSSM_CL_MEMORY_ERROR
Not enough memory to allocate the CRL.

CSSM_CL_CRL_ADD_CERT_FAIL
Unable to add certificate to CRL.

SEE ALSO

CSSM_CL_CrlRemoveCert

NAME

CSSM_CL_CrlRemoveCert

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlRemoveCert
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Cert,
    const CSSM_DATA_PTR OldCrl)
```

DESCRIPTION

This function reinstates a certificate by removing it from the specified CRL. The operation is valid only if the CRL has not been closed by the process of signing the CRL (by executing the CSSM_CL_CrlSign function). Once the CRL has been signed, entries cannot be added or removed.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be reinstated.

OldCrl (input)
A pointer to the CSSM_DATA structure containing the CRL from which the certificate is to be removed.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the updated CRL. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
Invalid Certificate.

CSSM_CL_CERT_NOT_FOUND_IN_CRL
Certificate not referenced by the CRL.

CSSM_CL_INVALID_CRL
Invalid CRL.

CSSM_CL_MEMORY_ERROR
Not enough memory to allocate the CRL.

CSSM_CL_CRL_REMOVE_CERT_FAIL
Unable to remove certificate from CRL.

SEE ALSO

CSSM_CL_CrlAddCert

NAME

CSSM_CL_CrlSign

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlSign
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR UnsignedCrl,
    const CSSM_DATA_PTR SignerCert,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize)
```

DESCRIPTION

This function signs, in accordance with the specified cryptographic context, the fields of the CRL indicated in the SignScope parameter. Once the CRL has been signed it may not be modified. This means that entries cannot be added or removed from the CRL through application of the *CSSM_CL_CrlAddCert* or *CSSM_CL_CrlRemoveCert* operations. A signed CRL can be verified, applied to a data store, and searched for values.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

UnsignedCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to be signed.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be used to sign the CRL.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be signed. If the signing scope is null, the certificate library module includes a default set of CRL fields in the signing process.

ScopeSize (input)
The number of entries in the sign scope list. If the signing scope is not specified, the input scope size must be zero.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the signed CRL. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Context Handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
Invalid Certificate.

CSSM_CL_INVALID_CRL_PTR
Invalid CRL pointer.

CSSM_CL_INVALID_SCOPE
: Signing scope is invalid.

CSSM_CL_MEMORY_ERROR
: Not enough memory to allocate the CRL.

CSSM_CL_CRL_SIGN_FAIL
: Unable to sign CRL.

SEE ALSO

CSSM_CL_CrlVerify

NAME

CSSM_CL_CrlVerify

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_CL_CrlVerify
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR CrlToBeVerified,
    const CSSM_DATA_PTR SignerCert,
    const CSSM_FIELD_PTR VerifyScope,
    uint32 ScopeSize)
```

DESCRIPTION

This function verifies that the signed CRL has not been altered since it was signed by the designated signer. It does this by verifying the digital signature over the fields specified by the VerifyScope parameter.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

CrlToBeVerified (input)
A pointer to the CSSM_DATA structure containing the CRL to be verified.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate used to sign the CRL.

VerifyScope (input/optional)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be verified. If the verification scope is null, the certificate library module assumes that a default set of fields were used in the signing process and those same fields are used in the verification process.

ScopeSize (input)
The number of entries in the verify scope list. If the verification scope is not specified, the input value for scope size must be zero.

RETURN VALUE

A CSSM_TRUE return value signifies that the certificate revocation list verifies successfully. When CSSM_FALSE is returned, either the CRL verified unsuccessfully or an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Context Handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
Invalid Certificate.

CSSM_CL_INVALID_CRL_PTR
Invalid CRL pointer.

CSSM_CL_INVALID_SCOPE
: Verify scope is invalid.

CSSM_CL_MEMORY_ERROR
: Not enough memory to allocate the CRL.

CSSM_CL_CRL_VERIFY_FAIL
: Unable to verify CRL.

SEE ALSO

CSSM_CL_CrlSign

NAME

CSSM_CL_IsCertInCrl

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_CL_IsCertInCrl
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Cert,
    const CSSM_DATA_PTR Crl)
```

DESCRIPTION

This function searches the CRL for a record corresponding to the certificate. The operation will fail if neither the CRL or the revocation records in the CRL have been signed. If a signature exists, the application is responsible for verifying that the signature was created by a trust party. The *CSSM_TP_CrlVerify* function can be invoked to perform this service.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be located.

Crl (input)
A pointer to the CSSM_DATA structure containing the CRL to be searched.

RETURN VALUE

A CSSM_TRUE return value signifies that the certificate is in the CRL. When CSSM_FALSE is returned, either the certificate is not in the CRL or an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
Invalid Certificate.

CSSM_CL_INVALID_CRL_PTR
Invalid CRL pointer.

NAME

CSSM_CL_CrlGetFirstFieldValue

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlGetFirstFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Crl,
    const CSSM_OID_PTR CrlField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedCrls)
```

DESCRIPTION

This function returns the value of the designated CRL field. If more than one field matches the CrlField OID, the first matching field will be returned. The number of matching fields is an output parameter, as is the ResultsHandle to be used to retrieve the remaining matching fields.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Crl (input)
A pointer to the CSSM_DATA structure which contains the CRL from which the field is to be retrieved.

CrlField (input)
An object identifier which identifies the field value to be extracted from the Crl.

ResultsHandle (output)
A pointer to the CSSM_HANDLE which should be used to obtain any additional matching fields.

NumberOfMatchedFields (output)
The number of fields which match the CrlField OID.

RETURN VALUE

Returns a pointer to a CSSM_DATA structure containing the first field which matched the CrlField. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_UNKNOWN_TAG
Unrecognized field tag in OID.

CSSM_CL_NO_FIELD_VALUES
No fields match the specified OID.

CSSM_CL_INVALID_CRL_PTR
Invalid CRL pointer.

CSSM_CL_CRL_GET_FIELD_VALUE_FAIL
Unable to get first field value.

SEE ALSO

CSSM_CL_CrlGetNextFieldValue, *CSSM_CL_CrlAbortQuery*

NAME

CSSM_CL_CrlGetNextFieldValue

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_CL_CrlGetNextFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function returns the value of a CRL field, when that field occurs multiple times in a CRL. CRL with repeated fields (such as revocation records) have multiple field values corresponding to a single OID. A call to the function *CSSM_CL_CrlGetFirstFieldValue* initiates the process and returns a results handle identifying the CRL from which values are being obtained and the OID corresponding to those values. The *CSSM_CL_CrlGetNextFieldValue* function can be called repeatedly to obtain these values, one at a time.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a CRL query.

RETURN VALUE

Returns a pointer to a CSSM_DATA structure containing the next field in the CRL that matched the CrlField specified in the *CL_CrlGetFirstFieldValue* function. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_NO_FIELD_VALUES
No more matches in the CRL.

CSSM_CL_INVALID_CRL_PTR
Invalid CRL pointer.

CSSM_CL_CRL_GET_FIELD_VALUE_FAIL
Unable to get next value.

SEE ALSO

CSSM_CL_CrlGetFirstFieldValue, *CSSM_CL_CrlAbortQuery*

NAME

CSSM_CL_CrlAbortQuery

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_CL_CrlAbortQuery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function terminates the query initiated by *CL_CrlGetFirstFieldValue* and allows the CL to release all intermediate state information associated with the get operation.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ResultsHandle (input)
The handle which identifies the results of a CRL query.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_RESULTS_HANDLE
Invalid query handle.

CSSM_CL_CRL_ABORT_QUERY_FAIL
Unable to get next item.

SEE ALSO

CSSM_CL_CrlGetFirstFieldValue, *CSSM_CL_CrlGetNextFieldValue*

NAME

CSSM_CL_CrlDescribeFormat

SYNOPSIS

```
CSSM_OID_PTR CSSMAPI CSSM_CL_CrlDescribeFormat
    (CSSM_CL_HANDLE CLHandle,
    uint32 *NumberOfFields)
```

DESCRIPTION

This function returns a list of the object identifiers used to describe the CRL format supported by the specified CL.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

NumberOfFields (output)
The length of the returned array of OIDs.

RETURN VALUE

A pointer to the array of CSSM_OIDs which represent the supported CRL format. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid handle.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_CRL_DESCRIBE_FORMAT_FAIL
Unable to return the list of fields.

13.5 Extensibility Functions

The manpages for Certificate Library extensibility functions follow on the next page.

NAME

CSSM_CL_PassThrough

SYNOPSIS

```
void * CSSMAPI CSSM_CL_PassThrough
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    uint32 PassThroughId,
    const void *InputParams)
```

DESCRIPTION

This function allows applications to call certificate library module-specific operations. Such operations may include queries or services that are specific to the domain represented by the CL module.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input/optional)
The handle that describes the context of the cryptographic operation. If the module-specific operation does not perform any cryptographic operations a cryptographic context is not required.

PassThroughId (input)
An identifier assigned by the CL module to indicate the exported function to perform.

InputParams (input)
A pointer to a module implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested CL module.

RETURN VALUE

A pointer to a module implementation-specific structure containing the output from the pass-through function. The output data must be interpreted by the calling application based on externally available information. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Cryptographic Context Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_PASS_THROUGH_FAIL
Unable to perform pass through.

Chapter 14

Data Storage Library Services API

14.1 Overview

The primary purpose of a data storage library (DL) module is to provide persistent storage of security-related objects including certificates, certificate revocation lists (CRLs), keys, and policy objects. A DL module is responsible for the creation and accessibility of one or more data stores. A single DL module can be tightly tied to a CL and/or TP module, or can be independent of all other module types. A single data store can contain a single object type in one format, a single object type in multiple formats, or multiple object types. The persistent repository can be local or remote.

CSSM stores and manages meta-information about a DL in the CSSM registry. This information describes the storage and retrieval capabilities of a DL. Applications can query the CSSM registry to obtain information about the available DLs and attach to a DL that provides the needed services. Some DL services can acquire and store meta-information about each of the data stores it manages. When this information is available it is stored in the CSSM registry. Not all DL service providers can supply this information.

The DL APIs define a data storage model that can be implemented using a custom storage device, a traditional local or remote file system service, a database management system package, or a complete (local or remote) information management system. The abstract data model defined by the DL APIs partitions all values stored in a data record into two categories: one or more mutable attributes and one opaque data object. The attribute values can be directly manipulated by the application and the DL module. Values stored within the opaque data object must be accessed using parsing functions. A DL module that stores certificates can, but should not, interpret the format of those certificates. A set of parsing functions such as those defined in a certificate library module can be used to parse the opaque certificate object. The DL module defines a default set of parsing functions. An application can define a CSSM module to be used for parsing or can define its own set of parsing functions to be used during a data storage session.

To ensure a minimal level of interoperability among applications and DL modules, CSSM requires that all DL modules recognize and support two pre-defined attribute names for all record types. All applications can use these strings as valid attribute names even if no value is stored in association with this attribute name.

14.2 Data Storage Data Structures

14.2.1 CSSM_DL_HANDLE

A unique identifier for an attached module that provide data storage library services.

```
typedef uint32 CSSM_DL_HANDLE /* data storage library Handle */
```

14.2.2 CSSM_DB_HANDLE

A unique identifier for an open data store.

```
typedef uint32 CSSM_DB_HANDLE /* Data storage Handle */
```

14.2.3 CSSM_DL_DB_HANDLE

This data structure holds a pair of handles, one for a data storage library and another for a data store opened and being managed by the data storage library.

```
typedef struct cssm_dl_db_handle {
    CSSM_DL_HANDLE DLHandle;
    CSSM_DB_HANDLE DBHandle;
} CSSM_DL_DB_HANDLE, *CSSM_DL_DB_HANDLE_PTR;
```

Definition

DLHandle
Handle of an attached module that provides DL services.

DBHandle
Handle of an open data store that is currently under the management of the DL module specifies by the DLHandle.

14.2.4 CSSM_DL_DB_LIST

This data structure defines a list of handle pairs of (data storage library handle, data store handle).

```
typedef struct cssm_dl_db_list {
    uint32 NumHandles;
    CSSM_DL_DB_HANDLE_PTR DLDBHandle;
} CSSM_DL_DB_LIST, *CSSM_DL_DB_LIST_PTR;
```

Definition

NumHandles
Number of DL module and data store pairing in the list.

DLDBHandle
List of data library module and data store pairs.

14.2.5 CSSM_DB_ATTRIBUTE_NAME_FORMAT

This enumerated list defines the two formats used to represent an attribute name. The name can be represented by a character string in the native string encoding of the platform or the name can be represented by an opaque OID structure that is interpreted by the DL module.

```
typedef enum cssm_db_attribute_name_format {
    CSSM_DB_ATTRIBUTE_NAME_AS_STRING = 0,
    CSSM_DB_ATTRIBUTE_NAME_AS_OID = 1,
} CSSM_DB_ATTRIBUTE_NAME_FORMAT, *CSSM_DB_ATTRIBUTE_NAME_FORMAT_PTR;
```

14.2.6 CSSM_DB_ATTRIBUTE_INFO

This data structure describes an attribute of a persistent record. The description is part of the schema information describing the structure of records in a data store. The description includes the format of the attribute name and the attribute name itself. The attribute name implies the underlying data type of a value that may be assigned to that attribute. The attribute name is of one of two formats, not both.

```
typedef struct cssm_db_attribute_info {
    CSSM_DB_ATTRIBUTE_NAME_FORMAT AttributeNameFormat;
    union cssm_db_attribute_label {
        char * AttributeName; /* e.g., "record label" */
        CSSM_OID AttributeID; /* e.g., CSSMOID_RECORDLABEL */
    } Label;
    CSSM_DB_ATTRIBUTE_FORMAT AttributeFormat;
} CSSM_DB_ATTRIBUTE_INFO, *CSSM_DB_ATTRIBUTE_INFO_PTR;
```

Definition

AttributeNameFormat
Indicates which of the two format was selected to represent the attribute name.

AttributeName
A character string representation of the attribute name.

AttributeID
An OID representation of the attribute name.

AttributeFormat
Indicates the format of the attribute.The Data Storage Library may not support more than one format, typically CSSM_DB_ATTRIBUTE_FORMAT_STRING. In this case, the library module can ignore any format specification provided by the caller.

14.2.7 CSSM_DB_ATTRIBUTE_DATA

This data structure holds an attribute value that can be stored in an attribute field of a persistent record. The structure contains a value for the data item and a reference to the meta information (typing information and schema information) associated with the attribute.

```
typedef struct cssm_db_attribute_data {
    CSSM_DB_ATTRIBUTE_INFO Info;
    CSSM_DATA Value;
} CSSM_DB_ATTRIBUTE_DATA, *CSSM_DB_ATTRIBUTE_DATA_PTR;
```

Definition

Info
A reference to the meta-information/schema describing this attribute in relationship to the data store at large.

Value
The data-present value assigned to the attribute.

To ensure a minimal level of interoperability among applications and DL modules, CSSM requires that all DL modules recognize and support two pre-defined attribute names for all record types:

- PrintName: a printable or viewable string name associated with the record.
- Alias: an arbitrary value associated with the record. The value can be non-printable.

Applications that create new data stores and define the associated schema are encouraged to define these attributes as part of the schema. If the data store creator does not define these attributes, the DL module must add these attributes with the following minimum storage size requirements:

- PrintName: the associated value is a string of maximum length 16 characters.
- Alias: the associated value is an arbitrary data type of maximum length 8 bytes.

Applications are encouraged to provide values for these attributes when creating data store records, but values are not required. All applications can use these strings as valid attribute names even if no value is stored in association with this attribute name. When no value is associated with a pre-defined attribute name, it is possible for a DL module that encapsulates a data store schema to return one of the following:

- A module-defined default value
- A value selected from a database-key attribute in the data store
- A NULL value

The CSSM_DB_ATTRIBUTE_DATA structure for the pre-defined attribute name "PrintName" contains the following values:

```
{  AttributeNameFormat = CSSM_DB_ATTRIBUTE_NAME_AS_STRING
    AttributeName = "PrintName"
    Value = <a value in a CSSM_DATA structure>  }
```

14.2.8 CSSM_DB_RECORDTYPE

This enumerated list defines the categories of persistent security-related objects that can be managed by a data storage library module. These categories are in one-to-one correspondence with types of records that can be managed by a data store.

```
typedef enum cssm_db_recordtype {
    CSSM_DL_DB_RECORD_GENERIC = 0,
    CSSM_DL_DB_RECORD_CERT = 1,
    CSSM_DL_DB_RECORD_CRL = 2,
    CSSM_DL_DB_RECORD_KEY = 3,
    CSSM_DL_DB_RECORD_POLICY = 4,
} CSSM_DB_RECORDTYPE;
```

14.2.9 CSSM_DB_CERTRECORD_SEMANTICS

These bit masks define a list of usage semantics for how certificates may be used. It is anticipated that additional sets of bit masks will be defined listing the usage semantics of how other record types can be used, such as CRL record semantics, key record semantics, policy record semantics, and so on.

```
#define CSSM_DB_CERT_USE_TRUSTED 0x00000001 /* application-defined
                                        as trusted */
#define CSSM_DB_CERT_USE_SYSTEM  0x00000002 /* the CSSM system
                                        cert */
#define CSSM_DB_CERT_USE_OWNER   0x00000004 /* private key owned
                                        by system user*/
#define CSSM_DB_CERT_USE_REVOKED 0x00000008 /* revoked cert -
                                        used w CRL APIs */
#define CSSM_DB_CERT_USE_SIGNING 0x00000010 /* use cert for
                                        signing only */
#define CSSM_DB_CERT_USE_PRIVACY 0x00000020 /* use cert for
                                        confidentiality only */
```

Record semantic designations are advisory only. For example, the designation CSSM_DB_CERT_USE_OWNER suggests that the private key associated with the public key contained in the certificate is local to the system. This statement was probably true when the certificate was created. Various actions could make this assertion false. The private key could have expired, been revoked, or be stored in a portable cryptographic storage device that is not currently resident on the system. The validity of the advisory designation CSSM_DB_CERT_USE_TRUSTED should be verified using standard certificate verification procedures. Although these designators are advisory, application or trust policies can choose to use this information if it is useful for their purpose. For example, a trust policy can define how advisory designations can be used when full policy evaluation requires connection to a remote facility that is currently inaccessible.

Management practices for record semantic designators define the agent and the time when a data store record can be assigned a particular designator value. Reasonable usage is described as follows:

Designation Value	**Assigning Time**	**Assigning Agents**
CSSM_DB_CERT_USE_TRUSTED	Local record creation time Remote record creation time Reset at any time	Sys Admin App App/Record Owner
CSSM_DB_CERT_USE_SYSTEM	Local record creation time Should not be reset	Sys Admin App
CSSM_DB_CERT_USE_OWNER	Local record creation time Reset at any time	App/Record Owner
CSSM_DB_CERT_USE_REVOKED	Set once only	System Administrator App Application/Record Owner

CSSM_DB_CERT_SIGNING	Local record creation time	Remote Authority Local Authority Record Owner
CSSM_DB_CERT_PRIVACY	Local record creation time	Remote Authority Local Authority Record Owner

14.2.10 CSSM_DB_RECORD_ATTRIBUTE_INFO

This structure contains the meta information or schema information about all of the attributes in a particular record type. The description specifies the record type, the number of attributes in the record type, and a type information for each attribute. This description includes the CSSM pre-defined attributes named "PrintName" and "Alias".

```
typedef struct cssm_db_record_attribute_info {
    CSSM_DB_RECORDTYPE DataRecordType;
    uint32 NumberOfAttributes;
    CSSM_DB_ATTRIBUTE_INFO_PTR AttributeInfo;
} CSSM_DB_RECORD_ATTRIBUTE_INFO, *CSSM_DB_RECORD_ATTRIBUTE_INFO_PTR;
```

Definition

DataRecordType
A CSSM_DB_RECORDTYPE.

NumberOfAttributes
The number of attributes in a record of the specified type.

AttributeInfo
A list of pointers to the type (schema) information for each of the attributes.

14.2.11 CSSM_DB_RECORD_ATTRIBUTE_DATA

This structure aggregates the actual data values for all of the attributes in a single record. The structure includes the record type, optional semantic information on how the record can and cannot be used, the number of attributes in the records, and the actual data value for each attribute.

```
typedef struct cssm_db_record_attribute_data {
    CSSM_DB_RECORDTYPE DataRecordType;
    uint32 SemanticInformation;
    uint32 NumberOfAttributes;
    CSSM_DB_ATTRIBUTE_DATA_PTR AttributeData;
} CSSM_DB_RECORD_ATTRIBUTE_DATA, *CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR;
```

Definition

DataRecordType
A CSSM_DB_RECORDTYPE.

SemanticInformation
A bit mask of type CSSM_XXXRECORD_SEMANTICS defining how the record can be used. Currently these bit masks are defined only for CSSM_CERTRECORD_SEMANTICS. For all other records types, a bit masks of zero must be used or a set of semantically meaning masks must be defined.

NumberOfAttributes
The number of attributes in a record of the specified type.

AttributeData
A list of pointers to data values, one per attribute. If no stored value is associated with this attribute, the attribute data pointer is NULL.

14.2.12 CSSM_DB_RECORD_PARSING_FNTABLE

This structure defines the three prototypes for functions that can parse the opaque object stored in a record. The functions can parse the opaque objects in some or all of the distinct record types stored in the data store. Record types not supported by the data store need not be supported by the parsing functions. The DL module must designate a default parsing module for each record type stored in the data store. The default parsing module can parse multiple record types. The function CSSM_DbSetRecordParsingFunctions must be used to override the default parsing module each applicable record type.

```
typedef struct cssm_db_record_parsing_fntable {
    CSSM_DATA_PTR (CSSMAPI *RecordGetFirstFieldValue)
       (CSSM_HANDLE Handle,
        const CSSM_DATA_PTR Data,
        const CSSM_OID_PTR DataField,
        CSSM_HANDLE_PTR ResultsHandle,
        uint32 *NumberOfMatchedFields);
    CSSM_DATA_PTR (CSSMAPI *RecordGetNextFieldValue)
       (CSSM_HANDLE Handle,
        CSSM_HANDLE ResultsHandle);
    CSSM_RETURN (CSSMAPI *RecordAbortQuery)
       (CSSM_HANDLE Handle,
        CSSM_HANDLE ResultsHandle);
} CSSM_DB_RECORD_PARSING_FNTABLE,  *CSSM_DB_RECORD_PARSING_FNTABLE_PTR;
```

Definition

**RecordGetFirstFieldValue*
A function to retrieve a value from a field in the opaque object. The field is specified by attribute name. The results handle holds the state information required to retrieve subsequent values having the same attribute name.

**RecordGetNextFieldValue*
A function to retrieve subsequent values having the same attribute name from a record parsed by the first function in this table.

**RecordAbortQuery*
Stop subsequent retrievals of values having the same attribute name from within an opaque

object in a CSSM record.

14.2.13 CSSM_DB_PARSING_MODULE_INFO

This structure aggregates the persistent subservice ID of a default parsing module with the record type that it parses. A parsing module can parse multiple records types. The same ID would be repeated with each record type parsed by the module.

```
typedef cssm_db_parsing_module_info {
    CSSM_DB_RECORDTYPE RecordType;
    CSSM_SUBSERVICE_UID ModuleSubserviceUid;
} CSSM_DB_PARSING_MODULE_INFO, *CSSM_DB_PARSING_MODULE_INFO_PTR;
```

Definition

RecordType
The type of record parsed by the module specified by GUID.

ModuleSubserviceUid
A persistent subservice ID identifying the default parsing module for the specified record type.

14.2.14 CSSM_DB_INDEX_TYPE

This enumerated list defines two types of indexes: indexes with unique values (such as, primary database keys) and indexes with non-unique values. These values are used when creating a new data store and defining the schema for that data store.

```
typedef enum cssm_db_index_type {
    CSSM_DB_INDEX_UNIQUE = 0,
    CSSM_DB_INDEX_NONUNIQUE = 1
} CSSM_DB_INDEX_TYPE;
```

14.2.15 CSSM_DB_INDEXED_DATA_LOCATION

This enumerated list defines where within a CSSM record the indexed data values reside. Indexes can be constructed on attributes or on fields within the opaque object in the record. However, the logical location of the index value between these two categories may be unknown by the user of this enumeration.

```
typedef enum cssm_db_indexed_data_location {
    CSSM_DB_INDEX_ON_UNKNOWN = 0,
    CSSM_DB_INDEX_ON_ATTRIBUTE = 1,
    CSSM_DB_INDEX_ON_RECORD = 2
} CSSM_DB_INDEXED_DATA_LOCATION;
```

14.2.16 CSSM_DB_INDEX_INFO

This structure contains the meta information or schema description of an index defined on an attribute. The description includes the type of index (for example, unique key or non-unique key), the logical location of the indexed attribute in the CSSM record (for example, an attribute or a field within the opaque object in the record), and the meta information on the attribute itself.

```
typedef struct cssm_db_index_info {
    CSSM_DB_INDEX_TYPE IndexType;
    CSSM_DB_INDEXED_DATA_LOCATION IndexedDataLocation;
    CSSM_DB_ATTRIBUTE_INFO Info;
} CSSM_DB_INDEX_INFO, *CSSM_DB_INDEX_INFO_PTR;
```

Definition

IndexType
A CSSM_DB_INDEX_TYPE.

IndexedDataLocation
A CSSM_DB_INDEXED_DATA_LOCATION.

Info
The meta information description of the attribute being indexed.

14.2.17 CSSM_DB_UNIQUE_RECORD

This structure contains an index descriptor and a module-defined value. The index descriptor may be used by the module to enhance the performance when locating the record. The module-defined value must uniquely identify the record. For a DBMS, this may be the record data. For a PKCS #11 DL, this may be an object handle. Alternately, the DL may have a module-specific scheme for identifying data which has been inserted or retrieved.

```
typedef struct cssm_db_unique_record {
    CSSM_DB_INDEX_INFO RecordLocator;
    CSSM_DATA RecordIdentifier;
} CSSM_DB_UNIQUE_RECORD, *CSSM_DB_UNIQUE_RECORD_PTR;
```

Definition

RecordLocator
The information describing how to locate the record efficiently.

RecordIdentifier
A module-specific identifier which will allow the DL to locate this record.

14.2.18 CSSM_DB_RECORD_INDEX_INFO

This structure contains the meta information or schema description of the set of indexes defined on a single record type. The description includes the type of the record, the number of indexes and the meta information describing each index. The data store creator can specify an index over a CSSM pre-defined attribute. When no index has been defined, the DL module has the option to add an index over a CSSM pre-defined attribute or any other attribute defined by the data store creator.

```
typedef struct cssm_db_record_index_info {
    CSSM_DB_RECORDTYPE DataRecordType;
    uint32 NumberOfIndexes;
    CSSM_DB_INDEX_INFO_PTR IndexInfo;
} CSSM_DB_RECORD_INDEX_INFO, *CSSM_DB_RECORD_INDEX_INFO_PTR;
```

Definition

DataRecordType
A CSSM_DB_RECORDTYPE.

NumberOfIndexes
The number of indexes defined on the record of the given type.

IndexInfo
An array containing a description of each index defined over the specified record type.

14.2.19 CSSM_DB_ACCESS_TYPE

This bitmask describes a user's desired level of access to a data store.

```
typedef uint32 CSSM_DB_ACCESS_TYPE, *CSSM_DB_ACCESS_TYPE_PTR;

#define CSSM_DB_ACCESS_READ 0x00001
#define CSSM_DB_ACCESS_WRITE 0x00002
#define CSSM_DB_ACCESS_PRIVILEGED 0x00004 /* versus user mode */
#define CSSM_DB_ACCESS_ASYNCHRONOUS 0x00008 /* versus
                                        synchronous */
```

14.2.20 CSSM_DBINFO

This structure contains the meta-information about an entire data store. The description includes the types of records stored in the data store, the attribute schema for each record type, the index schema for all indexes over records in the data store, the type of authentication mechanism used to gain access to the data store, and other miscellaneous information used by the DL module to manage the data store.

```
typedef struct cssm_dbinfo {
      /* meta information about each record type stored in this
      data store including meta information about record
      attributes and indexes */
    uint32 NumberOfRecordTypes;
    CSSM_DB_PARSING_MODULE_INFO_PTR DefaultParsingModules;
    CSSM_DB_RECORD_ATTRIBUTE_INFO_PTR RecordAttributeNames;
    CSSM_DB_RECORD_INDEX_INFO_PTR RecordIndexes;

       /* access restrictions for opening this data store */
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;

 /* transparent integrity checking options for this data store */
    CSSM_BOOL RecordSigningImplemented;
    CSSM_DATA SigningCertificate;
    CSSM_SUBSERVICE_UID SigningCspSubserviceUid;
                                 /* additional information */
    CSSM_BOOL IsLocal;
    char *AccessPath; /* URL, dir path, etc. */
    void *Reserved;
} CSSM_DBINFO, *CSSM_DBINFO_PTR;
```

Definition

NumberOfRecordTypes
The number of distinct record types stored in this data store.

DefaultParsingModules
A pointer to a list of GUID-record-type pairs, defining the default parsing module for each record type.

RecordAttributeNames
The meta (schema) information about the attributes in each of the record types that can be stored in this data store.

RecordIndexes
The meta (schema) information about the indexes that are defined over each of the record types that can be stored in this data store.

AuthenticationMechanism
Defines the authentication mechanism required when accessing this data store.

RecordSigningImplemented
A flag indicating whether or not the DL module provides record integrity service based on digital signaturing of the data store records.

SigningCertificate
The certificate used to sign data store records when the transparent record integrity option is in effect.

SigningCspSubserviceUid
The persistent subservice ID for the cryptographic service provider to be used to sign data store records when the transparent record integrity option is in effect.

IsLocal
Indicates whether the physical data store is local.

AccessPath
A character string describing the access path to the data store, such as an URL, a file system path name, a remote directory service name, and so on.

Reserved
Reserved for future use.

14.2.21 CSSM_DB_OPERATOR

These are the logical operators which can be used when specifying a selection predicate.

```
typedef enum cssm_db_operator {
    CSSM_DB_EQUAL = 0,
    CSSM_DB_NOT_EQUAL = 1,
    CSSM_DB_APPROX_EQUAL = 2,
    CSSM_DB_LESS_THAN = 3,
    CSSM_DB_GREATER_THAN = 4,
    CSSM_DB_EQUALS_INITIAL_SUBSTRING = 5,
    CSSM_DB_EQUALS_ANY_SUBSTRING = 6,
    CSSM_DB_EQUALS_FINAL_SUBSTRING = 7,
    CSSM_DB_EXISTS = 8
} CSSM_DB_OPERATOR, *CSSM_DB_OPERATOR_PTR;
```

14.2.22 CSSM_DB_CONJUNCTIVE

These are the conjunctive operations which can be used when specifying a selection criterion.

```
typedef enum cssm_db_conjunctive{
    CSSM_DB_NONE = 0,
    CSSM_DB_AND = 1,
    CSSM_DB_OR = 2
} CSSM_DB_CONJUNCTIVE, *CSSM_DB_CONJUNCTIVE_PTR;
```

14.2.23 CSSM_SELECTION_PREDICATE

This structure defines the selection predicate to be used for data store queries.

```
typedef struct cssm_selection_predicate {
    CSSM_DB_OPERATOR DbOperator;
    CSSM_DB_ATTRIBUTE_DATA Attribute;
} CSSM_SELECTION_PREDICATE, *CSSM_SELECTION_PREDICATE_PTR;
```

Definition

DbOperator
The relational operator to be used when comparing a value to the values stored in the specified attribute in the data store.

Attribute
The meta information about the attribute to be searched and the attribute value to be used for comparison with values in the data store.

14.2.24 CSSM_QUERY_LIMITS

This structure defines the time and space limits a caller can set to control early termination of the execution of a data store query. The constant values CSSM_QUERY_TIMELIMIT_NONE and CSM_QUERY_SIZELIMIT_NONE should be used to specify no limit on the resources used in processing the query. These limits are advisory. Not all data storage library modules recognize and act upon the query limits set by a caller.

```
#define CSSM_QUERY_TIMELIMIT_NONE 0
#define CSSM_QUERY_SIZELIMIT_NONE 0

typedef struct cssm_query_limits {
    uint32 TimeLimit; /* in seconds */
    uint32 SizeLimit; /* max. number of records to return */
} CSSM_QUERY_LIMITS, *CSSM_QUERY_LIMITS_PTR;
```

Definition

TimeLimit
Defines the maximum number of seconds of resource time that should be expended performing a query operation. The constant value CSSM_QUERY_TIMELIMIT_NONE means no time limit is specified. All specific time values must be greater than zero, as any query requires greater than zero time to execute.

SizeLimit
Defines the maximum number of records that should be retrieved in response to a single query. The constant value CSSM_QUERY_SIZELIMIT_NONE means no space limit is

specified. All specific space values must be greater than zero, as any query requires greater than zero space in which to execute.

14.2.25 CSSM_QUERY_FLAGS

These flags may be used by the application to request query-related operation, such as the format of the returned data.

```
typedef uint32 CSSM_QUERY_FLAGS

#define CSSM_QUERY_RETURN_DATA 0x1 /* On = Return the data record
                    Off = Return a reference to the data record*/
```

14.2.26 CSSM_QUERY

This structure holds a complete specification of a query to select records from a data store.

```
typedef struct cssm_query {
    CSSM_DB_RECORDTYPE RecordType;
    CSSM_DB_CONJUNCTIVE Conjunctive;
    uint32 NumSelectionPredicates;
    CSSM_SELECTION_PREDICATE_PTR SelectionPredicate;
    CSSM_QUERY_LIMITS QueryLimits;
    CSSM_QUERY_FLAGS QueryFlags;
} CSSM_QUERY, *CSSM_QUERY_PTR;
```

Definition

RecordType
Specifies the type of record to be retrieved from the data store.

Conjunctive
The conjunctive operator to be used in constructing the selection predicate for the query.

NumSelectionPredicates
The number of selection predicates to be connected by the specified conjunctive operator to form the query.

SelectionPredicate
The list of selection predicates to be combined by the conjunctive operator to form the data store query.

QueryLimits
Defines the time and space limits for processing the selection query. The constant values CSSM_QUERY_TIMELIMIT_NONE and CSM_QUERY_SIZELIMIT_NONE should be used to specify no limit on the resources used in processing the query.

QueryFlags
Query-related requests from the application.

14.2.27 CSSM_DLTYPE

This enumerated list defines the types of underlying data management systems that can be used by the DL module to provide services. It is the option of the DL module to disclose this information. It is anticipated that other underlying data servers will be added to this list over time.

```
typedef enum cssm_dltype {
    CSSM_DL_UNKNOWN = 0,
    CSSM_DL_CUSTOM = 1,
    CSSM_DL_LDAP = 2,
    CSSM_DL_ODBC = 3,
    CSSM_DL_PKCS11 = 4,
    CSSM_DL_FFS = 5, /* flat file system or fast file system */
    CSSM_DL_MEMORY = 6,
    CSSM_DL_REMOTEDIR = 7
} CSSM_DLTYPE, *CSSM_DLTYPE_PTR;
```

14.2.28 CSSM_DL_PKCS11_ATTRIBUTES

Each type of DL module can define it own set of type specific attributes. This structure contains the attributes that are specific to a PKCS#11 compliant data storage device.

```
typedef void *CSSM_DL_CUSTOM_ATTRIBUTES;
typedef void *CSSM_DL_LDAP_ATTRIBUTES;
typedef void *CSSM_DL_ODBC_ATTRIBUTES;
typedef void *CSSM_DL_FFS_ATTRIBUTES;

typedef struct cssm_dl_pkcs11_attributes {
    uint32 DeviceAccessFlags;
} *CSSM_DL_PKCS11_ATTRIBUTE, *CSSM_DL_PKCS11_ATTRIBUTE_PTR;
```

Definition

DeviceAccessFlags
Specifies the PKCS#11-specific access modes applicable for accessing persistent objects in the PKCS#11 data store.

14.2.29 CSSM_DB_DATASTORES_UNKNOWN

Not all DL modules can maintain a summary of managed data stores. In this case, the DL module reports its number of data stores as CSSM_DB_DATASTORES_UNKNOWN. Data stores can (and probably do) exist, but the DL module cannot provide a list of them.

```
#define CSSM_DB_DATASTORES_UNKNOWN (0xFFFFFFFF)
```

14.2.30 CSSM_DL_WRAPPEDPRODUCT_INFO

This structure holds product information about all backend data base services used by the DL module. The DL module vendor is not required to provide this information, but may choose to do so.

```
typedef struct cssm_dl_wrappedproductinfo
    CSSM_VERSION StandardVersion; /* Ver of standard the product
                                    conforms to */
    CSSM_STRING StandardDescription; /* Descr of standard the
                                    product conforms to */
    CSSM_VERSION ProductVersion; /* Version of wrapped product or
                                    library */
    CSSM_STRING ProductDescription; /* Description of wrapped
                                    product or library */
    CSSM_STRING ProductVendor; /* Vendor of wrapped product or
                                    library */
    CSSM_NET_PROTOCOL NetworkProtocol; /* The network protocol
                         supported by a remote storage service */
    uint32 ProductFlags; /* Mask of selectable DB service
                              features actually used by the DL */
} CSSM_DL_WRAPPEDPRODUCT_INFO, *CSSM_DL_WRAPPEDPRODUCT_INFO_PTR
```

Definition

StandardVersion
If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription
If this product conforms to an industry standard, this is a description of that standard.

ProductVersion
Version number information for the actual product version used in this version of the DL module.

ProductDescription
A string describing the product.

ProductVendor
The name of the product vendor.

NetworkProtocol
The name of the network protocol supported by a remote storage service.

ProductFlags
A bit mask enumerating selectable features of the data base service that the DL module uses in its implementation.

14.2.31 CSSM_NAME_LIST

The CSSM_NAME_LIST structure is used to return the logical names of the data stores that a DL module can access.

```
typedef struct cssm_name_list {
    uint32 NumStrings;
    char** String;
} CSSM_NAME_LIST, *CSSM_NAME_LIST_PTR;
```

Definition

NumStrings
Number of strings in the array pointed to by String.

String
A pointer to an array of strings.

14.2.32 CSSM_DLSUBSERVICE

This structure contains the static information that describes a data storage library sub-service. This information is stored in the CSSM registry when the DL module is installed with CSSM. CSSM checks the integrity of the DL module description before using the information. A data storage library module may implement multiple types of services and organize them as sub-services.

The descriptive information stored in these structures can be queried using the function *CSSM_GetModuleInfo*( ) and specifying the data storage library module GUID.

```
typedef struct cssm_dlsubservice {
    uint32 SubServiceId;
    CSSM_STRING Description;
    CSSM_DLTYPE Type;
    union cssm_dlsubservice_attributes {
        CSSM_DL_CUSTOM_ATTRIBUTES CustomAttributes;
        CSSM_DL_LDAP_ATTRIBUTES LdapAttributes;
        CSSM_DL_ODBC_ATTRIBUTES OdbcAttributes;
        CSSM_DL_PKCS11_ATTRIBUTES Pkcs11Attributes;
        CSSM_DL_FFS_ATTRIBUTES FfsAttributes;
    } Attributes;

    CSSM_DL_WRAPPEDPRODUCTINFO WrappedProduct;
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;

    /* meta information about the query support provided by the
                                    module */
    uint32 NumberOfRelOperatorTypes;
    CSSM_DB_OPERATOR_PTR RelOperatorTypes;
    uint32 NumberOfConjOperatorTypes;
    CSSM_DB_CONJUNCTIVE_PTR ConjOperatorTypes;
    CSSM_BOOL QueryLimitsSupported;

    /* meta information about the encapsulated data
                                        stores (if known) */
    sint32 NumberOfDataStores;
    CSSM_NAME_LIST_PTR DataStoreNames;
    CSSM_DBINFO_PTR DataStoreInfo;

    /* additional information */
    void *Reserved;
} CSSM_DLSUBSERVICE, *CSSM_DLSUBSERVICE_PTR;
```

Definition

SubServiceId
A unique, identifying number for the sub-service described in this structure.

Description
A string containing a description name or title for this sub-service.

Type
An identifier for the type of underlying data server the DL module uses to provide persistent storage.

Attributes
A structure containing attributes that define additional parameter values specific to the DL module type.

WrappedProduct
Descriptions of the backend data store services used by this module.

AuthenticationMechanism
Defines the authentication mechanism required when using this DL module. This authentication mechanism is distinct from the authentication mechanism (specified in a DBInfo structure) required to access a specific data store.

NumberOfRelOperatorTypes
The number of distinct relational operators the DL module accepts in selection queries for retrieving records from its managed data stores.

RelOperatorTypes
The list of specific relational operators that can be used to formulate selection predicates for queries on a data store. The list contains NumberOfRelOperatorTypes operators.

NumberOfConjOperatorTypes
The number of distinct conjunctive operator the DL module accepts in selection queries for retrieving records from its managed data stores.

ConjOperatorTypes
A list of specific conjunctive operators that can be used to formulate selection predicates for queries on a data store. The list contains NumberOfConjOperatorTypes operators.

QueryLimitsSupported
A Boolean indicating whether query limits are effective when the DL module executes a query.

NumberOfDataStores
The number of data stores managed by the DL module. This information may not be known by the DL module and hence may not be available.

DataStoreNames
A list of names of the data stores managed by the DL module. This information may not be known by the DL module and hence may not be available. The list contains

NumberOfDataStores entries .

DataStoreInfo
A list of pointers information about each data store managed by the DL module. This information may not be known in advance by the DL module and hence may not be available through this structure. The list contains NumberOfDataStores entries.

Reserved
Reserved for future use.

14.3 Data Storage Functions

The manpages for Data Storage Functions follow on the next page.

NAME

CSSM_DL_DbOpen

SYNOPSIS

```
CSSM_DB_HANDLE CSSMAPI CSSM_DL_DbOpen
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS_PTR DbLocation,
    const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    const void *OpenParameters))
```

DESCRIPTION

This function opens the data store with the specified logical name under the specified access mode. If no DbName is provided, the default data store will be opened. If user authentication credentials are required, they must be provided. Also, additional open parameters may be required to open a given data store, and are supplied in the OpenParameters.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbName (input)
A pointer to the string containing the logical name of the data store.

DbLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbName (for existing data stores) or can assume a default storage service process location. If the DbName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

AccessRequest (input)
An indicator of the requested access mode for the data store, such as read-only or read-write.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

RETURN VALUE

The handle to the opened data store. If the handle is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_DATASTORE_NOT_EXISTS
The data store with the logical name does not exist.

CSSM_DL_INVALID_AUTHENTICATION
Caller is not authorized for specified access mode.

CSSM_DL_DB_OPEN_FAIL
Open caused an exception.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

CSSM_DL_DbClose

NAME

CSSM_DL_DbClose

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_DbClose
    (CSSM_DL_DB_HANDLE DLDBHandle)
```

DESCRIPTION

This function closes an open data store.

PARAMETERS

DLDBHandle (input)
A handle structure containing the DL handle for the attached DL module and the DB handle for an open data store managed by the DL. This specifies the open data store to be closed.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_DB_CLOSE_FAIL
Close caused an exception.

SEE ALSO

CSSM_DL_DbOpen

NAME

CSSM_DL_DbCreate

SYNOPSIS

```
CSSM_DB_HANDLE CSSMAPI CSSM_DL_DbCreate
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS_PTR DbLocation,
    const CSSM_DBINFO_PTR DBInfo,
    const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    const void *OpenParameters)
```

DESCRIPTION

This function creates and opens a new data store. The name of the new data store is specified by the input parameter DbName. The record schema for the data store is specified in the DBINFO structure. The newly-created data store is opened under the specified access mode. If user authentication credentials are required, they must be provided. Also, additional open parameters may be required and are supplied in the OpenParameters.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module used to perform this function.

DbName (input)
The logical name for the new data store.

DbLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbName (for existing data stores) or can assume a default storage service process location. If the DbName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

DBInfo (input)
A pointer to a structure describing the format/schema of each record type that will be stored in the new data store. If the schema definition does not specify the CSSM pre-defined attribute name "PrintName" and "Alias", these attributes are added by the DL module with the minimum associated storage size.

AccessRequest (input)
An indicator of the requested access mode for the data store, such as read-only or read-write.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

RETURN VALUE

The handle to the newly created and open data store. When NULL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
: Invalid DL handle.

CSSM_DL_INVALID_AUTHENTICATION
: Caller is not authorized for the operation.

CSSM_DL_INVALID_DBINFO
: Invalid meta information for the schema.

CSSM_DL_DB_CREATE_FAIL
: Create caused an exception.

CSSM_REGISTRY_ERROR
: Unable to add-update registry entry.

CSSM_DL_INVALID_CSP_HANDLE
: Invalid default CSP handle (integrity signing).

CSSM_DL_MEMORY_ERROR
: Error in allocating memory.

SEE ALSO

CSSM_DL_DbOpen, *CSSM_DL_DbClose*, *CSSM_DL_DbDelete*

NAME

CSSM_DL_DbDelete

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_DbDelete
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS_PTR DbLocation,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication))
```

DESCRIPTION

This function deletes all records from the specified data store and removes all state information associated with that data store.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbName (input)
A pointer to the string containing the logical name of the data store.

DbLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbName (for existing data stores) or can assume a default storage service process location. If the DbName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access (and consequently deletion capability) to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_INVALID_AUTHENTICATION
Caller is not authorized for operation.

CSSM_REGISTRY_ERROR
Unable to update registry entry.

CSSM_DL_DB_DELETE_FAIL
Delete caused an exception.

SEE ALSO

CSSM_DL_DbCreate, *CSSM_DL_DbOpen*, *CSSM_DL_DbClose*

NAME

CSSM_DL_DbImport

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_DbImport
    (CSSM_DL_HANDLE DLHandle,
    const char *DbDestinationName,
    const CSSM_NET_ADDRESS_PTR DbDestinationLocation,
    const char *DbSourceName,
    const CSSM_NET_ADDRESS_PTR DbSourceLocation,
    const CSSM_DBINFO_PTR DBInfo,
    const CSSM_BOOL InfoOnly,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication)
    const void *DestinationOpenParameters,
    const void *SourceOpenParameters)
```

DESCRIPTION

This function makes the contents of the source data store available from the destination data source. This may involve registering the source data store with this DL module or the transfer of records from the source to the destination.

If INFO_ONLY is TRUE, information about an existing data store is registered with the DL module but no records are imported. The DL module will update the CSSM registry with the DbDestinationName and DBInfo to inform applications that this data store is available. This method may be used to make existing data stores available via the CSSM DL interface.

If INFO_ONLY is FALSE, this function creates a new data store, or adds to an existing data store, by importing records from the specified data source. It is assumed that the data source contains records exported from a data store using the function *CSSM_DL_DbExport*.

The DbDestinationName specifies the name of a new or existing data store. If a new data store is being created, the DBInfo structure provides the meta information (schema) for the new data store. This structure describes the record attributes and the index schema for the new data store. If the data store already exists, then the existing meta information (schema) is used. (Dynamic schema evolution is not supported.)

Typically, user authentication is required to create a new data store or to write to an existing data store. An authentication credential is presented to the DL module in the form required by the module. The required form is documented in the capabilities and feature descriptions for this module. The resulting data store is not opened as a result of this operation.

PARAMETERS

DLHandle (input)
: The handle that describes the add-in data storage library module to be used to perform this function.

DbDestinationName (input)
: The name of the data store which will contain the imported records.

DbDestinationLocation (input/optional)
: A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbDestinationName or can assume a default storage service process location. If the DbDestinationName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

DbSourceName (input)
The name of the data source from which to obtain the records to be imported.

DbSourceLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbSourceName or can assume a default storage service process location. If the DbSourceName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

DBInfo (input/optional)
A data structure containing a detailed description of the meta information (schema) for the new data store. If a new data store is being created, then the caller must specify the meta information (schema), or the data source must include the meta information required for proper import of the records. If meta information is supplied by the caller and specified in the data source, then the meta information provided by the caller overrides the meta information recorded in the data source. If the data store exists and records are being added, then this pointer must be NULL. The existing meta information will be used and the schema cannot be evolved.

InfoOnly (input)
A Boolean value indicating what to import. If TRUE, import only the DBInfo, which describes the a data store. If FALSE, import both the DBInfo and all of the records exported from a data store.

UserAuthentication (input/optional)
The caller's credential as required for authorization to create a data store. If the DL module requires no additional credentials to create a new data store, then user authentication can be NULL.

DestinationOpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the destination data store.

SourceOpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the source data store.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully and the new data store was created. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_PTR
NULL source or destination names.

CSSM_REGISTRY_ERROR
Unable to add/update registry entry.

CSSM_DL_DB_IMPORT_FAIL
DB exception doing import function.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

CSSM_DL_DbExport

NAME

CSSM_DL_DbExport

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_DbExport
    (CSSM_DL_HANDLE DLHandle,
    const char *DbDestinationName,
    const CSSM_NET_ADDRESS_PTR DbDestinationLocation,
    const char *DbSourceName,
    const CSSM_NET_ADDRESS_PTR DbSourceLocation,
    const CSSM_BOOL InfoOnly,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication)
    const void *DestinationOpenParameters,
    const void *SourceOpenParameters)
```

DESCRIPTION

This function exports a copy of the data store records from the source data store to a data container that can be used as the input data source for the *CSSM_DL_DbImport* function. The DL module may require additional user authentication to determine authorization to snapshot a copy of an existing data store.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbDestinationName (input)
The name of the destination data container to contain a copy of the source data store's records.

DbDestinationLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbDestinationName or can assume a default storage service process location. If the DbDestinationName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

DbSourceName (input)
The name of the data store from which the records are to be exported.

DbSourceLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbSourceName or can assume a default storage service process location. If the DbSourceName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

InfoOnly (input)
A Boolean value indicating what to export. If TRUE, export only the DBInfo, which describes the a data store. If FALSE, export both the DBInfo and all of the records in the specified data store.

UserAuthentication (input/optional)
The caller's credential as required for authorization to snapshot/copy a data store. If the DL module requires no additional credentials to perform this operation, then user authentication can be NULL.

DestinationOpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the destination data store.

SourceOpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the source data store.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_PTR
NULL source or destination names.

CSSM_DL_DB_EXPORT_FAIL
DB exception doing export function.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

CSSM_DL_DbImport

NAME

CSSM_DL_Authenticate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_Authenticate
    (const CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication)
```

DESCRIPTION

This function allows the caller to provide authentication credentials to the DL module at a time other than data store creation, deletion, open, import, and export. AccessRequest defines the type of access to be associated with the caller. If the authentication credential applies to access and use of a DL module in general, then the data store handle specified in the DLDBHandle must be NULL. When the authorization credential is to apply to a specific data store, the handle for that data store must be specified in the DLDBHandle pair.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module used to perform this function and the data store to which access is being requested. If the form of authentication being requested is authentication to the DL module in general, then the data store handle must be NULL.

AccessRequest (input)
An indicator of the requested access mode for the data store or DL module in general.

UserAuthentication (input)
The caller's credential as required for obtaining authorized access to the data store or to the DL module in general.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_INVALID_ACCESS_MODE
Unrecognized access type.

CSSM_INVALID_AUTHENTICATION
Unrecognized or invalid authentication credential.

NAME

CSSM_DL_DbSetRecordParsingFunctions

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_DbSetRecordParsingFunctions
    (CSSM_DL_HANDLE DLHandle,
    const char* DbName,
    const CSSM_DB_RECORDTYPE RecordType,
    const CSSM_DB_RECORD_PARSING_FNTABLE_PTR FunctionTable)
```

DESCRIPTION

This function sets the records parsing function table, overriding the default parsing module, for records of the specified type, in the specified data store. Three record parsing functions can be specified in the table. The functions can be implemented to parse multiple record types. In this case, multiple calls to DbSetRecordParsingFunctions must be made, once for each record type that should be parsed using these functions. The DL module uses these functions to parse the opaque data object stored in a data store record. If no parsing function table has been set for a given record type, then the default parsing module is invoked for that record type.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbName (input)
The name of the data store with which to associate the parsing functions.

RecordType (input)
One of the record types parsed by the functions specified in the function table.

FunctionTable (input)
The function table referencing the three parsing functions to be used with the data store specified by DbName.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_NAME
Invalid DB name.

CSSM_DL_MEMORY_ERROR
Error allocating memory.

SEE ALSO

CSSM_DL_GetRecordParsingFunctions

NAME

CSSM_DL_DbGetRecordParsingFunctions

SYNOPSIS

```
CSSM_DB_RECORD_PARSING_FNTABLE_PTR CSSMAPI
CSSM_DL_DbGetRecordParsingFunctions
    (CSSM_DL_HANDLE DLHandle,
    const char* DbName,
    const CSSM_DB_RECORDTYPE RecordType)
```

DESCRIPTION

This function gets the records parsing function table, that operates on records of the specified type, in the specified data store. Three record parsing functions can be returned in the table. The functions can be implemented to parse multiple record types. In this case, multiple calls to DbGetRecordParsingFunctions must be made, once for each record type whose parsing functions are required by the caller. The DL module uses these functions to parse the opaque data object stored in a data store record. If no parsing function table has been set for a given record type, then a NULL value is returned.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbName (input)
The name of the data store with which the parsing functions are associated.

RecordType (input)
The record type whose parsing functions are requested by the caller.

RETURN VALUE

A function table for the parsing function appropriate to the specified record type. When CSSM_NULL is returned, either no function table has been set for the specified record type or an error has occurred. Use *CSSM_GetError* to obtain the error code and determine the reason for the NULL result.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_NAME
Invalid DB name.

CSSM_DL_MEMORY_ERROR
Error allocating memory.

SEE ALSO

CSSM_DL_SetRecordParsingFunctions

NAME

CSSM_DL_GetDbNames

SYNOPSIS

```
CSSM_NAME_LIST_PTR CSSMAPI CSSM_DL_GetDbNames
    (CSSM_DL_HANDLE DLHandle)
```

DESCRIPTION

This function returns a list of the logical data store names that the specified DL module can access and a count of the number of logical names in that list.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

RETURN VALUE

Returns a pointer to a CSSM_NAME_LIST structure that contains a list of data store names. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_MEMORY_ERROR
Error allocating memory.

CSSM_DL_NO_DATA_SOURCES
No known data store names.

CSSM_DL_GET_DB_NAMES_FAIL
Get DB names failed.

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

SEE ALSO

CSSM_DL_GetDbNameFromHandle, *CSSM_DL_FreeNameList*

NAME

CSSM_DL_GetDbNameFromHandle

SYNOPSIS

```
char * CSSMAPI CSSM_DL_GetDbNameFromHandle
    (CSSM_DL_DB_HANDLE DLDBHandle)
```

DESCRIPTION

This function retrieves the data source name corresponding to an opened data store handle.

PARAMETERS

DLDBHandle (input)
The handle pair that identifies the add-in data storage library module and the open data store whose name should be retrieved.

RETURN VALUE

Returns a string which contains a data store name. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_MEMORY_ERROR
Error allocating memory.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

SEE ALSO

CSSM_DL_GetDbNames

NAME

CSSM_DL_FreeNameList

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_FreeNameList
    (CSSM_DL_HANDLE DLHandle,
    CSSM_NAME_LIST_PTR NameList)
```

DESCRIPTION

This function frees the list of the logical data store names that was returned by *CSSM_DL_GetDbNames*.

PARAMETERS

DLHandle (input)
: The handle that describes the add-in data storage library module to be used to perform this function.

NameList (input)
: A pointer to the CSSM_NAME_LIST.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_MEMORY_ERROR
: Error allocating memory.

CSSM_DL_INVALID_PTR
: Invalid pointer to the name list.

CSSM_DL_INVALID_DL_HANDLE
: Invalid DL handle.

SEE ALSO

CSSM_DL_GetDbNames

14.4 Data Record Operations

The manpages for Data Record Operations follow on the next page.

NAME

CSSM_DL_DataInsert

SYNOPSIS

```
CSSM_DB_UNIQUE_RECORD_PTR CSSMAPI CSSM_DL_DataInsert
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_DB_RECORDTYPE RecordType,
    const CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    const CSSM_DATA_PTR Data)
```

DESCRIPTION

This function creates a new persistent data record of the specified type by inserting it into the specified data store. The values contained in the new data record are specified by the Attributes and the Data. The attribute value list contains zero or more attribute values. The DL module may require initial values for the CSSM pre-defined attributes. The DL module can assume default values for any unspecified attribute values or can return an error condition when DLM-required attribute values are not specified by the caller. The Data is an opaque object to be stored in the new data record.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store in which to insert the new data record.

RecordType (input)
Indicates the type of data record being added to the data store

Attributes (input/optional)
A list of structures containing the attribute values to be stored in that attribute and the meta information (schema) describing those attributes. The list contains at most one entry per attribute in the specified record type. The DL module can assume default values for those attributes that are not assigned values by the caller, or may return an error. If the specified record type does not contain any attributes, this parameter must be NULL.

DataRecord (input/optional)
A pointer to the CSSM_DATA structure which contains the opaque data object to be stored in the new data record. If the specified record type does not contain an opaque data object, this parameter must be NULL.

RETURN VALUE

A pointer to a CSSM_DB_UNIQUE_RECORD_POINTER containing a unique identifier associated with the new record. This unique identifier structure can be used in future references to this record. When NULL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid Data Store handle.

CSSM_DL_INVALID_RECORDTYPE
Invalid record type for this data store.

CSSM_DL_INVALID_ATTRIBUTE
Invalid attribute for this record type in this data store.

CSSM_DL_MISSING_VALUE
: Missing attribute or data value for this record type.

CSSM_DL_DATA_INSERT_FAIL
: Add caused an exception.

SEE ALSO

CSSM_DL_DataDelete

NAME

CSSM_DL_DataDelete

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_DataDelete
    (CSSM_DL__DB_HANDLE DLDBHandle,
    const CSSM_DB_UNIQUE_RECORD_PTR UniqueRecordIdentifier)
```

DESCRIPTION

This function removes the data record specified by the unique record identifier from the specified data store.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store from which to delete the specified data record.

UniqueRecordIdentifier (input)
A pointer to a CSSM_DB_UNIQUE_RECORD identifier containing unique identification of the data record to be deleted from the data store. Once the associated record has been deleted, this unique record identifier cannot be used in future references.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid Data Storage handle.

CSSM_DL_INVALID_RECORD_IDENTIFIER
Invalid data pointer.

CSSM_DL_DATA_DELETE_FAIL
Delete caused an exception.

SEE ALSO

CSSM_DL_DataInsert

NAME

CSSM_DL_DataModify

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_DataModify
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_DB_RECORDTYPE RecordType,
    CSSM_DB_UNIQUE_RECORD_PTR UniqueRecordIdentifier,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR AttributesToBeModified,
    CSSM_DB_DATA_PTR DataToBeModified)
```

DESCRIPTION

This function modifies the persistent data record identified by the UniqueRecordIdentifier. The modifications are specified by the Attributes and Data parameters. For each attribute in the Attributes list, the attribute is added if does not exist, or replaced if it does exist. If a Data value is specified, the record data value should be replaced. To remove a record or attribute, set the value to NULL.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store to search for records satisfying the query.

RecordType (input)
Indicates the type of data record being modified.

UniqueRecordIdentifier (input)
A pointer to a CSSM_DB_UNIQUE_RECORD identifier containing unique identification of the data record to be modified.

AttributesToBeModified (input/optional)
A list containing the names of the attributes to be modified and their new values. For each attribute in the Attributes list, the attribute is added if does not exist, or replaced if it does exist. If the attribute value is NULL, the attribute is deleted. If the Attributes parameter is NULL, no attribute modification occurs.

DataToBeModified (input/optional)
A pointer to the CSSM_DATA structure which contains the opaque data object to be stored in the data record. If this parameter is NULL, no Data modification occurs.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid Data Store handle.

CSSM_DL_INVALID_RECORDTYPE
Invalid record type for this data store.

CSSM_DL_INVALID_ATTRIBUTE
Invalid attribute for this record type in this data store.

CSSM_DL_DATA_MODIFY_FAIL
Modify caused an exception.

SEE ALSO

CSSM_DL_DataInsert, *CSSM_DL_DataDelete*

NAME

CSSM_DL_DataGetFirst

SYNOPSIS

```
CSSM_DB_UNIQUE_RECORD_PTR CSSMAPI CSSM_DL_DataGetFirst
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_QUERY_PTR Query,
    CSSM_HANDLE_PTR ResultsHandle,
    CSSM_BOOL *EndOfDataStore,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR Data)
```

DESCRIPTION

This function retrieves the first data record in the data store that matches the selection criteria. The selection criteria (including selection predicate and comparison values) is specified in the Query structure. The DL module can use internally-managed indexing structures to enhance the performance of the retrieval operation. This function selects the first record satisfying the query based on the list of Attributes and the opaque Data object. This function also returns a flag indicating whether additional records also satisfied the query and a results handle to be used when retrieving subsequent records satisfying the query. If the query selection criteria specifies time or space limits for executing the query, those limits also apply to retrieval of the additional selected data records retrieved using the CSSM_DL_DataGetNext function. Finally, this function returns a unique record identifier associated with the retrieved record. This structure can be used in future references to the retrieved data record.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store to search for records satisfying the query.

Query (input/optional)
The query structure specifying the selection predicate(s) used to query the data store. The structure contains meta information about the search fields and the relational and conjunctive operators forming the selection predicate. The comparison values to be used in the search are specified in the Attributes and Data parameter. The CSSM pre-defined attribute names "PrintName" and "Alias" are valid in any query, regardless of the stored value for those attributes. If no query is specified, the DL module can return the first record in the data store, performing sequential retrieval, or return an error.

ResultsHandle (output)
This handle should be used to retrieve subsequent records that satisfied this query.

EndOfDataStore (output)
A flag indicating whether a record satisfying this query was available to be retrieved in the current operation. If TRUE, then a record was available and was retrieved unless an error condition occurred. If FALSE, then all records satisfying the query have been previously retrieved, and no record has been returned by this operation.

Attributes (output)
A list of attribute values (and corresponding meta information) from the retrieved record.

Data (output)
The opaque object stored in the retrieved record.

RETURN VALUE

If successful and EndOfDataStore is FALSE, this function returns a pointer to a CSSM_UNIQUE_RECORD structure containing unique identifier associated with the retrieved record. This unique identifier structure can be used in future references to this record using this DLDBHandle pairing. It may not be valid for other DLHandles targeted to this DL module or to other DBHandles targeted to this data store. If the pointer is NULL and EndOfDataStore is TRUE, then a normal termination condition has occurred. If the pointer is NULL and EndOfDataStore is FALSE, then an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_INVALID_SELECTION_PRED
Invalid selection predicate.

CSSM_DL_NO_DATA_FOUND
No data records match the selection predicate.

CSSM_DL_DATA_GETFIRST_FAIL
An exception occurred when processing the query.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

CSSM_DL_DataGetNext, *CSSM_DL_DataAbortQuery*

NAME

CSSM_DL_DataGetNext

SYNOPSIS

```
CSSM_DB_UNIQUE_RECORD_PTR CSSMAPI CSSM_DL_DataGetNext
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_BOOL *EndOfDataStore,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR Data)
```

DESCRIPTION

This function returns the next data record referenced by the ResultsHandle. The ResultsHandle references a set of records selected by an invocation of the DataGetFirst function. The record values are returned in the Attributes and Data parameters. A flag indicates whether additional records satisfying the original query remain to be retrieved. The function also returns a unique record identifier for the return record.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function, and the open data store from which records were selected by the initiating query.

ResultsHandle (output)
The handle identifying a set of records retrieved by a query executed by the DataGetFirst function.

EndOfDataStore (output)
A flag indicating whether a record satisfying this query was available to be retrieved in the current operation. If TRUE, then a record was available and was retrieved unless an error condition occurred. If FALSE, then all records satisfying the query have been previously retrieved and no record has been returned by this operation.

Attributes (input/output)
The names of the attributes to be retrieved are input. The DL module fills in these attributes' values from the retrieved record and returns these values as output. If the Attributes pointer is NULL, no values are returned.

Data (output)
The opaque object stored in the retrieved record. If the pointer is NULL, no record is returned.

RETURN VALUE

If successful and EndOfDataStore is FALSE, this function returns a pointer to a CSSM_UNIQUE_RECORD structure containing a a unique identifier associated with the retrieved record. This unique identifier structure can be used in future references to this record using this DLDBHandle pairing. It may not be valid for other DLHandles targeted to this DL module or to other DBHandles targeted to this data store. If the pointer is NULL and EndOfDataStore is TRUE, then a normal termination condition has occurred. If the pointer is NULL and EndOfDataStore is FALSE, then an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid Data Store handle.

CSSM_DL_INVALID_RESULTS_HANDLE
Invalid query handle.

CSSM_DL_NO_MORE_RECORDS
No more records for this selection handle.

CSSM_DL_DATA_GETNEXT_FAIL
Opening the records caused an exception.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

CSSM_DL_DataGetFirst, *CSSM_DL_DataAbortQuery*

NAME

CSSM_DL_DataAbortQuery

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_DataAbortQuery
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function terminates the query initiated by *DL_DataGetFirst*, and allows a DL to release all intermediate state information associated with the query.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store from which records were selected by the initiating query.

ResultsHandle (input)
The selection handle returned from the initial query function.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid data store handle.

CSSM_DL_INVALID_RESULTS_HANDLE
Invalid results handle.

CSSM_DL_DATA_ABORT_QUERY_FAIL
Unable to abort query.

SEE ALSO

CSSM_DL_DataGetFirst, *CSSM_DL_DataGetNext*

NAME

CSSM_DL_DataGetFromUniqueRecordId

SYNOPSIS

```
CSSM_RETURN CSSMAPI DL_DataGetFromUniqueRecordId
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR Data)
```

DESCRIPTION

This function retrieves the data record and attributes associated with this unique record identifier. The DL module can use indexing structure identified in the UniqueRecord to enhance the performance of the retrieval operation.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store to search for the data record.

UniqueRecord (input)
The pointer to a unique record structure returned from a *DL_DataInsert*, *DL_DataGetFirst*, or *DL_DataGetNext* operation.

Attributes (input/output)
The calling application specifies the names of the attributes to be retrieved. The DL module fills in these attributes' values for the retrieved record. If the Attributes pointer is NULL, the DL module should not return the record's attributes.

Data (output)
The opaque object stored in the retrieved record. If the Data pointer is NULL, the DL module should not return the record's data.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_NO_DATA_FOUND
No data records match the unique record id.

CSSM_DL_DATA_GETFROMUNIQUEID_FAIL
An exception occurred when processing the query.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

CSSM_DL_DataInsert, *CSSM_DL_DataGetFirst*, *CSSM_DL_DataGetNext*

NAME

CSSM_DL_FreeUniqueRecord

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DL_FreeUniqueRecord
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord)
```

DESCRIPTION

This function frees the memory associated with the data store unique record structure.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store from which the UniqueRecord identifier was assigned.

UniqueRecord(input)
The pointer to the memory that describes the data store unique record structure.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_UNIQUE_RECORD_POINTER
Invalid data store unique record pointer.

SEE ALSO

CSSM_DL_DataInsert, *CSSM_DL_DataGetFirst*, *CSSM_DL_DataGetNext*

14.5 Extensibility Functions

The manpages for Extensibility Functions follow on the next page.

NAME

CSSM_DL_PassThrough

SYNOPSIS

```
void * CSSMAPI DL_PassThrough
    (CSSM_DL_DB_HANDLE DLDBHandle,
    uint32 PassThroughId,
    const void *InputParams)
```

DESCRIPTION

This function allows applications to call data storage library module-specific operations that have been exported. Such operations may include queries or services that are specific to the domain represented by a DL module.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store upon which the function is to be performed.

PassThroughId (input)
An identifier assigned by a DL module to indicate the exported function to be performed.

InputParams (input)
A pointer to a module implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested DL module.

RETURN VALUE

A pointer to a module implementation-specific structure containing the output from the pass-through function. The output data must be interpreted by the calling application based on externally-available information. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_INVALID_PASSTHROUGH_ID
Invalid passthrough ID.

CSSM_DL_INVALID_PTR
Invalid pointer.

CSSM_DL_PASS_THROUGH_FAIL
DB exception doing passthrough function.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

Appendix A

CSSM Error-Handling

A.1 Introduction

This chapter presents a specification for error handling in CSSM that provides a consistent mechanism across all layers of CSSM for returning errors to the caller.

All CSSM API functions will return one of the following:

1. CSSM_RETURN—an enumerated type consisting of CSSM_OK and CSSM_FAIL. If it is CSSM_FAIL, an error code indicating the reason for failure can be obtained by calling *CSSM_GetError*( ).

2. CSSM_BOOL—an enumerated type consisting of CSSM_TRUE and CSSM_FALSE. If it is CSSM_FALSE, an error code may be available (but not always) by calling *CSSM_GetError*.

3. A pointer to a data structure, a handle, a file size or whatever is logical for the function to return. An error code may be available (but not always) by calling *CSSM_GetError*.

Check documentation for individual functions to determine if error information will be available and what error values the function uses. Note that there will be additional error values defined by add-in modules. The information available from *CSSM_GetError* will include both the error number and a GUID (global unique ID) that will associate the error with the add-in module that set it. The GUID of each add-in module can be obtained by calling CSSM_XX_ListModules (where XX = CSP, CL, DL, or TP). *CSSM_CompareGuids* can then be called to determine from which module an error came.

Each add-in module must have a mechanism for reporting their errors to the calling application. In general, there are two types of errors an add-in module can return:

- Errors CSSM has defined for it to use (CSSM_CSP_INVALID_SECURITY_LIST)
- Errors particular to an add-in module (XXX_CSP_BAD_HW_TOKEN_SERIAL_NUMBER)

Since some errors are predefined by CSSM, those errors have a set of pre-defined numeric values which are reserved by CSSM, and cannot be used arbitrarily by add-in modules. For errors that are particular to an add-in module, a different set of predefined values has been reserved for their use.

It will be up to the calling application to determine how to handle the error returned by *CSSM_GetError*( ). Detailed descriptions of the error values will be available in the corresponding specification, the **<cssmerr.h>** header file, and the documentation for specific add-in modules. If a routine does not know how to handle the error, it may choose to pass the error on up the chain to its caller.

Error values should not be overwritten, if at all possible. Overwriting the return destroys valuable error handling and debugging information. This means an add-in module ot type A can return an error code defined by an add-in module of type B.

A.2 Data Structures

```
typedef enum cssm_bool {
    CSSM_FALSE = 0,
    CSSM_TRUE = 1,
} CSSM_BOOL

typedef enum cssm_return {
    CSSM_OK = 0,
    CSSM_FAIL = -1
} CSSM_RETURN

typedef struct cssm_error {
    uint32 error;
    CSSM_GUID guid;
} CSSM_ERROR, *CSSM_ERROR_PTR
```

A.3 Error Handling Functions

The manpages for Error Handling Functions follow on the next page.

NAME

CSSM_GetError

SYNOPSIS

```
CSSM_ERROR_PTR CSSMAPI CSSM_GetError
    (void)
```

DESCRIPTION

This function returns the current error information.

PARAMETERS

None.

RETURN VALUE

Returns the current error information. If there is no valid error, the error number will be CSSM_OK. A NULL pointer indicates that the *CSSM_InitError* was not called or that a call to *CSSM_DestroyError* has been made. No error information is available.

SEE ALSO

CSSM_InitError, *CSSM_DestroyError*, *CSSM_ClearError*, *CSSM_SetError*, *CSSM_IsCSSMError*, *CSSM_IsCLError*, *CSSM_IsTPError*, *CSSM_IsDLError*, *CSSM_IsCSPError*

NAME

CSSM_SetError

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SetError
    (CSSM_GUID_PTR guid,
    uint32 error_number)
```

DESCRIPTION

This function sets the current error information to error_number and guid.

PARAMETERS

guid (input)
: Pointer to the GUID (global unique ID) of the add-in module.

error_number (input)
: An error number. It should fall within one of the valid CSSM, CL, TP, DL, or CSP error ranges.

RETURN VALUE

CSSM_OK if error was successfully set. A return value of CSSM_FAIL indicates that the error number passed is not within a valid range, the GUID passed is invalid, *CSSM_InitError* was not called, or *CSSM_DestroyError* has been called. No error information is available.

SEE ALSO

CSSM_InitError, *CSSM_DestroyError*, *CSSM_ClearError*, *CSSM_GetError*

NAME

CSSM_ClearError

SYNOPSIS

```
void CSSMAPI CSSM_ClearError
    (void)
```

DESCRIPTION

This function sets the current error value to CSSM_OK. This can be called if the current error value has been handled and therefore is no longer a valid error.

PARAMETERS

None.

SEE ALSO

CSSM_SetError, CSSM_GetError

NAME

CSSM_InitError

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_InitError
    (void)
```

DESCRIPTION

This function initializes the error information for that thread/process and allocates any necessary memory. Should be called by the thread/process initialization function.

PARAMETERS

None.

RETURN VALUE

CSSM_OK if the error information was successfully initialized. If CSSM_FAIL is returned, no error information will be available.

Note: *CSSM_InitError* does not need to be called if you have loaded the CSSM DLL.

SEE ALSO

CSSM_DestroyError

NAME

CSSM_DestroyError

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DestroyError
    (void)
```

DESCRIPTION

This function destroys the error information for a thread/process and frees any necessary memory. It should be called by the function performing clean up before a thread/process exits.

PARAMETERS

None.

RETURN VALUE

CSSM_OK if the error information was successfully destroyed. If CSSM_FAIL is returned, no error information will be available.

Note: *CSSM_DestroyError* does not need to be called if you have loaded the CSSM DLL.

SEE ALSO

CSSM_InitError

NAME

CSSM_IsCSSMError

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_IsCSSMError
    (uint32 error_number)
```

DESCRIPTION

This function determines if error_number is within the CSSM range of errors.

PARAMETERS

error_number (input)
An error number.

RETURN VALUE

CSSM_TRUE if the error is a CSSM error; otherwise CSSM_FALSE.

SEE ALSO

CSSM_IsCLError, *CSSM_IsDLError*, *CSSM_IsTPError*, *CSSM_IsCSPError*

NAME

CSSM_IsCLError

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_IsCLError
    (uint32 error_number)
```

DESCRIPTION

This function determines if error_number is within the CL range of errors.

PARAMETERS

error_number (input)
An error number.

RETURN VALUE

CSSM_TRUE if the error is a CL error; otherwise CSSM_FALSE.

SEE ALSO

CSSM_IsCSSMError, *CSSM_IsDLError*, *CSSM_IsTPError*, *CSSM_IsCSPError*

NAME

CSSM_IsDLError

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_IsDLError
    (uint32 error_number)
```

DESCRIPTION

This function determines if error_number is within the DL range of errors.

PARAMETERS

error_number (input)
An error number.

RETURN VALUE

CSSM_TRUE if the error is a DL error; otherwise CSSM_FALSE.

SEE ALSO

CSSM_IsCLError, *CSSM_IsCSSMError*, *CSSM_IsTPError*, *CSSM_IsCSPError*

NAME

CSSM_IsTPError

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_IsTPError
    (uint32 error_number)
```

DESCRIPTION

This function determines if error_number is within the TP range of errors.

PARAMETERS

error_number (input)
An error number.

RETURN VALUE

CSSM_TRUE if the error is a TP error; otherwise CSSM_FALSE.

SEE ALSO

CSSM_IsCLError, *CSSM_IsDLError*, *CSSM_IsCSSMError*, *CSSM_IsCSPError*

NAME

CSSM_IsCSPError

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_IsCSPError
    (uint32 error_number)
```

DESCRIPTION

This function determines if error_number is within the CSP range of errors.

PARAMETERS

error_number (input)
An error number.

RETURN VALUE

CSSM_TRUE if the error is a CSP error; otherwise CSSM_FALSE.

SEE ALSO

CSSM_IsCLError, *CSSM_IsDLError*, *CSSM_IsTPError*, *CSSM_IsCSSMError*

NAME

CSSM_CompareGuids

SYNOPSIS

```
CSSM_BOOL CSSMAPI CSSM_CompareGuids
    (CSSM_GUID guid1,
    CSSM_GUID guid2)
```

DESCRIPTION

This function determines if two GUIDs are equal.

PARAMETERS

guid1 (input)
A GUID.

guid1 (input)
A GUID.

RETURN VALUE

CSSM_TRUE if the two GUIDs are equal, CSSM_FALSE otherwise.

Note: GUIDs are returned in the error information of *CSSM_GetError*. Once you know which type of error is returned (CSP, CL, TP, DL), you can call CSSM_XX_ListModules to get a list of all the modules that are registered and their GUIDs, in order to determine which module set the error. This can be useful for debugging purposes if there is more than one type of module for each add-in type installed on the system.

SEE ALSO

CSSM_GetError, *CSSM_CSP_ListModules*, *CSSM_CL_ListModules*, *CSSM_TP_ListModules*, *CSSM_DL_ListModules.*

Appendix B

Application Memory Functions

B.1 Introduction

When CSSM or add-in modules return memory structures to applications, that memory is maintained by the application. Instead of using a model where the application passes memory blocks to the add-in modules to work on, the CSSM model requires the application to supply memory functions. This frees the application from any requirement to specify memory block sizes to the CSSM and the add-ins. The memory that the application receives is in its process space, and this prevents the application from walking through the memory of the CSSM or the add-in modules. When the application no longer requires the memory, it is responsible for freeing it.

Applications will register memory functions with the add-in modules during attach time and with CSSM during initialization. A memory function table will be passed from the application to add-in modules through the CSSM_xxx_Attach functions associated with each add-in. The *CSSM_Init* function is where the CSSM will receive the application's memory function.

B.1.1 CSSM_API_MEMORY_FUNCS Data Structure

This structure is used by applications to supply memory functions for the CSSM and the add-in modules. The functions are used when memory needs to be allocated by the CSSM or add-ins for returning data structures to the applications.

```
typedef struct cssm_api_memory_funcs {
  void * (*malloc_func) (uint32 Size, void *AllocRef);
  void (*free_func) (void *MemPtr, void *AllocRef);
  void * (*realloc_func) (void *MemPtr, uint32 Size, void *AllocRef);
  void * (*calloc_func) (uint32 Num, uint32 Size, void *AllocRef);
  void *AllocRef;
} CSSM_API_MEMORY_FUNCS, *CSSM_API_MEMORY_FUNCS_PTR
```

Definition

malloc_func
: Pointer to function that returns a void pointer to the allocated memory block of at least size bytes from heap AllocRef.

free_func
: Pointer to function that deallocates a previously-allocated memory block (memblock) from heap AllocRef.

realloc_func
: Pointer to function that returns a void pointer to the reallocated memory block (memblock) of at least size bytes from heap AllocRef.

calloc_func
: Pointer to function that returns a void pointer to an array of num elements of length size initialized to zero from heap AllocRef.

AllocRef
Indicates the memory heap the function operates on.

CAE Specification

Part 3:

CSSM Key Recovery API

The Open Group

Chapter 15

Overview

Key recovery mechanisms serve many useful purposes. They may be used by individuals to recover lost or corrupted keys; they may be used by enterprises to deter corporate insiders from using encryption to bypass the corporate security policy regarding the flow of proprietary information. Corporations may also use key recovery mechanisms to recover employee keys in certain situations, for example, in the employee's absence. The use of key recovery mechanisms in web based transactional scenarios can serve as an additional technique of non-repudiation and audit, that may be admissible in a court of law. Finally, key recovery mechanisms may be used by jurisdictional law enforcement bodies to access the contents of confidentiality protected communications and stored data. Thus, there appear to be multiple incentives for the incorporation as well as adoption of key-recovery mechanisms in local and distributed encryption based systems.

15.1 Key Recovery Nomenclature

Denning and Brandstad [Key Escrow], present a taxonomy of key escrow systems. Here, a different scheme of nomenclature was adopted in order to exhibit some of the finer nuances of key recovery schemes. The term key recovery encompasses mechanisms that allow authorized parties to retrieve the cryptographic keys used for data confidentiality, with the ultimate goal of recovery of encrypted data. The remainder of this section will discuss the various types of key recovery mechanisms, the phases of key recovery, and the policies with respect to key recovery.

15.1.1 Key Recovery Types

There are two classes of key recovery mechanisms based on the way keys are held to enable key recovery:

- Key escrow—techniques based on the paradigm that the government or a trusted party called an *escrow agent*, holds the actual user keys or portions thereof.
- Key encapsulation—techniques based on the paradigm that a cryptographically encapsulated form of the key is made available to parties that require key recovery. The technique ensures that only certain trusted third parties called *recovery agents* can perform the unwrap operation to retrieve the key material buried inside.

There may also be hybrid schemes that use some escrow mechanisms in addition to encapsulation mechanisms.

An orthogonal way to classify key recovery mechanisms is based on the nature of the key:

- Long-term, private keys
- Ephemeral keys

Both types can be escrowed or encapsulated. Since escrow schemes involve the actual archival of keys, they typically deal with long-term keys, in order to avoid the proliferation problem that arises when trying to archive the myriad ephemeral keys. Key encapsulation techniques, on the other hand, usually operate on the ephemeral keys.

For a large class of key recovery (escrow as well as encapsulation) schemes, there are a set of *key recovery fields* that accompany an enciphered message or file. These key recovery fields may be used by the appropriate authorized parties to recover the decryption key and or the plaintext.

Typically, the key recovery fields comprise information regarding the key escrow or recovery agent(s) that can perform the recovery operation; they also contain other pieces of information to enable recovery.

In a key escrow scheme for long-term private keys, the "escrowed" keys are used to recover the ephemeral data confidentiality keys. In such a scheme, the key recovery fields may comprise the identity of the escrow agent(s), identifying information for the escrowed key, and the bulk encryption key wrapped in the recipient's public key (which is part of an escrowed key pair); thus the key recovery fields include the key exchange block in this case. In a key escrow scheme where bulk encryption keys are archived, the key recovery fields may comprise information to identify the escrow agent(s), and the escrowed key for that enciphered message.

In a typical key encapsulation scheme for ephemeral bulk encryption keys, the key recovery fields are distinct from the key exchange block, (if any.) The key recovery fields identify the recovery agent(s), and contain the bulk encryption key encapsulated using the public keys of the recovery agent(s).

The key recovery fields are generated by the party performing the data encryption, and associated with the enciphered data. To ensure the integrity of the key recovery fields, and its association with the encrypted data, it may be required for processing by the party performing the data decryption. The processing mechanism ensures that successful data decryption cannot occur unless the integrity of the key recovery fields are maintained at the receiving end. In schemes where the key recovery fields contain the key exchange block, decryption cannot occur at the receiving end unless the key recovery fields are processed to obtain the decryption key; thus the integrity of the key recovery fields are automatically verified. In schemes where the key recovery fields are separate from the key exchange block, additional processing must be done to ensure that decryption of the ciphertext occurs only after the integrity of the key recovery fields are verified.

15.1.2 Key Recovery Phases

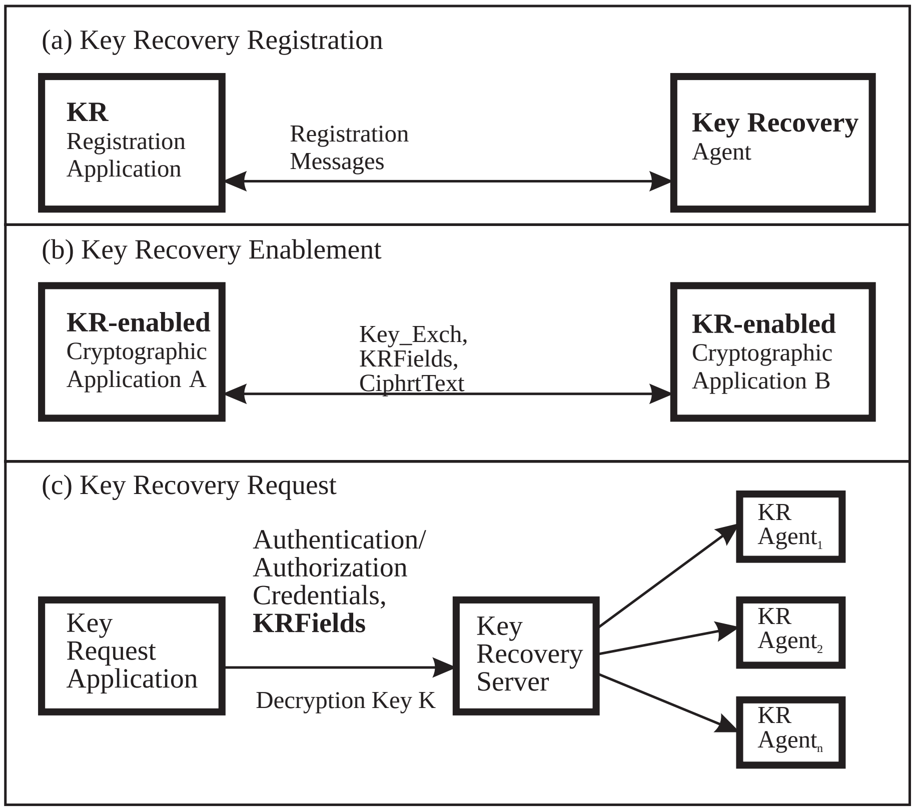
(a) Key Recovery Registration
KR Registration Application
Registration Messages
Key Recovery Agent
(b) Key Recovery Enablement
KR-enabled Cryptographic Application A
Key_Exch, KRFields, CiphrtText
KR-enabled Cryptographic Application B
(c) Key Recovery Request
Key Request Application
Authentication/ Authorization Credentials, KRFields
Decryption Key K
Key Recovery Server
KR Agent1
KR Agent2
KR Agentn

Figure 15-1 Key Recovery Phases

The process of cryptographic key recovery involves three major phases. First, there is an optional *key recovery registration* phase where the parties that desire key recovery perform some initialization operations with the escrow or recovery agents; these operations include obtaining a user public key certificate (for an escrowed key pair) from an escrow agent, or obtaining a public key certificate from a recovery agent . Next, parties that are involved in cryptographic associations have to perform operations to enable key recovery (such as the generation of key recovery fields, and so on)—this is typically called the key recovery enablement phase. Finally, authorized parties that desire to recover the data keys, do so with the help of a recovery server and one or more escrow agents or recovery agents—this is the *key recovery request* phase.

Figure 15-1 illustrates the three phases of key recovery. In Figure 15-1(a), a key recovery client registers with a recovery agent prior to engaging in cryptographic communication. In Figure 15-1(b), two key-recovery-enabled cryptographic applications are communicating using a key encapsulation mechanism; the key recovery fields are passed along with the ciphertext and key exchange block, to enable subsequent key recovery. The key recovery request phase is illustrated in Figure 15-1(c), where the key recovery fields are provided as input to the key recovery server along with the authorization credentials of the client requesting service. The key recovery server interacts with one or more local or remote key recovery agents to reconstruct the secret key that can be used to decrypt the ciphertext.

It is envisaged that governments or organizations will operate their own recovery server hosts independently, and that key recovery servers may support a single or multiple key recovery mechanisms. There are a number of important issues specific to the implementation and operation of the key recovery servers, such as vulnerability and liability. The focus of this documentation is a framework-based approach to implementing the key recovery operations pertinent to end parties that use encryption for data confidentiality. The issues with respect to the key recovery server and agents will not be discussed further here.

15.1.3 Lifetime of Key Recovery Fields

Cryptographic products fall into one of two fundamental classes: *archived-ciphertext products*, and *transient-ciphertext products*. When the product allows either the generator or the receiver of ciphertext to archive the ciphertext, the product is classified as an archived-ciphertext product. On the other hand, when the product does not allow the generator or receiver of ciphertext to archive the ciphertext, it is classified as a transient-ciphertext product.

It is important to note that the lifetime of key recovery fields should never be greater than the lifetime of the associated ciphertext. This is somewhat obvious, since recovery of the key is only meaningful if the key can be used to recover the plaintext from the ciphertext. Hence, when archived-ciphertext products are key recovery enabled, the key recovery fields are typically archived for the same duration as the ciphertext. Similarly, when transient-ciphertext products are key recovery enabled, the key recovery fields are associated with the ciphertext for the duration of its lifetime. It is not meaningful to archive key recovery fields without archiving the associated ciphertext.

15.1.4 Key Recovery Policy

Key recovery policies are mandatory policies that may be derived from enterprise-based or jurisdiction-based rules on the use of cryptographic products for data confidentiality. Political jurisdictions may choose to define key recovery policies for cryptographic products based on export, import, or use controls. Enterprises may define internal and external domains, and may mandate key recovery policies on the cryptographic products within their own domain.

Key recovery policies come in two flavors: *key recovery enablement policies* and *key recovery inter-operability policies*. Key recovery enablement policies specify the exact cryptographic protocol suites (algorithms, modes, key lengths and so on) and perhaps usage scenarios, where key recovery enablement is mandated. Furthermore, these policies may also define the number of bits of the cryptographic key that may be left out of the key recovery enablement operation; this is typically referred to as the *workfactor*. Key recovery inter-operability policies specify to what degree a key-recovery-enabled cryptographic product is allowed to interoperate with other cryptographic products.

15.1.5 Operational Scenarios for Key Recovery

There are three basic operational scenarios for key recovery:

- Enterprise key recovery
- Law enforcement key recovery
- Individual key recovery

Enterprise key recovery allows enterprises to enforce stricter monitoring of the use of cryptography, and the recovery of enciphered data when the need arises. The user in this scenario is the enterprise employee. Enterprise key recovery is based on a mandatory key recovery policy; however, this policy is set (typically through administrative means) by the organization or enterprise at the time of installation of a recovery-enabled cryptographic product. The enterprise key recovery policy should not be modifiable or by-passable by the individual using the cryptographic product. Enterprise key recovery mechanisms may use special, enterprise-authorized escrow or recovery agents.

In the law enforcement scenario, key recovery is mandated by the jurisdictional law enforcement authorities in the interest of national security and law enforcement. The user in this scenario is the private citizen in the jurisdiction where the product is being used. For a specific cryptographic product, the key recovery policies for multiple jurisdictions may apply simultaneously. The policies (if any) of the jurisdiction(s) of manufacture of the product, as well as the jurisdiction of installation and use, need to be applied to the product such that the most restrictive combination of the multiple policies is used. Thus, law enforcement key recovery is based on mandatory key recovery policies; these policies are logically bound to the cryptographic product at the time the product is shipped. There may be some mechanism for vendor-controlled updates of such law enforcement key recovery policies in existing products; however, organizations and end users of the product are not able to modify this policy at their discretion. The escrow or recovery agents used for this scenario of key recovery need to be strictly controlled in most cases, to ensure that these agents meet the eligibility criteria for the relevant political jurisdiction where the product is being used.

Individual key recovery is user-discretionary in nature, and is performed for the purpose of recovery of enciphered data by the owner of the data, if the cryptographic keys are lost or corrupted. The user in this scenario is the traditional end-user of the software product. Since this is a non-mandatory key recovery scenario, it is not based on any policy that is enforced by the cryptographic product; rather, the product may allow the user to specify when individual key recovery enablement is to be performed. There are few restrictions on the use of specific escrow or recovery agents.

Key recovery-enabled cryptographic products must be designed so that the key recovery enablement operation is mandatory and noncircumventable in the law enforcement and enterprise scenarios, and discretionary for the individual scenario. The escrow and recovery agent(s) that are used for law enforcement and enterprise scenarios must be tightly controlled. These agents must be validated as as authorized or approved agents. In the law enforcement and enterprise scenarios, the key recovery process typically needs to be performed without the knowledge and cooperation of the parties involved in the cryptographic association.

The components of the key recovery fields also varies somewhat between the three scenarios. In the law enforcement scenario, the key recovery fields must contain identification information for the escrow or recovery agent(s); whereas for the enterprise and individual scenarios, the agent identification information is not so critical, since this information may be available from the context of the recovery enablement operation. For the individual scenario, there needs to be a strong user authentication component in the key recovery fields, to allow the owner of the key recovery fields to authenticate themselves to the agents; however, for the enterprise and law enforcement scenarios, the authorization credentials checked by the agents may be in the form of legal documents, or enterprise-authorization documents for key recovery, that may not be tied to any authentication component in the key recovery fields. For the law enforcement and enterprise scenarios, the key recovery fields may contain recovery information for both the generator and receiver of the enciphered data; in the individual scenario, only the information of the generator of the enciphered data is typically included (at the discretion of the generating party).

15.2 Key Recovery in the Common Data Security Architecture

The Common Data Security Architecture (CDSA) defines an open infrastructure for security services. Within the four layer architecture, the Common Security Services Manager (CSSM) is the central layer that manages the range of security service options available to applications. CSSM allows applications to dynamically select:

- Categories of security services
- Mechanisms that perform desired security services
- Implementations of selected security mechanisms

CSSM acts as a broker between applications requesting security services and dynamically-loadable security service modules. The CSSM application programming interface (CSSM-API) defines the interface for accessing security services. The CSSM service provider interface (CSSM-SPI) defines the interface for service providers who develop plug-able security service products.

CSSM is extensible in that it also provides dynamic loading of module managers that provide elective categories of security services. Key recovery is an important security service for applications and institutions that choose to use it. CSSM accommodates key recovery as an elective category of security service.

A complete architectural description of CDSA and CSSM is contained in the *Common Data Security Architecture (CDSA) Specification*.

Chapter 16

Key Recovery Enablement in CSSM

Figure 16-1 shows the Key Recovery Module Manager (KRMM) as an elective service in CSSM. The KRMM defines a key recovery API (KR-API) on top and a key recovery SPI (KR-SPI) below. One or more Key Recovery Service Providers may be plugged-in under the KRMM. The KRMM manages these dynamic service modules and brokers their use by applications and layered security-aware services, such as SSL (Secure Sockets Layer) and SMIME (Secure MIME).

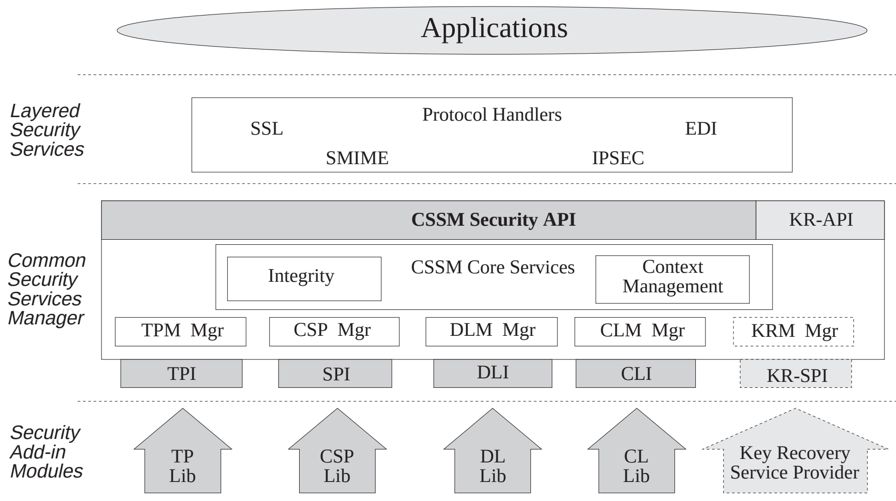

Figure 16-1 Elective Key Recovery Services in the CSSM

16.1 Functionality Definition

CDSA defines the expected functions for each layer of the four layer architecture. Processes, such as protocol handlers, in the security services layer that use key recovery services are assumed to perform the following functions with respect to key recovery:

- Determination of key recovery mechanism (perhaps through negotiation with peer) and selection of an appropriate key recovery service provider
- Identification of the peers in the cryptographic association
- Set up and update of key recovery parameters for the peers in the cryptographic association
- Invocation of the key recovery field generation function and associating the generated fields with the ciphertext
- Retrieval of the key recovery fields from the protocol message or file and invocation of the key recovery field processing function
- Understanding the semantics of the opaque input parameters for the key recovery registration and recovery request operations
- Providing callbacks to allow the KRSP to dynamically obtain additional input from the application layer code, and interact with the human interface, if necessary

The KRMM in the CSSM layer performs the following functions with respect to key recovery:

- Storing and fetching user key recovery parameters from a persistent repository
- Maintaining key recovery context or state information
- Determination of when key recovery fields need to be generated or processed
- Invocation of the KR-SPI with appropriate parameters when key recovery operations are invoked

The Key Recovery Service Provider performs the following functions with respect to key recovery:

- Validation of any and all recovery agent certificates by selection of appropriate certificate library and trust policy service providers
- Choosing an appropriate CSP to use as a cryptographic engine for key recovery field generation
- Generation of the key recovery fields
- Processing of the key recovery fields
- Exchanging messages with a possibly remote key recovery agent/server for recovery registration and request operations
- Invocation of supplied callbacks to obtain additional input information, as necessary
- Maintaining state about asynchronous recovery registration and request operations to allow the application layer code to check (by polling) if the results of a registration or request operation are available

16.2 Extensions to the Cryptographic Module Manager

The Cryptographic Module Manager of the CSSM is responsible for handling the cryptographic functions of the CSSM. In order to introduce the necessary dependencies between the cryptographic operations and the key recovery enablement operations, the cryptographic module manager is extended with conditional behavior as specified below.

The cryptographic context data structure, which holds the many parameters that must be specified as input to a cryptographic function, has been augmented to include the following key recovery extension fields:

- An enterprise usability flag for key recovery
- A law enforcement usability flag for key recovery
- A workfactor field for law enforcement key recovery

The two flag parameters denote whether a cryptographic context needs to have key recovery enablement operations performed before it can be used for cryptographic operations such as encrypt or decrypt. The workfactor field holds the allowable workfactor value for law enforcement key recovery. These three additional fields of the cryptographic context are not available through the CSSM-API for modification. They are set by the KRMM when the latter makes the key recovery policy enforcement decision for enterprise and law enforcement policies.

Although the CSSM API has been left intact in the CSSM, the behavior of some of the cryptographic functions will change due to intervention of the KRMM and the cryptographic module manager, which sits between the caller and the service provider module. Behavioral changes in the cryptographic module manager are based on whether the KRMM is present in the system and the values stored in the cryptographic context extensions. The conditional behavior is as follows:

- Invoke key recovery policy enforcement functions for cryptographic context creation and update operations
- Flag cryptographic context as unusable if key recovery enablement operations are mandated
- Check cryptographic context usability flags for encrypt/decrypt operations

Whenever a cryptographic context is created or updated using the CSSM API and the KRMM is present in CSSM, the cryptographic module manager invokes a KRMM policy enforcement function module. The KRMM checks the enterprise and law enforcement policies to determine whether the cryptographic context defines an operation where key recovery is mandated. If so, the key recovery flags are set in the cryptographic context data structure to signify that the context is unusable until key recovery enablement operations are performed on this context. When the appropriate key recovery enablement operations are performed on this context, the flag values are toggled so that the cryptographic context becomes usable for the intended operations.

When the encryption/decryption operations are invoked through the CSSM-API and the KRMM is present in CSSM, the cryptographic module manager checks the key recovery usability flags in the cryptographic context to determine whether the context is usable for encryption/decryption operations. If the context is flagged as unusable, the cryptographic module manager does not dispatch the call to the CSP and returns an error to the caller. When the appropriate key recovery enablement operations are performed on that context, the KRMM resets the context flags making that context usable for encryption/decryption.

16.3 Key Recovery Module Manager

The Key Recovery Module Manager is responsible for handling the KR-API functions and invocation of the appropriate KR-SPI functions. The KRMM enforces the key recovery policy on all cryptographic operations that are obtained through the CSSM. It maintains key recovery state in the form of key recovery contexts.

16.3.1 Operational Scenarios

The CSSM architecture supports three distinct operational scenarios for key recovery, namely, key recovery for law enforcement purposes, enterprise purposes, and individual purposes. The law enforcement and enterprise scenarios for key recovery are mandatory in nature, thus the CSSM layer code enforces the key recovery policy with respect to these scenarios through the appropriate sequencing of KR-API and cryptographic API calls. On the other hand, the individual scenario for key recovery is completely discretionary, and is not enforced by the CSSM layer code. The application/user requests key recovery operations using the KR-APIs at their discretion.

CSSM allows authorized applications to request and be granted exemption from built-in policy checks performed by CSSM module managers such as the KRMM. Applications with appropriate credentials can request exemption from the key recovery checks defined for the enterprise, for law enforcement, or for both. Exemption is granted if the caller provides credentials that:

- Are successfully authenticated by CSSM

- Carry implied authorization for the requested exemptions

Applications use a CSSM_EXEMPTION_MASK to represents a set of requested exemptions. The Key Recovery Module Manager defines the following exemption request flags in addition to those already defined by CSSM and by other elective module managers:

- CSSM_EXEMPT_LE_KR
- CSSM_EXEMPT_ENT_KR

The CSSM_RequestCssmExemption function is used to request exemptions. Applications can invoke this function at any time after invoking the CSSM_Init function. This allows applications to change exemption status as appropriate during execution. Authentication and implied authorization are checked by CSSM at each request.

16.3.2 Key Recovery Profiles

The KRSPs require certain pieces of information related to the parties involved in a cryptographic association in order to generate and process key recovery fields. These pieces of information (such as the public key certificates of the key recovery agents) are contained in *key recovery profiles*. A key recovery profile contains all of the per-user parameters for key recovery field generation and processing for a specific KRSP. In other words, each user has a distinct profile for each KRSP.

The information contained in the profile comprises the following:

- A user identity
- The public key certificate chain for the user
- A set of Key Recovery Agent (KRA) certificate chains for enterprise key recovery
- A set of Key Recovery Agent (KRA) certificate chains for law enforcement key recovery
- An authentication information field for enterprise key recovery
- A set of Key Recovery Agent (KRA) certificate chains for individual key recovery
- An authentication information field for individual key recovery
- A set of key recovery flags that fine tune the behavior of a KRSP
- An extension field

The key recovery profiles support a list of KRA certificate chains for each of the law enforcement, enterprise, and individual key recovery scenarios, respectively. While the profile allows full certificate chains to be specified for the KRAs, it also supports the specification of leaf certificates; in such instances, the KRSP and the appropriate TP modules are expected to dynamically discover the intermediate certificate authority certificates up to the root certificate of trust. One or more of these certificate chains may be set to NULL, if they are not needed or supported by the KRSP involved.

The user public key certificate chain is also part of a profile. This is a necessary parameter for certain key escrow and encapsulation schemes. Similarly certain schemes support the notion of an authentication field for enterprise as well as individual key recovery. This field is used by the key recovery server and/or agent(s) to verify the authorization of the individual/enterprise requesting key. One or more fields can be set to NULL, if their use is not required or supported by the KRSP involved.

The key recovery flags are defined values that are pertinent for a large class of escrow and recovery schemes. The extension field is for use by the KRSPs to define additional semantics for the key recovery profile. These extensions may be flag parameters or value parameters. The semantics of these extensions are defined by a KRSP; the application that uses profile extensions has to be cognizant of the specific extensions for a particular KRSP. However, it is envisioned that these extensions will be for optional use only. KRSPs are expected to have reasonable defaults for all such extensions; this is to ensure that applications do not need to be aware of specific KRSP profile extensions in order to get basic key recovery enablement services from a KRSP. Whenever the extensions field is set to NULL, the defaults should be used by a KRSP.

16.3.3 Key Recovery Context

All operations performed by the KRSPs are performed within a *key recovery context*. A key recovery context is programmatically equivalent to a cryptographic context; however the attributes of a key recovery context are different from those of other cryptographic contexts. There are three kinds of key recovery contexts— registration contexts, enablement contexts and recovery request contexts. A key recovery context contains state information that is necessary to perform key recovery operations. When the KR-API functions are invoked by application layer code, the KRMM passes the appropriate key recovery context to the KRSP using the KR-SPI function parameters.

A key recovery registration context contains no special attributes. A key recovery enablement context maintains information about the profiles of the local and remote parties for a cryptographic association. When the KR-API function to create a key recovery enablement context is invoked, the key recovery profiles for the specified communicating peers are specified by the application layer code using the API parameters. A key recovery request context maintains a set of key recovery fields, which are being used to perform a recovery request operation, and a set of flags that denotes the operational scenario of the recovery request operation. Since the establishment of a context implies the maintaining of state information within the CSSM, contexts acquired should be released as soon as their need is over.

16.3.4 Key Recovery Policy

The CSSM enforces the applicable key recovery policy on all cryptographic operations. There are two key recovery policies enforced by the CSSM, a law enforcement (LE) key recovery policy, and the enterprise (ENT) key recovery policy. Since the requirements for these two mandatory key recovery scenarios are somewhat different, they are implemented by different mechanisms within the CSSM.

The law enforcement key recovery policy is predefined (based on the political jurisdictions of manufacture and use of the cryptographic product) for a given product. The parameters on which the policy decision is made are predefined as well. Thus, the LE key recovery policy is implemented using a key recovery policy table and a key recovery policy enforcement function, both of which are used by the CSSM in making a key recovery policy decision. The LE policy table is implemented as a separate physical file for ease of implementation and upgrade (as law enforcement policies evolve over time); however, this file is protected using the same integrity mechanisms as the CSSM module.

The ENT key recovery policy, could vary anywhere between being set to NULL, and being very complex (for example, based on parameters such as time of day.) Enterprises are allowed total flexibility with respect to the enterprise key recovery policy. The enterprise policy is implemented within the CSSM by invoking a key recovery policy function that is defined by the enterprise administrator. The KR-API provides a function that allows an administrator to specify the name of a file that contains the enterprise key recovery policy function. The first time this function is used, the administrator can establish a passphrase for all subsequent calls on this function. This mechanism assures a level of access control on the enterprise policy, once a policy function has been established. It goes without saying that the file containing the policy function should be protected using the maximal possible protection afforded by the operating system platform. The actual structure of the policy function file is operating system platform-specific.

Every time a cryptographic context handle is returned to application layer code, the CSSM enforces the LE and ENT key recovery policies. For the LE policy, the CSSM policy enforcement function and the LE policy table are used. For the ENT policy, the ENT policy function file is invoked in an operating system platform-specific way. If the policy check determines that key recovery enablement is required for either LE or ENT scenarios, then the context is flagged as unusable, otherwise, the context is flagged as usable. An unusable context handle becomes flagged as usable only after the appropriate key recovery enablement operation is completed using that context handle. A usable context handle can then be used to perform cryptographic operations.

16.3.5 Key Recovery Enablement Operations

The CSSM key recovery enablement operations comprise the generation and processing of key recovery fields. Within a cryptographic association, key recovery field generation is performed by the sending side; key recovery field processing is performed on the receiving side to ensure that the integrity of the recovery fields have been maintained in transmission between the sending and receiving sides. These two vital operations are performed via the *CSSM_KR_GenerateRecoveryFields*( ) and the *CSSM_KR_ProcessRecoveryFields*( ) functions, respectively. These functions are covered summarily in a subsequent section of this chapter.

The key recovery fields generated by the CSSM potentially comprise three sub-fields, for law enforcement, enterprise, and individual key recovery scenarios, respectively. The law enforcement and enterprise key recovery sub-fields are generated when the law enforcement and enterprise usability flags are appropriately set in the cryptographic context used to generate the key recovery fields. When an application invokes the API function to generate the key recovery fields, a certain flag value is set indicating the fields have been generated. The processing of the key recovery fields only applies to the law enforcement and enterprise key recovery sub-fields; the individual key recovery sub-fields are ignored by the key recovery fields processing function.

16.3.6 Key Recovery Registration and Request Operations

The CSSM also supports the operations of registration and recovery requests. The KRSP exchanges messages with the appropriate key recovery agent/server to obtain the results required. If additional inputs are required for the completion of the operation, the supplied callback may be used by the KRSP. The recovery request operation can be used to request a batch of recoverable keys . The result of the registration operation is a key recovery profile data structure, while the results of a recovery request operation are a set of recovered keys.

Chapter 17

Key Recovery APIs

17.1 Module Management Operations

The generic CSSM module management functions are used to install and attach a Key Recovery add-in service module. These functions are specified in detail in the *CSSM Application Programming Interface*. The applicable generic management functions include:

- CSSM_ModuleInstall
- CSSM_ModuleUninstall
- CSSM_ListModules
- CSSM_ModuleAttach
- CSSM_ModuleDetach
- CSSM_GetModuleInfo
- CSSM_FreeModuleInfo

The new management function, CSSM_KR_SetEnterpriseRecoveryPolicy, is directly supported by the Key Recovery Module Manager.

CSSM_DATA_PTR CSSMAPI CSSM_KR_SetEnterpriseRecoveryPolicy( )
: Establishes the filename which contains the enterprise-based key recovery policy function for use by the KRMM in CSSM.

17.2 Key Recovery Context Operations

CSSM_BOOL CSSMAPI CSSM_KR_CreateRecoveryRegistrationContext( )
: Accepts as input the handle to the KRSP and returns a handle to a key recovery registration context. This context must be used when registering with a key recovery server or agent.

CSSM_DATA_PTR CSSMAPI CSSM_KR_CreateRecoveryEnablementContext( )
: Accepts as input the handle to the KRSP and the key recovery profiles of the local and remote parties, and returns a handle to the key recovery context for the given parties under the key recovery mechanism specified.

CSSM_BOOL CSSMAPI CSSM_KR_CreateRecoveryRequestContext( )
: Accepts as input the handle to the KRSP, the key recovery fields (from which the key is to be recovered), and the profile of the local party, and returns a handle to the key recovery context for the given party and key recovery fields.

CSSM_DATA_PTR CSSMAPI CSSM_KRPolicyInfo( )
: Returns the key recovery policy information pertaining to a given cryptographic context.

17.3 Key Recovery Registration Operations

CSSM_RETURN CSSMAPI CSSM_KR_RegistrationRequest( )

Performs a recovery registration request operation. A callback may be supplied to allow the registration operation to query for additional input information, if necessary. The result of the registration request operation is a reference handle that may be used to invoke the CSSM_KR_RegistrationRetrieve function.

CSSM_RETURN CSSMAPI CSSM_KR_RegistrationRetrieve( )

Completes a recovery registration operation. The result of the registration operation is returned in the form of a key recovery profile.

17.4 Key Recovery Enablement Operations

CSSM_RETURN CSSMAPI CSSM_KR_GenerateRecoveryFields( )

Accepts as input the key recovery context handle, the session-based recovery parameters and the cryptographic context handle, and several other parameters of relevance to the KRSP, and outputs a buffer of the appropriate mechanism-specific key recovery fields in a format defined and interpreted by the specific KRSP involved. It returns a cryptographic context handle, which can be input to the encryption APIs in the cryptographic framework.

CSSM_RETURN CSSMAPI CSSM_KR_ProcessRecoveryFields( )

Accepts as input the key recovery context handle, the cryptographic context handle, several other parameters of relevance to a KRSP, and the unparsed buffer of key recovery fields. It returns with a cryptographic context handle, which can then be used for the decryption APIs in the cryptographic framework.

17.5 Key Recovery Request Operations

CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRequest( )

Performs a recovery request operation for one or more recoverable keys. A callback may be supplied to allow the recovery request operation to query for additional input information, if necessary. The result of the recovery request operation is a results handle that may be used to obtain each recovered key and its associated meta information using the CSSM_KR_GetRecoveredObject function.

CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRetrieve( )

Completes a recovery request operation for one or more recoverable keys. The result of the recovery operation is a results handle that may be used to obtain each recovered key and its meta information using the CSSM_KRGetRecoveredObject function.

CSSM_RETURN CSSMAPI CSSM_KR_GetRecoveredObject( )

Retrieves a single recovered key and its associated meta information.

CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRequestAbort( )

Terminates a recovery request operation and releases any state information associated with it.

17.6 Extensibility Functions

CSSM_RETURN CSSMAPI CSSM_KR_PassThrough( )

Accepts as input an operation ID and an arbitrary set of input parameters. The operation ID may specify any type of operation the KR wishes to export. Such operations may include queries or services specific to the key recovery mechanism implemented by the KR module.

17.7 An Example Application Using Key Recovery APIs

To understand the role of key recovery in encrypted data communication, consider the following scenario, illustrated in Figure 17-1. A communication protocol running on behalf of party A sends an encrypted message to its counterpart running on behalf of party B. To encrypt/decrypt message data, the communication protocol implementations use the CSSM APIs as follows:

- A invokes "CSSM_CSP_CreateSymmetricContext" and obtains a cryptographic context handle (HA1) representing the encryption key.
- A invokes the "CSSM_EncryptData" API and provides the cryptographic context handle (HA1) as a parameter along with the message to be encrypted.
- A obtains the encrypted message and sends it to B. A also sends B the data key via the key exchange mechanism. The encrypted message can be intercepted by law-enforcement agencies.
- B obtains the data key from A through the key exchange mechanism and invokes the "CSSM_CSP_CreateSymmetricContext" to obtain a cryptographic context handle (HB1) representing the encryption key used by A.
- B invokes the "CSSM_DecryptData" and provides the key handle (HB1) as a parameter along with the message to be decrypted.
- B obtains the decrypted message sent by A.

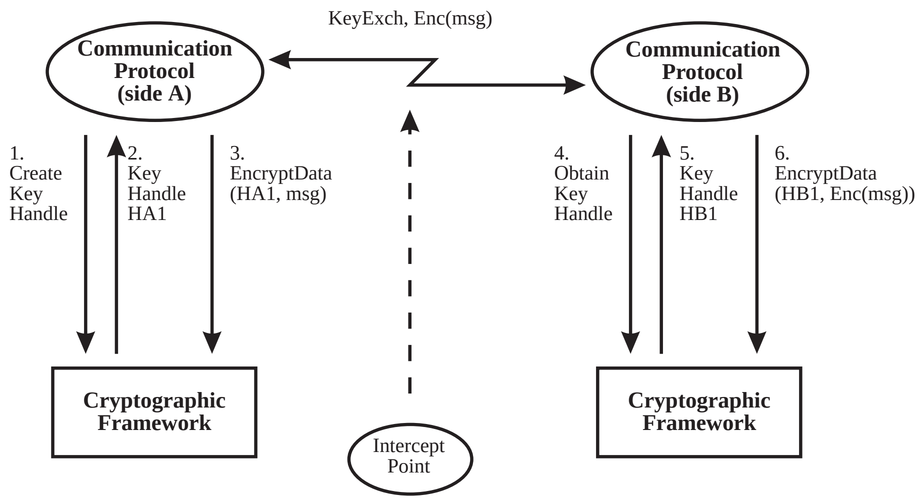

Figure 17-1 Encrypted Communications without Key Recovery

In the above scenario, after the key handles (and keys) are destroyed there is no practical way to decipher the contents of the encrypted message A sent to B by any law-enforcement agency. If good or strong encryption is used, deciphering the encrypted message is impractical (for example, either too expensive or impossible to decipher in useful time). Hence, key recovery techniques must be employed.

To illustrate the use of key recovery, we modify the scenario of Figure 17-1 to take advantage of KR-API functions, as illustrated in Figure 17-2. The CSSM ensures that key recovery can be performed using the messages being passed between A and B, as seen from the intercept point.

- A invokes the "CSSM_CSP_CreateSymmetricContext" and obtains a cryptographic context handle (HA1) representing the encryption key. In contrast to the previous scenario (Fig. 1(a)) where A could use the handle HA1 to encrypt the message, here, the direct use of key handle HA1 would be rejected by the "CSSM_EncryptData". The encrypt API will only accept a separate cryptographic context handle generated by the CSSM.
- A invokes the "CSSM_KR_GenerateRecoveryFields" to obtain the new cryptographic context handle, HA2, that would be used for encryption. The "CSSM_KR_GenerateRecoveryFields" also generates a set of key recovery fields that are returned along with the HA2 to A.

  Note that this is a simplified example. In reality, the "CSSM_KR_GenerateRecoveryFields" function requires a key recovery context handle in addition to the cryptographic context handle.
- A invokes the "CSSM_EncryptData" and provides as parameters the cryptographic context handle (HA2), and the message to be encrypted.
- A obtains the encrypted message and KR fields, and sends them to B. The data key is also sent to B using the key exchange mechanism. The encrypted message and KR fields can be intercepted by law enforcement agencies.
- B retrieves the data key using the key exchange mechanism and invokes the "CSSM_CSP_CreateSymmetricContext" to obtain a cryptographic context handle (HB1) for the encryption key used by A. In contrast to the previous scenario (Fig. 1(a)) where B could use the handle HB1 to decrypt the message, here the direct use of HB1 would be rejected by the decrypt operation. The decrypt will only accept a separate cryptographic context handle generated by the CSSM.

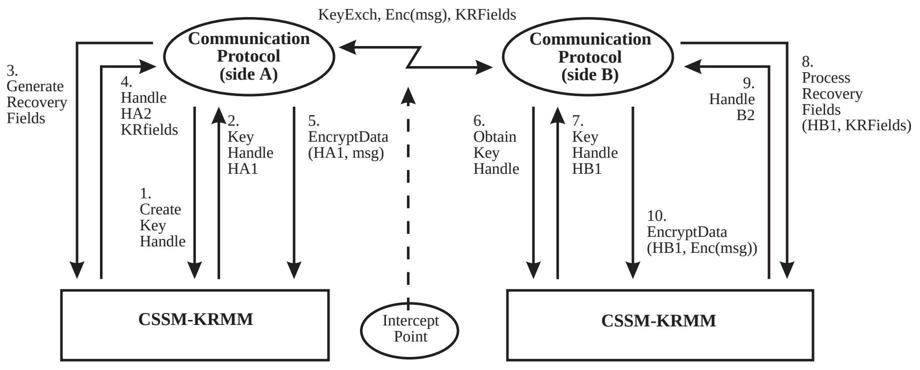
KeyExch, Enc(msg), KRFields
Communication Protocol (side A)
Communication Protocol (side B)
3. Generate Recovery Fields
4. Handle HA2 KRfields
2. Key Handle HA1
5. EncryptData (HA1, msg)
1. Create Key Handle
6. Obtain Key Handle
7. Key Handle HB1
9. Handle B2
8. Process Recovery Fields (HB1, KRFields)
10. EncryptData (HB1, Enc(msg))
CSSM-KRMM
Intercept Point
CSSM-KRMM

Figure 17-2 Encrypted Communications with Key Recovery Enablement

- B invokes the "CSSM_KR_ProcessRecoveryFields" of the CSSM and provides the handle (HB1) as a parameter along with the KR fields to be processed. If the recovery fields process correctly, a new cryptographic context handle HB2 is returned, which B must use to decrypt the message. Note that without processing the KR fields, B could not obtain handle HB2 and, consequently, could not decrypt the message.

Note that this is a simplified example. In reality, the "CSSM_KR_ProcessRecoveryFields" function requires a key recovery context handle in addition to the cryptographic context handle.

- B invokes the "CSSM_DecryptData" and provides the handle (HB2) as a parameter along with the message to be decrypted.
- B obtains the decrypted message sent by A.
- law enforcement picks up the recovery fields and obtains the key used by A and B with the help of one or more trusted third parties. To do so, law enforcement must authenticate itself to the recovery service, must present the KR fields and must demonstrate that it has the legal credentials (for example, Court warrant) for recovering the key.

The second scenario discussed above points out one of the salient features of the CSSM, namely that a key cannot be used to encrypt or decrypt a message without mediation by the CSSM. Hence, the CSSM cannot be bypassed.

17.8 Data Structures

17.8.1 CSSM_KR_HANDLE

This data structure represents the key recovery module handle. The handle value is a unique pairing between a key recovery module and an application that has attached that module. KR handles can be returned to an application as a result of the CSSM_ModuleAttach function.

```
typedef uint32 CSSM_KRSP_HANDLE /* Key Recovery Service
                                    Provider Handle */
```

17.8.2 CSSM_KR_NAME

This data structure contains a typed name. The namespace type specifies what kind of name is contained in the third parameter.

```
typedef struct cssm_kr_name {
     uint8 type; /* namespace type */
     uint8 length; /* name string length */
     char *name; /* name string */
} CSSM_KR_NAME
```

Definition

type
    The type of the key recovery name space.

length
    The length of the name (in bytes).

name
    The name represented in a string.

17.8.3 CSSM_KR_PROFILE

This data structure encapsulates the key recovery profile for a given user and a given key recovery mechanism.

```
typedef struct cssm_kr_profile {
    CSSM_KR_NAME UserName; /* name of the user */
    CSSM_DATA_PTR UserCertificate; /* public key certificate
                                    of the user */

    uint8 LE_KRANum; /* number of KRA cert chains in the
                                    following list */
    CSSM_CERT_LIST_PTR LE_KRACertChainList; /* list of Law
                         enforcement KRA certificate chains*/

    uint8 ENT_KRANum; /* number of KRA cert chains in the
                                    following list */
    CSSM_CERT_LIST_PTR ENT_KRACertChainList; /* list of
                          Enterprise KRA certificate chains*/

    CSSM_DATA_PTR ENTAuthenticationInfo; /* authentication
                   information for enterprise key recovery */

    uint8 INDIV_KRANum; /* number of KRA cert chains in the
                                            following list */
    CSSM_CERT_LIST_PTR INDIV_KRACertChainList; /* list of
                          Individual KRA certificate chains*/

    CSSM_DATA_PTR INDIVAuthenticationInfo; /* authentication
                   information for individual key recovery */

    uint32 KRFlags;   /* flag values to be interpreted by KRSP */

    CSSM_DATA_PTR Extensions; /* reserved for extensions specific
                                                  to KRSPs */

} CSSM_KR_PROFILE, *CSSM_KR_PROFILE_PTR;
```

Definition

UserName
The user's name.

UserCertificate
The user's certificate chain, used for identity and authentication when performing policy evaluation.

LE_KRANum
The number of LE Key Recovery agents in the following list.

LE_KRACertChainList
A list of certificate chains, one per Key Recovery Agent authorized for LE key recovery.

ENT_KRANum
The number of ENT Key Recovery agents in the following list.

ENT_KRACertChainList
A list of certificate chains, one per Key Recovery Agent authorized for ENT key recovery.

ENTAuthenticationInfo
Authentication information to be used for ENT key recovery.

INDIV_KRANum
The number of INDIV Key Recovery agents in the following list.

INDIV_KRACertChainList
A list of certificate chains, one per Key Recovery Agent authorized for INDIV key recovery.

INDIVAuthenticationInfo
Authentication information to be used for INDIV key recovery.

KRFlags
A bit mask specifying the user's selected service options specific to the selected key recovery service module.

Extensions
Reserved for future use.

17.8.4 CSSM_EXEMPTION_MASK

The Key Recovery Module Manager defines these CSSM_EXEMPTION_MASK flags in addition to those defined by CSSM and other CSSM module managers. These flags represent exemption from specified, built-in checks performed by the KRMM. Authorized applications use the CSSM_RequestCssmExemption function to request exemptions. Exemption is granted if the application's credentials can be authenticated by CSSM based on selected roots of trust.

```
typedef uint32 CSSM_EXEMPTION_MASK

#define CSSM_EXEMPT_LE_KR 0x00000004   /* ask exemption from LE
                                       key recovery */
#define CSSM_EXEMPT_ENT_KR 0x00000008  /* ask exemption from ENT
                                       key recovery */
```

17.8.5 CSSM_CERT_LIST

This data structure encapsulates a generic list of items.

```
typedef struct cssm_cert_list {
    uint32 NumberCerts;
    CSSM_DATA_PTR CertList;
} CSSM_CERT_LIST, *CSSM_CERT_LIST_PTR;
```

Definition

NumberCerts
Count of the number of certs in the list.

CertList
Pointer to a list of certificate items.

17.8.6 CSSM_CONTEXT_ATTRIBUTE Extensions

The key recovery context creation operations return key recovery context handles that are represented as cryptographic context handles. In order to use the CSSM_CONTEXT data structure to implement key recovery contexts, the CSSM_CONTEXT will be used to hold new types of attributes, as shown below:

```
typedef struct cssm_context_attribute {
    uint32 AttributeType;
    uint32 AttributeLength;
    union cssm_context_attribute_value {
      char *String;
      uint32 Uint32;
      CSSM_CRYPTO_DATA_PTR Crypto;
      CSSM_KEY_PTR Key;
      CSSM_DATA_PTR Data;
      CSSM_DATE_PTR Date;
      CSSM_RANGE_PTR Range;
      CSSM_KR_PROFILE_PTR KRProfile;
    } Attribute;
    } CSSM_CONTEXT_ATTRIBUTE, *CSSM_CONTEXT_ATTRIBUTE_PTR;
```

All but the last member of the union above are part of the core *CSSM Application Programming Interface*. The descriptions of these basic fields and members are in the *CSSM Application Programming Interface*. The KRProfile member of the union has been added specifically to support key recovery contexts, and is described below.

Definition

KRProfile
A pointer to the key recovery profile structure that defines the user parameters with respect to the key recovery process.

17.8.7 CSSM_ATTRIBUTE_TYPE Additions

Several new attribute types were defined to support the key recovery context attributes. The following definitions are added to the enumerated type CSSM_ATTRIBUTE_TYPE:

```
CSSM_ATTRIBUTE_KRPROFILE_LOCAL   = CSSM_ATTRIBUTE_LAST+1,
                /* local entity profile */
CSSM_ATTRIBUTE_KRPROFILE_REMOTE  = CSSM_ATTRIBUTE_LAST+2,
                /* remote entity profile */
```

17.8.8 CSSM_KRSUBSERVICE

Two structures are used to contain all of the static information that describes a key recovery add-in module: the krinfo structure and the krsubservice structure. This descriptive information is securely stored in the CSSM registry when the KR module is installed with CSSM. A key recovery module may implement multiple types of services and organize them as sub-services. For example, a KR module supporting an encapsulation mechanism and an escrow mechanism may organize its implementation as two subservices.

The descriptive information stored in these structures can be queried using the function **CSSM_GetModuleInfo()** and specifying the key recovery module GUID.

```
typedef struct cssm_krsubservice {
    uint32 SubServiceId;
    char *Description;      /* Description of this sub service */
    CSSM_CALLER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
} CSSM_KRSUBSERVICE, *CSSM_KRSUBSERVICE_PTR;
```

Definition

SubServiceId
A unique, identifying number for the sub-service described in this structure.

Description
A string containing a descriptive name or title for this sub-service.

AuthenticationMechanism
An enumerated value defining the credential format accepted by the KR module. When an authentication credential is required by a KR function, the presented credentials must be of the required format.

17.8.9 CSSM_KRINFO

Two structures are used to contain all of the static information that describes a key recovery add-in module: the krinfo structure and the krsubservice structure. This descriptive information is securely stored in the CSSM registry when the KR module is installed with CSSM. A key recovery module may implement multiple types of services and organize them as sub-services. For example, a KR module supporting an encapsulation mechanism and an escrow mechanism may organize its implementation as two subservices.

The descriptive information stored in these structures can be queried using the function **CSSM_GetModuleInfo()** and specifying the key recovery module GUID.

```
typedef struct cssm_krinfo {
    CSSM_VERSION Version;  /* major and minor version number */
    char *Description;     /* Detailed description of this KR */
    char *Vendor;          /* KRSP Vendor name */
    char *Jurisdiction;    /* Home jurisdiction of the
                                              KRSP installation */
    uint32 NumberSubService;
    CSSM_KRSUBSERVICE_PTR SubService;
} CSSM_KRINFO, *CSSM_KRINFO_PTR;
```

Definition

Version
The major and minor version number of the add-in module.

Description
A character string containing a general description of this key recovery module.

Vendor
A character string containing the name of the vendor who implemented and manufactured this key recovery module.

Jurisdiction
A character string describing the geographical region where the key recovery module is installed.

NumberOfSubServices
The number of sub-services implemented by this key recovery module. Every KR module implements at least one sub-service.

Subservices
A pointer to an array of sub-service structures. Each structure contains detailed information about that sub-service.

17.9 Key Recovery Module Management Operations

The manpages for Key Recovery Module Management Operations follow on the next page.

NAME

CSSM_KR_SetEnterpriseRecoveryPolicy

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_SetEnterpriseRecoveryPolicy
    (const CSSM_DATA_PTR RecoveryPolicyFileName,
    const CSSM_CRYPTO_DATA_PTR OldPassPhrase)
    const CSSM_CRYPTO_DATA_PTR NewPassPhrase)
```

DESCRIPTION

This call establishes the identity of the file that contains the enterprise key recovery policy function. The first time this function is invoked, the old passphrase is established for access control purposes. Subsequent invocations of this function will require the original passphrase to be supplied in order to update the filename of the policy function. Optionally the passphrase can be changed from the oldpassphrase to the newpassphrase on subsequent invocations.

The policy function module is operating system platform specific (for Windows 95 and Windows NT, it may be a DLL, for UNIX platforms, it may be a separate executable which gets launched by the KRMM. It is expected that the policy function file will be protected using the available protection mechanisms of the operating system platform. The policy function is expected to conform to the following interface:

boolean EnterpriseRecoveryPolicy(CSSM_CONTEXT CryptoContext);

The Boolean return value of this policy function will determine whether enterprise-based key recovery is mandated for the given cryptographic operation.

PARAMETERS

RecoveryPolicyFileName (input)
A pointer to a CSSM_DATA structure that contains the file name of the module that contains the enterprise key recovery policy function. The filename may be a fully qualified pathname or a partial pathname.

OldPassPhrase (input)
The current, active passphrase that controls access to this operation.

NewPassPhrase (input/optional)
A new passphrase that becomes the current, active passphrase after the execution of this function. It must be used to control access to future invocations of this operation.

RETURN VALUES

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_KR_INVALID_FILENAME
Invalid policy file name.

CSSM_MEMORY_ERROR
Memory error.

17.10 Key Recovery Context Operations

Key recovery contexts are essentially cryptographic contexts. The following API functions deal with the creation of these special types of cryptographic contexts. Once these contexts are created, the regular CSSM context API functions may be used to manipulate these key recovery contexts.

NAME

CSSM_KR_CreateRecoveryRegistrationContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_KR_CreateRecoveryRegistrationContext
    (CSSM_KRSP_HANDLE KRSPHandle)
```

DESCRIPTION

This call creates a key recovery registration context based on a KRSP handle (which determines the key recovery mechanism that is in use). This context may be used for performing registration with key recovery servers and/or agents.

PARAMETERS

KRSPHandle (input)
The handle to the KRSP that is to be used.

RETURN VALUES

A handle to the key recovery registration context is returned. If the handle is NULL, it signifies that an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid handle.

CSSM_MEMORY_ERROR
Memory error.

NAME

CSSM_KR_CreateRecoveryEnablementContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_KR_CreateRecoveryEnablementContext
    (CSSM_KRSP_HANDLE KRSPHandle,
    const CSSM_KR_PROFILE LocalProfile,
    const CSSM_KR_PROFILE RemoteProfile)
```

DESCRIPTION

This call creates a key recovery enablement context based on a KRSP handle (which determines the key recovery mechanism that is in use), and key recovery profiles for the local and remote parties involved in a cryptographic exchange. A handle to the key recovery enablement context is returned. It is expected that the LocalProfile will contain sufficient information to perform LE, ENT and IND key recovery enablement, whereas the RemoteProfile will contain information to perform LE and ENT key recovery enablement only. However, any and all of the fields within the profiles may be set to NULL—in this case, default values for these fields are to be used when performing the recovery enablement operations.

PARAMETERS

KRSPHandle (input)
The handle to the KRSP that is to be used.

LocalProfile (input)
The key recovery profile for the local client.

RemoteProfile (input)
The key recovery profile for the remote client.

RETURN VALUES

A handle to the key recovery enablement context is returned. If the handle is NULL, it signifies that an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid handle.

CSSM_KR_INVALID_PROFILE
Invalid profile structure.

CSSM_KR_INVALID_PTR
Bad pointer.

CSSM_MEMORY_ERROR
Memory error.

NAME

CSSM_KR_CreateRecoveryRequestContext

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_KR_CreateRecoveryRequestContext
    (CSSM_KRSP_HANDLE KRSPHandle,
    const CSSM_KR_PROFILE LocalProfile)
```

DESCRIPTION

This call creates a key recovery request context based on a KRSP handle (which determines the key recovery mechanism that is in use) and the profile for the local client. A handle to the key recovery request context is returned.

PARAMETERS

KRSPHandle (input)
The handle to the KRSP that is to be used.

LocalProfile (input)
The key recovery profile for the local client. This parameter is relevant only when the KRFlags value is set to KR_INDIV.

RETURN VALUES

A handle to the key recovery context is returned. If the handle is NULL, it signifies that an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid handle.

CSSM_KR_INVALID_PROFILE
Invalid profile.

CSSM_MEMORY_ERROR
Memory error.

NAME

CSSM_KRPolicyInfo

SYNOPSIS

```
CSSM_RETURN CSSM_KRPolicyInfo
    (CSSM_CC_HANDLE CCHandle,
    CSSM_BOOL *LE_KRFlag,
    CSSM_BOOL *ENT_KRFlag,
    uint32 *LE_WorkFactor)
```

DESCRIPTION

This call returns the key recovery policy information for a given cryptographic context. The information returned constitutes the key recovery extension fields of a cryptographic context.

PARAMETERS

CCHandle (input)
The handle to the cryptographic context that is to be used.

LE_KRFlag (output)
The usability flag for law enforcement key recovery. Possible values are:

- TRUE—signifies that law enforcement key recovery enablement needs to be done
- FALSE—signifies that law enforcement key recovery enablement is either not required or has already been done.

ENT_KRFlag (output)
The usability flag for enterprise key recovery. Possible values are:

- TRUE—signifies that enterprise key recovery enablement needs to be done
- FALSE—signifies that enterprise key recovery enablement is either not required or has already been done.

LE_WorkFactor (output)
The workfactor value to use for law enforcement key recovery.

RETURN VALUES

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_KR_INVALID_CC_HANDLE
Invalid crypto context handle.

CSSM_MEMORY_ERROR
Memory error.

17.11 Key Recovery Registration Operations

The manpages for Key Recovery Registration Operations follow on the next page.

NAME

CSSM_KR_RegistrationRequest

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_RegistrationRequest
    (CSSM_CC_HANDLE RecoveryRegistrationContext,
    CSSM_DATA_PTR KRInData,
    CSSM_CRYPTO_DATA_PTR UserCallback,
    Uint8 KRFlags,
    uint32 *EstimatedTime,
    CSSM_HANDLE_PTR ReferenceHandle )
```

DESCRIPTION

This function initiates a key recovery registration operation. The KRInData contains known input parameters for the recovery registration operation. A UserCallback function can be supplied to allow the registration operation to interact with the user interface, if necessary.

This function returns a ReferenceHandle and an EstimatedTime for completion of the request. The ReferenceHandle must be used to retrieve the registration result using the **CSSM_KR_RegistrationRetrieve()** function after the EstimatedTime has elapsed. The return value for this function indicates whether the request was successfully initiated.

PARAMETERS

RecoveryRegistrationContext (input)
The handle to the key recovery registration context.

KRInData (input)
Input data for key recovery registration.

UserCallback (input/optional)
A callback function that may be used to collect further information from the user interface.

KRFlags (input)
Flag values for recovery registration. Defined values are:

- KR_INDIV—registration for individual key recovery
- KR_ENT—registration for enterprise key recovery
- KR_LE—registration for law enforcement key recovery

EstimatedTime (output)
The estimated time after which this call should be repeated to obtain registration results. This is set to a non-zero value only when the KRProfile parameter is NULL.

ReferenceHandle (output)
A handle that references the outstanding registration request. This handle must be used to retrieve the registration result using the function CSSM_KR_RegistrationRetrieve.

RETURN VALUES

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_KR_INVALID_HANDLE
Invalid registration handle.

CSSM_KR_INVALID_POINTER
Invalid pointer.

CSSM_MEMORY_ERROR
Memory error.

NAME

CSSM_KR_RegistrationRetrieve

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_RegistrationRetrieve
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE ReferenceHandle
    uint32 *EstimatedTime
    CSSM_KR_PROFILE_PTR KRProfile)
```

DESCRIPTION

This function completes a key recovery registration operation by returning the profile information generated as a result of a successful key recovery registration process. It is possible that the key recovery registration process has not yet completed. In this case, the returned EstimatedTime is the updated estimate for completion of the registration procedure. If the profile pointer is NULL and the estimated time is greater than zero, the caller should repeat this call after the specified time to retrieve the profile structure.

PARAMETERS

KRSPHandle (input)
The handle of the KR module to perform this operation.

ReferenceHandle (input)
The handle that specifies the corresponding call to CSSM_KR_RegistrationRequest, which initiated the key recovery registration procedure.

EstimatedTime (output)
The estimated time after which this call should be repeated to obtain registration results. This is set to a non-zero value only when the KRProfile result is NULL.

KRProfile (output)
The key recovery profile that is filled in by the registration operation.

RETURN VALUES

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_KR_INVALID_HANDLE
Invalid reference handle.

CSSM_MEMORY_ERROR
Memory error.

17.12 Key Recovery Enablement Operations

The manpages for Key Recovery Enablement Operations follow on the next page.

NAME

CSSM_KR_GenerateRecoveryFields

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_KR_GenerateRecoveryFields
    (CSSM_CC_HANDLE KeyRecoveryContext,
    CSSM_CC_HANDLE CryptoContext,
    CSSM_DATA_PTR KRSPOptions,
    uint32 KRFlags,
    CSSM_DATA_PTR KRFields)
```

DESCRIPTION

This function generates the key recovery fields for a cryptographic association given the key recovery context, the session specific key recovery attributes, and the handle to the cryptographic context containing the key that is to be made recoverable. The session attributes and the flags are not interpreted at the KRMM layer. A non-NULL cryptographic context handle is returned if the key recovery field generation was successful. This returned handle can be used for the encrypt APIs of the CSSM. The generated key recovery fields are returned as an output parameter. The KRFlags parameter may be used to fine tune the contents of the KRFields produced by this operation.

PARAMETERS

KeyRecoveryContext (input)
The handle to the key recovery context for the cryptographic association.

CryptoContext (input)
The cryptographic context handle that points to the session key.

KRSPOptions (input)
The key recovery service provider specific options. These options are not interpreted by the KRMM, but passed on to the KRSP.

KRFlags (input)
Flag values for key recovery fields generation. Defined values are:

- KR_INDIV—signifies that only the individual key recovery fields are to be generated
- KR_ENT—signifies that only the enterprise key recovery fields are to be generated
- KR_LE—signifies that only the law enforcement key recovery fields are to be generated
- KR_OPTIMIZE—signifies that performance optimization options are to be adopted by a KRSP while implementing this operation
- KR_DROP_WORKFACTOR—signifies that the key recovery fields should be generated without using the key size work factor.

KRFields (output)
The key recovery fields in the form of an uninterpreted data blob.

RETURN VALUES

A cryptographic context handle is returned. This handle is NULL if the generation of the key recovery fields was not successful.

ERRORS

CSSM_KR_INVALID_CC_HANDLE
Invalid crypto context handle.

CSSM_KR_INVALID_KRC_HANDLE
Invalid key recovery context handle.

CSSM_KR_INVALID_OPTIONS
Invalid recovery options.

CSSM_MEMORY_ERROR
Memory error.

NAME

CSSM_KR_ProcessRecoveryFields

SYNOPSIS

```
CSSM_CC_HANDLE CSSMAPI CSSM_KR_ProcessRecoveryFields
    (CSSM_CC_HANDLE KeyRecoveryContext,
    CSSM_CC_HANDLE CryptoContext,
    CSSM_DATA_PTR KRSPOptions,
    uint32 KRFlags,
    CSSM_DATA_PTR KRFields)
```

DESCRIPTION

This call processed a set of key recovery fields given the key recovery context, and the cryptographic context for the decryption operation, and returns a non-NULL cryptographic context handle if the processing was successful. The returned handle may be used for the decrypt API calls of the CSSM.

PARAMETERS

KeyRecoveryContext (input)
The handle to the key recovery context.

CryptoContext (input)
A handle to the cryptographic context for which the key recovery fields are to be processed.

KRSPOptions (input)
The key recovery service provider specific options. These options are not interpreted by the KRMM, but passed on to the KRSP.

KRFlags (input)
Flag values for key recovery fields processing. Defined values are:

- KR_ENT—signifies that only the enterprise key recovery fields are to be processed
- KR_LE—signifies that only the law enforcement key recovery fields are to be processed
- KR_ALL—signifies that all of the key recovery fields are to be processed
- KR_OPTIMIZE—signifies that performance optimization options are to be adopted by a KRSP while implementing this operation.

KRFields (input)
The key recovery fields to be processed.

RETURN VALUES

A cryptographic context handle for the session key is returned. This handle is NULL if the processing was unsuccessful.

ERRORS

CSSM_KR_INVALID_CC_HANDLE
Invalid crypto context handle.

CSSM_KR_INVALID_KRC_HANDLE
Invalid key recovery context handle.

CSSM_KR_INVALID_OPTIONS
Invalid recovery options.

CSSM_MEMORY_ERROR
Memory error.

17.13 Key Recovery Request Operations

The manpages for Key Recovery Request Operations follow on the next page.

NAME

CSSM_KR_RecoveryRequest

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRequest
    (CSSM_CC_HANDLE RecoveryRequestContext,
    const CSSM_DATA_PTR KRInData,
    const CSSM_CRYPTO_DATA_PTR UserCallback,
    uint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceHandle)
```

DESCRIPTION

This function initiates a key recovery request operation. The RecoveryRequestContext describes the operation to be performed. The KRInData contains known input parameters for the recovery request operation. A UserCallback function may be supplied to allow the recovery operation to interact with the user interface to obtain additional input, if necessary.

The results of a successful recovery operation are referenced by the ReferenceHandle parameter, which must be used with the CSSM_KR_RecoveryRetrieve function to obtain a cache of secured, recovered keys. The returned value of EstimatedTime specifies the amount of time the caller should wait before call the retrieve function.

PARAMETERS

RecoveryRequestContext (input)
The handle to the key recovery request context.

KRInData (input)
Input data for key recovery requests. For encapsulation schemes, the key recovery fields are included in this parameter.

UserCallback (input/optional)
A callback function that may be used to collect further information from the user interface.

EstimatedTime (output)
The estimated time after which the caller should invoke the CSSM_KR_RecoveryRetrieve function to obtain a cache of recovered keys.

ReferenceHandle (output)
Handle representing this outstanding recovery request. This handle should be used at input to the CSSM_KR_RecoveryRetrieve function.

RETURN VALUES

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid handle.

CSSM_KR_INVALID_HANDLE
Invalid recovery context handle.

CSSM_KR_INVALID_RECOVERY_CONTEXT
Invalid context value.

CSSM_KR_INVALID_POINTER
Invalid pointer.

CSSM_MEMORY_ERROR
Memory error.

NAME

CSSM_KR_RecoveryRetrieve

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRetrieve
    (CSSM_KR_HANDLE KRSPHandle,
    CSSM_HANDLE ReferenceHandle,
    uint32 *EstimatedTime,
    CSSM_HANDLE_PTR CacheHandle,
    uint32 *NumberOfRecoveredKeys)
```

DESCRIPTION

This function completes a key recovery request operation. The ReferenceHandle parameter indicates which outstanding recovery request is to be completed. The results of a successful recovery operation are referenced by the ResultsHandle parameter, which may be used with the CSSM_KR_GetRecoveredObject function to retrieve the recovered keys.

If the results are not available at the time this function is invoked, the CacheHandle is NULL, and the EstimatedTime parameter indicates when this operation should be repeated with the same ReferenceHandle.

PARAMETERS

KRSPHandle (input)
The handle of the KR module to perform this operation.

ReferenceHandle (input)
A reference handle which uniquely identifies the CSSM_KR_RecoveryRequest call that initiated recovery of the set of keys returned by this function.

EstimatedTime (output)
The number of seconds estimated before the set of recovered keys will be returned. A (default) value of zero indicates that the set has been returned as a result of this call.

CacheHandle (output)
A reference handle which uniquely identifies the cache of recovered keys. If the object retrieval process has not been completed, the returned cache handle is NULL. A non-NULL cache handle can be used in the CSSM_KR_GetRecoveredObject function to complete the recovery of an individual key.

NumberOfRecoveredKeys (output)
The number of keys in the cache.

RETURN VALUES

A CSSM_RETURN value indicating whether the operation returned a set of keys. If the result is CSSM_FAIL, and a NULL cache handle and a positive EstimatedTime are returned, then the calling application is expected to call this function again after the specified EstimatedTime. If the result is CSSM_FAIL and EstimatedTime is zero, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid KR handle.

CSSM_KR_INVALID_HANDLE
Invalid reference handle.

CSSM_MEMORY_ERROR
: Memory error.

CSSM_KR_FAIL
: Function failed.

NAME

CSSM_KR_GetRecoveredObject

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_GetRecoveredObject
    (CSSM_KR_HANDLE KRSPHandle,
    CSSM_HANDLE CacheHandle,
    uint32 IndexInResults,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_CRYPTO_DATA_PTR Passphrase,
    CSSM_KEY_PTR RecoveredKey,
    Uint32 Flags,
    CSSM_DATA_PTR OtherInfo)
```

DESCRIPTION

This function is used to step through the results of a recovery request operation in order to retrieve a single recovered key at a time along with its associated meta information. The cache handle returned from a successful CSSM_KR_RecoveryRetrieve operation is used. When multiple keys are recovered by a single recovery request operation, the index parameter indicates which item to retrieve through this function.

If the recovered key is a private key it is stored in the specified CSP secured by the passphrase. If the recovered key is a symmetric key it is returned to the caller in the RecoveredKey parameter. The OtherInfo parameter is used to return other meta data associated with the recovered key.

PARAMETERS

KRSPHandle (input)
The handle of the KR module to perform this operation.

CacheHandle (input)
The handle returned from a successful CSSM_KR_RecoveryRequest operation.

IndexInResults (input)
The index into the results that are referenced by the CacheHandle parameter.

CSPHandle (input/optional)
This parameter is used when recovering the private key in a keypair. This identifies the CSP that should store the recovered key.

Passphrase (input/optional)
This parameter is used when recovering the private key in a keypair. The passphrase is associated with the private key when it is securely stored in the specified CSP.

RecoveredKey (output)
This parameter is used when recovering a symmetric key. The recovered key is stored in the key structure provided by the caller.

Flags (input)
Flag values relevant for recovery of a key. Possible values are: CERT_RETRIEVE - if the recovered key is a private key, return the corresponding public key certificate in the OtherInfo parameter.

OtherInfo (output/optional)
Additional meta information can be associated with the recovered key. Any additional information is returned in this output parameter. The object is opaque and the caller must have knowledge of the expected structure of this result.

RETURN VALUES

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid KR Handle.

CSSM_KR_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_KR_INVALID_HANDLE
Invalid cache handle.

CSSM_KR_INVALID_INDEX
Cache index value is out of range.

CSSM_KR_PRIVATE_KEY_STORE_FAIL
Unable to store private key in CSP.

CSSM_MEMORY_ERROR
Not enough memory.

NAME

CSSM_KR_RecoveryRequestAbort

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_KR_RecoveryRequestAbort
    (CSSM_KR_HANDLE KRSPHandle,
    CSSM_HANDLE CacheHandle )
```

DESCRIPTION

This function terminates a recovery request operation. The function also destroys all intermediate state and secret information used during the key recovery process.

PARAMETERS

KRSPHandle (input)
The handle of the KR module to perform this operation.

CacheHandle (input)
The handle returned from a successful CSSM_KR_RecoveryRetrieve operation.

RETURN VALUES

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid KR handle.

CSSM_KR_INVALID_HANDLE
Invalid cache handle.

17.14 Extensibility Functions

The manpages for Extensibility Functions follow on the next page.

NAME

CSSM_KR_PassThrough

SYNOPSIS

```
CSSM_DATA_PTR CSSMAPI CSSM_KR_PassThrough
    (CSSM_KR_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KeyRecoveryContext,
    CSSM_CC_HANDLE CryptoContext,
    uint32 PassThroughId,
    const CSSM_DATA_PTR InputParams)
```

DESCRIPTION

This function allows applications to call key recovery module-specific operations that have been exported. Such operations may include queries or services specific to the recovery mechanism implemented by the KR module.

PARAMETERS

KRSPHandle (input)
The handle of the KR module to perform this operation.

KeyRecoveryContext (input/optional)
The handle that describes the context for the key recovery operation.

CryptoContext (input/optional)
The handle that describes the context for a cryptographic operation. The cryptographic context specifies the handle of the cryptographic service provider (CSP) that must be used to perform the operation. If no cryptographic context is specified, the KR module uses an assumed context, if required.

PassThroughId (input)
An identifier assigned by the KR module to indicate the exported function to perform.

InputParams (input)
A pointer to the CSSM_DATA structure containing parameters to be interpreted in a function-specific manner by the requested KR module. This parameter can be used as a pointer to an array of CSSM_DATA_PTRs.

RETURN VALUES

A pointer to the CSSM_DATA structure containing the output from the pass-through function. The output data must be interpreted by the calling application based on externally available information. If the pointer is NULL, an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid KR handle.

CSSM_KR_INVALID_CC_HANDLE
Invalid crypto context handle.

CSSM_KR_INVALID_KRC_HANDLE
Invalid key recovery context handle.

CSSM_KR_INVALID_OP_ID
Invalid operation ID.

CSSM_KR_INVALID_POINTER
Invalid pointer to input data.

CSSM_MEMORY_ERROR
: Error in allocating memory.

CSSM_KR_PASS_THROUGH_FAIL
: Unable to perform pass through.

CSSM_FUNCTION_NOT_IMPLEMENTED
: Function not implemented.

CAE Specification

Part 4:

CDSA Embedded Integrity Services Library API

The Open Group

Chapter 18

Introduction

18.1 Problem Statement

When attempting to establish a secure or trusted computing environment, the integrity of each software module in the environment must be verified. Digital signaturing and signature verification is a standard mechanism for demonstrating integrity and even authenticity (depending on the signing key). This is not a total solution. In a dynamic computing environment, modules are constantly being added to and removed from the environment. The verification process must be online and on-demand. Hence even when all modules are signed and signature verification is performed, there remains the question "Who is checking on the verifier?"

18.2 Extending Trust

To establish trust in a computing environment, it is essential to begin from a single trusted module and extend the perimeter of trust by verifying the integrity of each software module as it is added to the computing environment. One approach is to insert one or more integrity verification kernels (IVKs) into each module. The embedded IVK can verify digital signatures of itself and the module to improve the chances that any modification, whether accidental or malicious, can be detected prior to performing trusted operations within the scope of the module.

Cryptography is not useful in establishing a secure kernel. It assumes the existence of two secure end-points. It is assumed that the code signing environment is secure, by physical and software means. The problem is establishing a secure verification environment.

The starting point for verification should be one or more small kernels of code that are continually self-checking. This checking makes the IVKs more protected. They, in turn, are used to detect modification in the remainder of the program.

Many complex applications rely on dynamic linking to shared libraries to access program modules. These libraries are often created by diverse organizations and updated at asynchronous times. These libraries must be checked before they are added to the executing environment. It is also desirable to check these libraries after they are running in the system.

Checking is based on credentials. Credentials can also be used to convey authenticated attributes of the signing organization, the signed module, or even attributes of the signature itself. The software module can have some attributes, such as the version number or implementation restrictions, which are necessary for its partner modules. Finally, some attributes, such as the date and time when the signature was made, can be attributes of the signature itself.

A central authority with universal trust is not required. Each software organization can indicate which other organizations can produce trusted software by issuing certificates signed with its *digital signature*. Each module that evaluates credentials can contain the root public key, or keys, that it trusts. If it uses certificates as a means of introducing new partners, the number of vendors for partner modules need not be limited.

The security of these applications can be further enhanced by having IVKs in each module to check the integrity and credentials of other modules that it serves or that it uses to obtain services.

18.3 Why an Embedded Library?

The Embedded Integrity Services Library (EISL) is not extensible. It is intended to be implemented with position-independent code so that it can be used in constructing integrity verification kernels.

EISL implements a self-check procedure that verifies its own digital signature. The public key used for verification is embedded in the library code to avoid being easily modified.

The embedded integrity library contains the minimal set of services to locate partner modules and their credentials, verify credentials and obtain authenticated attributes, and securely link to partner modules. Because these services are used to establish trust in other modules, they must be statically bound to each module.

Once trusted contact has been established, a large, more general Integrity Services Library (ISL) can be used to implement the full range of integrity services. While compatible with the more general integrity library, the embedded integrity library is intended only to securely find other code modules and their attributes. Verification needs that exceed this scope should be met by the integrity services library.

18.4 A Phased Approach

The establishment of integrity between two dynamically loaded, executable objects proceeds in three phases:

- Self-check
- Bilateral authentication
- Extensible integrity services
- Secure linkage check

All three phases are discussed in greater detail in the *CSSM Add-in Module Structure and Administration Specification*. EISL defines APIs that support all three phases of the process to verify integrity between two objects dynamically loaded, executable objects.

18.4.1 Phase I. Establishing a Foothold: Self-Check

In the first phase, the self-check phase, the software module checks its own digital signature. The Embedded Integrity Services Library (EISL) defines a statically-linked library procedure to perform self-check.

18.4.2 Phase II. Finding our Friends: Bilateral Authentication

In the second phase, *bilateral authentication* routines in the EISL offer support for securely locating, verifying, and linking to partner software modules.

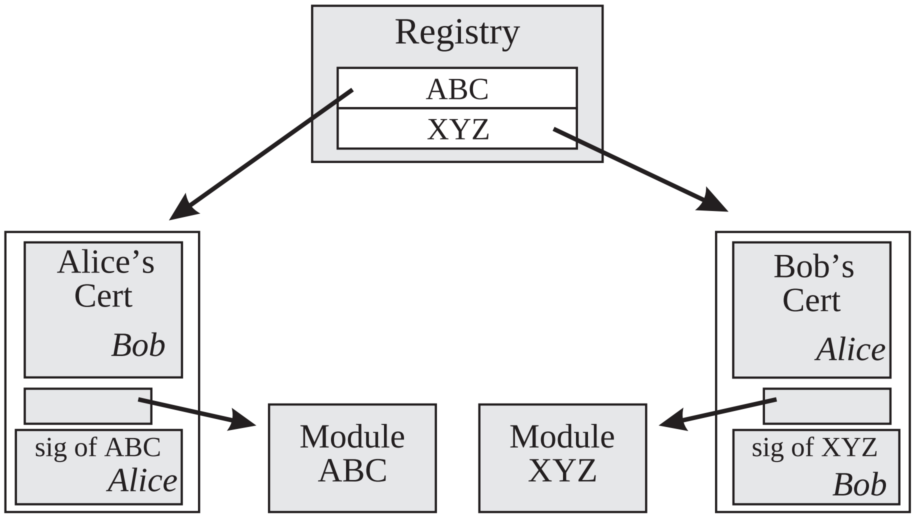

Figure 18-1 Bilateral Authentication Using Software Credentials

The process of bilateral authentication begins in the registry, where each program can find the credentials as well as the object code of the other.

Verification of the other module can be done prior to loading, or if it is already loaded, it can be verified in memory. Verification prior to loading prevents activating file viruses in infected modules. Verification in memory prevents *stealth* viral attacks where the file is healthy, but the loaded code is infected.

18.4.3 Phase III. Secure Linkage Check

Once verified, the programs can use the verified in-memory representation of the credentials to perform validity checks of addresses to provide secure linkage to modules. The addresses of both callers and procedures to be called can be verified using this facility.

18.5 Using Library Services

EISL defines a comprehensive set of services for extending the perimeter of trust based on integrity verification. EISL Users must make appropriate use of the library to obtain the full benefits of its services. This section discusses how to use the services defined by EISL.

18.5.1 Location of Modules and Credentials

The embedded integrity services library defines a service to *locate* a partner module in a central *registry*. This function assists applications in finding the module code as well as the credentials of a partner module. The credentials are *external* to the module's object code and publicly documented so that they can be verified by any party. Acceptable credentials are signed manifests and digital certificates. Each module must be issued a set of credentials as part of the module manufacturing process. Credentials consist of at least one digital certificate and one manifest. Over time, additional certificates can be added and the original manifest can be augmented with additional descriptions of the module. See the Signed Manifest Specification, Intel Architecture Lab, 1997, for an overview of manifests and their use in integrity verification.

While the credentials can be easily parsed and examined by the program directly, it is discouraged. External credentials are in a very public place, which allows multiple independent verifications, but they can therefore be easily modified between the time that they are verified and subsequent examination of them by the program. The library is intended to atomically retrieve, parse, and verify the credentials, and use (unspecified) methods to preserve the integrity of the attributes in memory after verification.

18.5.2 Verification of Modules and their Credentials

If a called partner module is not already loaded, the credentials and object code can be examined prior to loading and execution of the object code, preventing common file virus infections. Modules that are already loaded can be checked in memory as they execute.

Most aspects of the EISL specification can be implemented in a portable (platform-independent) manner. However, the object code format and return addresses are platform-specific.

18.5.3 Secure Linkage

Another service defined by EISL is secure linkage to a partner module. For the caller, this entails checking that the called address is in fact in the appropriate code module. For the called module, the return address can be verified to be within the appropriate calling module. Even in the case of self-checking, one can require that the return address be within the module being checked.

Linkage checks prevent attacks of the stealth class, where the object being verified is not the object that is being used. Also, the checks increase the difficulty of the man-in-the-middle attack, where a rogue module will insert itself between two communicating modules, masquerading itself as the other module to each module.

The specification supports modules that reside in a single address space, and have uncontrolled read and execute access to the code space of all modules.

18.5.4 Integrity Credentials

EISL integrity checks verify the integrity of an object code module and a set of credentials associated with that object module. These credentials must be signed manifests and digital certificates. A detailed description of these credentials are contained in two specifications:

- *CDSA Signed Manifest Specification*
- *CSSM Add-in Module Structure and Administration Specification*

An overview is provided here.

A credential is a set of persistent objects. A full set of credentials includes:

- A certificate, which can be part of a chain
- A manifest, which is a collection of references to the code modules that comprise the object and hashes of those executable objects
- A signer's information block, which contains references to sections of the manifest, a hash of that manifest section, and attributes describing the signer
- A signature block, which contains a signature over the signer's information block

The certificate must be verifiable based on a one or more specified public root keys. The complete certificate chain required for successful verification must be included in the signature block. This certificate must be used to sign the objects referenced by the manifest sections. This creates a tight integrity-binding between the certificate and the objects referenced by the manifest.

Each manifest section can contain additional descriptive information about the object referenced by the manifest section, such as their creation date.

The signature block is encoded in the format required by the signature block representation. For example, for a PKCS#7 signature block, the encoding format is BER / DER.

The manifest, signer information, and signature block are each stored in a separate file with an identifying suffix:

- The manifest filename suffix is .mf
- The signer information filename suffix is .sf
- The signature block filename suffix is .sig

It must be possible to specify a pathname and single common filename to locate the credential files. A convention for storing the credential files with the object code files could be adopted but is not required. For example, the credential files can be local to a system and the object code files could be remote. In this case the credential files and the object code files would not reside in the same file system path.

Based on these credentials EISL functions can be used to verify the identity and the data integrity of the object code modules referenced by the manifest sections.

18.6 EISL Uses Other Standards or Specifications

This specification uses other industry specifications or standards for certificates, keys, signatures, and cryptographic algorithms. Utilized standards include:

- X.509V3 certificates as identity credentials Signed Manifest Digital Signature Architecture [SM Spec] as integrity credentials
- PKCS#7 [PKCS] signatures
- DSA signature algorithm [DSA]
- SHA-1 message digest [SHA] algorithms
- OIW algorithm identifiers [OIW] and parameters to encode the DSA parameters and keys and to indicate the signature algorithms in certificates and PKCS#7 signature blocks

Chapter 19

Data Structures

19.1 Object Pointers

Many of the EISL objects form a hierarchical "contains" relationship. The larger, containing object defines an iterator object that enumerates the smaller objects. The smaller object defines a function that returns the larger object that contains it. A table summarizing the relationships among the EISL object types is provided at the end of this section.

19.1.1 Iterator Objects

Iterators are "disposable" objects created from verified objects that contain subordinate objects. They enumerate the manifest sections, or the attributes of the certificate, signature, or manifest section. The set of object references is determined when the iterator is created. Subsequent changes to the object from which it is created do not affect the set, the number of elements, or position in the iterator (this is not a problem in the embedded version of the library, which cannot change objects). Of course, many Iterators can be used to traverse the same set of object references independently.

The "get" function for each iterator object varies with each type of subordinate object referenced and returned by the function.

The object is recycled after the "get" function indicates that there are no more subordinate object references to enumerate.

Iterator objects are objects in their own right, but they are documented with their containing object.

```
typedef const void *ISL_ITERATOR_PTR
```

19.1.2 Verified Signature Root Object

A verified signature object is returned as the result of verifying a signature root. (This differs from the object type returned by the ISL_VerifySignatureRoot function.)

Valid operations on this object are to create an iterator to return manifest sections, or search for a specific signed object. The attributes of the unverified object have been verified, but the object itself has not been verified.

One can also create an iterator to enumerate the verified attributes of the signature itself.

```
typedef const void *ISL_VERIFIED_SIGNATURE_ROOT_PTR
```

19.1.3 Verified Certificate Chain Object

A verified certificate chain object is returned by functions that construct and verify a certificate chain. A certificate chain begins with the trusted signer certificate and ends with the certificate of the signer found in a signature block. Valid operations on this object are to return an array of verified certificate objects. This object can be contained in a Verified Signature Root Object.

```
typedef const void *ISL_VERIFIED_CERTIFICATE_CHAIN_PTR
```

19.1.4 Verified Certificate Object

A verified certificate object is returned as a result of requesting the verified certificates in a certificate chain. Valid operations on this object include obtaining public key and other attributes stored in the certificate. A verified certificate object cannot be modified. This object can be contained in a Verified Certificate Chain Object.

```
typedef const void *ISL_VERIFIED_CERTIFICATE_PTR
```

19.1.5 Manifest Section Object

A manifest section object is returned by an iterator that was created from a verified root signature. For each signed object, there is a manifest section which describes its attributes and how to retrieve and verify it.

Valid operations on this object are to verify the signed object, and to create an iterator which returns attributes of the signed object. Using the iterator, it is possible to check the attributes of a signed object prior to verifying the object itself. The manifest section object is always contained in a Verified Signature Root Object.

```
typedef const void *ISL_MANIFEST_SECTION_PTR
```

19.1.6 Verified Module Object

A verified module object is returned as a result of verifying the credentials for a module. This object is created by either ISL_VerifyAndLoadModuleAndCredentials, ISL_VerifyLoadedModuleAndCredentials, ISL_SelfCheck, ISL_VerifyAndLoadModule, or ISL_VerifyLoadedModule. This object is always contained in a Verified Signature Root Object.

Valid operations on this object include checking address ranges and obtaining the Manifest Section Object corresponding to the verified module. The verified module object cannot be modified in memory, and libraries must use various techniques to enforce this requirement.

```
typedef const void *ISL_VERIFIED_MODULE_PTR
```

19.1.7 EISL Object Relationships and Life Cycle

This is shown by the table which is on the following page.

OBJECT	CONTAINING OBJECT	CREATING FUNCTION(S)	RECYCLING FUNCTION
Verified Signature Root*	none	ISL_Self_Check*, ISL_VerifyAndLoadModuleAndCredentials*, ISL_VerifyLoadedModuleAndCredentials*	ISL_RecycleModuleAndCredentials*
Verified Signature Root	none	ISL_CreateVerifiedSignatureRoot, ISL_CreateVerifiedSignatureRootWithCertificate	ISL_RecycleVerifiedSignatureRoot
Manifest Section	Verified Signature Root	(implicit)	(implicit)
Verified Module	Manifest Section	(implicit)	(implicit)
Verified Certificate	none	ISL_CreateCertificateChain	ISL_RecycleCertificateChain
Verified Certificate Chain***	Verified Signature Root	(implicit)	(implicit)
Verified Certificate	Verified Certificate Chain	(implicit)	(implicit)
Manifest Section Iterator	Verified Signature Root	ISL_CreateManifestSectionEnumerator	ISL_RecycleManifestSectionEnumerator **
Signature Attribute Iterator	Verified Signature Root	ISL_Create Signature AttributeEnumerator	ISL_RecycleSignatureAttributeEnumerator **
Certificate Attribute Iterator	Verified Certificate	ISL_CreateCertificateAttributeEnumerator	ISL_RecycleCertificateAttributeEnumerator **
Manifest Section Attribute Iterator	Verified Signature Root	ISL_CreateManifestSection	ISL_RecycleManifestSectionAttribute

* A Verified Module object in the API function is used to reference its containing Verified Signature Root in these "simplified API" calls.

** The iterator is implicitly recycled if its parent object is recycled. The recycle API call is optional.

*** The object is created and recycled implicitly under the "simplified API" calls.

19.2 Low-Level Data Structures Used in API Functions

19.2.1 ISL_DATA

The ISL_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory.

```
typedef struct ISL_data{
    uint32 Length;  /* in bytes */
    uint8 *Data;
} ISL_DATA, *ISL_DATA_PTR
```

Definition

Length
Length of the data buffer in bytes.

Data
Points to the start of an arbitrary length data buffer.

19.2.2 ISL_CONST_DATA

The ISL_CONST_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous "read-only" memory.

Note: The data referenced by the ISL_CONST_DATA is read-only, but the ISL_CONST_DATA itself can be modified.

```
typedef struct ISL_data{
    uint32 Length;  /* in bytes */
    const uint8 *Data;
} ISL_CONST_DATA, *ISL_CONST_DATA_PTR
```

Definition

Length
Length of the data buffer in bytes.

Data
Points to the start of an arbitrary length data buffer.

Chapter 20

EISL Functions

20.1 Locator Services

The manpages for Locator Services follow on the next page.

NAME

ISL_FindRegistryAttribute

SYNOPSIS

```
ISL_STATUS ISL_FindRegistryAttribute
    (const ISL_DATA_PTR Name,
    ISL_DATA_PTR Value);
```

DESCRIPTION

This function searches the system registry for the attribute specified by Name. If successful, the value of the attribute is returned.

PARAMETERS

Name (input)
Full name of a registry entry.

Value (output)
Registry value corresponding to the given name.

RETURN VALUE

If the search was successful, ISL_OK is returned. Otherwise, ISL_FAIL is returned.

20.2 Credential and Attribute Verification Services

The functions for credential and attribute verification services provide a simplified verification for the common case where each code object is signed with its own signature file.

NAME

ISL_SelfCheck

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR ISL_SelfCheck
    ();
```

DESCRIPTION

This function returns a pointer to the verified module object if the module passed self-check, otherwise NULL. This function checks to see that the return address and the checking code itself are in the checked module.

Note: The public key used to verify the signature is embedded in the library code or can be referenced by it in an implementation-specific manner. The public key is not exposed in the API. The EISL takes additional measures that make it difficult to modify the public key. The self-check function in EISL implicitly knows how to obtain the credentials of the module the instance of EISL is contained within.

EISL also makes it difficult for each module that contains an instance of EISL to bypass the self-check function. After invoking the self-check function, the containing module should verify that the return address and the address of the function itself are within the module being verified using the ISL_CheckAddressWithinModule function.

PARAMETERS

None.

RETURN VALUE

Pointer to a verified object if verification is successful, or NULL if verification is unsuccessful.

SEE ALSO

ISL_CheckAddressWithinModule, *ISL_RecycleVerifiedModuleCredentials*

NAME

ISL_VerifyAndLoadModuleAndCredentials

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR ISL_VerifyAndLoadModuleAndCredentials
    (ISL_CONST_DATA Credentials,
    ISL_CONST_DATA SectionName,
    ISL_CONST_DATA Signer,
    ISL_CONST_DATA PublicKey)
```

DESCRIPTION

The purpose of this function is to verify the integrity of the credentials associated with an object code module and the integrity of the object code itself. If verified, the module is loaded into memory. Verification is accomplished as follows:

- Verify the credentials—the specified PublicKey is used to verify the signature on the specified Credentials. The Credentials parameter must specify a full file system path name to locate the signature and manifest files associated with the target module. If the signature has more than one signer, the Signer parameter selects the signer to be verified.
- Verify module integrity—if the credentials are valid, the integrity of the object code module referenced by the manifest section with the specified SectionName is verified. If successful, a verified module object pointer is returned. Otherwise, NULL is returned.

If the object module referenced by the manifest section is not already loaded, the object code is verified as an object module object using file system reads to obtain the image without loading it. If verified, the module is loaded.

If the module is already loaded, it is verified in memory.

Certificates embedded in the PKCS#7 signature as well as free-standing X.509 certificates in the credentials directory can be used in the certificate chain.

This function combines many smaller functions into one call for a common use case. If greater flexibility is needed, a series of calls that includes ISL_CreateCertificateChain, ISL_CopyCertificateChain, ISL_CreateVerifiedSignatureRootWithCertificate, ISL_FindManifestSection, and ISL_VerifyAndLoadModule provides the same functionality.

Cleanup is done by ISL_RecycleVerifiedModuleCredentials.

PARAMETERS

Credentials (input)
The full file name to the signature file.

SectionName (input)
The section name of the manifest that refers to the object code to be verified.

Signer (input)
The signer information (for directly signed signatures) or issuer name (if signed by certificates). If the Signer is NULL, a default value is assumed. For example, it could be the X.509V3 IssuerName in the root certificate, or the SignerID in the PKCS#7 specification if directly signed.

PublicKey (input)
This is the public key of the signer or root certificate authority. The representation for the key must be compatible with the format of public keys in the selected certificate format. If the PublicKey is NULL, a default value is assumed.

RETURN VALUE

Pointer to a verified object if verification is successful, or NULL if verification is unsuccessful.

SEE ALSO

ISL_CreateCertificateChain, *ISL_FindManifestSection*, *ISL_CopyCertificateChain*, *ISL_VerifyAndLoadModule*, *ISL_CreateVerifiedSignatureRootWithCertificate*, *ISL_RecycleVerifiedModuleCredentials*, *ISL_FindRegistryAttribute*,

NAME

ISL_VerifyLoadedModuleAndCredentials

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR ISL_VerifyLoadedModuleAndCredentials
    (ISL_CONST_DATA Credentials,
    ISL_CONST_DATA SectionName,
    ISL_CONST_DATA Signer,
    ISL_CONST_DATA PublicKey)
```

DESCRIPTION

The purpose of this function is to verify the integrity of the credentials associated with a loaded object code module and the integrity of the object code itself. Verification is accomplished as follows:

- Verify the credentials—the specified PublicKey is used to verify the signature on the specified Credentials. The Credentials parameter must specify a full file system path name to locate the signature and manifest files associated with the target module. If the signature has more than one signer, the Signer parameter selects the signer to be verified.
- Verify module integrity—if the credentials are valid, the integrity of the loaded object code module referenced by the manifest section with the specified SectionName is verified. If successful, a verified module object pointer is returned. Otherwise, NULL is returned.

Certificates embedded in the PKCS#7 signature as well as free-standing X.509 certificates in the credentials directory can be used in the certificate chain.

This function combines many smaller functions into one call for a common case. If greater flexibility is needed, a series of calls that includes ISL_CreateCertificateChain, ISL_CopyCertificateChain, ISL_CreateVerifiedSignatureRootWithCertificate, ISL_FindManifestSection, and ISL_VerifyLoadedModule provides the same functionality. Cleanup is done by ISL_RecycleVerifiedModuleCredentials.

PARAMETERS

Credentials (input)
The full file name to the signature file.

SectionName (input)
The section name of the manifest that refers to the object code to be verified.

Signer (input)
The signer information (for directly signed signatures) or issuer name (if signed by certificates). If the Signer is NULL, a default value is assumed.

PublicKey (input)
This is the public key of the signer or root certificate authority. The representation for the key must be compatible with the format of public keys in the selected certificate format. If the PublicKey is NULL, a default value is assumed.

RETURN VALUE

Pointer to a verified object if verification is successful, or NULL if verification is unsuccessful.

SEE ALSO

ISL_CreateCertificateChain, ISL_FindManifestSection, ISL_CopyCertificateChain, ISL_VerifyLoadedModule, ISL_CreateVerifiedSignatureRoot, ISL_RecycleVerifiedModuleCredentials, ISL_FindRegistryAttribute

NAME

ISL_GetCertficateChain

SYNOPSIS

```
ISL_VERIFIED_CERTIFICATE_CHAIN_PTR ISL_GetCertificateChain
    (ISL_VERIFIED_MODULE_PTR Module)
```

DESCRIPTION

This function returns a reference to the certificate chain that was constructed and verified by ISL_VerifyLoadedModuleAndCredentials or ISL_VerifyAndLoadModuleAndCredentials.

PARAMETERS

Module (input)

A verified module object returned by the ISL_SelfCheck, ISL_VerifyLoadedModuleAndCredentials, or ISL_VerifyAndLoadModuleAndCredentials function.

Verified module objects created by ISL_VerifyAndLoadModule, ISL_VerifyLoadedModule, and ISL_VerifyData return a NULL certificate chain.

RETURN VALUE

A pointer to the verified certificate chain object is returned if successful, otherwise NULL.

SEE ALSO

ISL_VerifyLoadedModuleAndCredentials, *ISL_VerifyAndLoadModuleAndCredentials*, *ISL_SelfCheck*

NAME

ISL_ContinueVerification

SYNOPSIS

```
uint32 ISL_ContinueVerification
    ISL_VERIFIED_MODULE_PTR Module,
    uint32 WorkFactor)
```

DESCRIPTION

The purpose of this function is to permit ongoing verification of an object which has been already verified by the ISL_VerifyAndLoadModuleAndCredentials, ISL_SelfCheck, ISL_VerifyLoadedModuleAndCredentials, ISL_VerifyAndLoadModule, or ISL_VerifyLoadedModule functions. The WorkFactor parameter increases the amount of verification for an individual call by an implementation-specific amount proportional to the parameter value. The result variable returns the cummulative number of complete, successful verification passes which have been performed on the verified module, or zero if a failure was ever detected.

The application can dynamically adjust the amount of time spent in verification by adjusting the work factor. The return value permits monitoring the rate at which the entire object is verified.

PARAMETERS

Module (input)
A verified module object returned by the ISL_SelfCheck, ISL_VerifyAndLoadModuleAndCredentials, ISL_VerifyAndLoadModuleAndCredentials, ISL_VerifyAndLoadModule, or ISL_VerifyLoadedModule function.

WorkFactor (input)
The amount of work spent in the partial verification increases in proportion to the value of this parameter. The actual rate of verification depends on the platform and implementation.

RETURN VALUE

The number of verification passes that have been completed successfully, or zero if verification is unsuccessful.

SEE ALSO

ISL_SelfCheck, ISL_VerifyAndLoadModuleAndCredentials, ISL_VerifyLoadedModuleAndCredentials, ISL_VerifyAndLoadModule, ISL_VerifyLoadedModule

NAME

ISL_RecycleVerifiedModuleCredentials

SYNOPSIS

```
ISL_STATUS ISL_RecycleVerifiedModuleCredentials
    (ISL_VERIFIED_MODULE_PTR Verification)
```

DESCRIPTION

This function destroys and recycles the memory for the module verification object, its containing Signature Root Object and Certificate Chain Object, and all subordinate objects. Related iterator objects and certificate objects must be recycled before recycling the module verification object. Once recycled, this object must not be referenced. All pointers to certificates, manifest sections, iterators, and the information returned by iterators are invalid after this call has completed.

PARAMETERS

Verification (input)

A verified module object returned by the ISL_SelfCheck, ISL_VerifyAndLoadModuleAndCredentials, or ISL_VerifyLoadedModuleAndCredentials function.

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

ISL_VerifyAndLoadModuleAndCredentials, *ISL_VerifyLoadedModuleAndCredentials*, *ISL_SelfCheck*

20.3 Signature Root Methods

The manpages for Signature Root Methods follow on the next page.

NAME

ISL_CreateVerifiedSignatureRoot

SYNOPSIS

```
ISL_VERIFIED_SIGNATURE_ROOT_PTR ISL_CreateVerfiedSignatureRoot
    (ISL_CONST_DATA Credentials,
    ISL_CONST_DATA Signer,
    ISL_CONST_DATA PublicKey)
```

DESCRIPTION

This function uses the PublicKey to verify the digital signature specified by the Credentials. It does not construct certificate chains, but must use the key directly. If the credentials support multiple signers, the Signer parameter can be used to determine which signer to verify.

This function does not verify the objects referenced in the manifest sections. However, the manifest sections are verified, and the attributes in the sections can be trusted.

The manifest sections can be enumerated using the object created by ISL_CreateManifestSectionEnumerator.

PARAMETERS

Credentials (input)
The complete path name to the digital signature file to be verified.

Signer (input)
The signer information for directly signed signatures. If the Signer is NULL, a default value is assumed.

PublicKey (input)
This is the public key of the signer or root certificate authority. The representation for the key must be compatible with the format of public keys in the selected certificate format. If the PublicKey is NULL, a default value is assumed.

RETURN VALUE

Pointer to a verified signature root object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_CreateManifestSectionEnumerator, ISL_CreateSignatureAttributeEnuerator

NAME

ISL_CreateVerifiedSignatureRootWithCertificate

SYNOPSIS

```
ISL_VERIFIED_SIGNATURE_ROOT_PTR
                         ISL_CreateVerfiedSignatureRootWithCertificate
    (ISL_CONST_DATA Credentials,
    ISL_VERIFIED_CERTIFICATE_PTR Cert)
```

DESCRIPTION

This function uses the PublicKey to verify the digital signature specified by the Credentials. It does not construct certificate chains, but must use the signer identification and public key in the certificate directly.

The function does not verify the objects referenced in the manifest sections. However, the manifest sections are verified, and the attributes in the sections can be trusted.

The manifest sections can be enumerated using the object created by ISL_CreateManifestSectionEnumerator.

PARAMETERS

Credentials (input)
The complete path name to the digital signature file to be verified.

Cert (input)
The certificate used to directly verify the digital signature.

RETURN VALUE

Pointer to a verified signature root object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_CreateManifestSectionEnumerator, ISL_CreateSignatureAttributeEnumerator

NAME

ISL_FindManifestSection

SYNOPSIS

```
ISL_MANIFEST_SECTION_PTR ISL_FindManifestSection
    (ISL_VERIFIED_SIGNATURE_ROOT_PTR Root,
    ISL_CONST_DATA Name)
```

DESCRIPTION

This function returns a pointer to the Manifest Section Object with the given name, or NULL if there is no such section.

PARAMETERS

Root (input)
A verified signature root explicitly created by ISL_CreateVerifiedSignatureRoot or ISL_CreateVerifiedSignatureRootWithCertificate, or implicitly by ISL_SelfCheck, ISL_VerifyAndLoadModuleAndCredentials, or ISL_VerifyLoadedModuleAndCredentials.

Name (input)
The name of the manifest section that is requested.

RETURN VALUE

The specified Manifest Section Object is returned, or NULL if no section exists.

SEE ALSO

ISL_CreateVerifiedSignatureRoot, *ISL_CreateVerifiedSignatureRootWithCertificate*, *ISL_SelfCheck*, *ISL_VerifyAndLoadModuleAndCredentials*, *ISL_VerifyLoadedModuleAndCredentials*

NAME

ISL_CreateManifestSectionEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR ISL_CreateManifestSectionEnumerator
    (ISL_VERIFIED_SIGNATURE_ROOT_PTR Root)
```

DESCRIPTION

This function creates a dynamic object whose purpose is to list references to the sections of the manifest referenced by the *Verification* parameter. The resulting iterator object is activated by invoking the ISL_GetNextManifestSection function. The object should be recycled using the ISL_RecycleManifestSectionEnumerator call when it is no longer needed.

PARAMETERS

Root (input)

A verified signature root explicitly created by ISL_CreateVerifiedSignatureRoot or ISL_CreateVerifiedSignatureRootWithCertificate, or implicitly by ISL_SelfCheck, ISL_VerifyAndLoadModuleAndCredentials, or ISL_VerifyLoadedModuleAndCredentials.

RETURN VALUE

Pointer to a manifest section iterator object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_GetNextManifestSection, *ISL_RecycleManifestSectionEnumerator*

NAME

ISL_GetNextManifestSection

SYNOPSIS

```
ISL_MANIFEST_SECTION_PTR ISL_GetNextManifestSection
    (ISL_ITERATOR_PTR Iterator)
```

DESCRIPTION

This function returns a pointer to the next Manifest Section Object, or NULL if there are no more sections. The state of the iterator is updated such that the next call to this function will return the next manifest section object.

PARAMETERS

Iterator (input)
A certificate attribute iterator created by ISL_CreateManifestSectionEnumerator.

RETURN VALUE

The next Manifest Section Object is returned, or NULL if no more sections exist.

SEE ALSO

ISL_CreateManifestSectionEnumerator

NAME

ISL_RecycleManifestSectionEnumerator

SYNOPSIS

```
ISL_STATUS ISL_RecycleManifestSectionEnumerator
    (ISL_ITERATOR_PTR Iterator)
```

DESCRIPTION

This function destroys and recycles the memory for the manifest section iterator. It must be the last call that references the iterator.

PARAMETERS

Iterator (input)
A manifest section iterator created by ISL_CreateManifestSectionEnumerator.

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateManifestSectionEnumerator

NAME

ISL_FindSignatureAttribute

SYNOPSIS

```
ISL_STATUS ISL_FindSignatureAttribute
    (ISL_VERIFIED_SIGNATURE_ROOT_PTR Root,
    ISL_CONST_DATA Name,
    ISL_CONST_DATA_PTR Value)
```

DESCRIPTION

This function returns the value associated with the signature attribute specified by Name. The value and its length are returned in the Value pointer. The function returns ISL_FAIL if the specified attribute does not exist.

PARAMETERS

Root (input)
A verified signature root explicitly created by ISL_CreateVerifiedSignatureRoot or ISL_CreateVerifiedSignatureRootWithCertificate, or implicitly by ISL_SelfCheck, ISL_VerifyAndLoadModuleAndCredentials, or ISL_VerifyLoadedModuleAndCredentials.

Name (input)
The name of the attribute that is requested. The representation of the attribute name must be consistent with the representation of certificates. For example, attribute names for signatures associated with X.509V3 certificates would be DER-encoded object identifiers.

Value (input/output)
The data pointer and length are updated to point to a read-only copy of the attribute.

RETURN VALUE

ISL_OK is returned if the attribute is found, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateVerifiedSignatureRoot, *ISL_CreateVerifiedSignatureRootWithCertificate*, *ISL_SelfCheck*, *ISL_VerifyAndLoadModuleAndCredentials*, *ISL_VerifyLoadedModuleAndCredentials*, *ISL_GetModuleManifestSection*, *ISL_GetManifestSignatureRoot*

NAME

ISL_CreateSignatureAttributeEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR ISL_CreateSignatureAttributeEnumerator
    (ISL_VERIFIED_SIGNATURE_ROOT_PTR Root)
```

DESCRIPTION

This function creates a dynamic object whose purpose is to list references to the attributes of the signature referenced by the *Verification* parameter. The resulting iterator object is activated by invoking the ISL_GetNextSignatureAttribute function. The object should be recycled using the ISL_RecycleSignatureEnumerator call when it is no longer needed.

PARAMETERS

Root (input)

A verified signature root explicitly created by ISL_CreateVerifiedSignatureRoot or ISL_CreateVerifiedSignatureRootWithCertificate, or implicitly by ISL_SelfCheck, ISL_VerifyAndLoadModuleAndCredentials, or ISL_VerifyLoadedModuleAndCredentials.

RETURN VALUE

Pointer to a signature attribute iterator object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_GetNextSignatureAttribute, ISL_RecycleSignatureAttributeEnumerator, ISL_SelfCheck, ISL_VerifyAndLoadModuleAndCredentials, ISL_VerifyLoadedModuleAndCredentials, ISL_GetModuleManifestSection, ISL_GetManifestSignatureRoot

NAME

ISL_GetNextSignatureAttribute

SYNOPSIS

```
ISL_STATUS ISL_GetNextSignatureAttribute
    (ISL_ITERATOR_PTR Iterator,
    ISL_CONST_DATA_PTR Name,
    ISL_CONST_DATA_PTR Value)
```

DESCRIPTION

This function returns the next attribute name and value for the signature referenced by the iterator object. The state of the iterator is updated such that the next call to this function will return the next attribute. The name and value cannot be modified by the program. If no more attribute values are present, the function returns ISL_FAIL.

PARAMETERS

Iterator (input)
A signature attribute iterator created by ISL_CreateSignatureAttributeEnumerator.

Name (output)
A pointer to a result variable that is updated to refer to the attribute name. The representation of the attribute name must be consistent with the representation of certificates. For example, with X.509V3 certificates, the name is a DER-encoded object identifier for a PKCS#7 authenticated attribute.

Value (output)
A pointer to a result variable that is updated to refer to the attribute value. The representation of the attribute name must be consistent with the representation of certificates. For example, with X.509V3 certificates, it is a DER-encoded value (or values).

RETURN VALUE

The function result is ISL_OK if successful in returning a name and value pair, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateSignatureAttributeEnumerator

NAME

ISL_RecycleSignatureAttributeEnumerator

SYNOPSIS

```
ISL_STATUS ISL_RecycleSignatureAttributeEnumerator
    (ISL_ITERATOR_PTR Iterator)
```

DESCRIPTION

This function destroys and recycles the memory for the signature attribute iterator. It must be the last call referencing the iterator.

PARAMETERS

Iterator (input)
A signature attribute iterator created by ISL_CreateSignatureAttributeEnumerator.

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateSignatureAttributeEnumerator

NAME

ISL_RecycleVerifiedSignatureRoot

SYNOPSIS

```
ISL_STATUS ISL_RecycleVerifiedSignatureRoot
    (ISL_VERIFIED_SIGNATURE_ROOT_PTR Root)
```

DESCRIPTION

This function destroys and recycles the memory for the verified signature root. It must be the last call referencing the signature root, or any objects derived from or contained in the signature root.

PARAMETERS

Root (input)
A verified signature root explicitly created by ISL_CreateVerifiedSignatureRoot or ISL_CreateVerifiedSignatureRootWithCertificate.

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateVerifiedSignatureRoot, *ISL_CreateVerifiedSignatureRootWithCertificate*

20.4 Certificate Chain Methods

The manpages for functions to manipulate certificate chains in a PKCS#7 signature block follow on the next page.

NAME

ISL_CreateCertificateChain

SYNOPSIS

```
CONST ISL_VERIFIED_CERTIFICATE_CHAIN_PTR ISL_CreateCertificateChain
    (ISL_CONST_DATA RootIssuer,
    ISL_CONST_DATA PublicKey,
    ISL_CONST_DATA Credential)
```

DESCRIPTION

This function constructs and verifies a certificate chain which starts with the root certificate authority (issuer) and ends with the certificate of the signer of the Credential. During the construction process, each certificate is verified, beginning with the root certificate.

PARAMETERS

RootIssuer (input)
The distinguished name of the root certificate authority.

PublicKey (input)
The public key of the root certificate authority.

Credential (input)
The full path filename of a module's signature file.

RETURN VALUE

A pointer to the verified certificate chain object is returned if successful, otherwise NULL.

SEE ALSO

ISL_RecycleCertificateChain, *ISL_FindRegistryAttribute*

NAME

ISL_CopyCertificateChain

SYNOPSIS

```
uint32 ISL_CopyCertificateChain
    (ISL_VERIFIED_CERTIFICATE_CHAIN_PTR Verification,
    ISL_VERIFIED_CERTIFICATE_PTR Certs[],
    uint32 MaxCertificates)
```

DESCRIPTION

This function copies pointers to the verified certificates in the certificate chain. The first certificate (subscript zero) is signed by the root certificate authority. The last certificate is the signer's certificate.

PARAMETERS

Verification (input)
A verified certificate chain returned by the ISL_CreateCertificateChain or ISL_GetCertificateChain function.

Certs (input/output)
An array of certificate object pointers sufficiently large to contain the expected certificate chain.

MaxCertificates (input)
The dimension of the certificate object pointer array.

RETURN VALUE

The number of certificates returned in the Certs array as a result of the copy process.

SEE ALSO

ISL_CreateCertificateChain, *ISL_GetCertificateChain*

NAME

ISL_RecycleCertificateChain

SYNOPSIS

```
ISL_STATUS ISL_RecycleVerifiedCertificateChain
    (ISL_VERIFIED_CERTIFICATE_CHAIN_PTR Chain)
```

DESCRIPTION

This function destroys and recycles the memory for the verified certificate chain. It must be the last call referencing the certificate chain, or any objects derived from or contained in the certificate chain.

PARAMETERS

Chain (input)
A verified certificate chain explicitly created by ISL_CreateCertificateChain.

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateCertificateChain

20.5 Certificate Attribute Methods

The manpages for Certificate Methods follow on the next page.

NAME

ISL_FindCertificateAttribute

SYNOPSIS

```
ISL_STATUS ISL_FindCertificateAttribute
    (ISL_VERIFIED_CERTIFICATE_PTR Cert,
    ISL_CONST_DATA Name,
    ISL_CONST_DATA_PTR Value)
```

DESCRIPTION

This function returns the value associated with the certificate attribute specified by Name. The value and its length are returned in the Value pointer. The function returns ISL_FAIL if the specified attribute does not exist.

PARAMETERS

Cert (input)
A reference to a certificate returned by the ISL_CopyCertificateChain function.

Name (input)
The name of the attribute that is requested. The name representation must be consistent with the certificate representation. For example, for X.509V3 certificates, an attribute name is represented as a DER-encoded object identifier.

Value (input/output)
The address and length are updated to refer to the attribute value within the verified certificate.

RETURN VALUE

ISL_OK is returned if the specified certificate attribute is found, or ISL_FAIL if the attribute is not found.

SEE ALSO

ISL_CopyCertificateChain

NAME

ISL_CreateCertificateAttributeEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR ISL_CreateCertificateAttributeEnumerator
    (ISL_VERIFIED_CERTIFICATE_PTR Cert)
```

DESCRIPTION

This function creates a dynamic object whose purpose is to list references to the attributes of the certificate. The iterator object is activated using the ISL_GetNextCertificateAttribute function call. The object must be recycled using the ISL_RecycleCertificateAttributeEnumerator call when it is no longer needed.

PARAMETERS

Cert (input)
A reference to a certificate returned by the ISL_CreateCertificateChain function.

RETURN VALUE

Pointer to an iterator object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_RecycleCertificateAttributeEnumerator, *ISL_CopyCertificateChain*, *ISL_GetNextCertificateAttribute*

NAME

ISL_GetNextCertificateAttribute

SYNOPSIS

```
ISL_STATUS ISL_GetNextCertificateAttribute
    (ISL_ITERATOR_PTR CertIterator,
    ISL_CONST_DATA_PTR Name,
    ISL_CONST_DATA_PTR Value)
```

DESCRIPTION

This function returns the next attribute name and value. The state of the iterator is updated such that the next call to this function will return the next attribute. The name and value cannot be modified by the program. If no more attribute values are present, the function returns ISL_FAIL.

PARAMETERS

CertIterator (input)
A certificate attribute iterator created by ISL_CreateCertificateAttributeEnumerator.

Name (output)
A pointer to a result variable that is updated to refer to the attribute name. The representation of the attribute name must be consistent with the representation of certificates. For example, with X.509V3 certificates, the name is a DER-encoded object identifier.

Value (output)
A pointer to a result variable that is updated to refer to the attribute value. The representation of the attribute name must be consistent with the representation of certificates. For example, with X.509V3 certificates, it is a DER-encoded value (or values).

RETURN VALUE

The function result is ISL_OK if successful in returning a name and value pair, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateCertificateAttributeEnumerator

NAME

ISL_RecycleCertificateAttributeEnumerator

SYNOPSIS

```
ISL_STATUS ISL_RecycleCertificateAttributeEnumerator
    (ISL_ITERATOR_PTR CertIterator)
```

DESCRIPTION

This function destroys and recycles the memory for the certificate attribute iterator. It must be the last call that references the iterator.

PARAMETERS

CertIterator (input)
A certificate attribute iterator created by ISL_CreateCertificateAttributeEnumerator.

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateCertificateAttributeEnumerator

20.6 Manifest Section Object Methods

The manpages for Manifest Section Object Methods follow on the next page.

NAME

ISL_GetManifestSignatureRoot

SYNOPSIS

```
ISL_VERIFIED_SIGNATURE_ROOT_PTR ISL_GetManifestSignatureRoot
    (ISL_MANIFEST_SECTION_PTR Section)
```

DESCRIPTION

This function gets the Verified Signature Root which contains this manifest section.

PARAMETERS

Section (input)
A manifest section pointer returned by ISL_GetNextManifestSection, ISL_GetModuleManifestSection, or ISL_FindManifestSection.

RETURN VALUE

Pointer to a signature root object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_GetNextManifestSection, *ISL_FindManifestSection*, *ISL_GetModuleManifestSection*

NAME

ISL_VerifyAndLoadModule

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR ISL_VerifyAndLoadModule
    (ISL_MANIFEST_SECTION_PTR Section)
```

DESCRIPTION

If the module referenced by the manifest section is already loaded, it is verified in memory. Otherwise, the module is verified on the file system, and, if successful, the module is loaded.

PARAMETERS

Section (input)
A manifest section returned by the ISL_GetNextManifestSection or ISL_FindManifestSection functions.

RETURN VALUE

Pointer to a verified module object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_GetNextManifestSection, *ISL_FindManifestSection*

NAME

ISL_VerifyLoadedModule

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR ISL_VerifyLoadedModule
    (ISL_MANIFEST_SECTION_PTR Section)
```

DESCRIPTION

This function verifies a memory-resident object code module referenced in the specified manifest section.

PARAMETERS

Section (input)
A manifest section returned by the ISL_GetNextManifestSection or ISL_FindManifestSection functions.

RETURN VALUE

Pointer to a verified module object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_GetNextManifestSection, *ISL_FindManifestSection*

NAME

ISL_VerifyData

SYNOPSIS

```
ISL_VERIFIED_MODULE_PTR ISL_VerifyData
    (ISL_MANIFEST_SECTION_PTR Section)
```

DESCRIPTION

This function verifies a data object referenced by the specified manifest section. This function can verify executable code modules, but it's intended use is to verify non-executable data objects.

PARAMETERS

Section (input)
A manifest section returned by the ISL_GetNextManifestSection or ISL_FindManifestSection functions.

RETURN VALUE

Pointer to a verified module object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_VerifyLoadedModule, ISL_VerifyAndLoadModule, ISL_GetNextManifestSection, ISL_FindManifestSection, ISL_CheckAddressWithinModule

NAME

ISL_FindManifestSectionAttribute

SYNOPSIS

```
ISL_STATUS ISL_FindManifestSectionAttribute
    (ISL_MANIFEST_SECTION_PTR Section,
    ISL_CONST_DATA Name,
    ISL_CONST_DATA_PTR Value)
```

DESCRIPTION

This function updates the length and pointer to refer to the Manifest Section Attribute (or metadata) Value corresponding to the given name, or returns ISL_FAIL if there is no such attribute.

PARAMETERS

Section (input)
A manifest section object returned by the ISL_FindManifestSection or ISL_GetNextManifestSection functions.

Name (input)
The name of the attribute that is requested. The name representation must be consistent with the manifest representation. Manifests are human-readable. The attribute name is represented as an alphanumeric (and underscore, minus, and period) ASCII character string.

Value (output)
A pointer to a result variable whose length and pointer are updated to refer to the attribute value.

RETURN VALUE

ISL_OK is returned if the attribute was found, or ISL_FAIL if unsuccessful.

SEE ALSO

ISL_FindManifestSection, *ISL_GetNextManifestSection*

NAME

ISL_CreateManifestSectionAttributeEnumerator

SYNOPSIS

```
ISL_ITERATOR_PTR ISL_CreateManifestSectionAttributeEnumerator
    (ISL_MANIFEST_SECTION_PTR Section)
```

DESCRIPTION

This function creates a dynamic object whose purpose is to list references to the attributes of the manifest Section. The iterator object is activated using the ISL_GetNextManifestSectionAttribute function call. The object must be recycled using the ISL_RecycleManifestSectionEnumerator call when it is no longer needed.

PARAMETERS

Section (input)
A manifest section object returned by the ISL_FindManifestSection or ISL_GetNextManifestSection functions.

RETURN VALUE

Pointer to a signed object attribute iterator object if successful, or NULL if unsuccessful.

SEE ALSO

ISL_FindManifestSection, ISL_GetNextManifestSection

NAME

ISL_GetNextManifestSectionAttribute

SYNOPSIS

```
ISL_STATUS ISL_GetNextManifestSectionAttribute
    (ISL_ITERATOR_PTR Iterator,
    ISL_CONST_DATA_PTR Name,
    ISL_CONST_DATA_PTR Value)
```

DESCRIPTION

This function returns the next attribute name and value. The state of the iterator is updated such that the next call to this function will return the next attribute. The name and value cannot be modified by the program. If no more attribute values are present, the function returns ISL_FAIL.

PARAMETERS

Iterator (input)
A signed object attribute iterator created by ISL_CreateManifestSectionAttributeEnumerator.

Name (output)
A pointer to a result variable that is updated to refer to the attribute name. The name representation must be consistent with the manifest representation. Manifests are human-readable. The attribute name is represented as an alphanumeric (and underscore, minus, and period) ASCII character string.

Value (output)
A pointer to a result variable that is updated to refer to the attribute value. The value is an arbitrary binary object.

RETURN VALUE

The function result is ISL_OK if successful in returning a name and value pair, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateManifestSectionAttributeEnumerator

NAME

ISL_RecycleManifestSectionAttributeEnumerator

SYNOPSIS

```
ISL_STATUS ISL_RecycleManifestSectionAttributeEnumerator
    (ISL_ITERATOR_PTR Iterator)
```

DESCRIPTION

This function destroys and recycles the memory for the Manifest Section Attribute iterator. It must be the last call which references the iterator.

PARAMETERS

Iterator (input)
A signed object attribute iterator created by ISL_CreateManifestSectionAttributeEnumerator.

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

ISL_CreateManifestSectionAttributeEnumerator

NAME

ISL_GetModuleManifestSection

SYNOPSIS

```
ISL_MANIFEST_SECTION_PTR ISL_GetModuleManifestSection
    (ISL_VERIFIED_MODULE_PTR Module)
```

DESCRIPTION

This function returns the manifest section that describes the integrity of the specified Module. This is the section that is used to verify module integrity.

PARAMETERS

Module (input)

A verified object module created by ISL_SelfCheck, ISL_VerifyLoadedModuleAndCredentials, ISL_VerifyAndLoadModuleAndCredentials, ISL_VerifyLoadedModule, or ISL_VerifyAndLoadModule.

RETURN VALUE

ISL_OK is returned if successful, otherwise ISL_FAIL.

SEE ALSO

ISL_SelfCheck, ISL_VerifyLoadedModuleAndCredentials, ISL_VerifyAndLoadModuleAndCredentials, ISL_VerifyAndLoadModule, ISL_VerifyLoadedModule

20.7 Secure Linkage Services

The manpages for Secure Linkage Services follow on the next page.

NAME

ISL_LocateProcedureAddress

SYNOPSIS

```
void * ISL_LocateProcedureAddress
    (ISL_VERIFIED_MODULE_PTR Module,
    ISL_CONST_DATA Name)
```

DESCRIPTION

This function returns the address of a function in a verified object code module. The function of interest is specified by Name. The address returned is read from the symbol table associated with the module.

To complete a secure linkage check before invoking the loaded module, the returned address must be checked to determine whether it is actually within the bounds of the verified object code module. If the symbol table associated with the object code module has been modified, the address can reference code outside of the verified module. The function ISL_CheckAddressWithinModule can to check the address for containment in the verified module.

PARAMETERS

Module (input)
A handle to a verified object module returned by ISL_VerifyLoadedModuleAndCredentials, ISL_VerifyAndLoadModuleAndCredentials, ISL_SelfCheck, ISL_VerifyAndLoadModule, or ISL_VerifyLoadedModule.

Name (input)
An entry point name as required by the platform.

RETURN VALUE

Pointer to the procedure entry point, or NULL if unsuccessful.

SEE ALSO

ISL_CheckAddressWithinModule, ISL_VerifyLoadedModuleAndCredentials, ISL_VerifyAndLoadModuleAndCredentials, ISL_SelfCheck, ISL_VerifyAndLoadModule, ISL_VerifyLoadedModule

NAME

ISL_GetReturnAddress

SYNOPSIS

```
void * ISL_GetReturnAddress()
```

or:

```
void ISL_GetReturnAddress(void *Address)
```

DESCRIPTION

This function facilitates validating that a caller's return address is inside an authorized, verified module.

If function A calls function B at address R and function B calls ISL_GetReturnAddress, ISL_GetReturnAddress returns value R. Function B can validate that address R is within a verified module which should contain function A using ISL_CheckAddressWithinModule.

This function is platform and compiler dependent. The second form may be substituted on platforms and compilers where the first form cannot be realized.

PARAMETERS

Address (output)
If the first form cannot be realized, the value is returned in the Address argument of the second form.

RETURN VALUE

Pointer to a return address if successful, or NULL if unsuccessful. No value is returned in the second form.

SEE ALSO

ISL_CheckAddressWithinModule

NAME

ISL_CheckAddressWithinModule

SYNOPSIS

```
ISL_STATUS ISL_CheckAddressWithinModule
    (ISL_VERIFIED_MODULE_PTR Verification,
    void * Address)
```

DESCRIPTION

The address is checked against the list of address ranges that are valid addresses within the module.

PARAMETERS

Verification (input)
A verified module object returned by the ISL_SelfCheck, ISL_VerifyLoadedModule, ISL_VerifyLoadedModuleAndCredentials, ISL_VerifyAndLoadModuleAndCredentials, or ISL_VerifyAndLoadModule function.

Address (input)
An address to be checked.

RETURN VALUE

ISL_OK is returned if the address is a valid address within the bounds of the module, otherwise ISL_FAIL is returned.

SEE ALSO

ISL_SelfCheck, *ISL_VerifyLoadedModule*, *ISL_VerifyAndLoadModule*, *ISL_VerifyLoadedModuleAndCredentials*, *ISL_VerifyAndLoadModuleAndCredentials*

NAME

ISL_GetLibHandle

SYNOPSIS

```
void * ISL_GetLibHandle
    (ISL_VERIFIED_MODULE_PTR Verification)
```

DESCRIPTION

The system-dependent handle (or address) of the loaded object code module is returned.

PARAMETERS

Verification (input)
A verified module object returned by the ISL_VerifyLoadedModuleAndCredentials, ISL_VerifyAndLoadModuleAndCredentials, ISL_SelfCheck, ISL_VerifyLoadedModule, or ISL_VerifyAndLoadModule function.

RETURN VALUE

The handle to the loaded object code is returned, or NULL if failure.

SEE ALSO

ISL_SelfCheck, *ISL_VerifyLoadedModule*, *ISL_VerifyAndLoadModule*,
ISL_VerifyLoadedModuleAndCredentials, *ISL_VerifyAndLoadModuleAndCredentials*

CAE Specification

Part 5:

CDSA Signed Manifest

The Open Group

Chapter 21

Introduction

21.1 Signed Manifests—An Overview

Signed manifests are used to describe the integrity of a list of digital objects of any type and to associate arbitrary attributes with those objects in a manner that is tightly binding and offers non-repudiation. The integrity description does not change the object being described, rather it exists outside of the object. This means an object can exist in encrypted form and processes can inquire about the integrity and authenticity of an object or its attributes without decrypting the object.

Signed manifests are extensible. Attributes of arbitrary type can be associated with any given digital object. This specification defines the framework for a signed manifest with a minimal set of well known name:value pairs that are common to all signed manifests. The set of valid defined names for name:value pairs will increase over time.

Signed manifests are generated by an application using the Common Security Services Manager Integrity Services Library (CSSM ISL) and are verified by either using ISL or the Embedded Integrity Services Library (EISL). EISL may operate on only a subset of the signed manifest name:value pairs defined in this specification. For further details on manifest constraints for EISL verification, see the Appendix.

21.2 Overview of the Common Data Security Architecture

Signed manifests are essential to the integrity services provided by the Common Security Services Manager (CSSM) within the Common Data Security Architecture (CDSA). CDSA defines an open, extensible architecture in which applications can selectively and dynamically access security services. Figure 21-1 shows the three basic layers of the CDSA:

- System Security Services
- The Common Security Services Manager (CSSM)
- Security Add-in Modules (cryptographic service providers, trust policy modules, certificate library modules, and data storage library modules)

CDSA is intended to be the multi platform security architecture that's horizontally broad and vertically robust.

The CSSM is the core of CDSA. CSSM manages categories of security services and multiple discrete implementations of those services as add-in security modules. CSSM:

- Defines the application programming interface for accessing security services
- Defines the service provider's interface for security service modules
- Dynamically extends the security services available to an application, while maintaining an extended security perimeter for that application, based on integrity services that use signed manifests

Applications request security services through the CSSM security API or via layered security services and tools implemented over the CSSM API. The requested security services are performed by add-in security modules.

Over time, new categories of security services will be defined, and new module managers will be required. CSSM supports elective module managers that dynamically extend the system with new categories of security services. Again CSSM manages the extended security perimeter using signed manifests to ensure integrity and authenticity of the dynamic extensions.

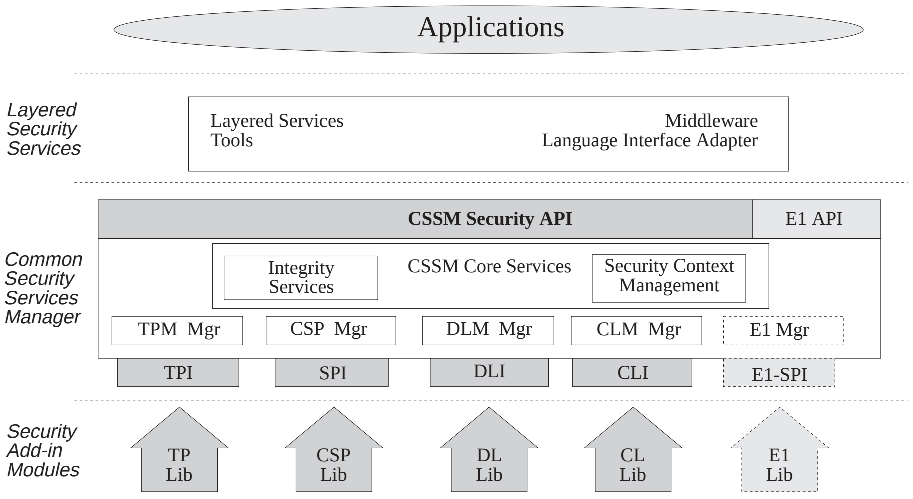

Figure 21-1 The Common Data Security Architecture for all Platforms

Chapter 22

Signed Manifests—Requirements

Signed manifests describe the integrity and authenticity of a collection of digital objects, where the collection is specified as an acyclic connected graph with an arbitrary number of nodes representing arbitrary typed digital objects. Digital signaturing based on a public key infrastructure is the basic integrity mechanism for verifying manifests.

22.1 Requirements

The following are requirements on the signed manifest:

- Manifest must sit outside the objects being signed
- Manifest must be capable of describing an acyclic graph representing an arbitrary number of arbitrary typed digital objects including:
  - Live objects
  - Dynamic objects
- Must be capable of specifying how the object is to be verified. Check the object's integrity by:
  - Reference (URL, pathname, and so on, not the contents of the object)
  - Value (only the contents of the object excluding the pathname)
  - Reference and value (check both the URL, pathname and the contents)
  - Must support one or more unordered signers
- Must support nested signing models. Objects being signed can themselves be signed objects, such as:
  - Signed manifests
  - Objects with embedded signatures
  - PKCS#7 signed messages
- Each signature must carry an unforgeable credential identifying the signer:
  - Digital certificate
  - Public key
  - Fingerprint
- Must be extensible in the type and format of accepted signer's credentials (certificate neutral):
  - X5.09 certificates
  - SDSI certificates
- Signer's credentials can be either:
  - Embedded
  - Referenced via URL
- Cryptographically neutral with respect to signing algorithms

- Performs complete integrity validation:
  — Verify the integrity of the object
  — Verify the integrity of the manifest
  — Runtime continuous verification for live objects
- Signature format must be based on standards
- Manifest format must be based on standards
- Support emerging standards:
  — New signature block formats
  — New certificate formats
  — use single pass verification of signature(s)
  — Verification must be capable of managing progressively rendered object referents

Chapter 23

Signed Manifests—The Architecture

Signed manifests describe the integrity and authenticity of a collection of digital objects, where the collection is specified as an acyclic connected graph with an arbitrary number of nodes representing arbitrary typed digital objects. Digital signaturing based on public key infrastructure is the basic integrity mechanism for manifests. The signed manifest is data type-agnostic allowing referents in the manifest to be other signed manifests or other types of signed objects.

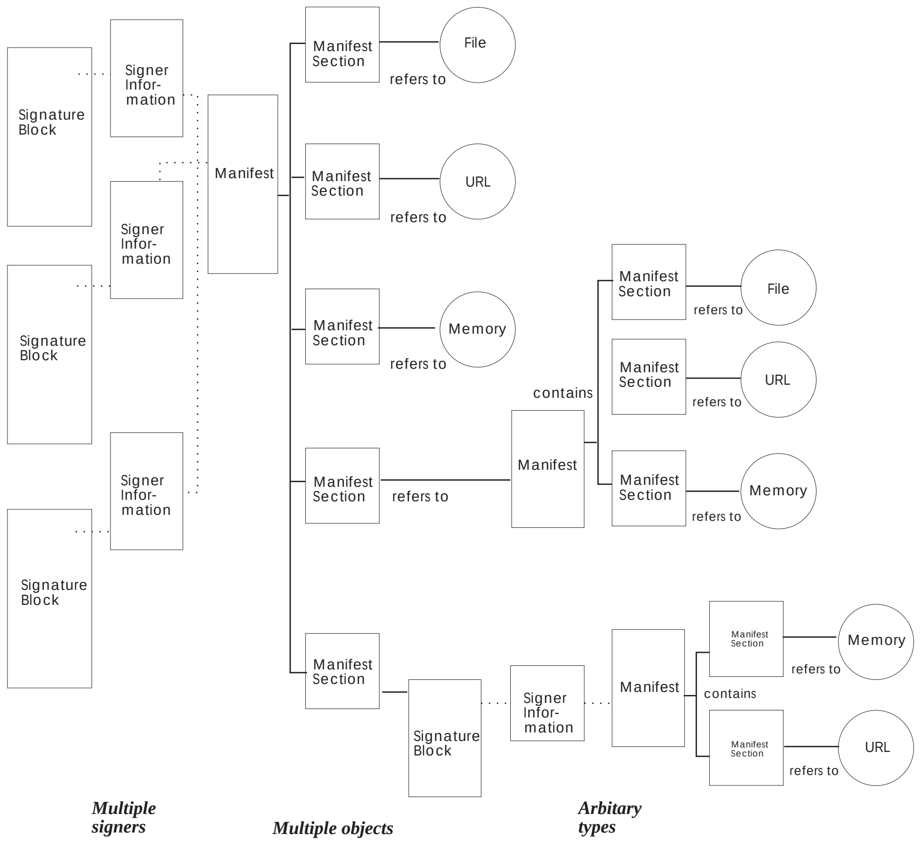

Figure 23-1 Signed Manifest Architectural View

The signed manifest is built from the following components:

- The *manifest* describes a collection of digital objects. It contains one or more *manifest sections*, where each section refers to one of the objects within the collection of objects being described. A section contains a reference to the object, attributes about the object, a list of digest algorithm identifiers that were used to digest the object, and a list of the associated digest values. The description is human-readable.
- The *signer's information* describes a list of references to one or more sections of the manifest. Each reference includes a signature information section which contains a reference to a manifest section, a list of digest algorithms identifiers used to digest the manifest section, a list of digest values for each specified algorithm identifier, and any other attributes that the signer may wish to be associated with the manifest section. It is possible for a signer to sign

only part of a manifest description. Using this structure, it is possible to add signer-specific assertions or attributes to the object being signed. This description is human-readable.

- The *signature block* contains a signature over the signer's information. The signature block is encoded in the particular format required by the signature block representation, for example, for a PKCS#7 signature block, the encoding format is BER / DER.

The relationship of these components is shown in Figure 23-2.

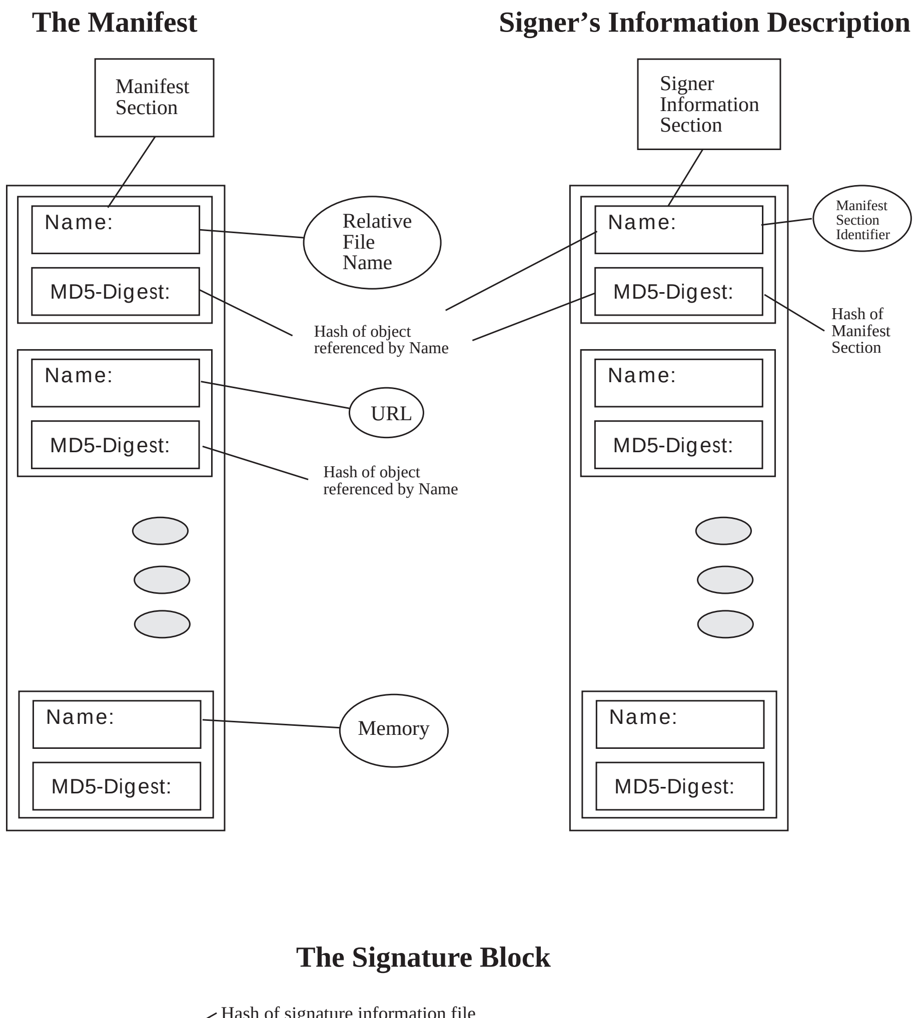

Figure 23-2 Relationships of Manifest, Signer's Information and Signature Block

These three objects must be zipped to form a single set of credentials. Multiple implementations of standard zip algorithms interoperate on one or more platforms, hence a zipped, signed manifest retains a substantial degree of interoperability.

The format used to describe both the manifest and the signer’s information are a series of Name:Value pairs, (RFC 822). Binary data of any form is represented in base64. Continuations are required for binary data which causes line length to exceed 72 bytes. Examples of binary data are digests and signatures.

Chapter 24

Format Specification

This section presents the format specification for the components that make up a signed manifest.

24.1 The Manifest

The purpose of the manifest is to unambiguously describe a list of referents so that its integrity and authenticity may be established. This is accomplished by including:

- The name of the referent
- Metadata about the referent
- How the message digest is to be computed on the object:
  — Message digest algorithm identifier
  — Message digest value

A manifest is composed of header information followed by a list of sections. A section unambiguously describes a referent. The use of metadata is defined below.3.2.1

24.1.1 Manifest Header Specification

A manifest begins with the manifest header, which contains at a minimum the version number:

```
Manifest-Version: 2.0
```

Optionally, a version required for use may be specified:

```
Required-Version: 2.0
```

24.1.2 Manifest Sections

The manifest section describes a referent, attributes about that referent, and the integrity of the referent (hash value). A manifest section is extensible, therefore it is not possible to define the entire list of headers that may be used. The minimum required headers and a list of well-known extended headers is provided to support interoperability with other implementations.

Well formed manifest sections begin with the **Name** token and a corresponding referent as a value.

For a listing of the common headers and their meanings see the appendix.

Multiple hash algorithms may be listed and the corresponding hash value must be present for each algorithm used.

Name values must be unique within a manifest. For example:

```
Name: SomeObject
MAGIC: UsesMetaData
Integrity-TrustedSigner: Some Certificate

Name: SomeObject
Digest_Algorithms: MD5
MD5-Digest: xxxx
```

is not a valid construction, because the sections cannot be distinguished.

If duplicate sections are encountered only the first is recognized. Nonrecognized headers are ignored.

24.1.3 Format Specification

The sections specifies the grammar for the manifest description and signer information descriptions. Each begins with a header which serves to distinguish its version or required version numbers followed by a list of sections. The header specification for both manifest and signer descriptions is presented first followed by the specification for sections.

In this specification, terminals are specified in all capital letters with non-terminals being specified in lower case. An asterisk indicates 0 or more of the item that follows, while a plus (+) indicates 1 or more of the item that follows.

The format specification for the header of a manifest description is:

```
manifest: "Manifest-Version: 2.0" newline
           +manifest-entry

manifest-start: section

; Optional header is
; Required-Version: number "." number
;
; Required-Version indicates that only tools of the given version
; or later can be used to manipulate the file.

; The value of Digest-Algorithms is a whitespace-separated-list:

whitespace-separated-list: +headerchar *whitespace
                    whitespace-separated-list
                    | +headerchar
```

The format specification for signer information is:

```
signer-information: "Signature-Version: 2.0" newline
          +signer-info-entry

signer-info-entry: section

; Optional header is
; Required-Version: number "." number
;
; Required-Version indicates that only tools of the given version
; or later can be used to manipulate the file.
```

The format specification for a section (both manifest and signer information) is:

```
section:nameheader  *header +newline
newline: CR LF
       | LF
       | CR (not followed by LF)

nameheader:"Name:" *header+newline
        *continuation

header: alphanum *headerchar ":" SPACE *otherchar newline
        *continuation

continuation: SPACE *otherchar newline

; RFC822 defines +(SPACE | TAB) as the continuation.
; Using SPACE *otherchar newline
; ensures that continuations are always recognized

alphanum: {"A"-"Z"} | {"a"-"z"} | {"0"-"9"}

headerchar: alphanum | "-" | "_"

otherchar: any Unicode character except NUL, CR and LF

whitespace: SPACE | TAB

; Also: To prevent damage to files sent via simple e-mail, no
; headers can begin with the four letters "From".

; When version numbering is used:

number: {"0"-"9"}+

; The number 1.11 is considered to be later than 1.9
; Both major and minor versions must be 3 digits or less.
```

A section begins with the **Name** token and ends when a new section begins or an end of file is encountered.

24.1.4 MAGIC—A Flagging Mechanism

The keyword **MAGIC** is used as a general flagging mechanism. It indicates to the verification mechanism that it must be able to parse and interpret the value associated with this keyword:value pair or the verifier cannot properly verify the integrity of the referent object. The **UsesMetaData** value indicates that this manifest section contains metadata statements which specify how to properly digest and verify the referent object.

24.1.5 Metadata

Metadata qualifies either the manifest or the referent object. Definition of a specification language for metadata is ongoing research. This specification uses the Dublin Core set and a new framework developed as part of this specification called the integrity core set. (See the appendix for details on these specification languages).

Metadata is described by using name:value pairs, where the format of name specifies both the metadata set being used as well as the name element from the set:

```
(Meta Data Set ID)-(Element Name):Value
```

For example the Integrity Core set element TrustedSigner would be described as:

```
Integrity-TrustedSigner: Some Certificate
```

24.1.6 Ordering Metadata Values

When metadata attributes must be processed in some order-dependent manner, the token **Ordered-Attributes** must be specified by the manifest definer and used by the manifest verifier. An example of an order-dependent process is a referent object that is first hashed and then compressed before being transmitted with the manifest. The verifier must decompress the referent before computing the digest value of the object. An example of a manifest section with ordering metadata is:

```
Name: ExampleFile
SectionName: Example of ordered operations on a referent
Ordered-Attributes: SHA1_Digest, Compression
Digest_Algorithms: SHA1
SHA1_Digest: <base64 encoded value>
Compression: SomeSuperFastAlgo
```

This manifest section specifies that the referent has ordered attributes of SHA1_Digest and Compression. The values that appear as the **Ordered-Attributes**, must be further qualified by other attributes appearing within this manifest section. The values of the **Ordered-Attributes** token must be an exact match with the names for other attributes within the section.

The listed order is relative to the signing operation, which implies that the verification operation must reverse the order of these operations.

24.1.7 Manifest Examples

```
  Manifest-Version: 2.0

  DublinCore-Title: Signed Manifest Format Proposals

Name: http://developer.intel.com/ial/security/CSSMSignedManifest.ps
  SectionName: Intel Manifest Format
  Digest_Algorithms: MD5
  MD5-Digest: (base64 representation of MD5 hash)
  MAGIC: UsesMetaData
  Integrity-Verifydata: Reference-Value
  DublinCore-Title: Signed Manifest File Format
  DublinCore-Subject: Manifest Format
  DublinCore-Author: CSSM Manifest Team
  DublinCore-Language: ENG
  DublinCore-Form: text/postscript
```

```
Name: http://www.javasoft.com/jdk/SignedManifest.html
SectionName: JavaSoft Manifest Format
Digest_Algorithms: MD5
MD5-Digest: (base64 representation of MD5 hash)
MAGIC: UsesMetaData
Integrity-Verifydata: Reference-Value
DublinCore-Title: JavaSoft Signed Manifest Specification
DublinCore-Subject: Manifest Format
DublinCore-Author: Someone from JavaSoft
DublinCore-Language: ENG
DublinCore-Format: text/html
```

24.2 Signer′s Information

The signer's information records the intent of a signer, when signing a manifest. This allows the signer to indicate which sections of the manifest are being signed, and to embed attributes or assertions in headers supplied by individual signers, rather than the manifest owner.

24.2.1 Signing Information Header

The header is the first token in the signer's information description. It must contain the version number for this specification.

```
Signature-Version: 2.0
```

General information supplied by the signer that is not specific to any particular referent should be included in this header.

24.2.2 Signer′s Information Sections

Each section contains a list of manifest section names. Each named section must be present in the manifest file. Additional metadata statements may be included here. A digest value of the named manifest section is also present.

Referents appearing in the manifest sections but not in the signer's information are not included in the hash calculation. This allows subsets of the manifest to be signed.

A signature section begins with the **Name** token. There must be an exact match between a **Name:value** pair in the manifest file.

The following are required:

```
Name: URL or relative pathname
Digest_Algorithms: MD5
(algorithm)-Digest: (base-64 representation of hash)
```

24.2.3 Signing Information Examples

```
Signature-Version: 2.0

Name: ./MyFiles/File1
SectionName: File1 Section
Digest_Algorithms: MD5
MD5-Digest: (base64 representation of MD5 hash)

Name: ./MyFiles/File2
Digest_Algorithms: MD5
MD5-Digest: (base64 representation of MD5 hash)
```

24.3 Signature Blocks

A signature block contains the actual formatted signature generated as part of the digital signing process. The signature is computed by hashing the corresponding signer's information and then encrypting that hash using the signer's private key. Signature block encoding is determined by the type of signature block being used. For example, PKCS#7 signatures use BER / DER encoding.

Chapter 25

Signed Manifests—Verifying Signatures

Validating the integrity of a referent object is a two-step process. The first step is to validate the integrity of the manifest itself. Step two checks the integrity of the particular referent.

25.1 Verifying the Manifest

The procedure for verifying the signer’s information is:

1. Select the signer to be verified
2. Compute the digest of the corresponding signer’s information using the digest algorithm indicated in the signature block file
3. Compare computed digest against digest in the signature block

If the digest values match, the next step is to validate the integrity of the manifest sections as defined by signer’s information. The procedure for verifying the manifest sections is:

1. For each signature section in the signer’s information:
   - Locate the corresponding manifest section matching either by **Name** or **SectionName** values
   - Compute the digest of that section using the digest algorithm indicated in the signature information file
   - Compare the computed digest against the value listed in the signature information file

If the digest values match, the final step is to validate the integrity of the referents listed in the manifest sections.

25.2 Verifying Referents in the Manifest

Once the manifest has been successfully verified, individual referents in the manifest can be verified. The verification process requires the use of values provided in the manifest. If the **MAGIC** token appears in the manifest section, the verifier must interpret and correctly act upon the **MAGIC** value. If the value UsesMetaData is specified, the verifier must check for one or more **Integrity** tokens as metadata statements. If this token appears, the digest must be calculated according to the instructions provided by the **Integrity** token. Verification is completed by computing the digest of the referent (as controlled by the metadata) and comparing the result to the value recorded in the manifest section.

Chapter 26

File-Based Representation of Signed Manifests

This section describes the file system based representation of a signed manifest. A signed manifest consists of:

- A manifest description
- Zero or more signer information descriptions
- Zero or more signature blocks

There are two representations for a signed manifest in the file system. The first representation maintains compatibility with existing implementations of signed manifests, while the second representation relaxes some of the constraints imposed by the first.

26.1 The META-INF Directory—First File-Based Signed Manifest Representation

The first representation is as a file set which resides in a well-known directory called META-INF. This directory is relative to the file-based referents in the manifest. The manifest description is written in a file called MANIFEST.MF. All pathnames appearing in the sections of MANIFEST.MF are relative to the parent directory of META-INF.

The signer information is placed in the META-INF directory under the filename x.SF, for some string x containing only the characters A-Z 0-9 and dash or underscore. x must not be more than eight characters, for instance MySig.SF.

Signature block filenames must share the base filename of the corresponding signer's information file. The filename extension identifies the signaturing type:

```
.RSA      (PKCS7 signature, MD5 + RSA)
.DSA      (PKCS7 signature, DSA)
.PGP      (Pretty Good Privacy Signature)
```

26.2 The ESW File—Archive-Based Signed Manifest Representation

The constraints placed by the first file-based representation are relaxed by archiving the signed manifest file set into one file. This archive file is called an Electronic Shrink Wrap file and must end in the filename extension .ESW. The .ESW file must reside in the parent directory relative to all pathnames of file-based referents in the manifest.

The archive format of an .ESW file must conform to the archive format specified by PKWARE. (See *http://www.pkware.com/download.html* for additional information.)

The signed manifest file set that appears in a .ESW archive must conform to the filename formats stated in the previous section, for example, an .ESW archive must:

- Have all manifest file names must be relative to META-INF
- Contain only one META-INF/MANIFEST.MF
- Contain zero or more META-INF/x.SF files
- Contain zero or more META-INF/x.(RSA, DSA, and so on) files

It is the responsibility of the verification program to select the correct .ESW file for the objects to be verified.

26.3 Representation Constraints

Filenames appearing in the META-INF directory are restricted to the characters A-Z 0-9 and dash or underscore. Base filenames consist of at most eight characters.

The names "META-INF", "MANIFEST.MF", and the filetype ".SF" should be generated as upper case, but must be recognized in upper and lower case.

File system pathnames appearing in a manifest must be relative to the parent directory of META-INF.

There can exist only one MANIFEST.MF file in a META-INF directory.

For each *x*.SF file there must be a corresponding signature block file.

Before parsing:

- If the last character of the file is an EOF character (code 26), the EOF is treated as whitespace.
- Two new lines are appended (one for editors that don't put a new line at the end of the last line, and one so that the grammar doesn't have to special-case the last entry, which may not have a blank line after it).

Headers:

- In all cases for all sections, headers which are not understood are ignored.
- Header names are case insensitive. Programs which generate manifest and signer information sections should use the cases shown in this specification.
- Only one "Name:" header may appear in a given section.

Versions:

- Manifest-Version and Signature-Version must be the first token in a manifest and signer's information, respectively. These token names are case sensitive. All other token headers within a section can appear in any order.

Ordering:

- The order of manifest entries is significant only in that the original digest value is computed based on the original ordering.
- The order of signature information entries is significant only in that the original digest value is computed based on the original ordering.
  - — Manifest and signer information sections entries may not be re-ordered during transmission, because this will adversely effect the digest value.

Line length:

- The line length limit is 72 bytes (not characters), in its UTF8-encoded form. Continuation lines (each beginning with a single SPACE) must be used for longer values.

Errors:

- If a file cannot be parsed according to this specification, a warning should be generated and the signatures should not be trusted.

Limitations:

- Header names cannot be continued so the maximum length of a header name is 70 bytes (followed by a colon and a SPACE).
- Header names must not begin with the character "<".
- NUL, CR, and LF must not be embedded in header values.
- NUL, CR, LF, and ":" must not be embedded in a header.
- It is desirable to support 65535-byte (not character) header values, and 65535 headers-per-file.

Algorithms:

- No digest algorithm or signature algorithm is mandated by this specification. However, the following algorithms are expected to be in general use:
  — Digest: at least one of MD5 and SHA1
  — Signature block representation: PKCS#7

Chapter 27

Signed Manifests—Examples

The following is a list of examples that serve to illustrate how this specification meets the requirements for signed manifests.

27.1 Static Referent Objects

The manifest:

```
Manifest-Version: 2.0

Name: pictures/ocean.gif
SectionName: Ocean picture
Digest_Algorithms: MD5
MD5-Digest: base64(md5-hash of ocean.gif)

Name: audio/ocean.au
SectionName: Ocean Sounds Audio File
Digest_Algorithms: MD5 SHA1
MD5-Digest: base64(md5-hash of ocean.au)
SHA1-Digest: base64(sha1-hash of ocean.au)
```

The signer's information description:

```
Signature-Version: 2.0

Name: audio/ocean.au
SectionName: Ocean Sounds Audio File
Digest_Algorithms: MD5
MD5-Digest: base64(MD5 Digest of manifest section entitled "Ocean Sounds")
```

The signature block is not shown here, but it would be represented as an ANS.1 encoded PKCS#7 signature block.

Note that the manifest includes two digests for audio/ocean.au, and the signer's information includes only one. At verification time the manifest section that is hashed is treated as opaque data; hence SHA1 digest is included in the hash.

27.2 Dynamic Referent Objects with Verified Source

This example describes a dynamic data source (such as a stock quote service) and its integrity. The manifest names the dynamic data source and qualifies that name with the integrity core metadata set. There is no hash value associated with the dynamic referent, rather integrity is based on verifying trust in the source of the data. The data source is specified in the token **Integrity-TrustedSigner**.

27.2.1 Stock Quote Service

The manifest:

```
Manifest-Version: 2.0

Name: SomeCompany.cert
SectionName: Trusted Root Certificate
Digest_Algorithms: SHA1
SHA1-Digest: XXXX

Name: http://www.stockquote.com
SectionName: Dynamic Stock Quote Service
DublinCore-Format: message/x-pkcs7
MAGIC: UsesMetaData
Integrity-TrustedSigner: Trusted Root Certificate
```

Trusted signer specifies the key holder that must have signed the dynamic object. The manifest section entitled "Trusted Root Certificate" contains a referent to a file where the trusted signer's certificate resides. The integrity of the Trusted Root Certificate is specified by including the hash value of the actual certificate in the manifest. This verifies the identity of the signer.

In this example, the signer has signed all sections of the manifest. The signer's information description appears as follows:

```
Signature-Version: 2.0

Name: SomeCompany.cert
Digest_Algorithms: MD5
MD5-Digest: xxxx

Name: http://www.stockquote.com
Digest_Algorithms: MD5
MD5-Digest: xxxx
```

The PKCS#7 signature block is not shown.

27.3 Embedded or Nested Referent Objects

27.3.1 Signed Objects Whose Signatures Serve to Carry the Object

PKCS#7 signed messages are objects that serve as a carrier for the object being signed as well as the signature for the object. When these enveloped objects are signed using the manifest, the whole object is hashed, treating it just as a generic blob of bits, ignoring its internal structure. To verify these types of objects, the entire object will be hashed and compared to the value in the manifest. If the digest values match, the next step is to verify the integrity of the enveloped object. This two-level verification check is described in the manifest by using the token **Integrity-Envelope** where the token value defines how the internal object must be verified. In the case where the internal object is enveloped by a PKCS#7 signed message, the value would indicate PKCS-7. The manifest description for a PKCS-12 signed object is similar to the manifest description for the PKCS-7 referent shown here.

```
Manifest-Version: 2.0

Name: ExamplePKCS7Data.pk7
SectionName: PKCS#7 Signed Message
Digest_Algorithms: MD5
MD5-Digest: (base64 representation of MD5 hash)
MAGIC: UsesMetaData
Integrity-Envelope: PKCS-7
```

27.3.2 Signed Objects Whose Signature Blocks are Embedded

Referent objects can be other signed objects, where the signature is embedded inside the object itself. When including these objects in a manifest, the entire object (including the embedded signature) is treated as a generic blob of bits during the digest process. However, during verification, it is desirable to verify the embedded signature after all of the manifest components have been verified. This is accomplished by delegating the verification of the embedded signature to the proper verification routines. These verification routines must be identified by the value of the **Integrity-Envelope** token.

```
Manifest-Version: 2.0

Name: http://www.activecontrols.com/shareware/KillerControl.ocx
SectionName: Embedded Signature Object
Digest_Algorithms: MD5
MD5-Digest: (base64 representation of MD5 hash)
MAGIC: UsesMetaData
Integrity-Envelope: Authenticode
```

The manifest section representing the object with an embedded signature indicates this using the **Integrity-Envelope** token. The token specifies that the signature was generated by and can be verified by the Authenticode system from Microsoft. No trusted signer is specified because the knowledge of "who" is trusted to have signed the executable is embedded in the specialized signature checker.

27.3.3 Nested Manifests

Nesting a signed manifest within another signed manifest is used to associate additional signatures and attributes with a package as it travels through its normal channel of handling. For example, in electronic software distribution, the software publisher creates a manifest representing their software product. The product and the manifest are archived together and electronically transmitted to several distributors. Distributors add advertisements, logos, and so on, and create a new manifest that references all the newly-added material and the original archive (including the signed manifest) from the publisher. The distributor transmits this new archive to its resellers who add branding information specific to their location. The reseller creates a manifest referencing their branding material and the material from the distributor, creating three levels of nested manifests.

An example manifest for a software publisher's release includes:

```
Manifest-Version: 2.0

Name: KillerApp.exe
SectionName: Killer Internet Application
Digest_Algorithms: SHA1
SHA1-Digest: XXXX

Name: KillerApp.hlp
Digest_Algorithms: SHA1
SHA1-Digest: XXXX

Name: KillerApp.doc
Digest_Algorithms: SHA1
SHA1-Digest: XXXX

Name: Readme.txt
Digest_Algorithms: SHA1
SHA1-Digest: XXXX

Name: EULA.txt
Digest_Algorithms: SHA1
SHA1-Digest: XXXX
```

The signer information description is:

```
Signature-Version: 2.0

Name: KillerApp.exe
SectionName: Killer Internet Application
Digest_Algorithms: SHA1
SHA1-Digest: YYYY

Name: KillerApp.hlp
Digest_Algorithms: SHA1
SHA1-Digest: YYYY

Name: KillerApp.doc
Digest_Algorithms: SHA1
SHA1-Digest: YYYY
```

```
Name: Readme.txt
Digest_Algorithms: SHA1
SHA1-Digest: YYYY

Name: EULA.txt
Digest_Algorithms: SHA1
SHA1-Digest: YYYY
```

Once the manifest has been created and signed the publisher archives the software release and the signed manifest, and transmits them to a set of distributors.

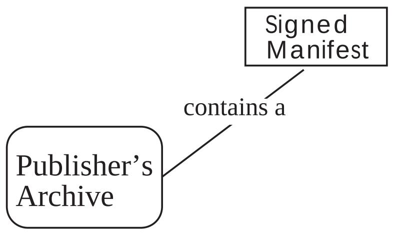

Figure 27-1 Relationship of Publisher’s Archive and Signed Manifest

The distributor creates a new manifest referencing the archive sent by the publisher:

```
Manifest-Version: 2.0

Name: distributor1logo.gif
SectionName: Distributor 1’s logo
Digest_Algorithms: SHA1
SHA1-Digest: XXXX

Name: KillerAppArchive
Digest_Algorithms: SHA1
SHA1-Digest: XXXX
```

The distributor’s signature information is:

```
Signature-Version: 2.0

Name: distributor1logo.gif
SectionName: Distributor 1’s logo
Digest_Algorithms: SHA1
SHA1-Digest: YYYY

Name: KillerAppArchive
Digest_Algorithms: SHA1
SHA1-Digest: YYYY
```

The distributor creates a new archive, combining the new manifest and the original archive sent by the publisher. This new archive is transmitted to resellers.

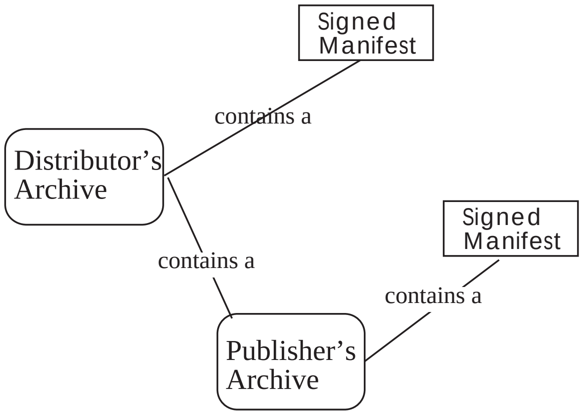

Figure 27-2 Relationship of Distributor's Archive to Publisher's Archive

The reseller creates another new archive, adding their own specific digital objects and including the archive sent by the distributor:

```
Manifest-Version: 2.0

Name: reseller1logo.gif
SectionName: Reseller 1’s logo
Digest_Algorithms: SHA1
SHA1-Digest: XXXX

Name: distributorarchive
Digest_Algorithms: SHA1
SHA1-Digest: XXXX
```

The reseller's signature information is:

```
Signature-Version: 2.0

Name: reseller1logo.gif
SectionName: Reseller 1’s logo
Digest_Algorithms: SHA1
SHA1-Digest: YYYY

Name: distributorarchive
Digest_Algorithms: SHA1
SHA1-Digest: YYYY
```

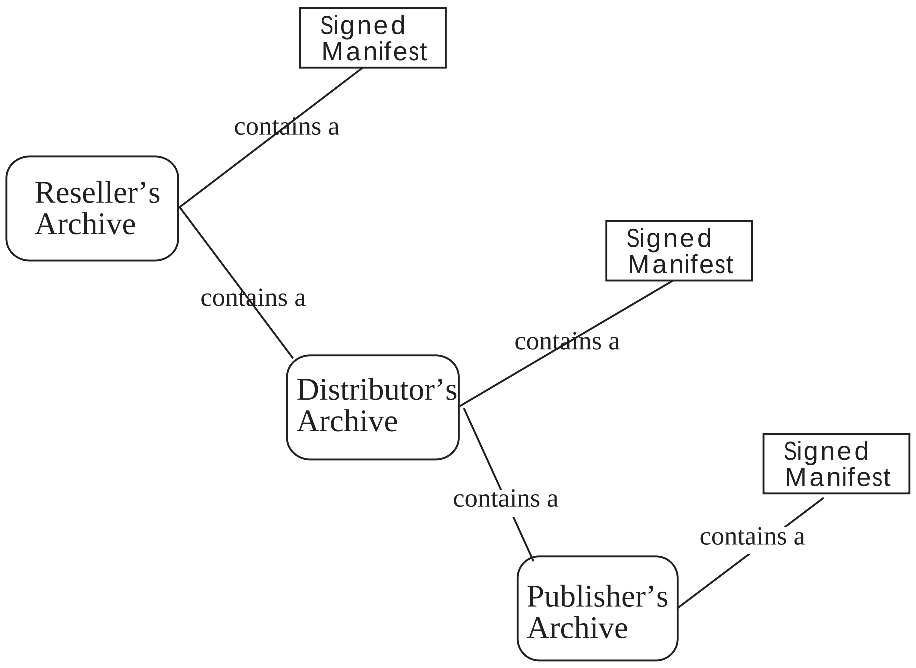

Figure 27-3 Relationship of Reseller to Distributor to Publisher

The reseller's archive includes the distributor's archive, which contains the distributor's manifest. The distributor's archive includes the publisher's archive which contains the publisher's manifest. This results in a manifest being implicitly embedded within another manifest which has in it an implicitly embedded manifest. The embedding is implicit because the manifests are referenced indirectly as part of the archive files.

27.3.4 Signed Portion of an HTML Page

```
Manifest-Version: 2.0

Name: http://www.scripts.com/index#script1
SectionName: Useful Javascripts demo home page
Digest_Algorithms: SHA1
SHA1-Digest: xxx
MAGIC: UsesMetaData
Integrity-VerifyData: namedsectionvalue
Integrity-NamedSectionForm: javascript
```

Only the named section "script1" is used in calculating the signature.

27.3.5 Foreign Language Support/Multiple Hash Values for a Referent

URLs are not unique names for objects. When a browser activates an URL, different documents are returned based on the language preference set in the browser. If the Catalan page is requested, it may not be returned. If there is no Catalan page for that referent, then the default language page is returned. A manifest section must unambiguously describe a referent, therefore the manifest must include a hash value for each of the language representations for a document.

```
Manifest-Version: 2.0

Name: http://www.intel.com/developer/ial/security/
Section Name: Intel’s Data Security Home Page
Digest_Algorithms: SHA1
SHA1-Digest: xxx
SHA1-Digest: yyy
SHA1-Digest: zzz
MAGIC: UsesMetaData
Integrity-VerifyIntegrity: match
```

Three hash values are provided, each for a different language representation of the referent object. The integrity token **Integrity-VerifyIntegrity** specifies that the hash of the referent must match one of the three hash values.

27.3.6 Dynamic Sources with no Associated Data

It is possible to have dynamic referent objects that do not provide associated data. This example is distinct from the stock quote service where the dynamic referent provided data.

```
Manifest-Version: 2.0

Name: telnet://mit.edu/
SectionName: Blessed telnet site
```

It is not feasible to hash the results of a telnet session. It is useful to list the telnet session as a referent of a manifest because it aggregates the session with other referent objects in the manifest. No hash values are provided for the telnet session because the section hash and hence the referent URL hash are provided in the signature information description.

27.3.7 Resources that Transform Locations

A referent in a manifest section can describe a resource that is either near (a memory image or local file) or far (an http address to a web server). A manifest section can also describe the integrity of an object without specifying its exact location. Consider a referent to an audio file. The file can be on a local file system or on a remote audio file server accessible using the Internet. A single manifest can be used to describe the integrity of this object using the token **ResourceProxy**.

```
Name: MyAudioFile.hqa
Section Name: High Quality Audio File
MAGIC: UsesMetaData
Integrity-ResourceProxy: http://www.HighQualityAudio.com\
                         /cgi-bin/StreamAudio?SKU=21339191XW

Name: http://www.HighQualityAudio.com/cgi-bin/StreamAudio?\
                                              SKU=21339191XW
Digest_Algorithms: SHA1
```

```
SHA1-Digest: xxx
```

Integrity-ResourceProxy informs the integrity verifier of two facts concerning the referent:

- If the referent does not exist in the location specified, then defer to the reference specified by **Integrity-ResourceProxy**
- When comparing digest values, use the value associated with the referent identified by the resource proxy.

When verifying the referent MyAudioFile.hqa, if the file does not exist in the local directory, then it can be found at: *http://www.HighQualityAudio.com/cgi-bin/StreamAudio?SKU=21339191XW.*

No digest value is indicated in the manifest section for MyAudioFile.hqa. The digest value is specified in the section describing the ResourceProxy.

It is an error to specify a digest value within the same manifest section where **Integrity-ResourceProxy** has been specified. If encountered the specified digest value will be ignored.

Appendix C

Signed Manifests

C.1 Extensions to the JavaSoft/Netscape Specification

The JavaSoft signed manifest specification states that:

> ''*It is technically possible that different entities may use different signing algorithms to share a single signature file. This violates the standard, and the extra signature may be ignored.*''

The Intel-signed manifest specification allows multiple signers to be included in the PKCS#7 signature block as long as each signer is signing the same manifest sections.

The only recognized valid **MAGIC** value for this specification is UsesMetaData.

C.2 Core Set of Name:Value Pairs

Name
This token specifies the referent for the manifest section.

SectionName
This token is informational only to the section it appears in.

(Digest algorithm ID)
Well-known digest algorithm identifiers are:
MD5, SHA, SHA1, MD2, MD4

Ordered-Attributes
This token specifies that some metadata values appearing within this manifest section must be processed in an order-specific manner. The order indicated is relative to the signing operation. The verification operation must reverse the order indicated.

MAGIC
This token is used as a general flagging mechanism. The only associated value is UsesMetaData.

Integrity
DublinCore
These tokens specify metadata contexts within which the name:value pairs have meaning.
SchemaInfo
This is a well known name that should be defined in every metadata set. It points to a resource that provides human readable text describing the metadata set. For instance:

```
Integrity-SchemaInfo: http://developer.intel.com/ial/security/
                                                IntegritySchema.html
```

points to a resource where a human readable description of the Integrity set resides.

C.3 Metadata

Metadata is used to qualify the referent by providing additional information that cannot be included in the name. The definition of valid metadata values is an ongoing effort. This specification incorporates the Dublin Core metadata set *http://www.ckm.ucsf.edu/meta/mguide3.html* and a new integrity core set to describe the integrity of the referents.

C.3.1 Integrity Core

The Integrity Core is a set of minimal values used to describe the integrity of information resources. The metadata name for this set is **Integrity**.

The core elements are:

- VerifyData
- TrustedSigner
- VerifyIntegrity
- NamedSectionForm
- NamedSection
- Envelope
- ResourceProxy

Integrity-VerifyData
This token describes how to retrieve the referent object for hashing. Valid values are:

- Reference—hash only the reference, exclude the contents.
- Reference-value—this is the default, hash both the name and contents.
- Match—match exactly one of the hash values provided for the referent.
- Namedsectionvalue—hash the contents identified by the named section specified.
- Manifest—the referent is itself a signed manifest.
- Signedarchive—the referent is an archive which contains a manifest.

Integrity-TrustedSigner
A token defines trusted signers for signed dynamic data sources. The signer must be described in another manifest section as an information resource. The value for this name:value pair must be the value of the referent (the value of the **NAME** token) in the manifest section where the trusted signer is described.

Integrity-VerifyIntegrity
This token is used to create descriptions, which cannot be expressed using **VerifyData** or **TrustedSigner**. Valid values are:

- Match—indicates that the hash value computed must match one of the values listed.
- Ondemand—this serves as a flag indicating that verification of the referent should be deferred until the point of rendering. This is useful when the referent is a large streaming object which will be incrementally verified as well as rendered.

Integrity-NamedSectionForm
This token defines the format of the partial section to be hashed. This is used to describe integrity of a portion of a compound object, such as a Microsoft PowerPoint slide residing in

a Microsoft Word document.

Integrity-NamedSection
This token identifies the section to be hashed.

Integrity-Envelope
This token indicates that the referent itself is a signed object, where the signature envelopes the object or is embedded within the object. Valid values are:

- PKCS-7—the object is a signed message conforming to PKCS#7 specification.
- Authenticode—object has been signed by Microsoft's Authenticode system.

Integrity-ResourceProxy
This token indicates that the location of the referent object changes over time. An example is an executable image. To describe the integrity of the object, a manifest must correctly reference the object as a file (which is far away) and as a loaded, executing memory image (which is nearby).

C.3.2 Dublin Core

Details of the specification of the Dublin Core set is outside the scope of this document. Refer to *http://www.oclc.org:5046/research/dublin_core/* for details on this metadata set.

C.3.3 PKWARE Archive File Format Specification

Reference documentation can be found at: *http://www.pkware.com/download.html*.

CAE Specification

Part 6:

CSSM Elective Module Manager

The Open Group

Chapter 28

Introduction

CDSA defines an interoperable, extensible architecture in which applications can selectively and dynamically access security services. The architecture is extensible is two dimensions:

- New categories of security services can be installed and accessed through the infrastructure.
- Independent and competitive implementations of specific security services can be installed and accessed through the infrastructure.

Figure 28-1 shows the three basic layers of the Common Data Security Architecture:

- System Security Services
- The Common Security Services Manager (CSSM)
- Security Add-in Modules

The Common Security Services Manager (CSSM) is the core of CDSA. CSSM manages categories of security services and multiple discrete implementations of those services as add-in security modules. CSSM:

- Defines the application programming interface for accessing security services
- Defines the service providers interface for security service modules
- Dynamically extends the categories of security services available to an application

Applications request security services through the CSSM security API or via layered security services and tools implemented over the CSSM API. The requested security services are performed by add-in security modules. Four basic types of module managers are defined:

- Cryptographic Services Manager
- Trust Policy Services Manager
- Certificate Library Services Manager
- Data Storage Library Services Manager

Over time, new categories of security services may be defined, and new module managers may be required. CSSM supports elective module managers that dynamically extend the system with new categories of security services.

Below CSSM are add-in security modules that perform cryptographic operations, manipulate certificates, manage application-domain-specific trust policies, and perform new, elective categories of security services. Add-in security modules can be provided by independent software and hardware vendors as competitive products. Applications use CSSM module managers to direct their requests to add-in modules from specific vendors or to any add-in module that performs the required services. A single add-in module can provide one or more categories of service. Modules implementing more than one category of service are called multi-service modules.

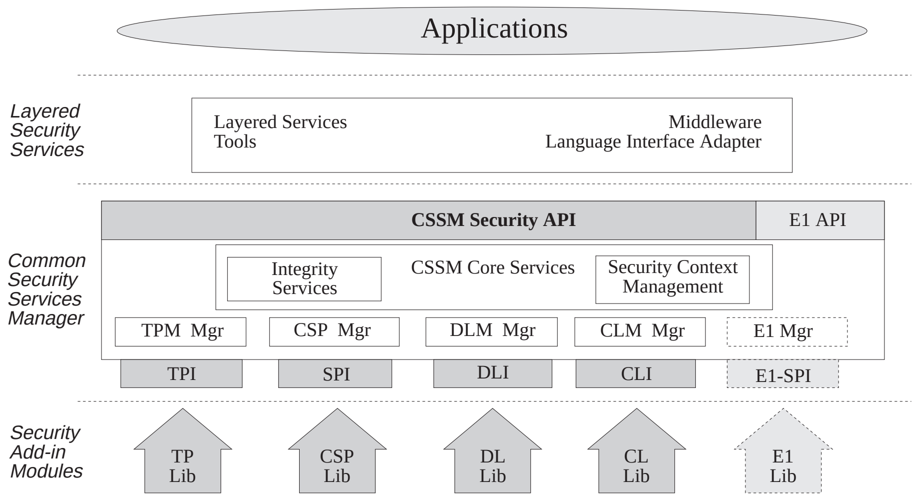

Figure 28-1 Common Data Security Architecture for all Platforms

CSSM core services support:

- Module management
- Security context management
- System integrity services

The module management functions are used by applications and by add-in modules to support module installation, registration of module features and attributes, and queries to retrieve information on module availability and features.

Security context management provides secured runtime caching of user-specific, cryptographic state information for use by multi-step cryptographic operations, such as staged hashing. These operations require multiple calls to a CSP and produce intermediate state that must be managed. CSSM manages this state information for the CSP, enabling more CSPs to easily support multiple concurrent callers.

The CSSM Embedded Integrity Services Library (EISL) provides secure verification services. CSSM, add-in modules, elective module managers, and optionally applications use EISL to verify the identity and integrity of components of CDSA. CSSM uses EISL functions to check dynamic components as they are added to the system. These components include: elective module managers, add-in service modules, and CSSM itself. In the future, applications will also be verified by CSSM before providing service to the application. The EISL services focus on detecting impostors or unauthorized components and tampering of authorized components.

Chapter 29

Overview of Elective Module Managers

To ensure long-lived utility of CDSA and CSSM APIs, the architecture includes several extensibility mechanisms. Elective module managers is a transparent mechanism supporting the dynamic addition of new categories of service. Elective service categories create areas for totally new products. When an elective service category is defined, at least one instance of an add-in module will also be developed to provide that service.

Elective services extend CSSM. They define their own application programming interfaces and service provider interfaces. For example, key recovery can be an elective service. Some applications will use key recovery services (by explicit invocation) and other applications will not use it. Audit logs can be an elective service. Applications wishing to maintain a log can do so, other applications will not use that facility. CSSM's elective module management facilities provide a set of mechanisms that support runtime inclusion of new APIs and their corresponding SPIs. Standardization of the new APIs and SPIs is in addition to the current CDSA standards. The additions can be standardized as an enhancement to CDSA or as an independent standard that is adopted and used within CDSA. The elective module management mechanisms allow CDSA implementations to easily and quickly incorporate these new standards. CSSM does not have a priori knowledge of the elective APIs, but applications have complete knowledge of the new APIs in order to explicitly invoke the services provided through those APIs.

29.1 Built-In Policies and Application Exemptions

The elective module manager can define built-in checks for normal controlled functioning of the new security services defined by the module manager. The elective module manager can define categories of exemption corresponding to these checks. Applications request exemption from these checks using the CSSM_RequestCssmExemption function. Exemptions are granted if the requester provides credentials that:

- Are successfully authenticated by CSSM
- Carry implied authorization for the requested exemptions

Credentials are said to carry implied authorization if they can be verified based on points of trust specified by the authorizing entity.

An elective module managers can define trust points in addition to CSSM's trust points by including application authentication keys in the module manager's description contained in its signed manifest credentials. CSSM can use these public keys as additional trust points for authenticating applications requesting exemptions. When an exemption has been granted, it is the responsibility of the elective module manager to not enforce the built-in check corresponding to the granted exemption.

To define new categories of exemption, the elective module manager defines a new, unique CSSM_EXEMPTION_MASK flag. This bitmask represents the set of exemptions requested by an application. Applications can change their exemption status multiple times during execution. Each request for exemption requires a separate authentication check .

29.2 Transparent, Dynamic Attach

Applications are not explicitly aware of module managers within CSSM. Applications see a uniform set of interface management services provided by CSSM across all types of security service categories. In reality, some of those services are provided by the CSSM core functions (that is, applicable to all service types) and the remainder are provided by each module manager for their respective security service category.

Applications are aware of instances of add-in modules, not the module managers that control access to those modules. Before requesting services from an add-in service provider (via APIs defined by a module manager), the application invokes *CSSM_attach* to obtain an instance of the add-in service provider. Figure 29-1 shows the sequence of processing steps. If the module is of an elective category of service, then CSSM transparently attaches the module manager for that category of service (if that manager is not currently loaded). The module manager must perform the CSSM-defined bilateral authentication protocol. This protocol is used to ensure CSSM-wide integrity when any component is dynamically added to the CSSM runtime environment. (This protocol is described in more detail in a later section of this specification.) Once the manager is loaded, the APIs defined by that module are available to the application.

The dynamic nature of the elective module manager is transparent to the add-in module also. This is important. It means that an add-in module vendor need not modify their module implementation to work with an elective module manager versus a basic module manager.

There is at most one module manager for each category of service loaded in CSSM at any given time. When an elective module manager is dynamically added to serve an application, that module manager is a peer of all other module managers and can cooperate with other managers as appropriate.

Elective module management defines a set of mechanisms that support runtime inclusion of new APIs and their corresponding SPIs. Standardization of the new APIs and SPIs is in addition to the current CDSA standards. The additions can be standardized as an enhancement to CDSA or as an independent standard that is adopted and used within CDSA. The elective module management mechanisms allow CDSA implementations to easily and quickly incorporate these new standards. CSSM does not have a priori knowledge of the elective APIs, but applications have complete knowledge of the new APIs in order to explicitly invoke the services provided through those APIs. The elective module manager is responsible for checking instance compatibility with the CSSM that loaded the manager. Compatibility can be based on a combination of the CSSM's GUID, CSSM's Interface GUID, and CSSM's major and minor version number. CSSM APIs can be invoked to obtain these values. These values also represent the instance level of the basic module managers that are always present in the CSSM. In rare cases, elective module managers may have dependencies on each other. In this case compatibility between elective module managers is the responsibility of the elective module managers. These checks must be performed in a manner that does not depend on the order in which the caller attaches depend services that are supported by elective module managers. Compatibility checks among dependent, elective module managers can be checked using the event notification interface for communication among module managers. When an attached application detaches from an add-in service module, CSSM will also unload the associated module manager if it is not in use by another thread, process, or application.

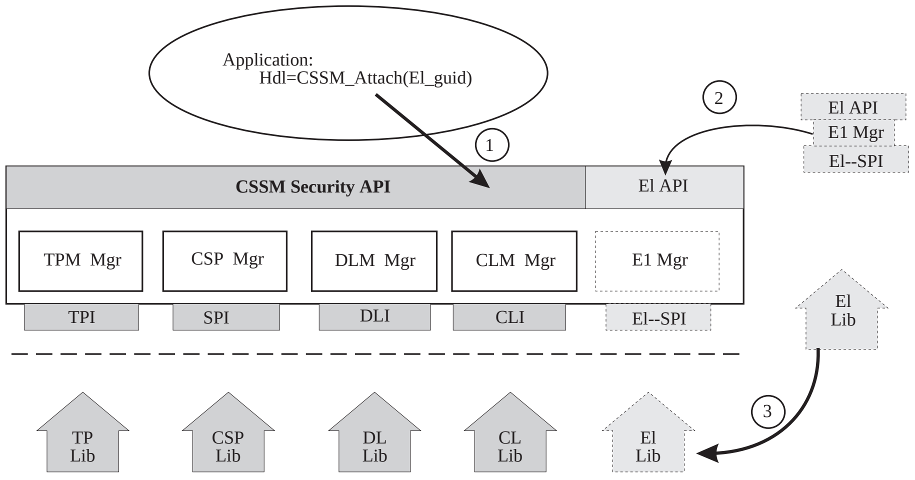

Figure 29-1 Steps to Attach an Add-In Module and load its EMM

29.3 Registering Module Managers

Module managers are installed and registered with CSSM in a similar manner to add-in modules. CSSM records module manager information in the CSSM registry. This information can be queried, but typically only system administration applications will use registry information about module managers. For example, a smart installer for an add-in module may confirm that the corresponding module manager is also installed on the local system. If not, then the installer can install the required module manager with the add-in module. This does not effect the implementation of the add-in module itself, just the install program for that add-in module.

29.4 State Sharing Among Module Managers

Module managers may be required to share state information in order to correctly perform their services.

When two or more module managers share state, each manager must be able to:

- Inform the other module managers of its presence in the system
- Request notification of certain states or activities taking place in the domain of another module manager
- Gather event information from other module managers
- Inform the other module managers of its imminent removal from the system

The other module managers must be able to:

- Change their behavior based on the presence or non-presence of other module managers in the system
- Accept and honor requests from other module managers for ongoing state and activity information

- Issue event notifications to other module managers when events of interest occur

When module managers share state information they must implement conditional logic to interact with each other. Two module managers can share state information by several different mechanisms:

- Invoking known, internal, module manager interfaces
- Using operating system supported state-sharing mechanisms, such as shared memory, RPC, event notification, and general interrupts
- Using a CSSM-supported event notification service

The first two mechanisms depend on platform services outside of CDSA. Module managers that share state information can use all of these mechanisms.

CSSM-supported event notifications require that all module managers implement and register with CSSM an event notification entry point. Module managers issue notifications by invoking a CSSM function, specifying:

- The source manager
- The destination manager
- The event type
- Notification ID (optional)
- Data Values (optional)

CSSM delivers the notification to the destination module manager by invoking the manager's notification entry point.

Typical event types include:

- Module manager loaded
- Module manager unloaded
- Selected Service Request
- Reply

Module managers that share state information are not required to use the CSSM event notification mechanism. These types of events, requests, and notifications can be shared using the other platform dependent mechanisms. CSSM provides this simple mechanism specifically for situations where other platform services are not readily available.

Chapter 30

Administration of Elective Module Managers

30.1 Integrity Verification

CSSM provides a set of integrity services that can be used by elective module managers to verify the integrity of themselves and of other components in the CSSM environment. CSSM requires the use of a strong verification mechanism to screen all components as they are dynamically added to the CSSM environment. This aids in CSSM's detection and protection against the classic forms of attack:

- Class attacks
- Stealth attacks
- Man-in-the-middle attacks

CSSM's verification mechanism is provided by the Embedded Integrity Services Library (EISL). This library defines basic integrity services and packaged services that implement standard integrity protocols. CSSM extends these services by defining additional layered protocols, such as bilateral authentication, to perform identity, integrity, and authorization checks during dynamic binding.

CSSM verifies elective module managers prior to loading them. Verification prior to loading prevents activating file viruses in infected modules. Once verified, CSSM can use the module manager's signed manifest to perform address validity checks, insuring secure linkage to the module manager.

A module manager is required to verify the integrity of its own subcomponents and of CSSM as part of the transparent attach process. To verify its own components, the module manager should use the Embedded Integrity Services Library's self check function. This in-memory verification prevents *stealth* attacks where the disk-resident object file is unaltered, but the loaded code is tampered. Additional functions are provided by the EISL for verifying the integrity of and secure linkage with CSSM. CSSM initiates this part of the verification process by invoking the ModuleManagerAuthenticate function implemented by the elective module manager.

30.2 Module Manager Credentials

Integrity verification is based on the module manager's credentials. A complete set of credentials must be created for each CSSM elective module manger as part of the software manufacturing process. These credentials are required by CSSM in order to maintain the integrity of the CDSA system. A full set of credentials includes:

- A set of object code files, which contain the executables for a module manager and the hashes of the object code files
- A *Manifest file*, which records a description of the module manager
- A signer's information file, which contains a reference to the manifest, a hash of the manifest, and the hash algorithm identifier

- A signature file, which contains a signature on the signer's information file and the complete set of X.509 certificates comprising the module manager's credentials

These three files must be zipped to form a single set of credentials. Multiple implementations of standard zip algorithms interoperate on one or more platforms, hence a zipped, signed manifest retains a substantial degree of interoperability.

The module manager's certificate is the leaf in a certificate chain. The chain is rooted at one of a small number of known, trusted, cross-certified certificates. A simple case is shown in Figure 30-1. A CSSM vendor issues a certificate to the elective module manager vendor, signed with the private key of the CSSM vendor's certificate. The elective module manager vendor issues a certificate for each of its products, signing the product certificate with its own certificate. The CSSM Embedded Integrity Services Library embeds a set of CSSM vendor public root keys. These key are recognized points of trust and are used when verifying a module manager's certificate. By incorporating multiple certificate chains in the signature file an elective module manager can be verified by multiple CSSM installations, no just those created by one specific root vendor.

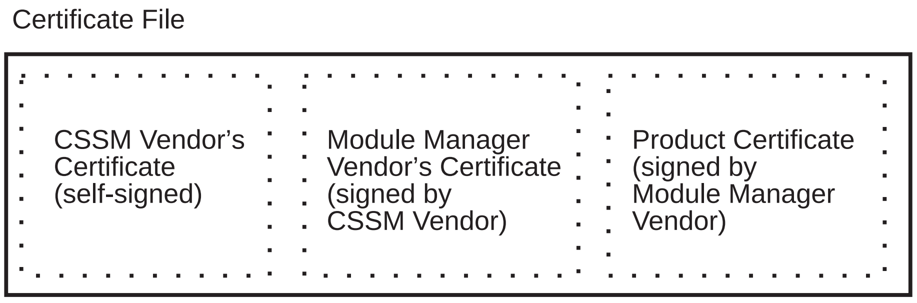
Certificate File
CSSM Vendor's Certificate (self-signed)
Module Manager Vendor's Certificate (signed by CSSM Vendor)
Product Certificate (signed by Module Manager Vendor)

Figure 30-1 Certificate Chain for an Elective Module Manager

The manifest associated with an elective module manager describes the module manager component. A manifest file includes:

- A reference to each object code file that is part of the module manager implementation
- A set of SHA-1 digital hashs, one per object code file
- The SHA-1 hash algorithm identifier
- Vendor-specified information about the elective module manager

The object code files are standard OS-managed entities. Object files do not embed their digital signatures, instead, signatures are stored in a manifest separate from, but related to, the object files.

A digest of each manifest section is then computed and stored in the signature info file.

The signature file contains the last PKCS#7 signature computed over all of the related manifest entries, including the signatures contained in the manifest.

This set of credentials must be manufactured when the module manager is manufactured. Assuming the elective module manager vendor already has a certificate from a CSSM manufacturer, the manufacturing process for an elective module manager proceeds as follows:

1. Generate an X.509 product certificate for the module manager and sign it with the manufacturer's certificate.

2. Create an optional description of the elective module manager for inclusion in the manifest.
3. Compute the SHA-1 hash for the implementation components (object code files) used in the module manager.
4. Build the signature info file containing the SHA-1 hash of each manifest section.
5. Compute a digital signature over the signature info file using the private key of the product's certificate.
6. Create the PKCS#7 signature file containing the signature info file digest, the signature over the signature info file, and all of the elective module manager certificates.

It is of the utmost importance that the object code files and the manifest be signed using the private key associated with the product certificate. This tightly binds the identity in the certificate with "what the elective module manager is" (that is, the object code files themselves) and the vendor identified in the certificate.

The structure and manipulation of manifests and certificate credentials is specified in the *CSSM Embedded Integrity Services Library API Spec* and the *CDSA Signed Manifest Specification*, which are included in the CDSA document set.

30.3 Installing an Elective Module Manager

Although the dynamic nature of an elective module manager is transparent to application, the elective module manager must be installed with CSSM before an application can use an add-in module of the service category defined and managed by the elective module manager. The name given to an elective module manager includes both a logical name and a globally-unique identifier (GUID). The logical name is a string chosen by the module manager developer to describe the module. The GUID is a structure used to differentiate among all components (for example, elective module managers and add-in modules) recorded in the CSSM registry. GUIDs are discussed in more detail below. The location of the module manager implementation is required at installation time so CSSM can locate the module manager and its credentials when the module manager must be loading into the system.

The module manager must also define and register a service mask with CSSM. This mask names the new category of security services defined by the elective module manager. CSSM defines a service mask for each of the four basic security service categories. The remainder of the name space for service masks is managed outside of CDSA. Module manager vendors are responsible for selecting a unique service mask for their new category of security service.

Add-in modules that implement a new category of security service and applications that use the new category of service use the service mask defined by the manager's vendor to identify the selected class of service. A single instance of an elective module manager can be registered with CSSM. Versions are not permitted. Applications and add-in modules have compile-time knowledge of the service mask value that identifies the module manager's category of service. CSSM core learns of new service masks during the installation of an elective module manager.

30.3.1 Global Unique Identifiers (GUIDs)

Each module manager must have a globally-unique identifier (GUID) that CSSM and the module manager itself uses to uniquely identify the manager for install and de-install operations. A module manager can also use its GUID to identify itself when it sets an error.

A GUID is defined as:

```
typedef struct cssm_guid {
    uint32 Data1;
    uint16 Data2;
    uint16 Data3;
    uint8 Data4[8];
} CSSM_GUID, *CSSM_GUID_PTR;
```

GUID generators are publicly available for Windows* 95, Windows NT*, and on many UNIX* platforms.

30.4 Loading an Elective Module Manager

Before an application can use the functions of a specific add-in module, it must use the *CSSM_ModuleAttach* function to request that CSSM attach to the module. If the add-in module implements an elective category of service and its module manager is not currently loaded, CSSM searches the CSSM registry for an appropriate module manager and loads it using the following process:

1. CSSM verifies the integrity of the elective module manager code prior to loading the object code module, performing the first half of the CSSM bilateral authentication protocol.
2. Once the module manager has been loaded, CSSM initiates a call to an OS-specific main entry point in the module manager. Within the main function, the elective module manager must perform integrity self-check using EISL. Upon return, CSSM invokes the ModuleManagerAuthenticate function. The elective module manager must implement this function. The function must verify CSSM and its credentials completing the second half of the CSSM bilateral authentication protocol.
3. Upon successful completion of the bilateral authentication protocol, the elective module manager registers a small table of function pointers for CSSM to module manager communications.
4. Using the function table provided by the elective module manager, CSSM invokes the module manager's *Initialize* function, allowing the module manager to complete additional initialization processing, if required.

The Embedded Integrity Services Library (EISL) must be used to perform bilateral authentication. The module manager is responsible for verifying the CSSM that is attempting to load the module manager. If verification fails, the module manager is responsible for terminating the attach process. When EISL returns a failure condition, then either the CSSM has been tampered or the attaching module manager does not recognize the CSSM's certificate. The module manager must terminate the attach process. The module manager should not register it interface function table with the suspect CSSM. The module manager should perform clean-up operations and exit voluntarily. The module manger has refused to provide service in an environment that it could not verify. If verification succeeds, then the module manager should proceed to register with CSSM.

30.4.1 Elective Module Manager Entry Point

When CSSM loads a module manager, it initiates an OS-specific entry point. For the Windows NT* operating system, *DLLMain* is the entry point. For SunOS, *_init* and *_fini* are the entry points. Upon load, this function will be responsible for performing self-check and returning to CSSM. The module manager function ModuleManagerAuthenticate is responsible for authenticating CSSM and then calling *CSSM_RegisterManagerServices* to register a function table with CSSM. Upon unload, the corresponding OS-specific entry point is invoked. This function is responsible for calling *CSSM_DeregisterManagerServices*. To avoid OS-related conflicts, any setup or cleanup operations should be performed in the module's *Initialize* and *Terminate* functions.

30.4.2 Bilateral Authentication

Upon load, CSSM and the elective module manager verify their own and each other's credentials by following CSSM's bilateral authentication protocol. The practice of self-checking and cross-checking by other parties increases the level of tamper detection provided by CDSA. The CSSM bilateral authentication protocol is supported by the services of the CSSM Embedded Integrity Services Library (EISL).

The basic steps in bilateral authentication during module attach are defined as follows:

1. CSSM performs a self integrity check.
2. CSSM performs an integrity check of the attaching elective module manager.
3. CSSM verifies secure linkage by checking that the initiation point is within the verified object code.
4. CSSM invokes the elective module manager.
5. The elective module manager performs a self integrity check.
6. The elective module manager performs an integrity check of CSSM.
7. The elective module manager verifies secure linkage by checking that the function call originated from the verified CSSM.

The EISL is an embedded subset of the Integrity Services Library (ISL). Each authenticating entity invokes EISL functions to carry out the steps in this process. The following EISL functions are used to carry out the seven step bilateral authentication protocol:

- *ISL_SelfCheck*( )
- *ISL_VerifyAndLoadModuleAndCredentials*( )
- *ISL_VerifyLoadedModuleAndCredentials*( )
- *ISL_RecycleVerifiedModuleCredentials*( )
- *ISL_LocateProcedureAddress*( )
- *ISL_GetReturnAddress*( )
- *ISL_CheckAddressWithinModule*( )

The EISL Verify functions check all aspects of a module manager's credentials, including the certificate chain, the signature on the manifest, the signature on the product description, and the signature on each object code file. The EISL Verify functions cannot check for secure linkage. CSSM and the elective module manager must use the EISL address checking functions to verify secure linkage with the party being verified. The purpose of the secure linkage check is to verify that the object code just verified is either the code you are about to invoke or the code that invoked you. To free the data structures used in bilateral authentication, EISL provides a Recycle function.

30.4.3 Module Manager Function Table Registration

Upon load, a module manager must register its function tables with CSSM by calling *CSSM_RegisterServices*. Its function table contains module management function pointers to *Initialize*, *Terminate*, *RegisterDispatchTable*, *DeregisterDispatchTable*, and *EventNotifyManager*.

CSSM invokes the initialize function providing its major and minor version numbers as input. The elective module manager must determine compatibility with the specified CSSM version and performs any additional, required initialization operations.

When an application attaches an add-in service module of the category managed by the elective module manager, CSSM invokes the *RegisterDispatchTable* function. The functions passes to the module manager:

- The module handle
- A memory-management function table supplied by the application
- A service function table supplied by the add-in module

The module handle uniquely identifies the session between the application and the add-in module instance.

All memory allocation and de-allocation for data passed between the application and any part of CSSM is ultimately the responsibility of the calling application. If the elective module manager provides direct services to an application in addition to those services provided by the add-in modules it manages, and the module manager needs to allocate memory to return data to the application, the application-provided memory management functions.

The functions are provided as a set of memory management upcalls. The functions are the application's equivalent of malloc, free, calloc, and re-alloc. The supplied functions are expected to have the same behavior as those functions. The function parameters will consist of the normal parameters for that function. The function return values should be interpreted in the standard manner. A module manager is responsible for making the memory management functions available to all of its internal functions that require it.

The service function table is the function table of an add-in service module. The table entries correspond to the Service Provider Interfaces (SPIs) defined by the elective module manager for the new category of security service. The elective module manager uses the function table to dispatch application calls for service to attached add-in modules. Multiple applications and multiple instances of an add-in module can be concurrently active. The single elective module manager is responsible for managing all of these concurrent sessions.

CSSM invokes the *DeregisterDispatchTable* to inform the elective module manager of the termination of a particular application and add-in module session. The module handle is used to identify the terminating session.

CSSM invokes the terminate function to inform the elective module manager that all of the application and add-in module sessions of the service types managed by that elective module manager have terminated and the CSSM is going to unload the elective module manager. The elective module manager must perform any cleanup tasks and make all preparations for unloading.

30.5 Error Handling

When an error occurs inside a module manager, the manager should call *CSSM_SetError*. The *CSSM_SetError* function takes the manager's GUID and an error number as inputs. The manager's GUID is used to identify where the error occurred. The error number will be used to describe the error.

The error number set by a module manager should fall into one of two ranges. The first range of error numbers is pre-defined by CSSM. These are errors that are common to all module managers in CSSM. The second range of error numbers is used to define error codes specific to the service category. These category-specific error codes must be in the range of CSSM_XX_PRIVATE_ERROR to CSSM_XX_END_ERROR, where XX stands for the service category abbreviation (such as, CSP, TP, CL, DL), defined by the elective module manager. Each service developer is responsible for making the definition and interpretation of their category-specific error codes available to applications.

When no error has occurred, but the appropriate return value from a function is CSSM_FALSE, that function should call *CSSM_ClearError* before returning. When the application receives a CSSM_FALSE return value, it is responsible for checking whether an error has occurred by calling *CSSM_GetError*. If the module function has called *CSSM_ClearError*, the calling application receives a CSSM_OK response from the *CSSM_GetError* function, indicating no error has occurred.

Chapter 31

Elective Module Manager Operations

31.1 Data Structures

31.1.1 CSSM_BOOL

This data type is used to indicate a true or false condition.

```
typedef enum cssm_bool {
        CSSM_TRUE = 1,
        CSSM_FALSE = 0
} CSSM_BOOL
```

Definition

CSSM_TRUE
Indicates a true result or a true value.

CSSM_FALSE
Indicates a false result or a false value.

31.1.2 CSSM_RETURN

This data type is used to indicate whether a function was successful.

```
typedef enum cssm_return {
        CSSM_OK = 0,
        CSSM_FAIL = -1
} CSSM_RETURN
```

Definition

CSSM_OK
Indicates operation was successful.

CSSM_FAIL
Indicates operation was unsuccessful.

31.1.3 CSSM_DATA

The CSSM_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory. This memory must be allocated and freed using the memory management routines provided by the calling application via CSSM.

```
typedef struct cssm_data{
    uint32 Length;  /* in bytes */
    uint8 *Data;
} CSSM_DATA, *CSSM_DATA_PTR
```

Definition

Length
Length of the data buffer in bytes.

Data
Points to the start of an arbitrary length data buffer.

31.1.4 CSSM_GUID

This structure designates a global unique identifier (GUID) that uniquely identifies each add-in module and elective module manager. All GUID values should be computer-generated to guarantee uniqueness (the GUID generator in Microsoft Developer Studio* and the RPC UUIDGEN/uuid_gen program on a number of UNIX* platforms can be used).

```
typedef struct cssm_guid{
    uint32 Data1;
    uint16 Data2;
    uint16 Data3;
    uint8  Data4[8];
} CSSM_GUID, *CSSM_GUID_PTR
```

Definition

Data1
Specifies the first eight hexadecimal digits of the GUID.

Data2
Specifies the first group of four hexadecimal digits of the GUID.

Data3
Specifies the second group of four hexadecimal digits of the GUID.

Data4
Specifies an array of eight elements that contains the third and final group of eight hexadecimal digits of the GUID in elements 0 and 1, and the final 12 hexadecimal digits of the GUID in elements 2 through 7.

31.1.5 CSSM_MODULE_HANDLE

A unique identifier for a session between an application and an attached, add-in service provider module.

```
typedef uint32 CSSM_MODULE_HANDLE
```

31.1.6 CSSM_SERVICE_MASK

This defines a bit mask of all the basic service types recognized by CSSM. The elective module manager must define a unique service mask for the new type of service it defines.

```
typedef uint32 CSSM_SERVICE_MASK;

#define CSSM_SERVICE_CSSM  0x1
#define CSSM_SERVICE_CSP   0x2
#define CSSM_SERVICE_DL    0x4
#define CSSM_SERVICE_CL    0x8
#define CSSM_SERVICE_TP    0x10
```

```
#define CSSM_SERVICE_LAST  CSSM_SERVICE_Tp
```

31.1.7 CSSM_EXEMPTION_MASK

The CSSM defines the exemption mask flags listed above. Other elective module managers define flags for their respective exemption categories. It is the responsibility of the elective module manager developer to ensure uniqueness of their defined flag values and to document those values for use by applications.

```
typedef uint32 CSSM_EXEMPTION_MASK

#define CSSM_EXEMPT_NONE 0x00000001
#define CSSM_EXEMPT_MULTI_ENCRYPT_CHECK 0x00000002
#define CSSM_EXEMPT_ALL 0xFFFFFFFF
```

31.1.8 CSSM_MODULE_MANAGER_INFO

This structure aggregates a description of the module manager.

```
typedef struct cssm_module_manager_info {
    CSSM_VERSION Version; /* Module Manager version */
    CSSM_VERSION CompatibleCSSMVersion; /* CSSM version the
                                              manager works with */
    CSSM_GUID_PTR CompatibleInterfaceGUID, /* opt GUID for
                                            compatible CSSM interface */
    CSSM_STRING Description; /* Module Manager description */
    CSSM_STRING Vendor; /* Vendor name */
    CSSM_SERVICE_MASK ServiceType; /* Bit mask of supported services */
    CSSM_EXEMPTION_FLAGS ExemptionFlags; /* Flags for exemption
                                                          categories */
    CSSM_KEY_PTR AppAuthenRootKeys, /* Mgr-specific keys to
                                                   authenticate apps */
    uint32 NumberOfAppAuthenRootKeys,/* Number of Manager-specific
                                                           root keys */
    void *Reserved;
} CSSM_MODULE_MANAGER_INFO, *CSSM_MODULE_MANAGER_INFO_PTR;
```

Definition

Version
The major and minor version numbers of this module manager.

CompatibleCSSMVersion
The version of CSSM that this module manager was written to.

CompatibleInterfaceGUID
An optional GUID describing the CSSM interface this module was written to.

Description
A text description of this module manager and its functionality.

Vendor
The name and description of the module manager vendor.

ServiceType
A bit mask identifying the types of services available through this module manager.

ExemptionFlags
A bit mask identifying the exemption categories offered by this module manager.

AppAuthenRootKeys
Public root keys defined by the module manager as additional roots for authenticating an application's request for exemptions.

NumberOfAppAuthenRootKeys
The number of public root keys in the AppAuthenRootKeys list.

Reserved
This field is reserved for future use. It should always be set to NULL.

31.1.9 CSSM_MEMORY_FUNCS

This structure is used by applications to supply memory functions for the CSSM and the add-in modules. The functions are used when memory needs to be allocated by the CSSM or add-ins for returning data structures to the applications.

```
typedef struct cssm_memory_funcs {
  void *(*malloc_func) (uint32 Size, void *AllocRef);
  void  (*free_func)   (void *MemPtr, void *AllocRef);
  void *(*realloc_func)(void *MemPtr, uint32 Size, void *AllocRef);
  void *(*calloc_func) (uint32 Num, uint32 Size, void *AllocRef);
  void *AllocRef;
} CSSM_MEMORY_FUNCS, *CSSM_MEMORY_FUNCS_PTR;
```

Definition

malloc_func
Pointer to function that returns a void pointer to the allocated memory block of at least size bytes from heap AllocRef.

free_func
Pointer to function that deallocates a previously-allocated memory block (memblock) from heap AllocRef.

realloc_func
Pointer to function that returns a void pointer to the reallocated memory block (memblock) of at least size bytes from heap AllocRef.

calloc_func
Pointer to function that returns a void pointer to an array of num elements of length size initialized to zero from heap AllocRef.

AllocRef
Indicates which memory heap the function operates on.

31.1.10 CSSM_MODULE_FUNCS

This structure is used by add-in modules to pass a table of function pointers for a single service to CSSM. CSSM core service forwards this table to the module manager responsible for dispatching function calls to the add-in module using this function table.

```
typedef struct cssm_module_funcs {
    uint32 SubServiceID;
    uint32 ServiceType;
    void *ModuleServices;
} CSSM_MODULE_FUNCS, *CSSM_MODULE_FUNCS_PTR;
```

Definition

SubServiceId
A module-specific identifier.

ServiceType
The type of add-in module services accessible via the ModuleServices function table.

31.1.11 CSSM_MANAGER_EVENT_TYPES

This enumerated list defines a standard set of event types used by module managers when informing other module managers of this event.

```
typedef enum cssm_manager_event_types {
        CSSM_MANAGER_LOADED = 1,          /* source manager
                                        just loaded */
        CSSM_MANAGER_UNLOADED = 2,        /* source manager about
                                        to be unloaded */
        CSSM_MANAGER_SERVICE_REQUEST = 3, /* source mgr asking
                                        standard service */
        CSSM_MANAGER_REPLY = 4,           /* event is a reply to
                                        earlier event notice */
} CSSM_MANAGER_EVENT_TYPES;
```

31.1.12 CSSM_MANGER_EVENT_NOTIFICATION

This structure contains all the information about a notification event between two module managers.

```
typedef struct cssm_manager_event_notification {
    CSSM_SERVICE_MASK DestinationModuleManagerUsage;
    CSSM_SERVICE_MASK SourceModuleManagerUsage;
    CSSM_MANAGER_EVENT_TYPES Event;
    uint32 EventId;
    CSSM_DATA_PTR EventData;
    } CSSM_MANAGER_EVENT_NOTIFICATION,
*CSSM_MANAGER_EVENT_NOTIFICATION_PTR;
```

Definition

DestinationModuleManagerType
A service mask identifying the module manager to receive the event notification.

SourceModuleManagerType
A service mask identifying the module manager that initiated the event notification.

Event
An identifier specifying the type of event that has taken place or will take place.

EventId
A unique identifier associated with this event notification. It must be used in any reply notification that result from this event notification.

EventData
Arbitrary data (required or information) for this event.

31.1.13 CSSM_MANAGER_REGISTRATION_INFO

This structure defines the function prototypes that an elective module manager must implement to be dynamically loaded by CSSM.

```
typedef struct cssm_manager_registration_info {
/* loading, unloading, dispatch table, and event notification */
    CSSM_RETURN (CSSMAPI *Initialize) (uint32 VerMajor,
                                       uint32 VerMinor);
    CSSM_RETURN (CSSMAPI *Terminate) (void);
    CSSM_RETURN (CSSMAPI *RegisterDispatchTable)
        (CSSM_MODULE_HANDLE Modulehandle,
         CSSM_MEMORY_FUNCS_PTR AppMemoryCallTable,
         CSSM_MODULE_FUNCS_PTR AddInCallTable);
    CSSM_RETURN (CSSMAPI *DeregisterDispatchTable)
        (CSSM_MODULE_HANDLE Modulehandle);
    CSSM_RETURN (CSSMAPI *EventNotifyManager)
        (CSSM_SERVICE_MASK DestinationModuleManagerType,
         CSSM_SERVICE_MASK SourceModuleManagerType,
         CSSM_MANAGER_EVENT_TYPES Event,
         uint32 EventId,
         CSSM_DATA_PTR EventData);
} CSSM_MANAGER_REGISTRATION_INFO,  *CSSM_MANAGER_REGISTRATION_INFO_PTR;
```

Definition

Initialize
Function invoked by CSSM to initialize an elective module manager.

Terminate
Function invoked by CSSM before unloading an elective module manager.

RegisterDispatchTable
Function invoked by CSSM to pass a service provider function table to an elective module manager.

DeregisterDispatchTable
Function invoked by CSSM to inform an elective module manager that an application and add-in module session is no longer active and the service provider function table for that

add-in module is no longer for the terminating session.

EventNotifyManager
Function invoked by CSSM forwarding an event notification from one module manager to another target module manager.

31.2 Elective Module Manager Functions

The manpages for Elective Module Manager Functions follow on the next page.

NAME

Initialize

SYNOPSIS

```
CSSM_RETURN CSSMAPI Initialize
    uint32 VerMajor,
    uint32 VerMinor)
```

DESCRIPTION

This function checks whether the current version of the module is compatible with the CSSM version specified as input and performs any module-manager-specific setup activities.

PARAMETERS

VerMajor (input)
The major version number of the CSSM that is invoking this module manager.

VerMinor (input)
The minor version number of the CSSM that is invoking this module manager.

RETURN VALUE

A CSSM_OK return value signifies that the current version of the module is compatible with the input CSSM version numbers and all setup operations were successfully performed. When CSSM_FAIL is returned, either the current module manager is incompatible with the requested module version or an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_MANAGER_INITIALIZE_FAIL
Unable to initialize the module manager.

SEE ALSO

Terminate

NAME

Terminate

SYNOPSIS

```
CSSM_RETURN CSSMAPI Terminate
    (void)
```

DESCRIPTION

This function performs any module-manager-specific cleanup activities in preparation for unloading of the elective module manager.

PARAMETERS

None.

RETURN VALUE

A CSSM_OK return value signifies that all cleanup operations were successfully performed. When CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

SEE ALSO

Initialize

NAME

ModuleManagerAuthenticate

SYNOPSIS

```
CSSM_RETURN CSSMAPI  ModuleManagerAuthenticate
    (const char *CssmCredentialPath,
    const char *CssmSection,
    const char *AppFileName,
    const char *AppPathName)
```

DESCRIPTION

This function should perform the elective module manager's half of the bilateral authentication procedure with CSSM. The CSSM credential path and section information is used to locate the CSSM's credentials to be verified. The credentials are a zipped, signed manifest.

If the application filename and pathname are provided, the elective module manager has the option to perform an integrity and identity check of the attaching application. The filename and pathname can be used to locate the application's signed credentials.

This function is the first module manager interface invoked by CSSM after loading and invoking the main entry point. In particular, the elective module manager's initialize function is invoked by CSSM after this function has successfully completed execution.

PARAMETERS

CssmCredentialPath (input)
A string containing the path name for locating the calling CSSM's credentials. These credentials are a zipped, signed manifest. The service module should verify these credentials as part of the bilateral authentication process.

CssmSection (input)
A string containing the section name for the manifest section containing a description and cryptographic digest of the calling CSSM's object code.

AppFileName (input/optional)
The name of the file that implements the application (containing its main entry point). This file name can be used to locate the application's credentials for purposes of application authentication by the elective module manager. The application provides this input to CSSM if the application has credentials it wishes to present for verification to CSSM or to other components in the system.

AppPathName (input/optional)
The pathname to the file that implements the application (containing its main entry point). This pathname can be used to locate the application's credentials for purposes of application authentication by the elective module manager. The application provides this input to CSSM if the application has credentials it wishes to present for verification to CSSM or to other components in the system.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

NAME

RegisterDispatchTable

SYNOPSIS

```
CSSM_RETURN CSSMAPI RegisterDispatchTable
    (CSSM_MODULE_HANDLE ModuleHandle,
    CSSM_MEMORY_FUNCS_PTR AppMemoryCallTable,
    CSSM_MODULE_FUNCS_PTR AddInCallTable)
```

DESCRIPTION

This function receives information about and application and add-in module session. The information is provided by the CSSM core services. The module handle is created by the CSSM core services. It uniquely identifies the application and add-in module session. The memory functions are defined by the application. The elective module manager must use these if it allocates or frees any memory resources on behalf of the application. The function table is defined by the add-in service module. The elective module manager uses this table to dispatch application function calls from the application (invoking the APIs) to the add-in service module (via the SPIs).

PARAMETERS

ModuleHandle (input)
The unique module handle of the session between the application and the add-in service module.

AppMemoryCallTable (input)
The function table for operations on the application's memory space.

AddInCallTable (input)
The function table for operations serviced by the add-in service provider module.

RETURN VALUE

A CSSM_OK return value signifies that the information about the application and add-in module session has been received and the module manager will maintain the session by dispatching application requests to the add-in module. When CSSM_FAIL is returned, the module manager was unable to assume responsibility for managing the service session. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_SERVICE_REGISTRY_FAIL
Unable to accept the function tables.

SEE ALSO

DeregisterDispatchTable

NAME

DeregisterDispatchTable

SYNOPSIS

```
CSSM_RETURN CSSMAPI DeregisterDispatchTable
    (CSSM_MODULE_HANDLE ModuleHandle)
```

DESCRIPTION

This function informs the elective module manager that the session identified by the module handle is shutting down. The notification is provided by the CSSM core services. The module handle uniquely identifies an application and add-in module session that the module manager has been servicing.

PARAMETERS

ModuleHandle (input)
The unique module handle of the session between the application and the add-in service module.

RETURN VALUE

A CSSM_OK return value signifies the module manager has successfully acted on the notification of session shutdown between an application and a add-in service module. When CSSM_FAIL is returned, the module manager was unable to perform the operations required for shutdown of individual sessions. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_SERVICE_DEREGISTRY_FAIL
Unable to clean-up for session shutdown.

SEE ALSO

RegisterDispatchTable

NAME

EventNotifyManager

SYNOPSIS

```
CSSM_RETURN CSSMAPI EventNotifyManager
   (CSSM_MANAGER_EVENT_NOTIFICATION_PTR EventDescription)
```

DESCRIPTION

This function receives an event notification from another module manager. The source manger is identified by its service mask. The specified event type is interpreted by the received and the appropriate actions must be taken in response. EventId and EventData are optional. The EventId is specified by the source module manager when a reply is expected. The destination module manager must used this identifier when replying to the event notification. The EventData is additional data or descriptive information provided to the destination manager.

PARAMETERS

EventDescription—A structure containing the following fields:

DestinationModuleManagerType (input/optional)
The unique service mask identifying the destination module manager.

SourceModuleManagerType (input)
The unique service mask identifying the source module manager.

Event (input)
An identified indicating the event the has or will take place.

EventId (input/optional)
A unique identified associated with this event notification. It must be used in any reply notification that result from this event notification.

EventData (input/optional)
Arbitrary data (required or informational) for this event.

RETURN VALUE

A CSSM_OK return value signifies that the event notification was received, understood, and acted upon. When CSSM_FAIL is returned, the module manager was unable to act appropriately in response to the event notification. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_MANGER_EVENT_FAIL
Unable to process the event.

SEE ALSO

CSSM_DeliverModuleManagerEvent

Chapter 32

Managing Elective Module Managers

32.1 Installation Functions

The manpages for Installation Functions follow on the next page.

NAME

CSSM_ModuleManagerInstall

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleManagerInstall
    (const char *ModuleManagerName,
    const char *ModuleManagerFileName,
    const char *ModuleManagerPathName,
    const CSSM_SERVICE_MASK ModuleManagerType,
    const CSSM_MODULE_MANAGER_INFO ModuleManagerDescription,
    const void *Reserved1,
    const CSSM_DATA_PTR Reserved2)
```

DESCRIPTION

This function registers the elective module manager with CSSM. CSSM adds the manager's descriptive information to its persistent registry. This makes the add-in service modules managed by this manager available for use on the local system. The function accepts as input the name and unique identifier for the module manager, the location executable code for the manager, and the service category supported by the manager.

PARAMETERS

ModuleManagerName (input)
The name of the module manager.

ModuleManagerFileName (input)
The name of the file that implements the module manager.

ModuleManagerPathName (input)
The path to the file that implements the module manager.

ModuleManagerGuid (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the module manager.

ModuleManagerType (input)
A CSSM_SERVICE_MASK defining the security service category supported by the module manager. This mask must be unique.

ModuleManagerDescription (input)
A pointer to the CSSM_MODULE_MANAGER_INFO structure containing a description of the module manager.

Reserved1 (input)
Reserve data for the function.

Reserved2 (input)
Reserve data for the function.

RETURN VALUE

A CSSM_OK return value signifies that the information has been registered with CSM. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_REGISTRY_ERROR
Error in the registry.

SEE ALSO

CSSM_ModuleManagerUninstall

NAME

CSSM_ModuleManagerUninstall

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleManagerUninstall
    (const CSSM_SERVICE_MASK ModuleManagerType)
```

DESCRIPTION

This function deletes the persistent CSSM internal information about the module manager, removing it from the name space of available elective module managers in the CSSM system. The service mask uniquely identifies the module manager. Exactly one manger for the given service type can be installed at any given time. Before installing a new version, the old version must be uninstalled.

PARAMETERS

ModuleManagerType (input)
A CSSM_SERVICE_MASK identifying the elective module manager to be removed from the CSSM registry.

RETURN VALUE

A CSSM_OK return value means the elective module manager has been successfully uninstalled. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_SERVICE_MASK
Unknown service type.

CSSM_REGISTRY_ERROR
Unable to delete information.

SEE ALSO

CSSM_ModuleManagerInstall

32.2 Information Functions

The manpages for Information Functions follow on the next page.

NAME

CSSM_GetModuleManagerInfo

SYNOPSIS

```
CSSM_MODULEMANAGER_INFO_PTR CSSMAPI CSSM_GetModuleManagerInfo
    (const CSSM_GUID_PTR GUID,
    CSSM_SERVICE_MASK ServiceType);
```

DESCRIPTION

This function returns descriptive information about the elective module manager identified by the GUID or the service mask. Each is a unique identifier of an elective module manager. Either identifier is a sufficient specification. The returned information structure contains the basic descriptive information registered with CSSM during the module manager installation process.

PARAMETERS

GUID (input/optional)
A pointer to the CSSM_GUID structure containing the global unique identifier for the elective module manager.

ServiceType (input/optional)
A bit mask specifying the service category supported by the elective module manager.

RETURN VALUE

A CSSM_MODULEMANAGER_INFO_PTR to an info structure.

ERRORS

CSSM_INVALID_SERVICE_MASK
Invalid bit mask.

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_INVALID_GUID
Unknown GUID.

NAME

CSSM_ListAttachedModuleManagers

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ListAttachedModuleManagers
    (CSSM_GUID_PTR ModuleManagerGuids,
    uint32 *NumberOfModuleManagers);
```

DESCRIPTION

This function returns a list of GUIDs for the currently attached/active module managers in the CSSM environment.

PARAMETERS

ModuleManagerGuids (output)
A pointer to an array of CSSM_GUID structures, one per active module manager.

NumberOfModuleManagers (output)
The number of GUIDs in the array.

RETURN VALUE

A CSSM_OK return value means a GUID list has been returned If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_INVALID_GUID
Unknown GUID.

32.3 Registration Functions

The manpages for Registration Functions follow on the next page.

NAME

CSSM_RegisterManagerServices

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_RegisterManagerServices
    (const CSSM_GUID_PTR Guid,
    const CSSM_MANAGER_REGISTRATION_INFO_PTR FunctionTable,
    void *Reserved);
```

DESCRIPTION

This function is used by an elective module manager to register its management functions with CSSM. CSSM cores services invokes these functions when loading and unloading the module manager, creating and ending service sessions between applications and add-in modules managed by the module manager, and forwarding even notifications from one module manager to another module manager.

PARAMETERS

Guid (input)
The GUID of the module manager that is registering it function table with CSSM core services.

FunctionTable (input)
The function table for the module manager's functions that interact with CSSM.

Reserved (input/optional)
Reserved for future use.

RETURN VALUE

A CSSM_OK return value means CSSM has received and recorded the module manager's function table. If CSSM_FAIL is returned, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_GUID
Unknown GUID.

CSSM_FUNCTION_TABLE
Bad or incomplete function table.

CSSM_MEMORY_ERROR
Internal memory error.

SEE ALSO

CSSM_DeregisterManagerServices

NAME

CSSM_DeregisterManagerServices

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DeregisterManagerServices
    (const CSSM_GUID_PTR GUID)
```

DESCRIPTION

This function is used by an elective module manager to de-register its function table with CSSM core services prior to termination.

PARAMETERS

GUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for this module.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_GUID
Invalid GUID.

CSSM_DEREGISTER_SERVICES_FAIL
Unable to deregister services.

SEE ALSO

CSSM_RegisterManagerServices

32.4 Notification Functions

The manpages for Notification Functions follow on the next page.

NAME

CSSM_DeliverModuleManagerEvent

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DeliverModuleManagerEvent
    (const CSSM_MANAGER_EVENT_NOTIFICATION_PTR EventDescription)
```

DESCRIPTION

This function requests that CSSM core services deliver an event notification to a module manager. registers the module with CSSM. The event description parameter identified the source and destination module managers, the event the will take place or has taken place, optional additional data about the event, and an optional identified used when a reply is required.

A module manager calls this CSSM core services function to send an event to another module manager. CSSM core services forwards this event to the destination module manager by invoking the module manager's *EventNotifyManager* function, which must be defined by every module manager that can receive event notification through the CSSM mechanism.

PARAMETERS

EventDescription (input)
A structure containing the following fields:

DestinationModuleManagerType (input/optional)
The unique service mask identifying the destination module manager.

SourceModuleManagerType (input)
The unique service mask identifying the source module manager.

Event (input)
An identified indicating the event the has or will take place.

EventId (input/optional)
A unique identified associated with this event notification. It must be used in any reply notification that result from this event notification.

EventData (input/optional)
Arbitrary data (required or informational) for this event.

RETURN VALUE

A CSSM_OK return value signifies that the event notice has been delivered to the specified destination module manager. If CSSM_FAIL is returned, either the destination module manager is unknown, the destination module manager has not registered an event notification entry point with CSSM so the notification cannot be delivered, or an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_INVALID_SERVICE_MASK
Unknown service category.

CSSM_NOTIFICATION_ERROR
Unable to deliver the notification.

CSSM_REGISTRY_ERROR
Error in the registry.

SEE ALSO

EventNotifyManager

CAE Specification

Part 7:

CSSM Add-In Module Structure and Administration

The Open Group

Chapter 33

Introduction

33.1 Common Data Security Architecture

The Common Data Security Architecture (CDSA) defines the infrastructure for a comprehensive set of security services to address the needs of individual users and the business enterprise. CDSA is an extensible architecture that provides mechanisms to manage add-in security service modules. These modules provide cryptographic services and certificate services for use in building secure applications. Figure 33-1 shows the four basic layers of the Common Data Security Architecture: Applications, System Security Services, the Common Security Services Manager, and Security Add-in Modules. The Common Security Services Manager (CSSM) is the core of CDSA. It provides a means for applications to directly access security services through the CSSM security API, or to indirectly access security services via layered security services and tools implemented over the CSSM API. CSSM manages the add-in security modules and re-directs application calls through the CSSM API to the selected add-in modules that will service the request.

This four layer architecture defines four categories of basic add-in module security services. Basic services are required to meet the security needs of all applications. CSSM also supports the dynamic inclusion of APIs for new categories of security services, as required by selected, security-aware applications. These elective services are dynamically and transparently added to a running CSSM environment when required by an application. When an elective service is needed, CSSM attaches a module manager for that category of service and then attaches the requested add-in service module. Once attached to the system, the elective module manager is a peer with all other CSSM module managers. Applications interact uniformly with add-in modules of all types.

The four basic categories of security services modules are:

- Cryptographic Service Providers (CSP)
- Trust Policy Modules (TPM)
- Certificate Library Modules (CLM)
- Data Storage Library Modules (DLM)

Cryptographic Service Providers (CSPs) are add-in modules, that perform cryptographic operations including encryption, decryption, digital signaturing, key pair generation, random number generation, and key exchange. Trust Policy (TP) modules implement policies defined by authorities, institutions, and applications, such as your Corporate Information Technology Group* (as a certificate authority) or MasterCard* (as an institution), or Secure Electronic Transfer (SET) applications. Each trust policy module embodies the semantics of a trust environment based on digital credentials. A certificate is a form of digital credential. Applications may use a digital certificate as an identity credential and/or an authorization credential. Certificate Library (CL) modules provide format-specific, syntactic manipulation of memory-resident digital certificates and certificate revocation lists. Data Storage Library (DL) modules provide persistent storage for certificates, certificate revocation lists, and other security-related objects.

Examples of elective security service categories are key recovery and audit logging.

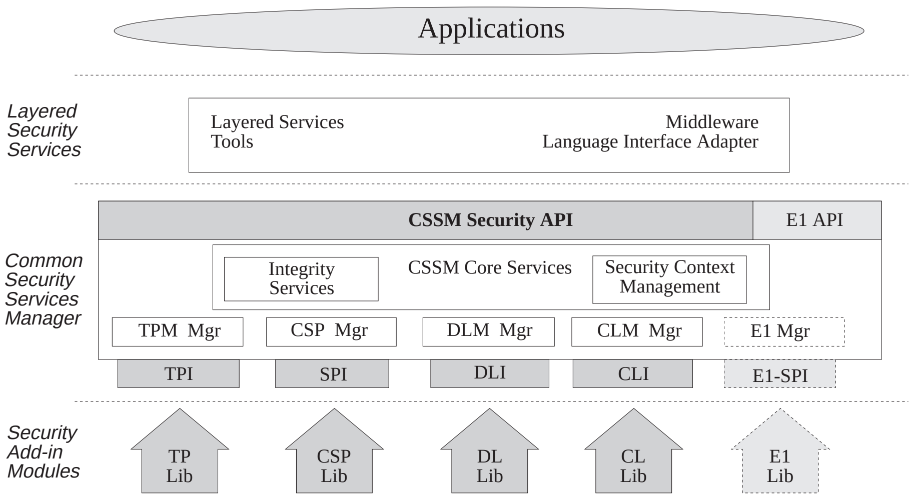

Figure 33-1 Common Data Security Architecture for all Platforms

Applications dynamically select the modules used to provide security services. These add-in modules can be provided by independent software and hardware vendors. A single add-in module can provide one or more categories of service. Modules implementing more than one category of service are called multi-service modules.

The majority of the CSSM API functions support service operations. Service operations are functions that perform a security operation, such as encrypting data, adding a certificate to a certificate revocation list, or verifying that a certificate is trusted/authorized to perform some action.

Modules can also provide services beyond those defined by the CSSM API. Module-specific operations are enabled in the API through pass-through functions whose behavior and use is defined by the add-in module developer. (For example, a CSP implementing signaturing with a fragmented private key can make this service available as a pass-through.) Existence as a pass-through function is viewed as a proving ground for potential additions to the CSSM APIs.

CSSM core services support:

- Module management
- Security context management
- System integrity services

The module management functions are used by applications and by add-in modules to support module installation, registration of module features and attributes, and queries to retrieve information on module availability and features.

Security context management provides runtime caching of user-specific, cryptographic state information for use by multi-step cryptographic operations, such as staged hashing. These operations require multiple calls to a CSP and produce intermediate state that must be managed. CSSM manages this state information for the CSP, enabling more CSPs to easily support multiple concurrent callers.

The CSSM Embedded Integrity Services Library (EISL) provides tamper resistant verification services. CSSM, add-in modules, and optionally applications use EISL to verify the identity and integrity of components of CDSA. Checkable components include: add-in service modules, CSSM itself, and in the future, applications that use CSSM. The EISL services focus on detecting impostors or unauthorized components and tampering of authorized components.

In summary, the direct services provided by CSSM through its API calls include:

- Comprehensive, extensible SPIs for each of four categories of security services
- Registration and management of all add-in security service modules available to applications
- Registration and management of elective module managers providing other security services
- Caching of runtime state for cryptographic operations
- Call-back functions used by add-in modules and CSSM to interact with an application process
- Notification services to inform add-in modules of selected actions taken by an application
- An Integrity Services Library providing tamper resistant test-and-check services for CDSA components
- Management support for concurrent security operations

33.2 Add-In Module Structure

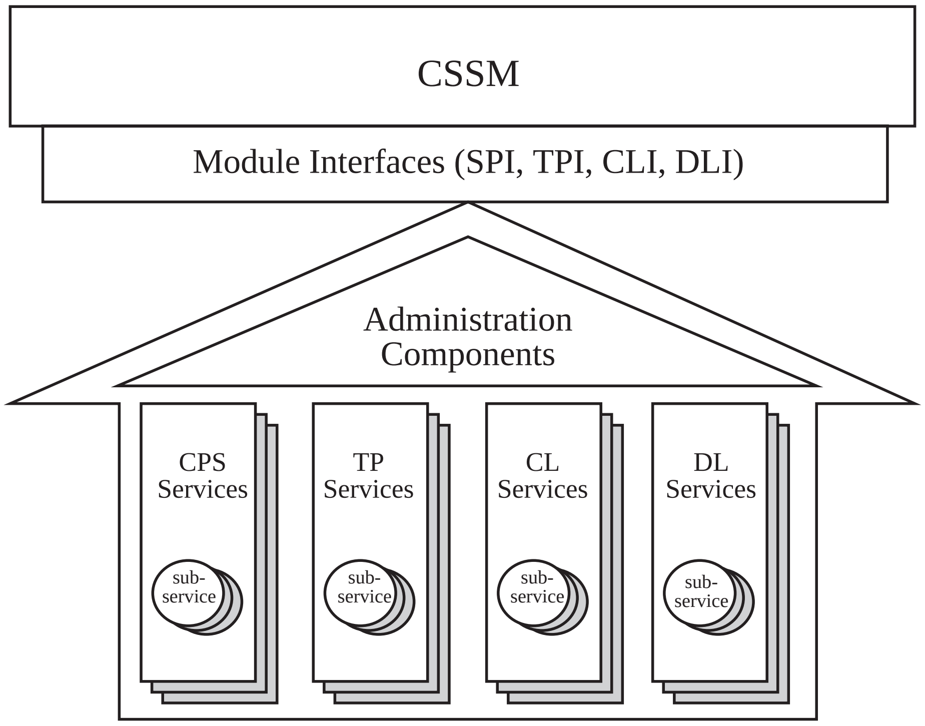

Figure 33-2 CDSA Add-In Module Structure

Add-in modules are composed of module administration components and implementations of security service interfaces in one or more categories of service. Module administration components include the tasks required during module installation, attach, and detach. The number, categories, and contents of the service implementations are determined by the module developer.

Every module implementation shares certain administrative tasks which must be performed during module installation, attach, and detach. As part of module installation, the module developer must register information about the module's services with CSSM. This information is stored in the CSSM registry and may be queried by applications using the CSSM_GetModuleInfo function. On attach, the module's administrative responsibilities include bilateral authentication, module registration, and module initialization. Bilateral authentication is a protocol whereby a module insures the integrity of its own components and of CSSM prior to attaching into the system. Following bilateral authentication, the module registers its functions with CSSM and performs any initialization operations. When the module is detached, it performs any necessary cleanup actions.

The remainder of the module implements one or more sub-services in one or more categories of service. A module developer may choose to implement a single service, such as a CSP, or may provide multiple services, such as Trust Policy and Certificate Library services. Within a single category of service, a module may implement multiple sets of capabilities, called sub-services. For example, a module might implement two Trust Policy sub-services in a single module in order to provide for two levels of authorization.

Additional utility libraries may be provided by a module developer for use by other module developers. Utility libraries are software components which contain functions that may be useful to several modules. For example, a utility library which performs DER encoding might be useful to several modules providing certificate library services. The utility library developer is responsible for making the definition, interpretation, and usage of their library available to other module developers.

33.3 Add-In Module Usage

33.3.1 Application Interaction

When a new module is installed on a system, information specific to the module and its services is stored in the CSSM registry. An application uses this information to find an appropriate module sub-service and to request that CSSM attach to it. When CSSM attaches to a module sub-service, it returns a module handle to the application that uniquely identifies the pairing of the application thread to the module sub-service instance. The application uses this handle to identify the module sub-service in future function calls. The module sub-service uses the handle to identify the calling application.

The calling application is responsible for the allocation and de-allocation of all memory that is passed into or out of the module. The application must register memory allocation and de-allocation upcalls with CSSM when it requests a module attach. These upcalls are passed to the module when the it calls the CSSM_RegisterServices function. These functions must be used whenever a module either allocates memory to be passed out of the module or de-allocates memory passed into the module.

33.3.2 CSSM Interaction

As a part of CSSM_ModuleAttach, CSSM and the add-in module perform a bilateral authentication protocol. In this protocol, CSSM insures that the module has not been altered since production by a trusted manufacturer. CSSM also verifies that the module is loaded into the appropriate memory space. The add-in module insures that the CSSM instantiation to which it is attaching is trusted, has not been altered, and is running in its appropriate memory space. The add-in service module implements this check in the AddInAuthenticate function, which is invoked by CSSM. The verification must succeed in order for a module to attach to CSSM.

Once bilateral authentication has been accomplished, the module uses CSSM_RegisterServices to register a function table with CSSM for each sub-service that it supports. The function tables consist of pointers to the sub-service functions supported by the module. During future function calls from the application, CSSM will use these function pointers to direct calls to the appropriate module sub-service.

33.3.3 Module to Module Interaction

Modules may make use of other CSSM add-in modules to implement their functionality. For example, a module implementing a certificate library may use the capabilities of a CSP add-in module to perform the cryptographic operations of sign and verify. In that case, the certificate library module could package the certificate or CRL fields to be signed or verified, attach to the appropriate CSP add-in module, and call CSSM_SignData or CSSM_VerifyData to perform the operation.

Similarly, that same module with certificate library capabilities may be used by other CSSM add-in modules to implement their functionality. For example, Trust Policy modules may choose to perform the syntactic verification of trust by calling a module with certificate library functionality.

Chapter 34

Add-In Module Structure

An add-in module is a dynamically-linkable library, which is composed of the following components:

- Security Services
- Module Administration Components

34.1 Security Services

The primary components of an add-in module are the security services that it offers. An add-in module may provide one to four categories of service, with each service having one or more available sub-services. The service categories are CSP services, TP services, CL services, and DL services. A sub-service consists of a unique set of capabilities within a certain service. For example, in a CSP service providing access to hardware tokens, each sub-service would represent a slot. A TP service may have one sub-service which supports the Secure Electronic Transfer (SET)* Merchant trust policy and a second sub-service which supports the Secure Electronic Transfer (SET)* Cardholder trust policy. A CL service may have different sub-services for different encoding formats. A DL service could use sub-services to represent different types of persistent storage. In all cases, the sub-service implements the basic service functions for its category of service.

Each service category contains from ten to sixty basic service functions. A library developer may choose to implement some or all of the functions specified in the service interface. A module developer may also choose to extend the basic interface functionality by exposing pass-through operations. Details of the Service Provider Interface Functions and their expected behavior can be found in the following specifications:

- *CSSM Cryptographic Service Provider Interface Specification*
- *CSSM Trust Policy Interface Specification*
- *CSSM Certificate Library Interface Specification*
- *CSSM Data Storage Library Interface Specification*
- *CSSM Key Recovery Interface Specification*

It may be necessary for sub-services to collaborate in order to perform certain operations. For example, a PKCS #11 module may require collaborating CSP and DL sub- services. Collaborating sub-services are assumed to share state. A module indicates that two or more sub-services collaborate by assigning them the same sub-service ID. When an application attaches one of the collaborating sub-services, it will receive a handle which may be used to access any of the sub-services having the same sub-service ID. This mechanism may be used for collaboration across categories of services, but is not available within a single category of service.

Sub-services may make use of other products or services as part of their implementation. For example, an ODBC DL sub-service may make use of a commercial database product, such as Microsoft Access*. A CL sub-service may make use of a CA service, such as the VeriSign DigitalID Center*, for filling certification requests. The encapsulation of these products and services is exposed to applications in the CSSM_XX_WRAPPEDPRODUCT_INFO data structure, available by querying the CSSM registry.

34.2 Module Administration Components

34.2.1 Integrity Verification

CDSA defines a dynamic environment where services are loaded on-demand. To ensure integrity under these conditions, CSSM defines and enforces a global integrity policy that aids in the detection of and protection against classic forms of attack, such as stealth and man-in-the-middle attacks. CSSM's global policy requires authentication checks and integrity checks at module attach time.

The policy requires successful certificate-based trust verification for:

- All add-in service modules
- All elective module managers

CSSM performs these checks during module attach. All verifications are based on CSSM-selected public root keys as points of trust.

When CSSM performs a verification check on any component in the CSSM environment, the verification process has three aspects:

- Verification of identity using a certificate chain naming the component's creator or manufacturer
- Verification of object code integrity based on a signed hash of the object code
- Tightly binding the verified identity with the verified object code

These steps are implemented by CSSM's Integrity Services. Integrity Services are packaged as a static library called the Embedded Integrity Services Library (EISL). EISL is available for use by the add-in module. EISL services support unilateral authentication, identity verification, object code integrity checks, and self-integrity-checks. EISL facilities are documented in the *CSSM Embedded Integrity Services Library API Spec.*

CDSA defines a layered bilateral authentication procedure by which CSSM and an add-in module can authenticate each other to achieve a mutual trust. EISL functions are used by the two parties to carry out the bilateral authentication procedure. An add-in module is strongly encouraged to verify its own components using the EISL self check function. This in-memory verification prevents stealth attacks where the file is unaltered, but the loaded copy is tampered. CSSM always verifies the add-in service module during attach processing. Add-in modules are also strongly encouraged to complete bilateral authentication with CSSM during module attach by verifying CSSM's credentials and object code module, and verifying secure linkage with the loaded, executing CSSM. CSSM initiates this last portion of the bilateral authentication process by invoking the AddInAuthenticate function.

Details of the attach process, including the bilateral authentication protocol, is presented later in this section.

34.2.2 Module-Defined Usage Policies

Service module vendors may wish to provide enhanced services to selected applications or classes of applications. A module-defined policy is in addition to the CSSM's general integrity policy.

Module-defined policies are enforced by one of the following authentication checks:

- CSSM authenticates the application that is requesting the module attach, based on CSSM trust points
- CSSM authenticates the application that is requesting the module attach, based on module-specified trust points
- The add-in module authenticates the attached application, based on module-specified trust points

The module specifies its policy by selecting one of these authentication checks. Options one and two use CSSM to enforce the module-defined policy during attach processing. Option three is carried out independently by the add-in module, using EISL services. The add-in module requests CSSM enforcement by setting MODULE_FLAGS corresponding to options one and two in the MODULE_INFO structure. When option two is selected, the MODULE_INFO structure should also contain a set of module-specific, public root keys corresponding to the module's points of trust.

The MODULE_INFO structure is presented to CSSM during module installation in two forms:

- As an attribute value in the service module's signed credentials
- As information for the CSSM registry

The policy is securely stored in the signed credentials. These credentials are authenticated by CSSM each time the module is attached. CSSM uses the signed policy description as the authoritative representation of the policy. The MODULE_INFO structure is also stored in the CSSM registry allowing applications to read the policy description by calling CSSM_GetModuleInfo.

If the CSSM_MODULE_CALLER_AUTHENTOCSSM flag is set, the module is declaring that all callers that attach this add-in module must be authenticated based on CSSM's known roots of trust. CSSM performs the authentication check on behalf of the add-in module.

If the CSSM_MODULE_CALLER_AUTHENTOMODULE flag is set, the module is declaring that all callers that attach this add-in module must be authenticated based on module-specified roots of trust. The add-in module must present the public root keys corresponding to these points of trust as input to CSSM during module installation and in the module's signed credentials.

CSSM performs the authentication check on behalf of the add-in module. The root keys specified by the add-in module are also stored in the CSSM registry, where they may be read by applications.

Add-in modules can independently authenticate applications based on module-defined points of trust. The application must incorporate a verifiable certificate in its credentials. To authenticate the application directly, the add-in module

- Locates the application's credential files using information passed to the add-in module during attach processing
- Invokes EISL facilities to verify the application credentials based on module-defined roots of trust

An application's verifiable credentials must be created during application manufacturing. The application vendor must obtain a manufacturing/signing certificate from all service module vendors and CSSM vendors who will provide it with privileged status. The application vendor uses the manufacturing certificates to create the certificate chains shown in Figure 34-1. The application must carry all of these certificate chains in the signature block for its persistent, signed manifest. When the application calls CSSM_ModuleAttach on a add-in module for which it has been granted special privileges, CSSM or the service module can verify at least one of the certificate chains in the application's credentials based on CSSM-defined or module-defined roots of trust.

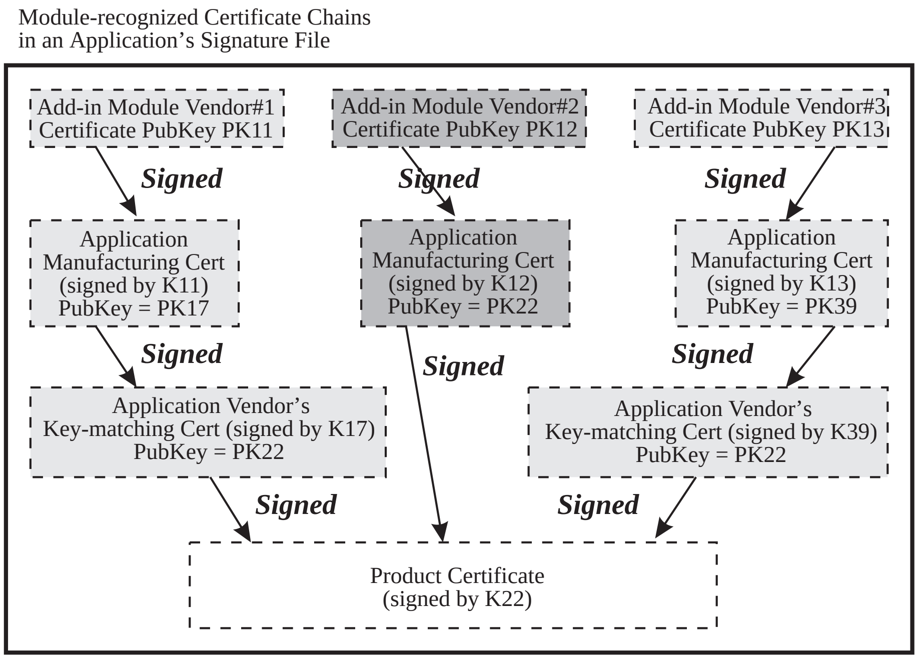
Module-recognized Certificate Chains
in an Application's Signature File
Add-in Module Vendor#1 Certificate PubKey PK11
Add-in Module Vendor#2 Certificate PubKey PK12
Add-in Module Vendor#3 Certificate PubKey PK13
Signed
Signed
Signed
Application Manufacturing Cert (signed by K11) PubKey = PK17
Application Manufacturing Cert (signed by K12) PubKey = PK22
Application Manufacturing Cert (signed by K13) PubKey = PK39
Signed
Signed
Signed
Application Vendor's Key-matching Cert (signed by K17) PubKey = PK22
Application Vendor's Key-matching Cert (signed by K39) PubKey = PK22
Signed
Signed
Product Certificate (signed by K22)

Figure 34-1 Three Module-Specific Certificate Chains

34.2.3 Initialization and Cleanup

Every module must include functions for module initialization and cleanup. The first time the module is attached, CSSM calls the module's*Initialize* function to allow the module to perform any necessary initialization operations. The last time the module is detached, CSSM calls the*Terminate* function which allows the module to perform any necessary cleanup actions. CSSM will call the module's*EventNotify* function as part of every attach and detach operation.

Chapter 35

Add-In Module Administration

Besides security services, there are several additional steps that must be performed by the module developer in order to insure access to the module via CSSM.

To insure system integrity, a module developer must create a set of digital credentials to be verified by CSSM when the module is attached.

The module developer will need to create an installation program to inform CSSM and applications of the module's identity and capabilities.

Finally, the module developer will need to insure that the appropriate sequence of component verification and module initialization steps occur prior to dynamic binding of the module with CSSM.

35.1 Manufacturing an Add-In Module

A complete set of credentials must be created for each CSSM add-in security service module as part of the module manufacturing process. These credentials are required by CSSM in order to maintain the integrity of the CDSA system. A full set of credentials are shown in Figure 35-1 and Figure 35-2. The set includes:

- The *manifest*, which is a collection of hashes of digital objects. It contains one or more *manifest sections*, where each section refers to one of the digital objects in the collection. A section contains a reference to the object, attributes about the object, a SHA-1 digest algorithm identifier, and a SHA-1 digest of the object.
- The *signer's information*, which contains a list of references to one or more sections of the manifest. Each reference includes a signature information section which contains a reference to a manifest section, a SHA-1 digest algorithms identifier, and a SHA-1 digest of the manifest section.
- The *signature block*, which contains a signature over the SHA-1 digest of the *signer's information* and the complete set of X.509 certificates comprising the module's credentials. The signature block is encoded in the particular format required by the signature block representation, for example, for a PKCS#7 signature block, the encoding format is BER / DER.

These three objects must be zipped to form a single set of credentials. Multiple implementations of standard zip algorithms interoperate on one or more platforms, hence a zipped, signed manifest retains a substantial degree of interoperability.

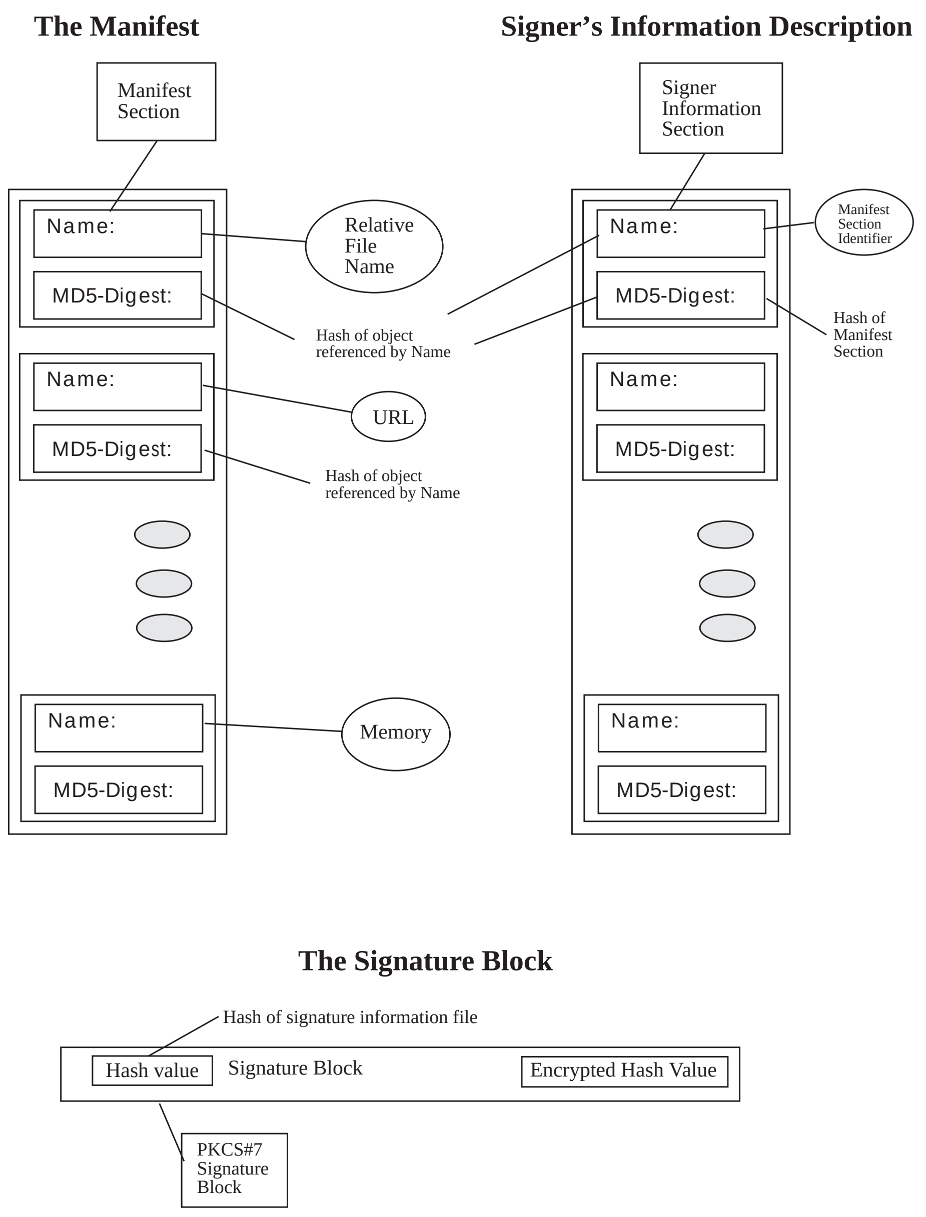

Figure 35-1 Credentials of an Add-In Service Module

The module's certificate is the leaf in one or more certificate chains. Each chain is rooted at one of a small number of known, trusted public keys. A single chain is shown in Figure 35-2. A CSSM vendor issues a certificate to the module vendor, signed with the private key of the CSSM vendor's certificate. The module vendor issues a certificate for each of its products, signing the product certificate with the module vendor's certificate. The CSSM Embedded Integrity Services Library (EISL) embeds a set of CSSM vendor public root keys. These keys are recognized roots of trust and are used when verifying a module's certificate. At runtime, EISL can also accepts additional public root keys as points of trust.

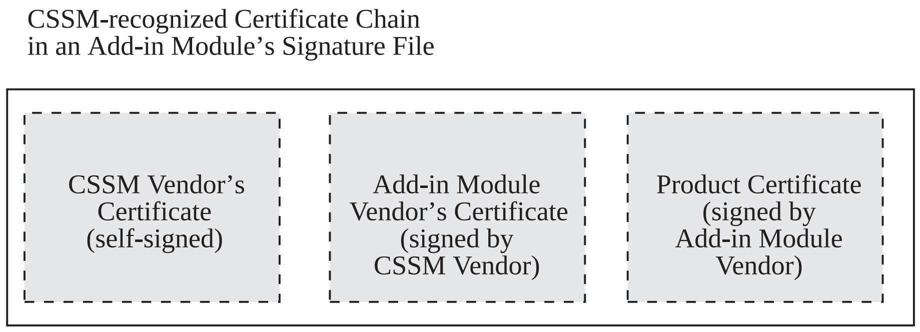
CSSM-recognized Certificate Chain
in an Add-in Module's Signature File
CSSM Vendor's Certificate (self-signed)
Add-in Module Vendor's Certificate (signed by CSSM Vendor)
Product Certificate (signed by Add-in Module Vendor)

Figure 35-2 Certificate Chain for an Add-In Service Module

The manifest forms a complete description of an add-in module. A manifest includes a manifest section for each object code file that is part of a module's implementation. Each manifest section contains:

- A reference to the object code file
- A list of the cryptographic functions and capabilities (attributes) supported in that file
- The SHA-1 digital hashing algorithm identifier
- A SHA-1 hash of the object code file

The object code files are standard OS-managed entities. Object files do not embed their digital signatures, instead, signatures are stored in a manifest separate from, but related to, the object files.

A digest of each manifest section is then computed and stored in the signature info file.

The signature file contains the PKCS#7 signature computed over the signature info file.

This set of credentials must be manufactured when the module is manufactured. Assuming a module manufacturer already has a certificate from a CSSM manufacturer, the module manufacturing process proceeds as follows:

1. Generate an X.509 product certificate for the module and sign it with the manufacturer's certificate.
2. Create a SHA-1 digest of each implementation component (object code file) used in the module.
3. Build a manifest which describes the module by referencing all object code files, digests of those files and the cryptographic capabilities (attributes) embedded in those files.
4. Build a signature info file which contains a SHA-1 digest of each manifest section.
5. Sign a SHA-1 digest of the signature info file using the private key of the product's certificate.
6. Create a PKCS #7 signature containing the signature info file digest, the product certificate and the signature.
7. Place the PKCS #7 signature in a signature file.

It is of the utmost importance that the object code files and the manifest be signed using the private key associated with the product certificate. This tightly binds the identity in the certificate with "what the module is" (that is, the object code files themselves) and with "what the module claims it is" (that is, the capability descriptions in the manifest).

35.1.1 Authenticating to Multiple CSSM Vendors

A single add-in module can authenticate with and attach to different instances of CSSM, even if these instances require add-in module credentials based on difference roots of trust. Figure 35-3. shows a complete set of credentials for an add-in module that can authenticate with a CSSM that accepts any one of three roots of trust. The credentials include three certificate chains. Each chain has a distinct root, but all chains share a common leaf certificate. This leaf certificate is used to sign the add-in module product. All three certificate chains are included in the signature file containing the credentials for this add-in module. When CSSM1 attempts to verify the add-in module's credential, a verified certificate chain will be constructed from the add-in module's leaf certificate to the root certificate containing public Key PK1, which is recognized as a point of trust by CSSM1. Hence the ad-in module's credentials will be successfully verified.

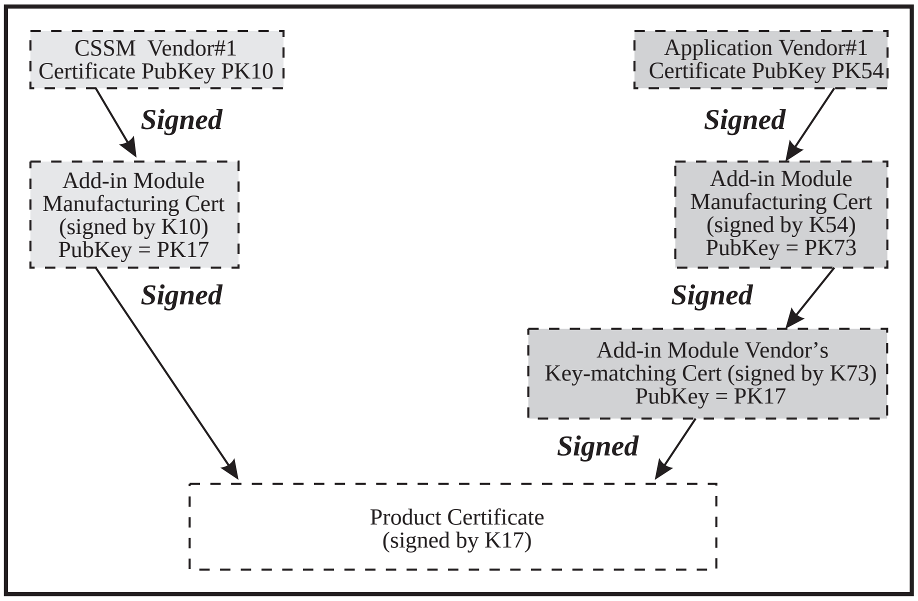

Figure 35-3 Signature File for an Add-In Module

that can authenticate with three distinct roots of trust

35.1.2 Obtaining an Add-In Module Manufacturing Certificate

Every add-in module must have an associated set of credentials, including a product certificate signed by the module manufacturer's certificate. If the module must be fully authenticated by the CSSM, then the module manufacturer must obtain a manufacturing certificate from each CSSM vendor it wishes to work with. The specific procedure for obtaining a manufacturing certificate depends on the CSSM vendor. The manufacturing certificate must be signed with the CSSM vendor's certificate and returned to the add-in module vendor.

35.1.3 Issuing an Add-In Module Product Certificate

A product certificate should be issued for each distinct product. What constitutes a distinct product is defined by the add-in module vendor. The product certificate must be directly or indirectly signed by the add-in module vendor's manufacturing certificate. Issuing a product certificate incorporates some of the processes of a Certificate Authority.

35.1.4 Manufacturing Add-In Modules

Manufacturing an add-in module is a three step process:

1. Incorporating integrity-checking facilities and roots of trust in the product software
2. Compiling the software components of the product
3. Generating integrity credentials for the add-in module product

An add-in module that performs self-check and/or authenticates CSSM during module attach must:

- Include and invoke integrity-checking software as part of the product module
- Incorporate knowledge of the roots of trust for module self-check and CSSM verification

The root of trust for self-check is the public key of the product certificate. The root of trust for authenticating a CSSM is the public root key of the CSSM vendor. Roots of trust can be presented as certificates or as keys. The add-in module should include the roots for all CSSM vendors that it trusts. This knowledge can be embedded as part of the module manufacturing process. Once the roots of trust are known, attach-time integrity checking is performed by invoking the Embedded Integrity Services Library (EISL).

CSSM invokes the module's AddInAuthenticate function to initiate the module's integrity check of CSSM. Although CSSM cannot determine that the add-in module has performed self-check and verified CSSM's credentials it is highly recommended that modules use EISL to perform these checks at attach-time and periodically during execution based on elapsed time or usage. Failure to perform these verifications during module attach processing compromises the integrity of the entire runtime environment.

After the roots of trust have been incorporated into the software component of the product and all product software components have been compiled and linked with EISL, the add-in module credentials should be created. These credentials are partitioned and persistently stored in three files:

- A manifest file
- A signer's information file
- a signature block file

The manifest file contains:

- A description of the add-in module's capabilities
- A reference to a separately link-able software component of the product
- A hash of the referenced software component

The capability description is mandatory for add-in modules that provide cryptographic services. It is highly recommended that all add-in modules, regardless of service type, include their capability description in their manifest file. The description is a flattened representation of the information defined by the CSSM_MODULE_INFO structure. This information is stored as an attribute value in the manifest. The hashes must be computed and included in the manifest file.

After the manifest file is created, the signer information file is created. The signer's information file must contain:

- References to one or more sections of a manifest file
- The hash of each reference section of a manifest file

Finally the signature block file is created. The signature block file must contain:

- A signed hash of the signer's information file
- All of the certificate chains that are trusted by the add-in module

The signing operation must be performed using the private key associated with the product certificate.

These credentials (three files) must be included with the add-in module that will be installed using CSSM_ModuleInstall. During installation these files should be placed in a subdirectory of the file system directory containing the add-in module object code files.

35.2 Installing an Add-In Module

Before an application can use a module, the module's name, location and description must be registered with CSSM by an installation application. The name given to a module includes both a logical name and a globally-unique identifier (GUID). The logical name is a string chosen by the module developer to describe the module. The GUID is a structure used to differentiate between library modules in the CSSM registry. GUIDs are discussed in more detail below. The location of the module is required at installation time so the CSSM can locate the module and its credentials when an application requests an attach. The module description indicates to CSSM the security services available within this module. The module description is clarified below.

35.2.1 Global Unique Identifiers (GUIDs)

Each module must have a globally-unique identifier (GUID) that the CSSM, applications, and the module itself use to uniquely identify a given module. The GUID is used by the CSSM registry to expose add-in module availability and capabilities to applications. A module uses its GUID to identify itself when it sets an error. When attaching the library, the application uses the GUID to identify the requested module.

A GUID is defined as:

```
typedef struct cssm_guid {
    uint32 Data1;
    uint16 Data2;
    uint16 Data3;
    uint8 Data4[8];
} CSSM_GUID, *CSSM_GUID_PTR;
```

GUID generators are publicly available for Windows* 95, Windows NT*, and on many UNIX* platforms.

35.2.2 The Module Description

At install time, the installation program must inform CSSM of the ways in which this module can be used. The module usage information includes indicators of the overall module capabilities and descriptions of the security services available from this module. The overall module capabilities include indicators such as the module's threading properties or exportability. The security service descriptions include information on each service, its sub-services, and any embedded products or services. For example, a module description might indicate that this is an exportable module containing a DL service and a CSP service, where the CSP service provides one sub-service to access a software token and a second sub-service to access a hardware token. The module description is made available to applications via queries to the CSSM registry.

35.3 Attaching an Add-In Module

Before an application can use the functions of a specific module sub-service, it must use the CSSM_ModuleAttach function to request that CSSM attach to the module's sub-service. On the first attach, CSSM verifies the integrity of the add-in module prior to loading the module. Loading the module initiates a call to an OS-specific main entry point in the module. The module must perform self-check and return to CSSM. CSSM invokes the AddInAuthenticate function implemented by the add-in module. Within that function, the add-in module must verify the integrity of CSSM.

The Embedded Integrity Services Library (EISL) must be used to perform this verification. If verification fails, the add-in module is responsible for terminating the attach process. When EISL returns a failure condition, then either the CSSM has been tampered or the attaching add-in module does not recognize the certificate of the CSSM that is attempting to attach the add-in module. The add-in module must terminate the attach. The module should not register it service function table with the suspect CSSM. The add-in module should perform clean-up operations and exit voluntarily. The module has refused to provide service in an environment that it could not verify. If verification succeeds, then the add-in module should proceed to register with CSSM.

On registration, the add-in module registers its tables of service function pointers with CSSM and receives the application's memory management upcalls. CSSM then uses the module function table to call the module's *Initialize* function to confirm version compatibility and calls the module's *EventNotify* function to indicate that an attach operation is occurring. Once these steps have successfully completed, CSSM returns a handle to the calling application which will identify the application to module sub-service pairing in future function calls. CSSM will notify the module of subsequent attach requests from the application by using the module's *EventNotify* function. Subsequent attach operations do not require integrity verification.

35.3.1 Module Entry Point

When CSSM first attaches to or last detaches from a module, it initiates an OS-specific entry point. For the Windows NT* operating system, DLLMain is the entry point. For SunOS, *_init* and *_fini* are the entry points. On attach, this function will be responsible for authenticating CSSM and then calling CSSM_RegisterServices. On detach, it will be responsible for calling CSSM_DeregisterServices. To avoid OS-related conflicts, any setup or cleanup operations should be performed in the module's *Initialize* and Terminate functions.

35.3.2 Bilateral Authentication

On attach, CSSM and the add-in module verify their own and each other's credentials by following CSSM's bilateral authentication protocol. These practices of self-checking and cross-checking by other parties increases the level of tamper detection provided by CDSA. This bilateral authentication protocol is supported by the services of the CSSM Embedded Integrity Services Library (EISL).

The basic steps in bilateral authentication during module attach are defined as follows:

1. CSSM performs a self integrity check
2. CSSM performs an integrity check of the attaching module
3. CSSM verifies secure linkage by checking that the initiation point is within the verified module
4. CSSM invokes the add-in module
5. The add-in module performs a self integrity check
6. The add-in module performs an integrity check of CSSM
7. The add-in module verifies secure linkage by checking that the function call originated from the verified CSSM

Each authenticating entity invokes ISL functions to carry out the steps in this process. The following ISL functions are used to carry out the seven step bilateral authentication protocol:

- ISL_SelfCheck
- ISL_VerifyAndLoadModuleAndCredentials
- ISL_LocateProcedureAddress, ISL_CheckAddressWithinModule
- ISL_SelfCheck
- ISL_VerifyLoadedModuleAndCredentials
- ISL_GetReturnAddress, ISL_CheckAddressWithinModule

The ISL Verify functions check all aspects of a module's credentials, including the certificate chain, the signature on the manifest, the signature on the capability descriptions, and the signature on each object code file. The ISL Verify functions cannot check for secure linkage. CSSM and the add-in module must use the ISL address checking functions to verify secure linkage with the party being verified. The purpose of the secure linkage check is to verify that the object code just verified is either the code you are about to invoke or the code that invoked you. To free the data structures used in bilateral authentication, the ISL provides a Recycle function.

35.3.3 Module Function Table Registration

On attach, a module must register its function tables with CSSM by calling CSSM_RegisterServices. Its function tables will consist of a table of module management function pointers, plus one table of service provider interface function pointers for each (service, sub-service) pair contained in this module. The module management functions include *Initialize*, *EventNotify*, and *Terminate*. The service provider interface functions reflect the CSSM API for each security service. The function prototypes and their descriptions are given in the Service Provider Interface Specifications, the *CSSM Cryptographic Service Provider Interface Specification*, *CSSM Trust Policy Interface Specification*, *CSSM Certificate Library Interface Specification*, *CSSM Data Storage Library Interface Specification*, *CSSM Key Recovery Interface Specification*. If a sub-service does not support a given function in its service provider interface, the pointer to that function should be set to NULL. These structures are specified in the CSSM header files, **<cssmspi.h>**, **<cssmcspi.h>**, **<cssmtpi.h>**, **<cssmcli.h>**, and **<cssmdli.h>**.

35.3.4 Memory Management Upcalls

All memory allocation and de-allocation for data passed between the application and a module via CSSM is ultimately the responsibility of the calling application. Since a module needs to allocate memory to return data to the application, the application must provide the module with a means of allocating memory that the application has the ability to free. It does this by providing the module with memory management upcalls.

Memory management upcalls are pointers to the memory management functions used by the calling application. They are provided to a module via CSSM as a structure of function pointers. The functions will be the calling application's equivalent of malloc, free, calloc, and re-alloc and will be expected to have the same behavior as those functions. The function parameters will consist of the normal parameters for that function. The function return values should be interpreted in the standard manner. A module is responsible for making the memory management functions available to all of its internal functions.

35.4 Error Handling

When an error occurs inside a module, the function should call CSSM_SetError. The CSSM_SetError function takes the module's GUID and an error number as inputs. The module's GUID is used to identify where the error occurred. The error number will be used to describe the error.

The error number set by a module sub-service should fall into one of two ranges. The first range of error numbers is pre-defined by CSSM. These are errors that are common to all modules implementing a given sub-service function. They are described in the *CSSM Cryptographic Service Provider Interface Specification* as part of the function definitions. They are defined in the header file **<cssmerr.h>**, which is distributed as part of CSSM. The second range of error numbers is used to define module-specific error codes. These module-specific error codes should be in the range of CSSM_XX_PRIVATE_ERROR to CSSM_XX_END_ERROR, where XX stands for the service category abbreviation (CSP, TP, CL, DL). CSSM_XX_PRIVATE_ERROR and CSSM_XX_END_ERROR are also defined in the header file **<cssmerr.h>**. A module developer is responsible for making the definition and interpretation of their module-specific error codes available to applications.

When no error has occurred, but the appropriate return value from a function is CSSM_FALSE, that function should call CSSM_ClearError before returning. When the application receives a CSSM_FALSE return value, it is responsible for checking whether an error has occurred by calling CSSM_GetError. If the module function has called CSSM_ClearError, the calling application receives a CSSM_OK response from the CSSM_GetError function, indicating no error has occurred.

35.5 Install Example

An installation program is responsible for registering a module's capabilities with CSSM. A sample code-segment for the installation of a CL Module is shown in the example below.

35.5.1 CL Module Install

```
#include "cssm.h"
CSSM_GUID clm_guid =
{ 0x5fc43dc1, 0x732, 0x11d0, { 0xbb, 0x14, 0x0, 0xaa, 0x0,
                                           0x36, 0x67, 0x2d } };
CSSM_BOOL CLModuleInstall()
{
    CSSM_VERSION      cssm_version = { CSSM_MAJOR, CSSM_MINOR };
    CSSM_VERSION  cl_version = { CLM_MAJOR_VER, CLM_MINOR_VER };
    CSSM_GUID           cl_guid = clm_guid;
    CSSM_CLSUBSERVICE   sub_service;
    CSSM_SERVICE_INFO   service_info;
    CSSM_MODULE_INFO    module_info;
    char                SysDir[_MAX_PATH];

        /* fill sub-service information */
    sub_service.SubServiceId = 0;
    strcpy(sub_service.Description, "X509v3 SubService");
    sub_service.CertType = CSSM_CERT_X_509v3;
    sub_service.CertEncoding = CSSM_CERT_ENCODING_DER;
    sub_service.AuthenticationMechanism = CSSM_AUTHENTICATION_NONE;
    sub_service.NumberOfTemplateFields = NUMBER_X509_CERT_OIDS;
    sub_service.CertTemplates = X509_CERT_OIDS_ARRAY;
    sub_service.NumberOfTranslationTypes = 0;
    sub_service.CertTranslationTypes = NULL;
    sub_service.WrappedProduct.EmbeddedEncoderProducts = NULL;
    sub_service.WrappedProduct.NumberOfEncoderProducts = 0;
    sub_service.WrappedProduct.AccessibleCAProducts = NULL;
    sub_service.WrappedProduct.NumberOfCAProducts = 0;

        /* fill service information */
    strcpy(service_info.Description, "CL Service");
    service_info.Type = CSSM_SERVICE_CL;
    service_info.Flags = 0;
    service_info.NumberOfSubServices = 1;
    service_info.ClSubServiceList = &sub_service;
    service_info.Reserved = NULL;

        /* fill module information */
    module_info.Version = cl_version;
    module_info.CompatibleCSSMVersion = cssm_version;
    strcpy(module_info.Description, "Vendor Module");
    strcpy(module_info.Vendor, "Vendor Name");
    module_info.Flags = 0;
    module_info.ServiceMask = CSSM_SERVICE_CL;
    module_info.NumberOfServices = 1;
    module_info.ServiceList = &service_info;
    module_info.Reserved = NULL;

        /* get system dir path */
    GetSystemDirectory(SysDir, _MAX_PATH);

        /* Install the module */
    if (CSSM_ModuleInstall(clm_fullname_string,
                clm_filename_string,
                SysDir,
                &clm_guid,
                &module_info,
                NULL,
                NULL) == CSSM_FAIL)
    {
        return CSSM_FALSE;
    }

    return CSSM_TRUE;
}
```

35.6 Attach/Detach and AddInAuthenticate Example

A module is responsible for performing certain operations when CSSM attaches to and detaches from it. Modules that have been developed for Windows-based systems use the DllMain routine to perform those operations, as shown in the DL Module example below.

35.6.1 DLLMain

```
#include "cssm.h"
CSSM_GUID dl_guid =
{ 0x5fc43dc1, 0x732, 0x11d0, { 0xbb, 0x14, 0x0, 0xaa, 0x0,
                                         0x36, 0x67, 0x2d } };
CSSM_SPI_DL_FUNCS FunctionTable;
CSSM_REGISTRATION_INFO  DLRegInfo;
CSSM_MODULE_FUNCS       Services;
CSSM_SPI_MEMORY_FUNCS   DLMemoryFunctions;

BOOL WINAPI DllMain ( HANDLE hInstance, DWORD dwReason,
                                               LPVOID lpReserved)
{
switch (dwReason)
{
case DLL_PROCESS_ATTACH:
{
    ISL_VERIFIED_MODULE_PTR VerifiedDLModulePtr = NULL;
    VerifiedDLModulePtr = ISL_SelfCheck();
    if(VerifiedDLModulePtr == NULL) return FALSE;
    ISL_RecycleVerifiedModuleCredentials
                                 (VerifiedDLModulePtr);

    break;
}
```

```
        case DLL_THREAD_ATTACH:
        break;

        case DLL_THREAD_DETACH:
        break;

        case DLL_PROCESS_DETACH:
              if (CSSM_DeregisterServices (&dl_guid) != CSSM_OK)
              return FALSE;
        break;
    }
        return TRUE;
    }

    CSSM_RETURN CSSMAPI AddInAuthenticate
        (char* cssmCredentialPath,
        char* cssmSection
        char* appFileName,
        char* appPathName)
    {

        ISL_VERIFIED_MODULE_PTR VerifiedCLModulePtr = NULL;
        ISL_STATUS islret;
        void* retAddress;
        ISL_CONST_DATA ConstData = {0, NULL};
        ISL_CONST_DATA ConstPathData = {0, NULL};
        ISL_CONST_DATA ConstSectionData = {0, NULL};

        ConstPathData.Length = strlen(cssmCredentialPath);
        ConstPathData.Data = (uint8*) cssmCredentialPath;
        ConstSectionData.Length = strlen(cssmSection);
        ConstSectionData.Data = (uint8*) cssmSection;

        /* Verify CSSM's static and dynamic footprint based on its
                                                        manifest */
        VerifiedCSSMModulePtr =
                             ISL_VerifyLoadedModuleAndCredentials
               (ConstPathDataConstSectionData,ConstData,ConstData);
        if(VerifiedCSSMModulePtr == NULL)
            return CSSM_FAIL;

        /* Verify secure linkage with CSSM */
        ISL_GetReturnAddress(retAddress);
        islret = ISL_CheckAddressWithinModule
                                (VerifiedCSSMModulePtr, retAddress);
        if(islret == ISL_FAIL)
        {
            ISL_RecycleVerifiedModuleCredentials(VerifiedCSSMModulePtr);
            VerifiedCSSMModulePtr = NULL;
            return CSSM_FAIL;
        }
        ISL_RecycleVerifiedModuleCredentials(VerifiedCSSMModulePtr);

        /* Authenticate application credentials directly if required*/
        if((appFileName == NULL) ]] (appPathName == NULL))
         return CSSM_FAIL;
        else
        {
         /* Verify the application’s credentials */
        }

        /* Fill in Registration information and register services
                                                             with CSSM*/
        DLRegInfo.Initialize            = DL_Initialize;
        DLRegInfo.Terminate             = DL_Uninitialize;
        DLRegInfo.EventNotify           = DL_EventNotify;
        DLRegInfo.GetModuleInfo         = NULL;
        DLRegInfo.FreeModuleInfo        = NULL;
        DLRegInfo.ThreadSafe            = CSSM_TRUE;
        DLRegInfo.ServiceSummary        = CSSM_SERVICE_DL;
        DLRegInfo.NumberOfServiceTables = 1;
        DLRegInfo.Services              = &Services;

        /* Fill in Services */

        Services.ServiceType  = CSSM_SERVICE_DL;
        Services.DlFuncs = &FunctionTable;

        /* Fill in FunctionTable with function pointers */
        FunctionTable.Authenticate = DL_Authenticate;
        FunctionTable.DbOpen = DL_DbOpen;
        FunctionTable.DbClose = DL_DbClose;
        /* initialize all the other function pointers */
        FunctionTable.PassThrough = DL_PassThrough;

        /* Call CSSM_RegisterServices to
                            register the FunctionTable */
        /* with CSSM and to receive the application’s
                                    memory upcall table*/
        if (CSSM_RegisterServices (&dl_guid, &DLRegInfo,
                         &DLMemoryFunctions,NULL) != CSSM_OK)
        return FALSE;

        /* Make the upcall table available to all
        functions in this library */

    }
```

Chapter 36

Add-In Module Interface Functions

An add-in module interfaces with CSSM via the two functions described below.

NAME

Initialize

SYNOPSIS

```
CSSM_RETURN CSSMAPI Initialize
    (CSSM_MODULE_HANDLE Handle,
    uint32 VerMajor,
    uint32 VerMinor)
```

DESCRIPTION

This function checks whether the current version of the module is compatible with the input version and performs any module-specific setup activities.

PARAMETERS

Handle (input)
The handle that identifies the module to application thread pairing.

VerMajor (input)
The major version number of the module expected by the calling application.

VerMinor (input)
The minor version number of the module expected by the calling application.

RETURN VALUE

A CSSM_OK return value signifies that the current version of the module is compatible with the input version numbers and all setup operations were successfully performed. When CSSM_FAIL is returned, either the current module is incompatible with the requested module version or an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INITIALIZE_FAIL
Unable to initialize the DL module.

SEE ALSO

Terminate, *EventNotify*

NAME

Terminate

SYNOPSIS

```
CSSM_RETURN CSSMAPI Terminate
    (CSSM_MODULE_HANDLE Handle)
```

DESCRIPTION

This function performs any module-specific cleanup activities.

PARAMETERS

Handle (input)
The handle that identifies the module to application thread pairing.

RETURN VALUE

A CSSM_OK return value signifies that all cleanup operations were successfully performed. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

SEE ALSO

Initialize, *EventNotify*

NAME

EventNotify

SYNOPSIS

```
CSSM_RETURN CSSMAPI EventNotify
    (CSSM_MODULE_HANDLE Handle, const CSSM_EVENT_TYPE Event, const
    uint32 Param)
```

DESCRIPTION

This function is used by CSSM to notify the module of certain events such as module attach and detach operations.

PARAMETERS

Handle (input)
The handle that identifies the module to application thread pairing.

Event (input)
The event which is occurring. The possible events are described in the table below.

Event	**Description**
CSSM_EVENT_ATTACH	The application has requested an attach operation.
CSSM_EVENT_DETACH	The application has requested a detach operation.
CSSM_EVENT_INFOATTACH	An application has requested module info and CSSM wants to obtain the module's dynamic capabilities. The add-in module cannot assume that Initialize or Terminate have been called.
CSSM_EVENT_INFODETACH	CSSM has finished obtaining the module's dynamic capabilities.
CSSM_EVENT_CREATE_CONTEXT	A context has been created.
CSSM_EVENT_DELETE_CONTEXT	A context has been deleted.

Table 36-1 Module Event Types

Param (input)
An event-specific parameter.

Event	**Parameter**
CSSM_EVENT_ATTACH	None.
CSSM_EVENT_DETACH	None.
CSSM_EVENT_INFOATTACH	None
CSSM_EVENT_INFODETACH	None
CSSM_EVENT_CREATE_CONTEXT	The context handle.
CSSM_EVENT_DELETE_CONTEXT	The context handle.

Table 36-2 Module Event Parameters

RETURN VALUE

A CSSM_OK return value signifies that the module's event-specific operations were successfully performed. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

SEE ALSO

Initialize, Terminate

NAME

AddInAuthenticate

SYNOPSIS

```
CSSM_RETURN CSSMAPI  AddInAuthenticate
    (const char *CssmCredentialPath,
    const char *CssmSection,
    const char *AppFileName,
    const char *AppPathName)
```

DESCRIPTION

This function should perform the add-in service module's half of the bilateral authentication procedure with CSSM. The CSSM credential path and section information is used to locate the CSSM's credentials to be verified. The credentials are a zipped, signed manifest.

If the application filename and pathname are provided, the add-in service has the option to perform an integrity and identity check of the attaching application. The filename and pathname can be used to locate the application's signed credentials. If this information is not provided and the add-in service module requires application verification, verification fails.

This function is the first module interface invoked by CSSM after loading and invoking the main entry point. In particular, the add-in service module's initialize function is invoked by CSSM after this function has successfully completed execution.

PARAMETERS

CssmCredentialPath (input)
A string containing the path name for locating the calling CSSM's credentials. These credentials are a zipped, signed manifest. The service module should verify these credentials as part of the bilateral authentication process.

CssmSection (input)
A string containing the section name for the manifest section containing a description and cryptographic digest of the calling CSSM's object code.

AppFileName (input/optional)
The name of the file that implements the application (containing its main entry point). This file name can be used to locate the application's credentials for purposes of application authentication by the add-in service module. The application provides this input to CSSM if the application has credentials it wishes to present for verification to CSSM or to the add-in service module. If application authentication is not required or the caller did not provide any file name information, this parameter is NULL.

AppPathName (input/optional)
The pathname to the file that implements the application (containing its main entry point). This pathname can be used to locate the application's credentials for purposes of application authentication by the add-in service module. The application provides this input to CSSM if the application has credentials it wishes to present for verification to CSSM or to the add-in service module. If application authentication is not required or the caller did not provide any file name information, this parameter is NULL.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

Appendix D

Relevant CSSM API Functions

D.1 Overview

Several API functions are particularly relevant to module developers, because they are used either by the application to access a module or by a module to access CSSM services, such as the CSSM registry or the error-handling routines. They are included in this appendix for quick-reference by module developers. For additional information, a module developer is encouraged to reference the CSSM Application Programming Interface.

D.2 Data Structures

D.2.1 CSSM_BOOL

This data type is used to indicate a true or false condition.

```
typedef uint32 CSSM_BOOL;
#define CSSM_TRUE 1
#define CSSM_FALSE 0
```

Definition

CSSM_TRUE
Indicates a true result or a true value.

CSSM_FALSE
Indicates a false result or a false value.

D.2.2 CSSM_RETURN

This data type is used to indicate whether a function was successful.

```
typedef enum cssm_return {
    CSSM_OK = 0,
    CSSM_FAIL = -1
} CSSM_RETURN
```

Definition

CSSM_OK
Indicates operation was successful.

CSSM_FAIL
Indicates operation was unsuccessful.

D.2.3 CSSM_STRING

This is used by CSSM data structures to represent a character string inside of a fixed-length buffer. The character string is expected to be NULL-terminated. The string size was chosen to accommodate current security standards, such as PKCS #11.

```
#define CSSM_MODULE_STRING_SIZE  64
typedef char CSSM_STRING [CSSM_MODULE_STRING_SIZE + 4];
```

D.2.4 CSSM_DATA

The CSSM_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory. This memory must be allocated and freed using the memory management routines provided by the calling application via CSSM. Trust policy modules and certificate libraries use this structure to hold certificates and CRLs. Other add-in service modules, such as CSPs use this same structure to hold general data buffers, and DLMs use this structure to hold persistent security-related objects.

```
typedef struct cssm_data{
    uint32 Length;  /* in bytes */
    uint8 *Data;
} CSSM_DATA, *CSSM_DATA_PTR
```

Definition

Length
Length of the data buffer in bytes.

Data
Points to the start of an arbitrary length data buffer.

D.2.5 CSSM_GUID

This structure designates a global unique identifier (GUID) that distinguishes one add-in module from another. All GUID values should be computer-generated to guarantee uniqueness (the GUID generator in Microsoft Developer Studio* and the RPC UUIDGEN/uuid_gen program on a number of UNIX* platforms can be used).

```
typedef struct cssm_guid{
    uint32 Data1;
    uint16 Data2;
    uint16 Data3;
    uint8  Data4[8];
} CSSM_GUID, *CSSM_GUID_PTR
```

Definition

Data1
Specifies the first eight hexadecimal digits of the GUID.

Data2
Specifies the first group of four hexadecimal digits of the GUID.

Data3
Specifies the second group of four hexadecimal digits of the GUID.

Data4
: Specifies an array of eight elements that contains the third and final group of eight hexadecimal digits of the GUID in elements 0 and 1, and the final 12 hexadecimal digits of the GUID in elements 2 through 7.

D.2.6 CSSM_VERSION

This structure is used to represent the version of CDSA components.

```
typedef struct cssm_version {
    uint32 Major;
    uint32 Minor;
} CSSM_VERSION, *CSSM_VERSION_PTR;
```

Definition

Major
The major version number of the component.

Minor
The minor version number of the component.

D.2.7 CSSM_SUBSERVICE_UID

This structure uniquely identifies a set of behaviors within a subservice within a CSSM add-in module.

```
typedef struct cssm_subservice_uid {
    CSSM_GUID Guid;
    CSSM_VERSION Version;
    uint32 SubserviceId;
    uint32 SubserviceFlags;
} CSSM_SUBSERVICE_UID, *CSSM_SUBSERVICE_UID_PTR;
```

Definition

Guid
A unique identifier for a CSSM add-in module.

Version
The version of the add-in module.

SubserviceId
An identifier for the subservice within the add-in module.

SubserviceFlags
An identifier for a set of behaviors provided by this subservice.

D.2.8 CSSM_HANDLE

A unique identifier for an object managed by CSSM or by an add-in module.

```
typedef uint32 CSSM_HANDLE, *CSSM_HANDLE_PTR
```

D.2.9 CSSM_MODULE_HANDLE

A unique identifier for an attached service provider module.

```
typedef uint32 CSSM_MODULE_HANDLE
```

D.2.10 CSSM_EVENT_TYPE

Events occur when an application calls a CSSM core service function. CSSM informs the attached module of this event using the EventNotify call to the Service provider module. Six types of events are defined:

```
typedef uint32 CSSM_EVENT_TYPE, *CSSM_EVENT_TYPE_PTR;

#define CSSM_EVENT_ATTACH          (0)
     /* application has requested an attach operation */
#define CSSM_EVENT_DETACH          (1)
     /* application has requested an detach operation */
#define CSSM_EVENT_INFOATTACH      (2)
     /* application has requested module info for dynamic module
                                                capabilities */
#define CSSM_EVENT_INFODETACH      (3)
     /* CSSM has completed obtaining dynamic module
                                                 capabilities */
#define CSSM_EVENT_CREATE_CONTEXT  (4)
     /* application has performed a create context operation */
#define CSSM_EVENT_DELETE_CONTEXT  (5)
     /* application has performed a delete context operation */
```

D.2.11 CSSM_SERVICE_MASK

This defines a bit mask of all the types of CSSM services a single module can implement.

```
typedef uint32 CSSM_SERVICE_MASK;

#define CSSM_SERVICE_CSSM  0x1
#define CSSM_SERVICE_CSP   0x2
#define CSSM_SERVICE_DL    0x4
#define CSSM_SERVICE_CL    0x8
#define CSSM_SERVICE_TP    0x10
#define CSSM_SERVICE_LAST  CSSM_SERVICE_TP
```

D.2.12 CSSM_SERVICE_TYPE

This data type is used to identify a single service from the CSSM_SERVICE_MASK options defined above.

```
typedef CSSM_SERVICE_MASK CSSM_SERVICE_TYPE
```

D.2.13 CSSM_SERVICE_FLAGS

This bitmask is used to identify characteristics of the service, such as whether it contains any embedded products.

```
typedef uint32 CSSM_SERVICE_FLAGS

#define CSSM_SERVICE_ISWRAPPEDPRODUCT 0x1
         /* On  = Contains one or more embedded products
                    Off = Contains no embedded products */
```

D.2.14 CSSM_SERVICE_INFO

This structure holds a description of a module service. The service described is of the CSSM service type specified by the module type.

```
typedef struct cssm_serviceinfo {
    CSSM_STRING Description;    /* Service description */
    CSSM_SERVICE_TYPE Type;     /* Service type */
    CSSM_SERVICE_FLAGS Flags;   /* Service flags */
    uint32 NumberOfSubServices; /* Number of sub services in SubService List */
    union cssm_subservice_list {    /* list of sub services */
        void *SubServiceList;
        CSSM_CSPSUBSERVICE_PTR CspSubServiceList;
        CSSM_DLSUBSERVICE_PTR  DlSubServiceList;
        CSSM_CLSUBSERVICE_PTR  ClSubServiceList;
        CSSM_TPSUBSERVICE_PTR  TpSubServiceList;
    } SubserviceList ;
    void *Reserved;
} CSSM_SERVICE_INFO, *CSSM_SERVICE_INFO_PTR;
```

Definition

Description
A text description of the service.

Type
Specifies exactly one type of service structure, such as CSSM_SERVICE_CSP, CSSM_SERVICE_CL, and so on.

Flags
Characteristics of this service, such as whether it contains any embedded products.

NumberOfSubServices
The number of elements in the module SubServiceList.

SubServiceList
A list of descriptions of the encapsulated SubServices which are not of the basic service types.

CspSubServiceList
A list of descriptions of the encapsulated CSP SubServices.

DlSubServiceList
A list of descriptions of the encapsulated DL SubServices.

ClSubServiceList
A list of descriptions of the encapsulated CL SubServices.

TpSubServiceList
A list of descriptions of the encapsulated TP SubServices.

Reserved
This field is reserved for future use. It should always be set to NULL.

D.2.15 CSSM_MODULE_FLAGS

This bitmask is used to identify characteristics of the module, such as whether or not it is threadsafe, exportable, and so on. The flags also describe the module vendor's policy for how CSSM process all module attach requests for this service module. The service module can select of the following authentication checks before allowing an instance of the service module to be attached by a requesting application:

- The attaching application must be successfully authenticated by CSSM, based on CSSM's roots of trust
- The attaching application must be successfully authenticated by CSSM, based on module-specified roots of trust

```
typedef uint32 CSSM_MODULE_FLAGS;

#define CSSM_MODULE_THREADSAFE  0x1
                    /* Module is threadsafe */
#define CSSM_MODULE_EXPORTABLE  0x2
                    /* Module can be exported outside the USA */
#define CSSM_MODULE_CALLER_AUTHENTOCSSM  0x04
                       /* CSSM authenticates the caller based */
                       /* on CSSM-known points of trust */
#define CSSM_MODULE_CALLER_AUTHENTOMODULE  0x08
                       /* CSSM authenticates the caller based */
                       /* on module-supplied points of trust */
```

D.2.16 CSSM_MODULE_INFO

This structure aggregates all service descriptions about all service types of a module implementation.

```
typedef struct cssm_moduleinfo {
    CSSM_VERSION Version;              /* Module version */
    CSSM_VERSION CompatibleCSSMVersion; /* CSSM version the
                                       module is written for*/
    CSSM_STRING Description;           /* Module description */
    CSSM_STRING Vendor;                /* Vendor name */
    CSSM_STRING ModuleFileName,        /* File name for module
                                       object code */
    CSSM_STRING ModulePathName,        /* Path name to module
                                       object code */
    CSSM_MODULE_FLAGS Flags;            /* Flags to describe and
                                        control module use */
    CSSM_KEY_PTR AppAuthenRootKeys,     /* Module-specific keys to
                                        authen apps */
    uint32 NumberOfAppAuthenRootKeys,   /* Number of module-
                                        specific root keys */
    CSSM_SERVICE_MASK ServiceMask;      /* Bit mask of supported
                                        services */
    uint32 NumberOfServices;            /* Number of services
                                        in ServiceList */
    CSSM_SERVICE_INFO_PTR ServiceList;  /* A list of service
                                        info structures */
    void *Reserved;
} CSSM_MODULE_INFO, *CSSM_MODULE_INFO_PTR;
```

Definition

Version
The major and minor version numbers of this add-in module.

CompatibleCSSMVersion
The version of CSSM that this module was written to.

Description
A text description of this module and its functionality.

Vendor
The name and description of the module vendor.

ModuleFileName
The name of the file that implements the add-in module. This file name is used to locate the add-in module's credentials for purposes module authentication.

ModulePathName
The name of the path to the file that implements the add-in module. This file name is used to locate the add-in module's credentials for purposes of module authentication.

Flags
Characteristics of this module, such as whether or not it is threadsafe.

AppAuthenRootKeys
Public root keys used by CSSM to verify an application's credentials when the service module has requested authentication based on module-specified root keys by setting the CSSM_MODULE_CALLER_AUTHENTOMODULE bit to true in its CSSM_MODULE_FLAGS mask. These keys should successfully authenticate only those applications that the service module wishes to recognize to receive the services the module has registered with CSSM during module installation.

NumberOfAppAuthenRootKeys
The number of public root keys in the AppAuthenRoot Keys list.

ServiceMask
A bit mask identifying the types of services available in this module.

NumberOfServices
The number of services for which information is provided. Multiple descriptions (as sub-services) can be provided for a single service category.

ServiceList
An array of pointers to the service information structures. This array contains NumberOfServices entries.

Reserved
This field is reserved for future use. It should always be set to NULL.

D.2.17 CSSM_ALL_SUBSERVICES

This data type is used to identify that information on all of the sub-services is being requested or returned.

```
#define CSSM_ALL_SUBSERVICES  (0xFFFFFFFF)
```

D.2.18 CSSM_INFO_LEVEL

This enumerated list defines the levels of information detail that can be retrieved about the services and capabilities implemented by a particular module. Modules can implement multiple CSSM service types. Each service may provide one or more sub-services. Modules can also have dynamically available services and features.

```
typedef enum cssm_info_level {
    CSSM_INFO_LEVEL_MODULE  = 0,
      /* values from CSSM_SERVICE_INFO struct */
    CSSM_INFO_LEVEL_SUBSERVICE  = 1,
      /* values from CSSM_SERVICE_INFO and XXsubservice struct */
    CSSM_INFO_LEVEL_STATIC_ATTR  = 2,
      /* values from CSSM_SERVICE_INFO and XXsubservice and
         all static-valued attributes of a subservice */
    CSSM_INFO_LEVEL_ALL_ATTR  = 3,
      /* values from CSSM_SERVICE_INFO and XXsubservice and
         all attributes, static and dynamic, of a subservice */
} CSSM_INFO_LEVEL;
```

D.2.19 CSSM_NET_ADDRESS_TYPE

This enumerated type defines representations for specifying the location of a service.

```
typedef enum cssm_net_address_type {
    CSSM_ADDR_NONE = 0,
    CSSM_ADDR_CUSTOM = 1,
    CSSM_ADDR_URL = 2, /* char* */
    CSSM_ADDR_SOCKADDR = 3,
    CSSM_ADDR_NAME = 4 /* char* - qualified by access method */
} CSSM_NET_ADDRESS_TYPE;
```

D.2.20 CSSM_NET_ADDRESS

This structure holds the address of a service. Typically the service is remote, but the value of the address field may resolve to the local system. The AddressType field defines how the Address field should be interpreted.

```
typedef struct cssm_net_address {
    CSSM_NET_ADDRESS_TYPE AddressType;
    CSSM_DATA Address;
} CSSM_NET_ADDRESS, *CSSM_NET_ADDRESS_PTR;
```

D.2.21 CSSM_NET_PROTOCOL

This enumerated list defines the application-level protocols that could be supported by a Certificate Library Module that communicates with Certification Authorities, Registration Authorities and other services, or by a Data Storage Library Module that communicates with service-based storage and directory services.

```
typedef enum cssm_net_protocol {
    CSSM_NET_PROTO_NONE = 0, /* local */
    CSSM_NET_PROTO_CUSTOM = 1, /* proprietary implementation */
    CSSM_NET_PROTO_UNSPECIFIED = 2, /* implementation default */
    CSSM_NET_PROTO_LDAP = 3, /* light weight directory access
                                        protocol */
    CSSM_NET_PROTO_LDAPS = 4, /* ldap/ssl where SSL initiates
                                        the connection */
    CSSM_NET_PROTO_LDAPNS = 5, /* ldap where ldap negotiates an
                                        SSL session */
    CSSM_NET_PROTO_X500DAP = 6, /* x.500 Directory access
                                        protocol */
    CSSM_NET_PROTO_FTPDAP = 7, /* file transfer protocol for
                                        cert/crl fetch */
    CSSM_NET_PROTO_FTPDAPS = 8, /* ftp/ssl where SSL initiates
                                        the connection */
    CSSM_NET_PROTO_NDS = 9, /* Novell directory services */
    CSSM_NET_PROTO_OCSP = 10, /* online certificate status
                                        protocol */
    CSSM_NET_PROTO_PKIX3 = 11, /* the cert request protocol
                                        in PKIX3 */
    CSSM_NET_PROTO_PKIX3S = 12, /* The ssl/tls derivative of
                                        PKIX3 */
    CSSM_NET_PROTO_PKCS_HTTP = 13, /* PKCS client <=> CA protocol
                                        over HTTP */
    CSSM_NET_PROTO_PKCS_HTTPS = 14, /* PKCS client <=> CA protocol
                                        over HTTPS */
} CSSM_NET_PROTOCOL;
```

D.2.22 CSSM_USER_AUTHENTICATION_MECHANISM

This enumerated list defines different methods an add-in module can require when authenticating a caller. The module specifies which mechanism the caller must use for each sub-service type provided by the module. CSSM-defined authentication methods include password-based authentication, a login sequence, or a certificate and passphrase. It is anticipated that new mechanisms will be add to this list as required.

```
typedef enum cssm_user_authentication_mechanism {
    CSSM_AUTHENTICATION_NONE = 0,
    CSSM_AUTHENTICATION_CUSTOM = 1,
    CSSM_AUTHENTICATION_PASSWORD = 2,
    CSSM_AUTHENTICATION_USERID_AND_PASSWORD = 3,
    CSSM_AUTHENTICATION_CERTIFICATE_AND_PASSPHRASE = 4,
    CSSM_AUTHENTICATION_LOGIN_AND_WRAP = 5,
} CSSM_USER_AUTHENTICATION_MECHANISM;
```

D.2.23 CSSM_CALLBACK

An application uses this data type to request that an add-in module call back into the application for certain cryptographic information.

```
typedef CSSM_DATA_PTR (CSSMAPI *CSSM_CALLBACK) (void *allocRef, uint32 ID);
```

Definition

allocRef
: Memory heap reference specifying which heap to use for memory allocation.

ID
: Input data to identify the callback.

D.2.24 CSSM_CRYPTO_DATA

This data structure is used to encapsulate cryptographic information, such as the passphrase to use when accessing a private key.

```
typedef struct cssm_crypto_data {
    CSSM_DATA_PTR Param;
    CSSM_CALLBACK Callback;
    uint32  ID;
}CSSM_CRYPTO_DATA, *CSSM_CRYPTO_DATA_PTR
```

Definition

Param
: A pointer to the parameter data and its size in bytes.

Callback
: An optional callback routine for the add-in modules to obtain the parameter.

ID
: A tag that identifies the callback.

D.2.25 CSSM_USER_AUTHENTICATION

This structure holds the user's credentials for authenticating to a module. The type of credentials required is defined by the module and specified as a CSSM_USER_AUTHENTICATION_MECHANISM.

```
typedef struct cssm_user_authentication {
    CSSM_DATA_PTR Credential; /* a cert, a shared secret, other */
    CSSM_CRYPTO_DATA_PTR MoreAuthenticationData;
} CSSM_USER_AUTHENTICATION, *CSSM_USER_AUTHENTICATION_PTR;
```

Definition

Credential
A certificate, a shared secret, a magic token or what every is required by an add-in service modules for user authentication. The required credential type is specified as a CSSM_USER_AUTHENTICATION_MECHANISM .

MoreAuthenticationData
A passphrase or other data that can be provided as immediate data within this structure or via a callback function to the user/caller.

D.2.26 CSSM_NOTIFY_CALLBACK

An application uses this data type to request that an add-in module call back into the application to notify it of certain events.

```
typedef CSSM_RETURN (CSSMAPI *CSSM_NOTIFY_CALLBACK)
    (CSSM_CSP_HANDLE ModuleHandle,
         uint32 Application,
         uint32 Reason,
         uint32 Param);
```

Definition

ModuleHandle
The handle of the attached add-in module.

Application
Input data to identify the callback.

Reason
The reason for the notification.

Reason	Description
CSSM_NOTIFY_SURRENDER	The add-in module is temporarily surrendering control of the process
CSSM_NOTIFY_COMPLETE	An asynchronous operation has completed
CSSM_NOTIFY_DEVICE_REMOVED	A device, such as a token, has been removed

CSSM_NOTIFY_DEVICE_INSERTED	A device, such as a token, has been inserted

Table D-1 Notification Reasons

Param
Any additional information about the event.

D.2.27 CSSM_MEMORY_FUNCS/CSSM_API_MEMORY_FUNCS

This structure is used by applications to supply memory functions for the CSSM and the add-in modules. The functions are used when memory needs to be allocated by the CSSM or add-ins for returning data structures to the applications.

```
typedef struct cssm_memory_funcs {
  void *(*malloc_func) (uint32 Size, void *AllocRef);
  void  (*free_func)   (void *MemPtr, void *AllocRef);
  void *(*realloc_func)(void *MemPtr, uint32 Size, void *AllocRef);
  void *(*calloc_func) (uint32 Num, uint32 Size, void *AllocRef);
  void *AllocRef;
} CSSM_MEMORY_FUNCS, *CSSM_MEMORY_FUNCS_PTR;

typedef CSSM_MEMORY_FUNCS CSSM_API_MEMORY_FUNCS;
typedef CSSM_API_MEMORY_FUNCS *CSSM_API_MEMORY_FUNCS_PTR;
```

Definition

malloc_func
Pointer to a function that returns a void pointer to the allocated memory block of at least Size bytes from heap AllocRef.

free_func
Pointer to a function that deallocates a previously-allocated memory block (*MemPtr*) from heap AllocRef.

realloc_func
Pointer to a function that returns a void pointer to the reallocated memory block (*MemPtr*) of at least Size bytes from heap AllocRef.

calloc_func
Pointer to a function that returns a void pointer to an array of Num elements of length Size initialized to zero from heap AllocRef.

AllocRef
Indicates which memory heap the function operates on.

D.2.28 CSSM_SPI_MEMORY_FUNCS

This structure is used by add-in modules to reference an application's memory management functions. The functions are used when an add-in module needs to allocate memory for returning data structures to the application or needs to de-allocate memory for a data structure passed to it from an application.

```
typedef struct cssm_spi_memory_funcs {
   void *(*malloc_func) (CSSM_HANDLE AddInHandle, uint32 Size);
   void  (*free_func)   (CSSM_HANDLE AddInHandle, void *MemPtr);
   void *(*realloc_func)(CSSM_HANDLE AddInHandle, void *MemPtr,
                                                   uint32 Size);
   void *(*calloc_func) (CSSM_HANDLE AddInHandle, uint32 Num,
                                                   uint32 Size);
} CSSM_SPI_MEMORY_FUNCS, *CSSM_SPI_MEMORY_FUNCS_PTR;
```

Definition

malloc_func
Pointer to a function that returns a void pointer to the allocated memory block of at least Size bytes from the heap of the application associated with AddInHandle.

free_func
Pointer to a function that de-allocates a previously-allocated memory block (*MemPtr*) from the heap of the application associated with AddInHandle.

realloc_func
Pointer to a function that returns a void pointer to the reallocated memory block (*MemPtr*) of at least Size bytes from the heap of the application associated with AddInHandle.

calloc_func
Pointer to function that returns a void pointer to an array of Num elements of length Size initialized to zero from the heap of the application associated with AddInHandle.

D.2.29 CSSM_MODULE_FUNCS

This structure is used by add-in modules to pass a table of function pointers for a single service to CSSM.

```
typedef struct cssm_module_funcs {
    CSSM_SERVICE_TYPE ServiceType;
    union cssm_function_table {
        void *ServiceFuncs;
        CSSM_SPI_CSP_FUNCS_PTR CspFuncs;
        CSSM_SPI_DL_FUNCS_PTR  DlFuncs;
        CSSM_SPI_CL_FUNCS_PTR  ClFuncs;
        CSSM_SPI_TP_FUNCS_PTR  TpFuncs;
        } FunctionTable;
    } CSSM_MODULE_FUNCS, *CSSM_MODULE_FUNCS_PTR;
```

Definition

ServiceType
The type of add-in module services accessible via the *XXFuncs* function table.

FunctionTable
A pointer to a function table of the type described by ServiceType. These function pointers are used by CSSM to direct function calls from an application to the appropriate service in the add-in module. These function pointer tables are described in the CSSM header files **<cssmcspi.h>**, **<cssmdli.h>**, **<cssmcli.h>**, and **<cssmtpi.h>**.

Value	Description
CSSM_SPI_CSP_FUNCS_PTR CspFuncs	Functions pointers to CSP services.
CSSM_SPI_DL_FUNCS_PTR DlFuncs	Functions pointers to DL services.
CSSM_SPI_CL_FUNCS_PTR ClFuncs	Functions pointers to CL services.
CSSM_SPI_TP_FUNCS_PTR TpFuncs	Functions pointers to TP services.

Table D-2 Service Access Tables

D.2.30 CSSM_HANDLEINFO

This structure is used by add-in modules to obtain information about a CSSM_HANDLE.

```
typedef struct cssm_handleinfo {
    uint32 SubServiceID;
    uint32 SessionFlags;
    CSSM_NOTIFY_CALLBACK Callback;
    uint32 ApplicationContext;
} CSSM_HANDLEINFO, *CSSM_HANDLEINFO_PTR;
```

Definition

SubServiceID
An identifier for this sub-service.

SessionFlags
Sessions flags set by CSSM during module attach processing.

Callback
A callback function registered by the application as part of the module attach operation. This function should be used to notify the application of certain events.

ApplicationContext
An identifier which should be passed back to the application as part of the Callback function.

D.2.31 CSSM_REGISTRATION_INFO

This structure is used by add-in modules to pass tables of function pointers and module information to CSSM. Note that the function table does not include a pointer to the AddInAuthenticate function. The AddInAuthenticate function is invoked by CSSM prior to the add-in service module presenting CSSM_REGISTRATION_INFO to CSSM by calling the CSSM_RegisterServices function.

```
typedef struct cssm_registration_info {
        /* Loading, Unloading and Event Notifications */
    CSSM_RETURN (CSSMAPI *Initialize) (CSSM_MODULE_HANDLE Handle,
                            uint32 VerMajor,
                            uint32 VerMinor);
    CSSM_RETURN (CSSMAPI *Terminate) (CSSM_MODULE_HANDLE Handle);
    CSSM_RETURN (CSSMAPI *EventNotify)(CSSM_MODULE_HANDLE Handle,
                             const CSSM_EVENT_TYPE Event,
                             const uint32 Param);
    CSSM_MODULE_INFO_PTR (CSSMAPI *GetModuleInfo)
                           (CSSM_MODULE_HANDLE ModuleHandle,
                            CSSM_SERVICE_MASK ServiceMask,
                            uint32 SubserviceID,
                            CSSM_INFO_LEVEL InfoLevel);
    CSSM_RETURN (CSSMAPI *FreeModuleInfo)
                                (CSSM_MODULE_HANDLE ModuleHandle,
                            CSSM_MODULE_INFO_PTR ModuleInfo);
    CSSM_BOOL ThreadSafe;
    uint32 ServiceSummary;
    uint32 NumberOfServiceTables;
    CSSM_MODULE_FUNCS_PTR Services;
} CSSM_REGISTRATION_INFO, *CSSM_REGISTRATION_INFO_PTR;
```

Definition

Initialize
Pointer to function that verifies compatibility of the requested module version with the actual module version and which performs module setup operations.

Terminate
Pointer to function that performs module cleanup operations.

EventNotify
Pointer to function that accepts event notification from CSSM.

GetModuleInfo
Pointer to function that obtains and returns dynamic information about the module.

FreeModuleInfo
Pointer to function that frees the module information structure.

ThreadSafe
A flag which indicates to CSSM whether or not the module is capable of handling multi-threaded access.

ServiceSummary
A bit mask indicating the types of services offered by this module. It is the bitwise-OR of the service types described in Figure 35-2 above.

NumberOfServiceTables
The number of distinct services provided by this module. This is also the length of the *Services* array.

Services
An array of CSSM_MODULE_FUNCS structures which provide the mechanism for accessing the module's services.

D.3 Function Definitions

The manpages for Function Definitions follow on the next page.

NAME

CSSM_ModuleInstall

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleInstall
    (const char *ModuleName,
    const char *ModuleFileName,
    const char *ModulePathName,
    const CSSM_GUID_PTR GUID,
    const CSSM_MODULE_INFO_PTR ModuleDescription,
    const void * Reserved1,
    const CSSM_DATA_PTR Reserved2)
```

DESCRIPTION

This function registers the module with CSSM. CSSM adds the module's descriptive information to its persistent registry. This makes the service module available for use on the local system. The function accepts as input the name and unique identifier for the module, the location executable code for the module, and a digitally signed list of capabilities supported by the module. The capabilities list includes flags defining the module's attach time policy. The module's attach time procedure requirements are defined by its MODULE_FLAGS that control authentication. In addition to the module-declared policy, CSSM always enforces its internal policy requiring integrity authentication for all service modules. CSSM evaluates it policy based on CSSM-selected public root keys as points of trust. The service module policy can require application authentication based on a set of module-selected public root keys as point of trust. A copy of these module-selected keys are included in the CSSM_MODULE_INFO structure. The effective module policy definition must be included in the module's signed credentials. The registry copy is only informational. The installation process records the module name and module info in the CSSM Registry, making the module available for use by applications.

PARAMETERS

ModuleName (input)
The name of the module.

ModuleFileName (input)
The name of the file that implements the module.

ModulePathName (input)
The path to the file that implements the module.

GUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the module.

ModuleDescription (input)
A pointer to the CSSM_MODULE_INFO structure containing a description of the module.

Reserved1 (input)
Reserve data for the function.

Reserved2 (input)
Reserve data for the function.

RETURN VALUE

A CSSM_OK return value signifies that information has been updated. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_REGISTRY_ERROR
Error in the registry.

SEE ALSO

CSSM_ModuleUninstall

NAME

CSSM_ModuleUninstall

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleUninstall
    (const CSSM_GUID_PTR GUID)
```

DESCRIPTION

This function deletes the persistent CSSM internal information about the module, removing it from the name space of available modules in the CSSM system.

PARAMETERS

GUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the module.

RETURN VALUE

A CSSM_OK return value means the module has been successfully uninstalled. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_INVALID_GUID
CSP module was not installed.

CSSM_REGISTRY_ERROR
Unable to delete information.

SEE ALSO

CSSM_ModuleInstall

NAME

CSSM_ModuleAttach

SYNOPSIS

```
CSSM_CSP_HANDLE CSSMAPI CSSM_ModuleAttach
    (const CSSM_GUID_PTR GUID,
    const CSSM_VERSION_PTR Version,
    const CSSM_API_MEMORY_FUNCS_PTR MemoryFuncs,
    uint32 SubserviceID,
    uint32 SubserviceFlags,
    uint32 Application,
    const CSSM_NOTIFY_CALLBACK Notification,
    const char *AppFileName,
    const char *AppPathName,
    const void * Reserved)
```

DESCRIPTION

This function attaches the service provider module and verifies that the version of the module expected by the application is compatible with the version on the system. The module can implement sub-services (as described in the service provider's documentation). The caller can specify a specific sub-service provided by the module. Sub-service flags may be required to set parameters for the service.

PARAMETERS

GUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the CSP module.

Version (input)
The major and minor version number of the service provider module that the application is compatible with.

MemoryFuncs (input)
A structure containing pointers to the memory routines.

SubserviceID (input)
The number of a sub-service provided by the module. This value should always be taken from the CSSM_MODULE_INFO structure to insure that a compatible identifier is used. (Service provider modules that implement only one service can use zero as the sub-service identifier.)

SubserviceFlags (input)
Bitmask of service options defined by a particular sub-service of the module. Legal values are described in module-specific documentation. A default set of flags is specified in the CSSM_MODULE_INFO structure for use by the caller.

Application (input/optional)
Nonce passed to the application when its callback is invoked allowing the application to determine the proper context of operation.

Notification (input/optional)
Callback provided by the application that is used by the add-in module to notify the application of certain events. For example, a CSP may use this callback in the following situations: a parallel operation completes, a token running in serial mode surrenders control to the application or the token is removed (hardware specific).

AppFileName (input/optional)
The name of the file that implements the application (containing its main entry point). This file name is used to locate the application's credentials for purposes of application authentication by CSSM or by CSSM on behalf of the target add-in module. This input must be provided if the target add-in module defines a usage policy that requires authentication of the application's credentials. The add-in module's declared policy is recorded by the MODULE_FLAGS contained in module's MODULE_INFO structure and in the module's signed credentials. If application authentication is not required by the target add-in module, this parameter should be NULL.

AppPathName (input/optional)
The path to the file that implements the application (containing its main entry point). This path name is used to locate the application's credentials for purposes of application authentication by CSSM or by CSSM on behalf of the target add-in module. This input must be provided if the target add-in module defines a usage policy that requires authentication of the application's credentials. The add-in module's declared policy is recorded by the MODULE_FLAGS contained in the module's MODULE_INFO structure and in the module's signed credentials. If application authentication is not required by the target add-in module, this parameter should be NULL.

Reserved (input)
A reserved input.

RETURN VALUE

A handle is returned for the attached service provider module. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_INCOMPATIBLE_VERSION
Incompatible version.

CSSM_EXPIRE
Add-in module has expired.

CSSM_ATTACH_FAIL
Unable to load service provider module.

CSSM_NOT_INITIALIZE
CSSM_Init has not been invoked.

SEE ALSO

CSSM_ModuleDetach

NAME

CSSM_ModuleDetach

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_ModuleDetach
    (CSSM_MODULE_HANDLE ModuleHandle)
```

DESCRIPTION

This function detaches the application from the service provider module.

PARAMETERS

ModuleHandle (input)
The handle that describes the service provider module.

RETURN VALUE

A CSSM_OK return value signifies that the application has been detached from the module. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_ADDIN_HANDLE
Invalid module handle.

SEE ALSO

CSSM_ModuleAttach

NAME

CSSM_GetModuleInfo

SYNOPSIS

```
CSSM_MODULE_INFO_PTR CSSMAPI CSSM_GetModuleInfo
    (const CSSM_GUID_PTR ModuleGUID,
    CSSM_SERVICE_MASK ServiceMask,
    sint32 SubserviceID,
    CSSM_INFO_LEVEL InfoLevel);
```

DESCRIPTION

This function returns descriptive information about the module identified by the GUID. The information returned can include all of the capability information, for each subservices, for each of the service types implemented by the selected module. The request for information can be limited to a particular set of services, as specified by the service bit mask. The request may be further limited to one or all of the sub-services implemented in one or all of the service categories. Finally the detail level of the information returned can be controlled by the InfoLevel input parameter. This is particularly important for module with dynamic capabilities. InfoLevel can be used to request static attribute values only or dynamic values.

PARAMETERS

ModuleGUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the service provider module.

ServiceMask (input)
A bit mask specifying the module service types used to restrict the capabilities information returned by this function. An input value of zero specifies all services for the specified module.

SubserviceID (input)
A single sub-service ID or the value CSSM_ALL_SUBSERVICES must be provided. If a sub-service ID is provided the get operation is limited to the specified sub-service. Note that the operation may already be limited by a service mask. If so, the sub-service ID applies to all service categories selected by the service mask. If CSSM_ALL_SUBSERVICES is specified, information for all sub-services (as limited by the service mask) are returned by this function.

InfoLevel (input)
Indicates the level of detail returned by this function. Information retrieval can be restricted as follows:

- CSSM_INFO_LEVEL_MODULE—returns only the information contained in the CSSM_SERVICE_INFOstructure.
- CSSM_INFO_LEVEL_SUBSERVICE—returns the information returned by CSSM_INFO_LEVEL_MODULE and the information contained in the XXsubservice structure, where XX corresponds to the module type, such as tpsubservice, clsubservice, dlsubservice, cpsubservice.
- CSSM_INFO_LEVEL_STATIC_ATTR—returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically defined for the module.
- CSSM_INFO_LEVEL_ALL_ATTR—returns the information returned by CSSM_INFO_LEVEL_SUBSERVICE and the attribute and capability values that are statically or dynamically defined for the module. Dynamic modules, whose capabilities

change over time, support a query function used by CSSM to interrogate the module's current capability status.

RETURN VALUE

A pointer to a module info structure containing a pointer to an array of zero or more service information structures. Each structure contains type information identifying the service description as representing certificate library services, data storage library services, and so on. The service descriptions are sub-classed into sub-service descriptions which describe the attributes and capabilities of a sub-service.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_INVALID_USAGE_MASK
Invalid bit mask.

CSSM_INVALID_SUBSERVICEID
Invalid sub-service ID.

CSSM_INVALID_INFO_LEVEL
Invalid info level indicator.

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_INVALID_GUID
Unknown GUID.

CSSM_NOT_INITIALIZE
CSSM_Init has not been invoked.

CSSM_MEMORY_ERROR
Internal Memory Error.

CSSM_REGISTRY_ERROR
A registry error occurred.

SEE ALSO

CSSM_SetModuleInfo, *CSSM_FreeModuleInfo*

NAME

CSSM_SetModuleInfo

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SetModuleInfo
    (const CSSM_GUID_PTR ModuleGUID,
    const CSSM_MODULE_INFO_PTR ModuleInfo);
```

DESCRIPTION

This function replaces all of the currently registered descriptive information about the module identified by the ModuleGUID with the newly specified information. The operation is a total replacement of all information for all service categories and all subservices.

If the caller wishes to retain any of the information registered prior to execution of this call, the caller must use the CSSM_GetModuleInfo function to retrieve the current information, update their private copy, and then use the CSSM_SetModuleInfo function to place the updated copy back into the CSSM registry.

This function should be used to incrementally update descriptive information that is unspecified at installation time.

PARAMETERS

ModuleGUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the service provider module.

ModuleInfo (input)
A pointer to the complete structured set of descriptive information about the module.

RETURN VALUE

A CSSM_RETURN value indicating pass or fail. CSSM_OK indicates success, otherwise use CSSM_GetError to determine the type of error that has occurred.

ERRORS

CSSM_INVALID_GUID
Unknown GUID.

CSSM_INVALID_MODULE_INFO
Invalid module info structure.

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_NOT_INITIALIZE
CSSM_Init has not been invoked.

CSSM_REGISTRY_ERROR
Registry error.

CSSM_INVALID_POINTER
Invalid input pointer.

SEE ALSO

CSSM_GetModuleInfo, *CSSM_FreeModuleInfo*

NAME

CSSM_FreeModuleInfo

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_FreeModuleInfo
    (CSSM_MODULE_INFO_PTR ModuleInfo)
```

DESCRIPTION

This function frees the memory allocated to hold all of the info structures returned by CSSM_GetModuleInfo. All sub-structures within the info structure are freed by this function.

PARAMETERS

ModuleInfo (input)
A pointer to the CSSM_MODULE_INFO structures to be freed.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_NOT_INITIALIZE
CSSM_Init has not been invoked.

CSSM_INVALID_POINTER
Invalid input pointer.

SEE ALSO

CSSM_GetModuleInfo, *CSSM_SetModuleInfo*

NAME

CSSM_RegisterServices

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_RegisterServices
    (const CSSM_GUID_PTR GUID,
    const CSSM_REGISTRATION_INFO_PTR FunctionTable,
    CSSM_SPI_MEMORY_FUNCS_PTR UpcallTable,
    void *Reserved)
```

DESCRIPTION

This function is used by an add-in module to register its function table with CSSM and to receive a memory management upcall table from CSSM.

PARAMETERS

GUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for the calling module.

FunctionTable (input)
A structure containing pointers to the interface functions implemented by this module, organized by interface type.

UpcallTable (output)
A pointer to the CSSM_SPI_MEMORY_FUNCS structure containing the memory management function pointers to be used by this module.

Reserved (input)
A reserved input.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_GUID
Invalid GUID.

CSSM_REGISTER_SERVICES_FAIL
Unable to register services.

CSSM_INVALID_POINTER
Invalid input pointer.

CSSM_MEMORY_ERROR
Internal Memory Error.

SEE ALSO

CSSM_DeregisterServices

NAME

CSSM_DeregisterServices

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_DeregisterServices
    (const CSSM_GUID_PTR GUID)
```

DESCRIPTION

This function is used by an add-in module to de-register its function table with CSSM.

PARAMETERS

GUID (input)
A pointer to the CSSM_GUID structure containing the global unique identifier for this module.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_GUID
Invalid GUID.

CSSM_DEREGISTER_SERVICES_FAIL
Unable to deregister services.

SEE ALSO

CSSM_RegisterServices

NAME

CSSM_GetHandleInfo

SYNOPSIS

```
CSSM_HANDLEINFO_PTR CSSMAPI CSSM_GetHandleInfo
    (CSSM_HANDLE ModuleHandle)
```

DESCRIPTION

Returns a structure that can contain a callback function and session flags provided by the application in association with the specified module handle.

PARAMETERS

ModuleHandle (input)
Handle of the module for which information should be returned.

RETURN VALUE

A pointer to the CSSM_HANDLEINFO structure containing information registered by the application for use by the add-in module. If the pointer is NULL, an error has occurred; use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_MODULE_HANDLE
Invalid add-in handle.

NAME

CSSM_GetError

SYNOPSIS

```
CSSM_ERROR_PTR CSSMAPI CSSM_GetError
    (void)
```

DESCRIPTION

This function returns the current error information.

PARAMETERS

None.

RETURN VALUE

Returns the current error information. If there is currently no valid error, the error number will be CSSM_OK. A NULL pointer indicates that the CSSM_InitError was not called by the CSSM Core or that a call to CSSM_DestroyError has been made by the CSSM Core. No error information is available.

SEE ALSO

CSSM_ClearError, *CSSM_SetError*

NAME

CSSM_SetError

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSSM_SetError
    (CSSM_GUID_PTR guid,
    uint32 error_number)
```

DESCRIPTION

This function sets the current error information to error_number and guid.

PARAMETERS

guid (input)
Pointer to the GUID (global unique ID) of the add-in module.

error_number (input)
An error number. It should fall within one of the valid CSSM, CL, TP, DL, or CSP error ranges.

RETURN VALUE

CSSM_OK if error was successfully set. A return value of CSSM_FAIL indicates that the error number passed is not within a valid range, the GUID passed is invalid, CSSM_InitError was not called by the CSSM Core, or CSSM_DestroyError has been called by the CSSM Core. No error information is available.

SEE ALSO

CSSM_ClearError, *CSSM_GetError*

NAME

CSSM_ClearError

SYNOPSIS

```
void CSSMAPI CSSM_ClearError
    (void)
```

DESCRIPTION

This function sets the current error value to CSSM_OK. This can be called if the current error value has been handled and therefore is no longer a valid error.

PARAMETERS

None.

SEE ALSO

CSSM_SetError, CSSM_GetError

CAE Specification

Part 8:
CDSA Mechanisms for Policy Compliance

The Open Group

Chapter 37

Introduction

The Common Data Security Architecture (CDSA) was defined to supply security services to applications in the widest possible range of computing platforms and application domains. When implemented and deployed in real-world, commercial environments, CDSA must support system-wide policy-based control over:

- The security services available on the platform
- Individual use of offered security services

Two major categories of security services are cryptographic operations and certificate creation and manipulation.

System-wide policies governing availability and use of these services can be defined by:

- A user
- A system administrator
- A site-wide administrator
- The global-enterprise administrator
- A government entity

For example, a site-wide administrator can require that the privacy mode (requiring encryption and decryption) in a communication service (such as Secure Sockets Layer) can only be used after 5:00 PM on Monday through Friday. This policy defines the extent of security services generally available to applications. Either the service is available to everyone or it is available to no one.

An example of controlling individual use of an available service is a government policy stating that financial applications can perform encryption and decryption with a key size greater than 56 effective bits.

CDSA defines a global, integrity-based policy for all CSSM systems. This policy is distinct from all locally defined system-wide policies. The CDSA integrity policy mandates the use of bilateral authentication when attaching add-in service modules and offers this option to applications. A complete description of this global policy, the mechanisms, and the interfaces used to implement it are described in the documents *Common Data Security Architecture (CDSA) Specification*, *CSSM Application Programming Interface*, and *CSSM Add-in Module Structure and Administration Specification*.

Policies are also defined by Trust Policy Modules (TPM) and add-in service modules. TPMs define and enforce policies over an application-specific domain. Enforcement is based on certificate verification. Add-in service modules define usage policies based on module capabilities presented in the module’s signed manifest.

These four sources form a hierarchy of policy definition applied in the following order:

- CSSM global-integrity policy
- Local, system-wide usage policy
- Add-in service module usage policy
- Domain-specific application action policy

The policy applied to a particular application request is the ordered evaluation of these policy definitions.

CDSA, as defined in the *Common Data Security Architecture (CDSA) Specification*, defines when and how three of these policy definitions are evaluated. The Common Security Services Manager (CSSM), which is the core of CDSA, can be enhanced to support the specification and application of a local, system-wide policy, controlling the offering and use of security services.

Use of system-wide policy statements is not required and the CSSM mechanisms to support them are optional. Vendors can choose to provide these mechanisms in their products. These mechanisms can assist in making their products more full-featured and attractive to enterprise customers. It may also assist these product vendors to achieve compliance with import, export, or use restrictions imposed by a relevant government entity.

37.1 Overview of CDSA

CDSA defines an interoperable, extensible architecture in which applications can selectively and dynamically access security services. Figure 37-1 shows the three basic layers of the Common Data Security Architecture:

- System Security Services
- The Common Security Services Manager (CSSM)
- Security Add-in Modules (cryptographic service providers, trust policy modules, certificate library modules, and data storage library modules)

CDSA is intended to be the multi-platform security architecture that's horizontally broad and vertically robust.

The Common Security Services Manager (CSSM) is the core of CDSA. CSSM manages categories of security services and multiple discrete implementations of those services as add-in security modules. CSSM:

- Defines the application programming interface for accessing security services
- Defines the service providers interface for security service modules
- Dynamically extends the security services available to an application

Applications request security services through the CSSM security API or via layered security services and tools implemented over the CSSM API. The requested security services are performed by add-in security modules. Four basic types of module managers are defined:

- Cryptographic Services Manager
- Trust Policy Services Manager
- Certificate Library Services Manager
- Data Storage Library Services Manager

Over time, new categories of security services will be defined, and new module managers will be required. CSSM supports elective module managers that dynamically extend the system with new categories of security services.

Below CSSM are add-in security modules that perform cryptographic operations, manipulate certificates, and manage application-domain-specific trust policies. Add-in security modules may be provided by independent software and hardware vendors as competitive products. Applications use CSSM to direct their requests to modules from specific vendors or to any module that performs the required services. Add-in modules augment the set of available security services.

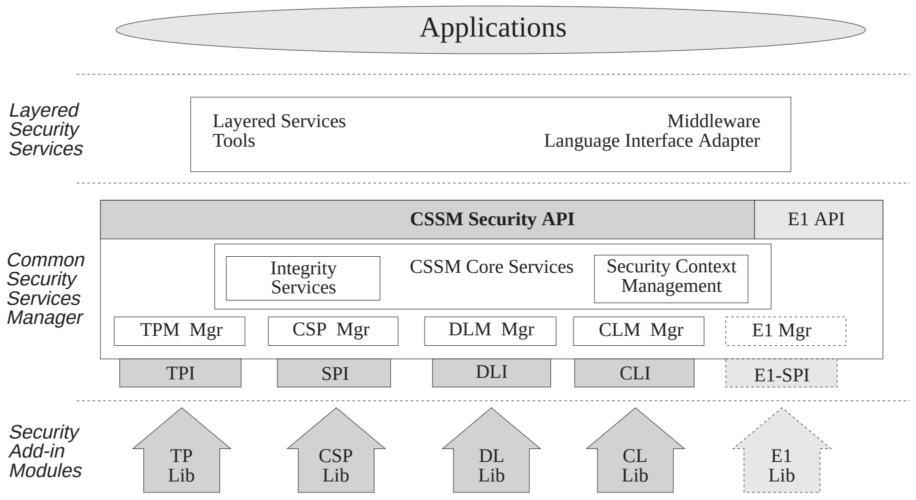

Figure 37-1 Common Data Security Architecture for all Platforms

Chapter 38

Goals and General Approach

The basic goal is to enhance CDSA with transparent support for system-wide, policy-based control of security services in a flexible and extensible manner. This means CSSM cannot hard-wire policy-specific mechanisms into the framework.

Even in the case of stable, long term policies, policy definition, interpretation, and enforcement can require complex procedures. In response, new mechanisms are continually under development to address these complex policy requirements.

38.1 Goals

The goals for an enhanced CDSA include:

- Support for a broad range of system-wide policies on the use of security services—different organizations will define distinctly different policies.
- Support for a broad range of mechanisms to evaluate and enforce system-wide policies—complex policy definitions can require more complex evaluation mechanisms to determine compliance. New mechanisms are continually under development and CSSM must be able to incorporate these new mechanisms.
- Create an interoperable business market for competing products for policy compliance mechanisms—if a sufficient number of new mechanisms are designed, a product market could emerge for these products. CSSM must be able to incorporate those products.
- Support changing policies—even stable, long term policy definitions evolve over time and are subject to reinterpretation.

38.2 Requirements

These CDSA goals generate requirements for enhanced CSSM mechanisms to perform the following services:

- Verify that each security service request is authorized according to the system-wide policy (this can include dependencies among CSSM add-in module service providers).
- Verify the correct (permitted) operation of a security service module (if required).
- Ensure that it is reasonably difficult to remove or alter the CSSM policy evaluation and enforcement mechanisms.
- Ensure that it is reasonably difficult to modify or delete the system-wide policy definition.
- Delay binding the system-wide policy to the runtime environment.
- Override/change an active policy.
- Support multiple, concurrently-active policies.
- Support a hierarchical or weighted relationship among concurrently-active policies.
- Provide plug-able policy mechanisms.
- Use a trusted component to determine trusted policy compliance.

38.3 Specifying a System-Wide Policy

Policies are stated as a set of restrictions on the use of security services. The restrictions are defined in terms of the attributes of the service being restricted. The primary attribute categories for security services are as follows:

- Service Representation—what is being restricted with respect to the service, its implementation, technical knowledge about it, or technical assistance with it
- Restriction Type—does the restriction apply to individual use of the service or to general availability
- Service User—is the service being requested by a special application (for example, system software performing authentication to make the platform more secure)
- Service Features—is the service generally available, but selected features of the service are restricted
- Service Strength—is the service weak or strong (for example, is the cryptographic cipher 56-bits or less; is the certificate management service capable of Certification Authority operations).

Corporations distinguish service representations in product licensing and the United States government has detailed definitions of the representation of cryptography. In a broad definition, an implementation is hardware or software that provides the security operation. Technical knowledge is the schematics or source code for the implementation, and technical assistance is the personal assistance given to another so that person can create an implementation. These do not constitute legal definitions but serve as a guideline to understanding these differences.

Various government entities may consider a cryptographic framework, such as CSSM, to be an implementation of cryptographic services. This may make CSSM subject to the same restrictions as a general purpose cryptographic library. CSSM is best described as "crypto with a hole"; software that provides a common, programmable interface for cryptographic operations where cryptography is added at a later time. While CSSM does not actually implement cryptographic operations, the enhanced CSSM mechanisms for system-wide policy control of security services may facilitate in complying with these government-defined policies.

Policies governing the use of security services can be defined in terms of any combination of the five aspects listed earlier. Every installation can run distinct system-wide policies. Clearly the CSSM-provided policy compliance mechanism(s) must be flexible, configurable, and relatively trustworthy.

38.4 Assumptions and Architectural Approach

The enhanced CDSA design assumes:

- Existence of a manufacturing infrastructure for components of CDSA (including elective module managers, add-in security service modules, layered application services, and applications)
- Add-in modules can declare the security services they can and will provide under specified conditions
- CSSM can evaluate and enforce policies without defining policies
- Complex policies require more complex mechanisms of evaluation and enforcement

Three policy enforcement mechanisms consistent with CDSA are shown in Figure 38-1.

These mechanisms include:

- Authentication checks on attaching add-in service modules and elective module managers
- Screened access to security service modules based on a system-wide policy (if specified)
- Use of existing CSSM extensibility mechanisms to add complex policy checks or services that enable the caller to be compliant with the policy

CSSM is uniquely positioned architecturally to provide these services, as it:

- Dynamically attaches add-in security service modules upon application request
- Manages the dispatching of application function calls for security services to appropriate add-in modules

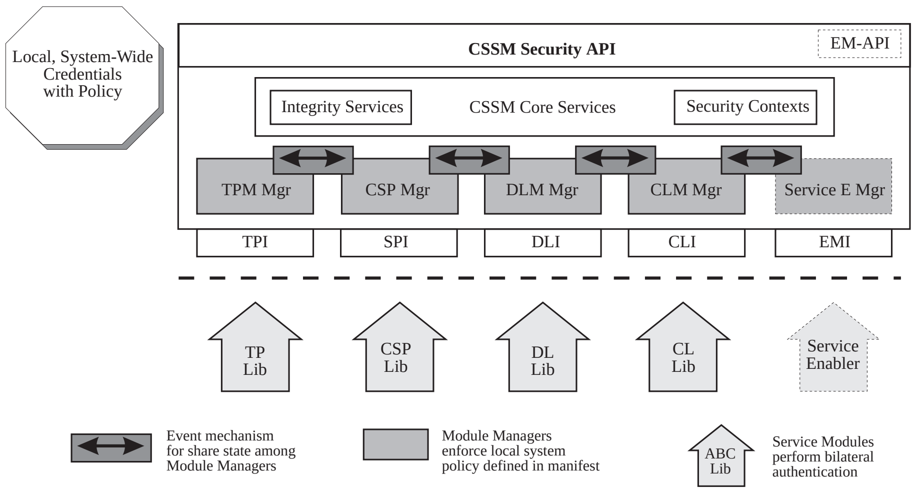
Local, System-Wide Credentials with Policy
CSSM Security API
EM-API
Integrity Services
CSSM Core Services
Security Contexts
TPM Mgr
CSP Mgr
DLM Mgr
CLM Mgr
Service E Mgr
TPI
SPI
DLI
CLI
EMI
TP Lib
CSP Lib
DL Lib
CL Lib
Service Enabler
Event mechanism for share state among Module Managers
Module Managers enforce local system policy defined in manifest
ABC Lib
Service Modules perform bilateral authentication

Figure 38-1 Enhanced Common Data Security Architecture

Chapter 39

CSSM Integrity Services—The Foundation

The fundamental CSSM mechanism supporting general, policy-based control of service offerings and service usage is authentication. Authentication is performed by a three step verification process:

- Verification of credentials for each dynamic component in CDSA
- Verification of manifests describing the capabilities of each add-in security service module
- Verification of signatures over the dynamic component's object code

The interfaces for these services are described in detail in the *CSSM Embedded Integrity Services Library API Spec* and is summarized here.

CSSM uses this mechanism to authenticate dynamic components that attach to CSSM.

CSSM Integrity Services can verify the identity and the integrity of each component that attaches to the CSSM. Identity verification of an add-in module is based on an X.509 certificate chain. Integrity verification of an add-in module is based on a sequence of signature verifications covering signed object code files and signed manifests, describing a module's capabilities.

A complete set of credentials must be created for each add-in security service module as part of the module manufacturing process. A full set of credentials includes:

- A certificate, which is part of a chain of X.509 certificates
- A set of digitally-signed code files, which contain the executables for a module
- A digitally signed manifest, which records the capabilities of the module
- A signature file, which records all of the signatures on the object files and manifest

39.1 A Module′s Certificate Chain

The certificate chain is constructed as follows:

1. A "root" certificate, owned by a CSSM vendor is used to sign a module manufacturer's certificate. The manufacturer's certificate identifies the manufacturer as a licensed vendor who has agreed to comply with all specified licensing conditions.
2. The manufacturer's certificate is used to sign the specific module's certificate. This is the manufacturer's certification of the product and assurance that the distribution and execution of the product will comply with all applicable export, import, and use restrictions. The root certificate owner is not responsible for the behavior of the manufacturer's product.

39.2 Checking a Module's Credentials

The certified module presents its complete credentials (certificate, manifest, and object code files) to CSSM during the installation process. CSSM verifies the credentials and if they are valid, the installation process is completed. It is of the utmost importance that the object code files and the manifest be signed using the private key associated with the module's certificate. This tightly binds the identity in the certificate with "what the module is" (in this case, the object code files themselves), and with "what the module claims it is" (in this case, the capability descriptions in the manifest).

When attaching a module, CSSM retrieves the module's credentials, verifies them and executes a bilateral authentication procedure with the attaching module. CSSM has the equivalent credentials which can be verified by the attaching module. If the bilateral verification is successful, the attach is completed. CSSM integrity services must embed a mechanism for validating module or application certificates. This mechanism verifies the certificate signature chain starting with the root public-key that is stored within CSSM. The removal or alteration of the public root key or the signature verification mechanism itself is deemed to be at least as hard as re-implementation of the entire CSSM infrastructure.

Applications can also be issued credentials during their manufacturing process. These credentials can certify that the application is exempt from a class of policy controls, can list required security services, and can identify the specific service modules required to perform those services.

Chapter 40

Defining the Local, System-Wide Policy

When CSSM is installed on a system, a local, system-wide policy, controlling the use of security services through CSSM, can be defined and installed with CSSM. Defining and installing a policy is optional.

If a system-wide policy is presented, it is represented in a set of credentials. These credentials include a digital certificate chain and a manifest. The certificate identifies the authorized local system administrator, and the manifest describes for each CSSM-defined category of security service the global restrictions on that category of service. The manifest contains one section for each type of CSSM security service supported by the local system. The section contains the CSSM_MODULE_INFO structure for each selected category of service. An additional manifest section can be added.

System-wide policy credentials are created by a manufacturing process. An enhanced CSSM that supports the definition of local policy must provide a policy signing certificate and signing tool with the CSSM system. The signing tool can be a complete manufacturing tool or the subset required to sign certificates and manifests. (Object code modules are not signed by this process.) The three policy credential files created by this process are stored in the file system directory with the CSSM credential files during CSSM installation.

If policy credentials are present at CSSM startup, the general CSSM authentication checking mechanism can be used to authenticate the source and definition of a local, system-wide policy credentials. The certificate chain must verify based on the CSSM-defined roots of trust and the manifest must be signed by the policy certificate. If verified, CSSM can use the policy manifest as the specification of a local system-wide policy.

Chapter 41

Screening Requests Based on Simple Policies

Given a verified system-wide policy definition, a policy enforcer must screen application requests for security services. The policy enforcer simply accepts or rejects each request based on the policy defined in the manifest. In the layered CDSA architecture, there are four candidates to perform policy enforcement:

- The application itself
- The security service module targeted to perform the service
- The CSSM
- An add-in service module that performs policy evaluation

To screen its own security service requests, an application must have a priori knowledge of the system-wide policy, runtime knowledge of the execution environment, and a willingness to follow the rules. Embedding the policy in the application makes the system-wide policy static. This approach also raises a concern about consistency of policy interpretation and enforcement when each application performs this task. It is often counter-productive for applications to screen /control their own security service request stream.

Each add-in security service module could screen the application requests it receives. This leads to the same problems and concerns encountered with applications screening their own requests. It is also a burden that CSSM should be able to remove from the module vendor community.

The remaining two options, CSSM and special add-in modules that perform policy evaluation, can be used in combination or alone to screen application requests according to a system-wide policy.

41.1 Simple Policies

CSSM can provide screening for simple policies. A policy is deemed simple if all of the following hold:

- It can be evaluated in a single atomic execution of an evaluation function.
- The input required for evaluation of the policy is localized and available when the evaluation must be made.
- The screening mechanism is transparent to the application (except for rejected service requests).

41.2 CSSM Mechanisms Supporting Simple Policies

When CSSM is installed on a system, it can receive a verifiable description of a system-wide policy specification. Three existing CSSM mechanisms are enhanced to support enforcement of that system-wide policy:

- Module installation check—the basic install time verification procedure is extended to include a comparison of the module's basic capabilities with the system-specified restrictions. This determines whether the module's capabilities are valid under current system constraints. If the module is found to be unacceptable then module installation is aborted.
- Module attach check—the basic attach time verification procedure is extended to include a comparison of the module's basic capabilities with the system-specified restrictions. This determines whether the module's capabilities can validly be attached to the CSSM framework, under current global policy constraints. If the module is found to be unacceptable then module attach is aborted. This is the same test that was performed at module installation. It is re-evaluated at module attach because the governing system-wide policy can change between the time of module install and module attach.
- Security service invocation—checking mechanisms are required to determine the validity of function calls.

CSSM enforces simple system-wide policies by screening function calls against:

- The signed system-wide policy description
- The signed capabilities description of the target security service module

This mechanism is:

- Transparent to the calling application—the application does not make additional calls to obtain pre-approval for their requests.
- Policy-neutral—it does not embed any specific policy, but can dynamically check different policies as they are installed. The permitted operations are specified by the administrator defining the system policy and the module vendor specifying the add-in module's capabilities. The CSSM mechanism is "table-driven".

Chapter 42

Screening Requests Based on Complex Policies

Not all policies can be served by the simple CSSM screening mechanisms described in the previous section. Complex policy definitions represent a challenge to clever systems designers. In response, these designers are building more complex protocols and mechanisms to provide applications with a broader range of security services while still complying with stated policies.

42.1 Complex Policies

A policy is deemed complex if policy conformance and evaluation requires any of the following:

- Evaluation of a sequence of state transitions to determine whether the security service request is permitted
- Additional, explicit API calls by the application, to establish required pre-conditions for performing policy controlled operations

Elective service modules and CSSM support for module manager communications can be used to support evaluation of this type of policy statement.

42.2 Evaluation of a Sequence of Events

When a policy definition requires checking a sequence of application operations, state must be maintained in or by the module managers of CSSM. Using information sharing, as described in the *Common Data Security Architecture (CDSA) Specification*, module managers can work together to maintain information on an application's sequence of requests. These same information-sharing mechanisms are used by elective module managers and basic module managers alike. This design approach allows a module manager to screen application requests by accumulating the required state information and evaluating compliance when the request is made, as if all of the required status information were simply available now, rather than having been collected over a period of time.

42.3 Services that Establish Pre-Conditions

Using the elective module manager features of CDSA, it is possible to define a new category of security service for mechanisms whose service it is to establish all pre-conditions required to use some other security service. This type of service is called a Service Enabler. Key Recovery is an example of a service enabler.

Some governmental entities are considering requiring the implementation and use of certain key recovery schemes as a pre-condition for granting an export, import, or use permit for certain encryption-based products. Private business entities may also use key-recovery schemes to ensure that their enterprise can recover confidential information important to the enterprise's operation. Key encapsulation and key escrow are two mechanisms that implement this new category of service.

As an elective module manager within CSSM, the Key Recovery Module Manager (KRMM) defines an API for use by applications. Applications must make explicit calls to the key recovery API to establish the pre-conditions required to perform strong encryption within the constraints of the policy. The CSSM Key Recovery APIs are specified in the *CSSM Key Recovery API*

Specification. Users requiring these services should consult that specification.

Applications establish the conditions required for policy compliance by making explicit calls to the service-enabling APIs. To verify that the required state has been achieved (that is, to determine that the appropriate service-enabling functions have been invoked in the proper order), appropriate module managers must share state information about the sequence of operations requested by the application. In the example of key recovery, the KRMM and the Cryptographic module manager must share state information about whether the application has enabled key recovery for a key that will be used to encrypt a communication message.

In summary, the enhanced services provided by CSSM to support system-wide policy compliance include:

- Enhanced manifest to include capability descriptions for security service modules
- Integrity checks on the capability descriptions in a manifest
- Capability screening at module installation and module attach time
- Elective module managers whose category of service is service-enablement

CAE Specification

Part 9:
CSSM Cryptographic Service Provider Interface

The Open Group

Chapter 43

Introduction

43.1 CDSA Add-In Module Overview

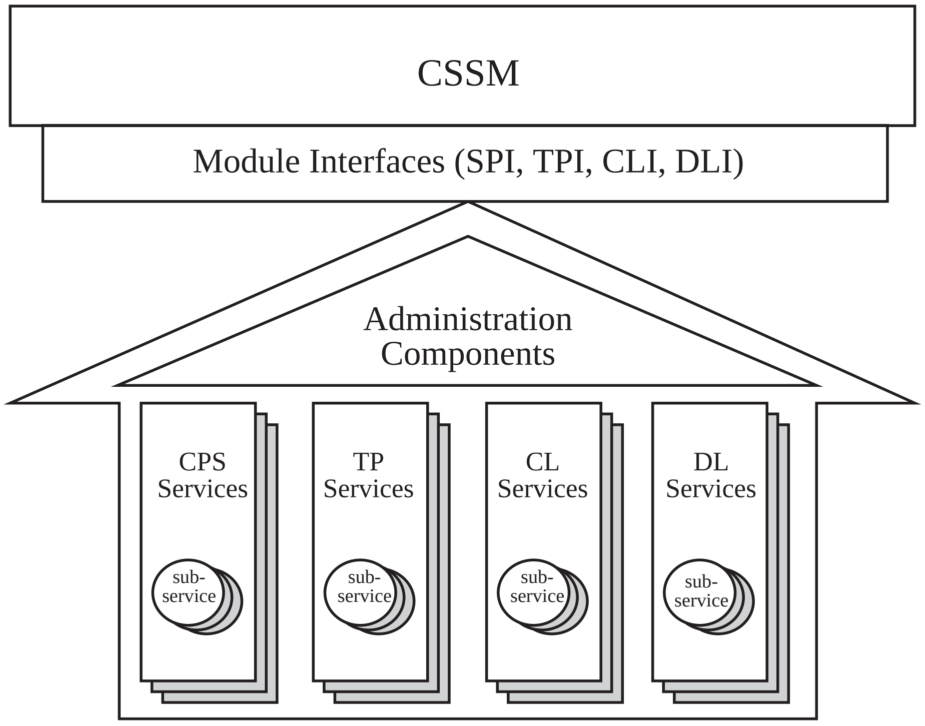

Figure 43-1 CDSA Add-In Module Structure

A CDSA add-in module is a dynamically-linkable library, composed of functions that implement some or all of the CSSM Module Interfaces. Add-in module functionality is partitioned into two areas:

- The provision of security services to applications
- Module administration

Add-in modules provide one or more categories of security services to applications. The service categories are Cryptographic Service Provider (CSP) services, Trust Policy (TP) services, Certificate Library (CL) services, and Data Storage Library (DL) services. Each security service contains one or more implementation instances, called sub-services. For a CSP service providing access to hardware tokens, a sub-service would represent a slot. For a DL service provider, a sub-service would represent a type of persistent storage. These sub-services each support the module interface for their respective service categories. This documentation-part describes the module interface functions in the CSP service category. More information about DL services can be found in the *CSSM Data Storage Library Interface Specification*. More information about TP services can be found in the *CSSM Trust Policy Interface Specification*. More information about CL services can be found in the *CSSM Certificate Library Interface Specification*.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions which allow CSSM to indicate events such as module attach and detach. In addition, as part of the attach operation, every module must be able to verify its own integrity, verify the integrity of CSSM, and register with CSSM. Detailed information about add-in module structure, administration, and interfaces can be found in the *CSSM Add-in Module Structure and Administration Specification*.

43.2 Cryptographic Service Provider Overview

The CSSM infrastructure does not implement any cryptography. It has been termed "crypto with a hole." The Cryptographic Services Manager provides applications with access to cryptographic functions that are implemented by Cryptographic Service Provider (CSP) modules. This achieves the objective of centralizing all the cryptography into exchangeable modules.

The Cryptographic Services Manager defines two categories of services:

- Module management—installation, feature registration, and query of CSP features
- Selection, initialization, and use of cryptographic operations, which are implemented by a CSP

The nature of the cryptographic functions contained in any particular CSP depends on what task the CSP was designed to perform. For example, a VISA* smart card* would be able to digitally sign credit card transactions on behalf of the card's owner, whereas a digital employee badge would be able to authenticate a user for physical or electronic access.

A CSP can perform one or more of these cryptographic functions:

- Bulk encryption
- Digital signature
- Cryptographic hash
- Key generation
- Random number generation

The Cryptographic Services Manager doesn't assume any particular form factor for a CSP. Indeed, CSPs can be instantiated in hardware, software or both. Operationally, the distinction must be transparent. The two visible distinctions between hardware and software implementations are the degree of trust the application receives by using a given CSP, and the cost of developing that CSP. A hardware implementation should be more tamper-resistant than a software implementation. Hence a higher level of trust is achieved by the application.

Software CSPs are the default and are portable in that they can be carried as an executable file. Additionally, the modules that implement a CSP must be digitally signed (to authenticate their origin and integrity), and they should be made as tamper-resistant as possible. This requirement extends to software implementations and hardware. Multiple CSPs may be loaded and active within the CSSM at any time. A single application may use multiple CSPs concurrently. Interpreting the resulting level of trust and security is the responsibility of the application or the trust-policy module used by the application.

A small (yet significant) number of CSPs existed prior to the definition of CSSM Cryptographic API. These legacy CSPs have defined their own API for cryptographic services. These interfaces are CSP-specific, non-standard, and in general low-level, key-based interfaces. Low-level, key-based interfaces present a considerable development effort to the application developer attempting to secure an application by using those services.

The Cryptographic Services Manager defines a high-level, certificate-based API for cryptographic services to better support application development. In consideration of legacy and divergent CSPs, the Cryptographic Services Manager defines a lower-level Service Provider Interface (SPI) that more closely resembles typical CSP APIs, and provides CSP developers with a single interface to support. A CSP may or may not support multithreaded applications.

Acknowledging legacy CSPs, the CSSM architecture defines an optional adaptation layer between the Cryptographic Services Manager and a CSP. The adaptation layer allows the CSP vendor to implement a shim to map the CSSM SPI to the CSP's existing API, and to implement any additional management functions that are required for the CSP to function as an add-in module in the extensible CSSM architecture. New CSPs may support the CSSM SPI directly (without the aid of an adaptation layer).

Chapter 44

Service Provider Interface

44.1 Overview

Cryptographic Service Providers (CSPs) are add-in modules which perform cryptographic operations including encryption, decryption, digital signaturing, key and key pair generation, random number generation, message digest, key wrapping, key unwrapping, and key exchange. Cryptographic services can be implemented by a hardware-software combination or by software only. Besides the traditional cryptographic functions, CSPs may provide other vendor-specific services. The set of services provided can be dynamic even after the CSP has been attached for service by a caller. This means the capabilities registered when the CSP was installed can change during execution, based on changes internal or external to the system.

The CSP is always responsible for the secure storage of private keys. Optionally the CSP may assume responsibility for the secure storage of other object types, such as symmetric keys and certificates. The implementation of secured persistent storage for keys can use the services of a Data Storage Library module within the CSSM framework or some approach internal to the CSP. Accessing persistent objects managed by the CSP, other than keys, is performed using CSSM's Data Storage Library APIs.

CSPs optionally support a password-based login sequence. When login is supported, the caller is allowed to change passwords as deemed necessary. This is part of a standard user-initiated maintenance procedure. Some CSPs support operations for privileged, CSP administrators. The model for CSP administration varies widely among CSP implementations. For this reason, CSSM does not define APIs for vendor-specific CSP administration operations. CSP vendors can makes these services available to CSP administration tools using the CSSM_Passthrough function.

The range and types of cryptographic services a CSP supports is at the discretion of the vendor. A registry and query mechanism is available through the CSSM for CSPs to disclose the services and details about the services. As an example, a CSP may register with the CSSM: Encryption is supported, the algorithms present are DES with cipher block chaining for key sizes 40 and 56 bits, triple DES with 3 keys for key size 168 bits.

All cryptographic services requested by applications will be channeled to one of the CSPs via the CSSM. CSP vendors only need target their modules to CSSM for all security-conscious applications to have access to their product.

Calls made to a Cryptographic Service Provider (CSP) to perform cryptographic operations occur within a framework called a *session*, which is established and terminated by the application. The *session context* (simply referred to as the *context* is created prior to starting CSP operations and is deleted as soon as possible upon completion of the operation. Context information is not persistent; it is not saved permanently in a file or database.

Before an application calls a CSP to perform a cryptographic operation, the application uses the query services function to determine what CSPs are installed, and what services they provide. Based on this information, the application then can determine which CSP to use for subsequent operations; the application creates a session with this CSP and performs the operation.

Depending on the class of cryptographic operations, individualized attributes are available for the cryptographic context. Besides specifying an algorithm when creating the context, the application may also initialize a session key, pass an initialization vector and/or pass padding information to complete the description of the session. A successful return value from the create function indicates the desired CSP is available. Functions are also provided to manage the created context.

When a context is no longer required, the application calls CSSM_DeleteContext. Resources that were allocated for that context can be reclaimed by the operating system.

Cryptographic operations come in two types—a single call to perform an operation and a staged method of performing the operation. For the single call method, only one call is needed to obtain the result. For the staged method, there is an initialization call followed by one or more update calls, and ending with a completion (final) call. The result is available after the final function completes its execution for most crypto operations—staged encryption/decryption are an exception in that each update call generates a portion of the result.

44.1.1 Cryptographic Operations

CSSM_RETURN CSP_SignData

CSSM_RETURN CSP_SignDataInit

CSSM_RETURN CSP_SignDataUpdate

CSSM_RETURN CSP_SignDataFinal
Accepts as input a handle to a cryptographic context describing the sign operation and the data to operate on. The result of the completed sign operation is returned in a CSSM_DATA structure.

CSSM_BOOL CSP_VerifyData

CSSM_RETURN CSP_VerifyDataInit

CSSM_RETURN CSP_VerifyDataUpdate

CSSM_BOOL CSP_VerifyDataFinal
Accepts as input a handle to a cryptographic context describing the verify operation and the data to operate on. The result of the completed verify operation is a CSSM_TRUE or CSSM_FALSE.

CSSM_RETURN CSP_DigestData

CSSM_RETURN CSP_DigestDataInit

CSSM_RETURN CSP_DigestDataUpdate

CSSM_RETURN CSP_DigestDataFinal
Accepts as input a handle to a cryptographic context describing the digest operation and the data to operate on. The result of the completed digest operation is returned in a CSSM_DATA structure.

CSSM_CC_HANDLE CSP_DigestDataClone
Accepts as input a handle to a cryptographic context describing the digest operation. A handle to another cryptographic context is created with similar information and intermediate result as described by the first context.

CSSM_RETURN CSP_GenerateMac

CSSM_RETURN CSP_GenerateMacInit

CSSM_RETURN CSP_GenerateMacUpdate

CSSM_RETURN CSP_GenerateMacFinal
Accepts as input a handle to a cryptographic context describing the MAC operation and the

data to operate on. The result of the completed MAC operation is returned in a CSSM_DATA structure.

CSSM_RETURN CSP_VerifyMac

CSSM_RETURN CSP_VerifyMacInit

CSSM_RETURN CSP_VerifyMacUpdate

CSSM_RETURN CSP_VerifyMacFinal
Accepts as input a handle to a cryptographic context describing the MAC operation and the data to operate on. The result of the completed verify operation is a CSSM_RETURN value.

CSSM_RETURN CSP_QuerySize
Accepts as input a handle to a cryptographic context describing the encryption or decryption operation, and an array of input block sizes. This function the output block sizes corresponding to the input sizes for the specified algorithm.

CSSM_RETURN CSP_EncryptData

CSSM_RETURN CSP_EncryptDataInit

CSSM_RETURN CSP_EncryptDataUpdate

CSSM_RETURN CSP_EncryptDataFinal
Accepts as input a handle to a cryptographic context describing the encryption operation and the data to operate on. The encrypted data is returned in CSSM_DATA structures.

CSSM_RETURN CSP_DecryptData

CSSM_RETURN CSP_DecryptDataInit

CSSM_RETURN CSP_DecryptDataUpdate

CSSM_RETURN CSP_DecryptDataFinal
Accepts as input a handle to a cryptographic context describing the decryption operation and the data to operate on. The decrypted data is returned in CSSM_DATA structures.

CSSM_RETURN CSP_GenerateKey
Accepts as input a handle to a cryptographic context describing the generate key operation and attributes of the new key. The key is returned in a CSSM_KEY structure.

CSSM_RETURN CSP_GenerateKeyPair
Accepts as input a handle to a cryptographic context describing the generate key operation and attributes of each key in the new keypair. The keys are returned in CSSM_KEY structures.

CSSM_RETURN CSP_GenerateRandom
Accepts as input a handle to a cryptographic context describing the generate random operation. The random data is returned in a CSSM_DATA structure.

CSSM_RETURN CSP_WrapKey
Accepts as input a handle to a symmetric/asymmetric cryptographic context describing the wrap key operation and the wrapping key to be used in the operation, the key to be wrapped, and a passphrase (if required by the CSP) that permits access to the private key to be wrapped.

CSSM_RETURN CSP_UnwrapKey
Accepts as input a handle to a cryptographic context describing the key unwrap operation, the wrapped key to be unwrapped, and a passphrase (if required by the CSP) that will be used to control access to the private key that will be unwrapped.

CSSM_RETURN CSP_DeriveKey

Accepts as input a handle to a cryptographic context describing the derive key operation and the base key that will be used to derive new keys.

CSSM_RETURN CSP_GenerateAlgorithmParams

Accepts as input a handle to a cryptographic context describing an algorithm and returns a set of algorithm parameters appropriate for that algorithm.

CSSM_RETURN CSP_QueryKeySizeInBits

Accepts as input a handle to a cryptographic context and the key. This function returns a pointer to a data structure containing the keysize and effective keysize in bits.

44.1.2 Cryptographic Sessions and Logon

CSSM_RETURN CSP_Login

Accepts as input a login password and a flag indicating the persistent or non-persistent status of keys and other objects created during the login session. CSPs are not required to support a login model. If a login model is supported, the CSP may request additional passwords at any time during the period of service.

CSSM_RETURN CSP_Logout

The caller is logged out of the current login session with the designated CSP.

CSSM_RETURN CSP_ChangeLoginPassword

Accepts as input a handle to a CSP, the caller's old login password for that CSP, and the caller's new login password. The old password is replaced with the new password. The caller's current login is terminated and another login session is created using the new password.

44.1.3 Extensibility Functions

CSSM_RETURN CSP_PassThrough

This performs the CSP module-specific function indicated by the operation ID. The operation ID specifies an operation which the CSP has exported for use by an application or module. Such operations should be specific to the key format of the private keys stored in the CSP module.

44.1.4 Key Formats for Public Key-Based Algorithms

To ensure interoperability among cryptographic service providers and portability for application developers, CSSM must mandate standard key formats for public key based cryptographic algorithms. Standard key formats have not been defined for many of the algorithms identified by CSSM because these algorithms are not yet in wide spread use. For those algorithms in wide spread use, CDSA adopts existing standard formats or defines a format when no standard exists.

The two PKI-based algorithms with wide spread usage are:

- RSA-based algorithms
- DSA-based algorithms

For RSA-based algorithms, CDSA adopts the PKCS#1 standard for key representation.

For DSA-based algorithms, no organization has published a standard and no *de facto* standard seems to exists. CDSA defines a standard representation for DSA key based on the DSA algorithm definitions in the FIPS 186 and FIPS 186a standards. Complete documentation on these standards can be found at *http://csrc.ncsl.nist.gov/fips/fips186.txt* and *http://csrc.ncsl.nist.gov/fips/fips186a.txt* respectively.

A DSA public key is represented as a BER-encoding of a sequence list containing:

```
PrimeModulus;    /* p */
PrimeDivisor;    /* q */
OrderQ;          /* g */
PublicKey;       /* y */
```

A DSA private key is represented as a BER-encoded sequence list containing:

```
PrimeModulus;    /* p */
PrimeDivisor;    /* q */
OrderQ;          /* g */
PrivateKey;      /* x */
```

These key components are defined by FIPS 186 and FIPS 186a as follows:

p = a prime modulus, where $2^{L-1} < p < 2^{L}$ for $512 \le L \le 1024$ and L is a multiple of 64.

PrimeModulus This is the public prime modulus.

q = a prime divisor of p-1, where $2^{159} < q < 2^{160}$

PrimeDivisor Another public prime number dividing (p-1).

g = $h^{(p-1)/q}$ mod p, where h is any integer with 1 < h < p-1 such that $h^{(p-1)/q}$ mod p > 1.

OrderQ This public number has order q mod p.

x = a pseudo-randomly generated integer with 0 < x < q.

PrivateKey The private key.

y = g^{x} mod p.

PublicKey The public key.

44.2 Data Structures

This section describes the data structures which may be passed to or returned from a CSP function. They will be used by applications to prepare data to be passed as input parameters into CSSM API function calls, that will be passed without modification to the appropriate CSP. The CSP is then responsible for interpreting them and returning the appropriate data structure to the calling application via CSSM. These data structures are defined in the header file **<cssmtype.h>** distributed with CSSM.

44.2.1 CSSM_CSP_HANDLE

The CSSM_CSP_HANDLE is used to identify the association between an application thread and an instance of a CSP module. It is assigned when an application causes CSSM to attach to a CSP. It is freed when an application causes CSSM to detach from a CSP. The application uses the CSSM_CSP_HANDLE with every CSP function call to identify the targeted CSP. The CSP uses the CSSM_CSP_HANDLE to identify the appropriate application’s memory management routines when allocating memory on the application’s behalf.

```
typedef uint32 CSSM_CSP_HANDLE
                    /* Cryptographic Service Provider Handle */
```

44.2.2 CSSM_DATA

The CSSM_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory. This memory must be allocated and freed using the memory management routines provided by the calling application via CSSM.

```
typedef struct cssm_data{
    uint32 Length;  /* in bytes */
    uint8 *Data;
} CSSM_DATA, *CSSM_DATA_PTR
```

Definition

Length
Length of the data buffer in bytes.

Data
Points to the start of an arbitrary length data buffer.

44.2.3 CSSM_CRYPTO_DATA

```
typedef struct cssm_crypto_data {
    CSSM_DATA_PTR Param;
    CSSM_CALLBACK Callback;
    uint32 CallbackID;
}CSSM_CRYPTO_DATA, *CSSM_CRYPTO_DATA_PTR
```

Definition

Param
A pointer to the parameter data and its size in bytes.

Callback
An optional callback routine for the add-in modules to obtain the parameter.

CallbackID
A tag that identifies the callback.

44.2.4 CSSM_DATE

```
typedef struct cssm_date {
    uint8 Year[4];
    uint8 Month[2];
    uint8 Day[2];
} CSSM_DATE, *CSSM_DATE_PTR;
```

Definition

Year
Four digit ASCII representation of the year.

Month
Two digit ASCII representation of the month.

Day
Two digit ASCII representation of the day.

44.2.5 CSSM_RANGE

```
typedef struct cssm_range {
    uint32 Min;          /* inclusive minimum value */
    uint32 Max;          /* inclusive maximum value */
} CSSM_RANGE, *CSSM_RANGE_PTR;
```

Definition

Min
Minimum value in the range.

Max
Maximum value in the range.

44.2.6 CSSM_QUERY_SIZE_DATA

```
typedef struct cssm_query_size_data {
    uint32 SizeInputBlock;  /* Input data block size */
    uint32 SizeOutputBlock; /* Output data block size */
} CSSM_QUERY_SIZE_DATA, *CSSM_QUERY_SIZE_DATA_PTR;
```

Definition

SizeInputBlock
Size of the data block to be input for processing.

SizeOutputBlock
Size of the data block that results from processing.

44.2.7 CSSM_HEADERVERSION

```
typedef uint32 CSSM_HEADERVERSION;
#define CSSM_KEYHEADER_VERSION (2)
```

Definition

Represents the version number of a key header structure. This version number is an integer that increments with each format revision. The current revision number is represented by the defined constant CSSM_KEYHEADER_VERSION.

44.2.8 CSSM_KEY_SIZE

This structure holds the key size and the effective key size for a given key. The metric used is bits. The number of effective bits is the number of key bits that can be used in a cryptographic operation compared with the number of bits that may be present in the key. When the number of effective bits is less than the number of actual bits, this is known as "dumbing down".

```
typedef struct cssm_key_size {
  uint32 KeySizeInBits;          /* Key size in bits */
  uint32 EffectiveKeySizeInBits; /* Effective key size in bits */
} CSSM_KEYSIZE, *CSSM_KEYSIZE_PTR;
```

Definition

KeySizeInBits
The actual number of bits in a key.

EffectiveKeySizeInBits
The number of key bits that can be used for cryptographic operations.

44.2.9 CSSM_KEYHEADER

The key header contains meta-data about a key. It contains the GUID of the CSP that owns the data. Attributes of the key are defined by the CSP and the application when the key is created. Most of these attributes describe both the CSP-stored copy of the key and the application's local copy of the key or the key reference. A subset of the attributes describe only the application-resident copy of the key or the key reference. A table at the end of this section summarizes the scope of each key header attribute.

```
typedef struct cssm_keyheader {
    CSSM_HEADERVERSION HeaderVersion;  /* Key header version */
    CSSM_GUID CspId;          /* GUID of CSP generating the key */
    uint32 BlobType;          /* See BlobType #define's */
    uint32 Format;            /* Raw or Reference format */
    uint32 AlgorithmId;       /* Algorithm ID of key */
    uint32 KeyClass;          /* Public/Private/Secret, etc. */
    uint32 EffectiveKeySizeInBits; /* Size of logical
                                   key/modulus/prime in bits */
    uint32 KeyAttr;           /* Attribute flags */
    uint32 KeyUsage;          /* Key use flags */
    CSSM_DATE StartDate;      /* Effective date of key */
    CSSM_DATE EndDate;        /* Expiration date of key */
    uint32 WrapAlgorithmId;   /* == CSSM_ALGID_NONE if clear key */
    uint32 WrapMode;          /* if alg supports multiple wrapping
                                     modes */
    uint32 Reserved;
} CSSM_KEYHEADER, *CSSM_KEYHEADER_PTR;
```

Definition

HeaderVersion
This is the version of the keyheader structure. The current version is represented by the defined constant CSSM_KEYHEADER_VERSION.

CspId
If known, the GUID of the CSP that generated the key. This value will not be known if a key is received from a third party, or extracted from a certificate.

BlobType
Describes the basic format of the key data. It can be any one of the following values:

Keyblob Type Identifier	**Description**
CSSM_KEYBLOB_RAW	The blob is a clear, raw key
CSSM_KEYBLOB_RAW_BERDER	The blob is a clear key, DER encoded
CSSM_KEYBLOB_REFERENCE	The blob is a reference to a key
CSSM_KEYBLOB_WRAPPED	The blob is a wrapped RAW key
CSSM_KEYBLOB_WRAPPED_BERDER	The blob is a wrapped DER encoded key
CSSM_KEYBLOB_OTHER	Other, CSP defined

Table 44-1 Keyblob Type Identifiers

Format

Describes the detailed format of the key data based on the value of the BlobType field. If the blob type has a non-reference basic type, then a CSSM_KEYBLOB_RAW_FORMAT identifier must be used, otherwise a CSSM_KEYBLOB_REF_FORMAT identifier is used. Any of the following values are valid as format identifiers.

Keyblob Format Identifier	**Description**
CSSM_KEYBLOB_RAW_FORMAT_NONE	Raw format is unknown
CSSM_KEYBLOB_RAW_FORMAT_PKCS1	RSA PKCS1 V1.5 See "RSA Encryption Standard", an RSA Laboratories publication *http://www.rsa.com/rsalabs/pubs/PKCS/*
CSSM_KEYBLOB_RAW_FORMAT_PKCS3	RSA PKCS3 V1.5 See"Diffie-Hellman Key-Agreement Standard", an RSA Laboratories publication *http://www.rsa.com/rsalabs/pubs/PKCS/*
CSSM_KEYBLOB_RAW_FORMAT_MSCAPI	Microsoft CAPI V2.0
CSSM_KEYBLOB_RAW_FORMAT_PGP	PGP See "PGP Cryptographic Software Development Kit (PGP sdk)", a PGP Publication
CSSM_KEYBLOB_RAW_FORMAT_FIPS186	US Gov. FIPS 186: DSS V
CSSM_KEYBLOB_RAW_FORMAT_BSAFE	RSA Bsafe V3.0 See "BSAFE, A Cryptographic Toolkit, Library Reference Manual", an RSA Data Security Inc. publication
CSSM_KEYBLOB_RAW_FORMAT_PKCS8	RSA PKCS8 V1.2 See "Private-Key Information Syntax Standard", an RSA Laboratories publication *http://www.rsa.com/rsalabs/pubs/PKCS/"*
CSSM_KEYBLOB_RAW_FORMAT_PKCS11	RSA PKCS11 V2.0 See "Cryptographic Token Interface Standard", an RSA Laboratories publication *http://www.rsa.com/rsalabs/pubs/PKCS/*

CSSM_KEYBLOB_RAW_FORMAT_CDSA	CDSA format See this specification and *CSSM Cryptographic Service Provider Interface Specification*
CSSM_KEYBLOB_RAW_FORMAT_OTHER	Other, CSP defined
CSSM_KEYBLOB_REF_FORMAT_INTEGER	Reference is a number or handle
CSSM_KEYBLOB_REF_FORMAT_STRING	Reference is a string or name
CSSM_KEYBLOB_REF_FORMAT_OTHER	Reference is a CSP-defined format

Table 44-2 Keyblob Format Identifiers

AlgorithmId
The algorithm for which the key was generated. This value does not change when the key is wrapped. Any of the defined CSSM algorithm IDs may be used.

KeyClass
Class of key contained in the key blob. Valid key classes are as follows:

Key Class Identifier	**Description**
CSSM_KEYCLASS_PUBLIC_KEY	Key is a public key
CSSM_KEYCLASS_PRIVATE_KEY	Key is a private key
CSSM_KEYCLASS_SESSION_KEY	Key is a session or symmetric key
CSSM_KEYCLASS_SECRET_PART	Key is part of secret key
CSSM_KEYCLASS_OTHER	Other

Table 44-3 Key Class Identifiers

EffectiveKeySizeInBits
This is the logical size of the key in bits. The logical size is the value referred to when describing the length of the key. For instance, an RSA key would be described by the size of its modulus and a DSA key would be represented by the size of its prime. Symmetric key sizes describe the actual number of bits in the key. For example, DES keys would be 64 bits and an RC4 key could range from 1 to 128 bits.

KeyAttr
Attributes of the key represented by the data. These attributes are used by CSPs and applications to convey information about stored or referenced keys. Some of the attribute values are used only as input or output values for CSP functions, can appear in a keyheader, and some can be used only by the CSP. The attributes are represented by a bitmask. The attribute name, its description, and its usage constraints are summarized in the following:

Attribute values valid only as inputs to functions and will never appear in a key header:	
Attribute	**Description**
CSSM_KEYATTR_RETURN_DEFAULT	Key is returned in CSP's default form.
CSSM_KEYATTR_RETURN_DATA	Key is returned with key bits present. The format of the returned key can be raw or wrapped.
CSSM_KEYATTR_RETURN_REF	Key is returned as a reference.
CSSM_KEYATTR_RETURN_NONE	Key is not returned.
Attribute values valid as inputs to functions and retained values in a key header:	
Attribute	**Description**
CSSM_KEYATTR_PERMANENT	Key is stored persistently in the CSP, such asa PKCS11 token object.
CSSM_KEYATTR_PRIVATE	Key is a private object and protected by either a user login, a password, or both.
CSSM_KEYATTR_MODIFIABLE	The key or its attributes can be modified.
CSSM_KEYATTR_SENSITIVE	Key is sensitive. It may only be extracted from the CSP in a wrapped state.
CSSM_KEYATTR_EXTRACTABLE	Key is extractable from the CSP. If this bit is not set, either the key is not stored in the CSP, or it cannot be extracted under any circumstances.
Attribute values valid in a key header when set by a CSP:	
Attribute	**Description**
CSSM_KEYATTR_ALWAYS_SENSITIVE	Key has always been sensitive.
CSSM_KEYATTR_NEVER_EXTRACTABLE	Key has never been extractable.

KeyUsage

A bitmask representing the valid uses of the key. Any of the following values are valid:

Usage Mask	**Description**
CSSM_KEYUSE_ANY	Key may be used for any purpose supported by the algorithm.
CSSM_KEYUSE_ENCRYPT	Key may be used for encryption.
CSSM_KEYUSE_DECRYPT	Key may be used for decryption.
CSSM_KEYUSE_SIGN	Key can be used to generate signatures. For symmetric keys this represents the ability to generate MACs.
CSSM_KEYUSE_VERIFY	Key can be used to verify signatures. For symmetric keys this represents the ability to verify MACs.
CSSM_KEYUSE_SIGN_RECOVER	Key can be used to perform signatures with message recovery. This form of a signature is generated using the CSSM_EncryptData API with the

	algorithm mode set to CSSM_ALGMODE_PRIVATE_KEY
CSSM_KEYUSE_VERIFY_RECOVER	Key can be used to verify signatures with message recovery. This form of a signature verified using the CSSM_DecryptData API with the algorithm mode set to CSSM_ALGMODE_PUBLIC_KEY.
CSSM_KEYUSE_WRAP	Key can be used to wrap another key.
CSSM_KEYUSE_UNWRAP	Key can be used to unwrap a key.
CSSM_KEYUSE_DERIVE	Key can be used as the source for deriving other keys.

Table 44-4 Key Usage Flags

StartDate
Date from which the corresponding key is valid. All fields of the CSSM_DATA structure are set to zero if the date is unspecified or unknown.

EndDate
Data that the key expires and can no longer be used. All fields of the CSSM_DATA structure are set to zero is the date is unspecified or unknown.

WrapAlgorithmId
If the key data contains a wrapped key, this field contains the algorithm used to create the wrapped blob. This field will be set to CSSM_ALGID_NONE if the key is not wrapped.

WrapMode
If the wrapping algorithm supports multiple wrapping modes, this field contains the mode used to wrap the key. This field is ignored if the WrapAlgorithmId is CSSM_ALGID_NONE.

Reserved
This field is reserved for future use. It should always be set to zero.

The scope of the key header attributes is summarized as follows:

Attribute Name	**Pertains to the Application's local copy of the key**	**Pertains to the CSP-stored copy of the key**
BlobType	X	
Format	X	
AlgorithmId	X	X
KeyClass	X	X
EffectiveKeySizeInBits	X	X
KeyAttr	Only the flag bits RETURN_XXX	All the flag bits except RETURN_XXX
KeyUsage	X	X
StartDate	X	X
EndDate	X	X
WrapAlgorithmId	X	
WrapMode	X	

44.2.10 CSSM_KEY

This structure is used to represent keys in CSSM.

```
typedef struct cssm_key    {
    CSSM_KEYHEADER KeyHeader;     /* Fixed length key header */
    CSSM_DATA KeyData;            /* Variable length key data */
} CSSM_KEY, *CSSM_KEY_PTR;
```

Definition

KeyHeader
Header describing the key.

KeyData
Data representation of the key.

44.2.11 CSSM_WRAP_KEY

This type is used to reference keys that are known to be in wrapped form.

```
typedef CSSM_KEY CSSM_WRAP_KEY, *CSSM_WRAP_KEY_PTR;
```

44.2.12 CSSM_CALLBACK

```
typedef CSSM_DATA_PTR (CSSMAPI *CSSM_CALLBACK)
                                    (void *allocRef, uint32 ID);
```

Definition

allocRef
Memory heap reference specifying which heap to use for memory allocation.

ID
Input data to identify the callback.

44.2.13 CSSM_CSP_TYPE

```
typedef enum cssm_csptype {
    CSSM_CSP_SOFTWARE = 1,
    CSSM_CSP_HARDWARE = CSSM_CSP_SOFTWARE+1,
    CSSM_CSP_HYBRID = CSSM_CSP_SOFTWARE+2,
}CSSM_CSPTYPE;
```

44.2.14 CSSM_CSP_SESSION_TYPE

A session flag is a valid input parameter to the CSSM_ModuleAttach function to declare the type of session requested by the caller.

```
#define CSSM_CSP_SESSION_EXCLUSIVE  0x0001
   /* single user CSP */
#define CSSM_CSP_SESSION_READWRITE  0x0002
   /* caller can read and write objects such as keys in the CSP */
#define CSSM_CSP_SESSION_SERIAL  0x0004
   /* multi-user, re-entrant CSP that requires serial access */
```

44.2.15 CSSM_NOTIFY_CALLBACK

An application uses this data type to request that an add-in module call back into the application to notify it of certain events.

```
typedef CSSM_RETURN (CSSMAPI *CSSM_NOTIFY_CALLBACK)
    (CSSM_CSP_HANDLE ModuleHandle,
      uint32 Application,
      uint32 Reason,
      void* Param)
```

Definition

ModuleHandle
Handle of the add-in to which the notification applies.

Application
Application specific context indicator. This value is specified when an add-in module is attached.

Reason
One of the values specified below.

```
#define CSSM_NOTIFY_SURRENDER  0
#define CSSM_NOTIFY_COMPLETE   1
#define CSSM_NOTIFY_DEVICE_REMOVED  2
#define CSSM_NOTIFY_DEVICE_INSERTED  3
```

Param
Used by the add-in that triggers the notification to pass relevant information about the notification to the application. This parameter will contain the cryptographic context handle for the CSSM_NOTIFY_SURRENDER / COMPLETE types and zero for the CSSM_NOTIFY_DEVICE_REMOVED / INSERTED types.

44.2.16 CSSM_HANDLEINFO

```
typedef struct cssm_handleinfo {
    uint32 SlotID;
    uint32 SessionFlags;
    CSSM_NOTIFY_CALLBACK Callback;
    uint32 ApplicationContext;
} CSSM_HANDLEINFO, *CSSM_HANDLEINFO_PTR;
```

44.2.17 CSSM_PADDING

Enumerates the padding options that can be provided by a CSP.

```
typedef enum cssm_padding {
    CSSM_PADDING_NONE = 0,
    CSSM_PADDING_CUSTOM = CSSM_PADDING_NONE+1,
    CSSM_PADDING_ZERO = CSSM_PADDING_NONE+2,
    CSSM_PADDING_ONE  = CSSM_PADDING_NONE+3,
    CSSM_PADDING_ALTERNATE = CSSM_PADDING_NONE+4,
    CSSM_PADDING_FF  = CSSM_PADDING_NONE+5,
    CSSM_PADDING_PKCS5 = CSSM_PADDING_NONE+6,
    CSSM_PADDING_PKCS7 = CSSM_PADDING_NONE+7,
    CSSM_PADDING_CipherStealing = CSSM_PADDING_NONE+8,
    CSSM_PADDING_RANDOM = CSSM_PADDING_NONE+9,
} CSSM_PADDING;
```

44.2.18 CSSM_CONTEXT_ATTRIBUTE

```
typedef struct cssm_context_attribute{
    uint32 AttributeType;
    uint32 AttributeLength;
    union cssm_context_attribute_value {
        char *String;
        uint32 Uint32;
        CSSM_CRYPTO_DATA_PTR Crypto;
        CSSM_KEY_PTR Key;
        CSSM_DATA_PTR Data;
        CSSM_DATE_PTR Date;
        CSSM_RANGE_PTR Range;
        CSSM_VERSION_PTR Version;
    } Attribute;
} CSSM_CONTEXT_ATTRIBUTE, *CSSM_CONTEXT_ATTRIBUTE_PTR;
```

Definition

AttributeType

An identifier describing the type of attribute. Valid attribute types are as follows:

Value	**Description**	**Data Type**
CSSM_ATTRIBUTE_NONE	No attribute	None
CSSM_ATTRIBUTE_CUSTOM	Custom data	Opaque pointer
CSSM_ATTRIBUTE_DESCRIPTION	Description of attribute	String
CSSM_ATTRIBUTE_KEY	Key Data	CSSM_KEY
CSSM_ATTRIBUTE_INIT_VECTOR	Initialization vector	CSSM_DATA
CSSM_ATTRIBUTE_SALT	Salt	CSSM_DATA
CSSM_ATTRIBUTE_PADDING	Padding information	CSSM_PADDING
CSSM_ATTRIBUTE_RANDOM	Random data	CSSM_DATA
CSSM_ATTRIBUTE_SEED	Seed	CSSM_CRYPTO_DATA
CSSM_ATTRIBUTE_PASSPHRASE	Pass phrase	CSSM_CRYPTO_DATA
CSSM_ATTRIBUTE_KEY_LENGTH	Key length specified in bits	uint32
CSSM_ATTRIBUTE_KEY_LENGTH_RANGE	Key length range specified in bits	CSSM_RANGE
CSSM_ATTRIBUTE_BLOCK_SIZE	Block size	uint32
CSSM_ATTRIBUTE_OUTPUT_SIZE	Output size	uint32
CSSM_ATTRIBUTE_ROUNDS	Number of runs or rounds	uint32
CSSM_ATTRIBUTE_IV_SIZE	Size of initialization vector	uint32
CSSM_ATTRIBUTE_ALG_PARAMS	Algorithm parameters	CSSM_DATA
CSSM_ATTRIBUTE_LABEL	Label placed on an object when it is created	CSSM_DATA
CSSM_ATTRIBUTE_KEY_TYPE	Type of key to generate or derive	uint32
CSSM_ATTRIBUTE_MODE	Algorithm mode to use for encryption	uint32
CSSM_ATTRIBUTE_EFFECTIVE_BITS	Number of effective bits used in the RC2	uint32

	cipher	
CSSM_ATTRIBUTE_START_DATE	Starting date for an object's validity	CSSM_DATE
CSSM_ATTRIBUTE_END_DATE	Ending date for an object's validity	CSSM_DATE
CSSM_ATTRIBUTE_KEYUSAGE	Usage restriction on the key	uint32
CSSM_ATTRIBUTE_KEYATTR	Key attribute	uint32
CSSM_ATTRIBUTE_VERSION	Version number	CSSM_VERSION
CSSM_ATTRIBUTE_PRIME	Prime value	CSSM_DATA
CSSM_ATTRIBUTE_BASE	Base Value	CSSM_DATA
CSSM_ATTRIBUTE_SUBPRIME	Subprime Value	CSSM_DATA
CSSM_ATTRIBUTE_ALG_ID	Algorithm identifier	uint32
CSSM_ATTRIBUTE_ITERATION_COUNT	Algorithm iterations	uint32
CSSM_ATTRIBUTE_ROUNDS_RANGE	Range of number of rounds possible	CSSM_RANGE

Table 44-5 Attribute Types

The data referenced by a CSSM_ATTRIBUTE_CUSTOM attribute must be a single continuous memory block. This allows the CSSM to appropriately release all dynamically allocated memory resources.

AttributeLength
Length of the attribute data.

Attribute
Union representing the attribute data. The union member used is named after the type of data contained in the attribute. See the attribute types table for the data types associated with each attribute type.

44.2.19 CSSM_CONTEXT

```
typedef uint32 CSSM_CC_HANDLE  /* Cryptographic Context Handle */

typedef struct cssm_context {
    uint32 ContextType;        /* context type */
    uint32 AlgorithmType;      /* algorithm type of context */
    uint32 Reserve;            /* reserved for future use */
    uint32 NumberOfAttributes; /* number of attributes associated
                                     with context */
    CSSM_CONTEXT_ATTRIBUTE_PTR ContextAttributes;  /* pointer to
                                     attributes */
} CSSM_CONTEXT, *CSSM_CONTEXT_PTR;
```

Definition

ContextType
An identifier describing the type of services for this context.

Value	Description
CSSM_ALGCLASS_NONE	Null Context type
CSSM_ALGCLASS_CUSTOM	Custom Algorithms
CSSM_ALGCLASS_KEYEXCH	Key Exchange Algorithms
CSSM_ALGCLASS_SIGNATURE	Signature Algorithms
CSSM_ALGCLASS_SYMMETRIC	Symmetric Encryption Algorithms
CSSM_ALGCLASS_DIGEST	Message Digest Algorithms
CSSM_ALGCLASS_RANDOMGEN	Random Number Generation Algorithms
CSSM_ALGCLASS_UNIQUEGEN	Unique ID Generation Algorithms
CSSM_ALGCLASS_MAC	Message Authentication Code Algorithms
CSSM_ALGCLASS_ASYMMETRIC	Asymmetric Encryption Algorithms
CSSM_ALGCLASS_KEYGEN	Key Generation Algorithms
CSSM_ALGCLASS_DERIVEKEY	Key Derivation Algorithms

Table 44-6 Context Types

AlgorithmType
An ID number describing the algorithm to be used.

Value	**Description**
CSSM_ALGID_NONE	Null algorithm
CSSM_ALGID_CUSTOM	Custom algorithm
CSSM_ALGID_DH	Diffie Hellman key exchange algorithm
CSSM_ALGID_PH	Pohlig Hellman key exchange algorithm
CSSM_ALGID_KEA	Key Exchange Algorithm
CSSM_ALGID_MD2	MD2 hash algorithm
CSSM_ALGID_MD4	MD4 hash algorithm
CSSM_ALGID_MD5	MD5 hash algorithm
CSSM_ALGID_SHA1	Secure Hash Algorithm
CSSM_ALGID_NHASH	N-Hash algorithm
CSSM_ALGID_HAVAL	HAVAL hash algorithm (MD5 variant)
CSSM_ALGID_RIPEMD	RIPE-MD hash algorithm (MD4 variant developed for the European Community's RIPE project)
CSSM_ALGID_IBCHASH	IBC-Hash (keyed hash algorithm or MAC)
CSSM_ALGID_RIPEMAC	RIPE-MAC
CSSM_ALGID_HASHwithHitachi	Hitachi hash algorithm
CSSM_ALGID_DES	Data Encryption Standard block cipher
CSSM_ALGID_DESX	DESX block cipher (DES variant from RSA)
CSSM_ALGID_RDES	RDES block cipher (DES variant)
CSSM_ALGID_3DES_3KEY	Triple-DES block cipher (with 3 keys)
CSSM_ALGID_3DES_2KEY	Triple-DES block cipher (with 2 keys)
CSSM_ALGID_3DES_1KEY	Triple-DES block cipher (with 1 key)
CSSM_ALGID_IDEA	IDEA block cipher
CSSM_ALGID_RC2	RC2 block cipher
CSSM_ALGID_RC5	RC5 block cipher
CSSM_ALGID_RC4	RC4 stream cipher
CSSM_ALGID_SEAL	SEAL stream cipher
CSSM_ALGID_CAST	CAST block cipher
CSSM_ALGID_BLOWFISH	BLOWFISH block cipher
CSSM_ALGID_SKIPJACK	Skipjack block cipher
CSSM_ALGID_LUCIFER	Lucifer block cipher
CSSM_ALGID_MADRYGA	Madryga block cipher
CSSM_ALGID_FEAL	FEAL block cipher
CSSM_ALGID_REDOC	REDOC 2 block cipher
CSSM_ALGID_REDOC3	REDOC 3 block cipher
CSSM_ALGID_LOKI	LOKI block cipher
CSSM_ALGID_KHUFU	KHUFU block cipher
CSSM_ALGID_KHAFRE	KHAFRE block cipher
CSSM_ALGID_MMB	MMB block cipher (IDEA variant)

CSSM_ALGID_GOST	GOST block cipher
CSSM_ALGID_SAFER	SAFER K-40, K-64, K-128 block cipher
CSSM_ALGID_CRAB	CRAB block cipher
CSSM_ALGID_MULTI2	MULTI2 block cipher algorithm (MULTI variant from Hitachi)
CSSM_ALGID_RSA	RSA public key cipher
CSSM_ALGID_CIPHERwithHitachiECCS	Hitachi's public key cipher algorithm with Elliptic Curve Cryptosystems
CSSM_ALGID_DSA	Digital Signature Algorithm
CSSM_ALGID_MD5WithRSA	MD5/RSA signature algorithm
CSSM_ALGID_MD2WithRSA	MD2/RSA signature algorithm
CSSM_ALGID_SIGwithHitachiECCS	Hitachi's signature algorithm with Elliptic Curve Cryptosystems
CSSM_ALGID_ElGamal	ElGamal signature algorithm
CSSM_ALGID_MD2Random	MD2-based random numbers
CSSM_ALGID_MD5Random	MD5-based random numbers
CSSM_ALGID_SHARandom	SHA-based random numbers
CSSM_ALGID_DESRandom	DES-based random numbers
CSSM_ALGID_MULTI2Random	MULTI2-based random numbers
CSSM_ALGID_SHA1WithRSA	SHA-1/RSA signature algorithm
CSSM_ALGID_RSA_PKCS	RSA as specified in PKCS #1
CSSM_ALGID_RSA_ISO9796	RSA as specified in ISO 9796
CSSM_ALGID_RSA_RAW	Raw RSA as assumed in X.509
CSSM_ALGID_CDMF	CDMF block cipher
CSSM_ALGID_CAST3	Entrust's CAST3 block cipher
CSSM_ALGID_CAST5	Entrust's CAST5 block cipher
CSSM_ALGID_GenericSecret	Generic secret operations
CSSM_ALGID_ConcatBaseAndKey	Concatenate two keys, base key first
CSSM_ALGID_ConcatKeyAndBase	Concatenate two keys, base key last
CSSM_ALGID_ConcatBaseAndData	Concatenate base key and random data, key first
CSSM_ALGID_ConcatDataAndBase	Concatenate base key and data, data first
CSSM_ALGID_XORBaseAndData	XOR a byte string with the base key
CSSM_ALGID_ExtractFromKey	Extract a key from base key, starting at arbitrary bit position
CSSM_ALGID_SSL3PreMasterGen	Generate a 48 byte SSL 3 pre-master key
CSSM_ALGID_SSL3MasterDerive	Derive an SSL 3 key from a pre-master key
CSSM_ALGID_SSL3KeyAndMacDerive	Derive the keys and MACing keys for the SSL cipher suite
CSSM_ALGID_SSL3MD5_MAC	Performs SSL 3 MD5 MACing
CSSM_ALGID_SSL3SHA1_MAC	Performs SSL 3 SHA-1 MACing

CSSM_ALGID_MD5_PBE	Generate key by MD5 hashing a base key
CSSM_ALGID_MD2_PBE	Generate key by MD2 hashing a base key
CSSM_ALGID_SHA1_PBE	Generate key by SHA-1 hashing a base key
CSSM_ALGID_WrapLynks	Spyrus LYNKS DES based wrapping scheme w/checksum
CSSM_ALGID_WrapSET_OAEP	SET key wrapping
CSSM_ALGID_BATON	Fortezza BATON cipher
CSSM_ALGID_ECDSA	Elliptic Curve DSA
CSSM_ALGID_MAYFLY	Fortezza MAYFLY cipher
CSSM_ALGID_JUNIPER	Fortezza JUNIPER cipher
CSSM_ALGID_FASTHASH	Fortezza FASTHASH
CSSM_ALGID_3DES	Generix 3DES
CSSM_ALGID_SSL3MD5	SSL3 with MD5
CSSM_ALGID_SSL3SHA1	SSL3 with SHA1
CSSM_ALGID_FortezzaTimestamp	Fortezza with Timestamp
CSSM_ALGID_SHA1WithDSA	SHA1 with DSA
CSSM_ALGID_SHA1WithECDSA	SHA1 with Elliptic Curve DSA
CSSM_ALGID_DSA_BSAFE	DSA with BSAFE Key format
CSSM_ALGID_Bcrypt	BSI algorithm
CSSM_ALGID_LUCpkcds	LUC Public key crypto and Dig Sig Alg
CSSM_ALGID_BARAS	
CSSM_ALGID_SxalMbal	Substitution Xor Alg / Multi Block Alg
CSSM_ALGID_MISTY1	Block Cipher
CSSM_ALGID_ENCRIP	

Table 44-7 Algorithms for a Session Context

Some of the above algorithms operate in a variety of modes. The desired mode is specified using an attribute of type CSSM_ATTRIBUTE_MODE. The valid values for the mode attribute are as follows:

Value	**Description**
CSSM_ALGMODE_NONE	Null Algorithm mode
CSSM_ALGMODE_CUSTOM	Custom mode
CSSM_ALGMODE_ECB	Electronic Code Book
CSSM_ALGMODE_ECBPad	ECB with padding
CSSM_ALGMODE_CBC	Cipher Block Chaining
CSSM_ALGMODE_CBC_IV8	CBC with Initialization Vector of 8 bytes
CSSM_ALGMODE_CBCPadIV8	CBC with padding and Initialization Vector of 8 bytes
CSSM_ALGMODE_CFB	Cipher FeedBack
CSSM_ALGMODE_CFB_IV8	CFB with Initialization Vector of 8 bytes
CSSM_ALGMODE_CFBPadIV8	CFB with Initialization Vector of 8 bytes and padding
CSSM_ALGMODE_OFB	Output FeedBack
CSSM_ALGMODE_OFB_IV8	OFB with Initialization Vector of 8 bytes
CSSM_ALGMODE_OFBPadIV8	OFB with Initialization Vector of 8 bytes and padding
CSSM_ALGMODE_COUNTER	Counter
CSSM_ALGMODE_BC	Block Chaining
CSSM_ALGMODE_PCBC	Propagating CBC
CSSM_ALGMODE_CBCC	CBC with Checksum
CSSM_ALGMODE_OFBNLF	OFB with NonLinear Function
CSSM_ALGMODE_PBC	Plaintext Block Chaining
CSSM_ALGMODE_PFB	Plaintext FeedBack
CSSM_ALGMODE_CBCPD	CBC of Plaintext Difference
CSSM_ALGMODE_PUBLIC_KEY	Use the public key
CSSM_ALGMODE_PRIVATE_KEY	Use the private key
CSSM_ALGMODE_SHUFFLE	Fortezza shuffle mode
CSSM_ALGMODE_ECB64	Electronic Code Book 64 bits
CSSM_ALGMODE_CBC64	Cipher BlockChaining 64 bits
CSSM_ALGMODE_OFB64	Output Feedback 64 bits
CSSM_ALGMODE_CBC64	Cipher Feedback 64 bits
CSSM_ALGMODE_CBC32	Cipher Feedback 32 bits
CSSM_ALGMODE_CBC16	Cipher Feedback 16 bits
CSSM_ALGMODE_CBC8	Cipher Feedback 8 bits
CSSM_ALGMODE_WRAP	
CSSM_ALGMODE_PRIVATE_WRAP	
CSSM_ALGMODE_RELAYX	

CSSM_ALGMODE_ECB128	Electronic Code Book 128 bits
CSSM_ALGMODE_ECB96	Electronic Code Book 96 bits
CSSM_ALGMODE_CBC128	Cipher Block Chaining 128 bits
CSSM_ALGMODE_OAEP_HASH	Algorithm mode for SET key wrapping

Table 44-8 PKCS #11 CSP Reader Flags

NumberOfAttributes
Number of attributes associated with this service.

ContextAttributes
Pointer to data that describes the attributes. To retrieve the next attribute, advance the attribute pointer.

44.2.20 CSSM_CSP_CAPABILITY

```
typedef CSSM_CONTEXT CSSM_CSP_CAPABILITY, *CSSM_CSP_CAPABILITY_PTR;
```

44.2.21 CSSM_SOFTWARE_CSPSUBSERVICE_INFO

```
typedef struct cssm_software_cspsubservice_info {
    uint32 NumberOfCapabilities;
    CSSM_CSP_CAPABILITY_PTR CapabilityList;
    uint32 Reserved;
} CSSM_SOFTWARE_CSPSUBSERVICE_INFO,
                          *CSSM_SOFTWARE_CSPSUBSERVICE_INFO_PTR;
```

Definition

NumberOfCapabilities
Number of capabilities available from the CSP.

CapabilityList
A context list that specifies the capabilities of the CSP.

Reserved
This field is reserved for future use and must always be set to zero.

44.2.22 CSSM_HARDWARE_CSPSUBSERVICE_INFO

```
typedef struct cssm_hardware_cspsubservice_info {
    uint32 NubmerOfCapabilities;
    CSSM_CSP_CAPABILITY_PTR CapabilityList;
    void* Reserved;

        /* Reader/Slot Info */
    char *ReaderDescription;
    char *ReaderVendor;
    char *ReaderSerialNumber;
    CSSM_VERSION ReaderHardwareVersion;
    CSSM_VERSION ReaderFirmwareVersion;
    uint32 ReaderFlags;
    uint32 ReaderCustomFlags;

    char *TokenDescription;
    char *TokenVendor;
    char *TokenSerialNumber;
    CSSM_VERSION TokenHardwareVersion;
    CSSM_VERSION TokenFirmwareVersion;

    uint32 TokenFlags;
    uint32 TokenCustomFlags;
    uint32 TokenMaxSessionCount;
    uint32 TokenOpenedSessionCount;
    uint32 TokenMaxRWSessionCount;
    uint32 TokenOpenedRWSessionCount;
    uint32 TokenTotalPublicMem;
    uint32 TokenFreePublicMem;
    uint32 TokenTotalPrivateMem;
    uint32 TokenFreePrivateMem;
    uint32 TokenMaxPinLen;
    uint32 TokenMinPinLen;
    char TokenUTCTime[16];

    CSSM_STRING UserLabel;
    CSSM_DATA UserCACertificate;
} CSSM_HARDWARE_CSPSUBSERVICE_INFO, *CSSM_HARDWARE_CSPSUBSERVICE_INFO_PTR;
```

Definition

NumberOfCapabilities
Number of capabilities available from the CSP.

CapabilityList
A list that specifies the capabilities of the CSP.

Reserved
This field is reserved for future use and must always be set to zero.

ReaderDescription
A NULL-terminated character string that contains a text description of the device reader.

ReaderVendor
A NULL-terminated string that contains the name of the reader vendor.

ReaderSerialNumber
A NULL-terminated string that contains the serial number of the reader.

ReaderHardwareVersion
Hardware version of the reader.

ReaderFirmwareVersion
Firmware version of the reader.

ReaderFlags
Bit mask containing information about the reader. The flags specified in the mask are as follows:

Reader Flag	**Description**
CSSM_CSP_RDR_TOKENPRESENT	Token is present in the reader
CSSM_CSP_RDR_TOKENREMOVABLE	Reader supports removable tokens
CSSM_CSP_RDR_HW	Reader is a hardware device

Table 44-9 PKCS #11 CSP Reader Flags

ReaderCustomFlags
Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

The following fields may not be valid if the CSSM_CSP_RDR_TOKENPRESENT flag is not set in the ReaderFlags field. Unknown string and CSSM_DATA fields will be set to NULL, integer and date fields will be set t zero and flag fields will have all flags set to false.

TokenDescription
A NULL-terminated character string that contains a text description of the token. This value may be NULL or equal to ReaderDescription if the token is not removable.

TokenVendor
A NULL-terminated string that contains the name of the token vendor. This value may be NULL or equal to ReaderVendor if the token is not removable.

TokenSerialNumber
A NULL-terminated string that contains the serial number of the token. This value may be NULL or equal to ReaderSerialNumber if the token is not removable.

TokenHardwareVersion
Hardware version of the token.

TokenFirmwareVersion
Firmware version of the token.

TokenFlags
Bit mask containing information about the token. The flags specified in the mask are as follows:

Token Flags	**Description**
CSSM_CSP_TOK_RNG	Token has random number generator
CSSM_CSP_TOK_WRITE_PROTECTED	Token is write protected
CSSM_CSP_TOK_LOGIN_REQUIRED	User must login to access private objects
CSSM_CSP_TOK_USER_PIN_INITIALIZED	User's PIN has been initialized
CSSM_CSP_TOK_EXCLUSIVE_SESSION	An exclusive session currently exists
CSSM_CSP_TOK_CLOCK_EXISTS	Token has built in clock
CSSM_CSP_TOK_ASYNC_SESSION	Token supports asynchronous operations
CSSM_CSP_TOK_PROT_AUTHENTICATION	Token has protected authentication path
CSSM_CSP_TOK_DUAL_CRYPTO_OPS	Token supports dual cryptographic operations

Table 44-10 PKCS #11 CSP Token Flags

TokenCustomFlags
Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

TokenMaxSessionCount
Maximum number of CSP handles referencing the token that may exist simultaneously.

TokenOpenedSessionCount
Number of existing CSP handles referencing the token.

TokenTotalPublicMem
Amount of public storage space in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE if the CSP does not wish to expose this information.

TokenFreePublicMem
Amount of public storage space available for use in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE if the CSP does not wish to expose this information.

TokenTotalPrivateMem
Amount of private storage space in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE if the CSP does not wish to expose this information.

TokenFreePrivateMem
Amount of private storage space available for use in the CSP. This value will be set to CSSM_VALUE_NOT_AVAILABLE if the CSP does not wish to expose this information.

TokenMaxPinLen
Maximum length of passwords that can be used for authentication to the CSP.

TokenMinPinLen
Minimum length of passwords that can be used for authentication to the CSP.

TokenUTCTime
Character array containing the current UTC time value in the CSP. The value is valid if the CSSM_CSP_TOK_CLOCK_EXISTS flag is true. The time is represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second; and 2 additional reserved '0' characters).

UserLabel
A NULL-terminated string containing the label of the token.

UserCACertificate
Certificate of the CA.

44.2.23 CSSM_HYBRID_CSPSUBSERVICE_INFO

```
CSSM_HARDWARE_CSPSUBSERVICE_INFO, *CSSM_HARDWARE_CSPSUBSERVICE_INFO_PTR;
```

44.2.24 CSSM_CSP_WRAPPEDPRODUCTINFO

```
typedef struct cssm_csp_wrappedproductinfo {
    CSSM_VERSION StandardVersion;
    CSSM_STRING StandardDescription;
    CSSM_VERSION ProductVersion;
    CSSM_STRING ProductDescription;
    CSSM_STRING ProductVendor;
    uint32 ProductFlags;
    uint32 ProductCustomFlags;
} CSSM_CSP_WRAPPEDPRODUCTINFO, *CSSM_CSP_WRAPPEDPRODUCTINFO_PTR;
```

Definition

StandardVersion
Version of the standard to which the wrapped product complies.

StandardDescription
A CSSM character string containing a text description of the standard to which the wrapped product complies.

ProductVersion
Version of the product wrapped by the CSP.

ProductDescription
A CSSM character string containing a text description of the product wrapped by the CSP.

ProductVendor
A CSSM character string containing the name of the wrapped product's vendor.

ProductFlags
This version of CSSM has no flags defined. This field must be set to zero.

ProductCustomFlags
Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

44.2.25 CSSM_CSP_FLAGS

A bit mask containing information about the CSP. The mask may be a combination of any of the following:

```
typedef uint32 CSSM_CSP_FLAGS;

#define CSSM_CSP_STORES_PRIVATE_KEYS
#define CSSM_CSP_STORES_PUBLIC_KEYS
#define CSSM_CSP_STORES_SESSION_KEYS
```

44.2.26 CSSM_CSPSUBSERVICE

```
typedef struct cssm_cspsubservice {
    uint32 SubServiceId;
    CSSM_STRING Description;
    CSSM_CSP_FLAGS CspFlags;
    uint32 CspCustomFlags;
    uint32 AccessFlags;
    CSSM_CSP_TYPE CspType;
    union cssm_subservice_info {
      CSSM_SOFTWARE_CSPSUBSERVICE_INFO SoftwareCspSubService;
      CSSM_HARDWARE_CSPSUBSERVICE_INFO HardwareCspSubService;
      CSSM_HYBRID_CSPSUBSERVICE_INFO HybridCspSubService;
    } SubserviceInfo;
    CSSM_CSP_WRAPPEDPRODUCTINFO WrappedProduct;
} CSSM_CSPSUBSERVICE, *CSSM_CSPSUBSERVICE_PTR;
```

Definition

SubServiceId
The sub-service ID required for an attach call to connect a CSP to an individual sub-service within a CSP.

Description
A CSSM character string containing a text description of the sub-service.

CspFlags
CSSM-defined flags indicating the key storage services provided by the CSP.

CspCustomFlags
Flags defined by the vendor. Consult the individual CSP user's guide for the list of valid flags.

AccessFlags
Flags that are required to be provided by the application during an attach call when specifying the sub-service ID given in SubServiceId.

CspType
Identifier that determines the type of CSP information structure contained in the union. The following values and their corresponding CSP information structures are currently defined.

CSP Information Structure Identifier	**Structure Type**
CSSM_CSP_TYPE_SOFTWARE	CSSM_SOFTWARE_CSPSUBSERVICE_INFO
CSSM_CSP_TYPE_HARDWARE	CSSM_HARDWARE_CSPSUBSERVICE_INFO

Table 44-11 CSP Information Type Identifiers and Associated Structure Types

SoftwareCspSubService or HardwareCspSubService
A CSP sub-service information structure of the type specified by CspType.

WrappedProduct
Pointer to a CSSM_CSP_WRAPPEDPRODUCTINFO structure describing a product that is wrapped by the CSP.

44.2.27 CSSM_SERVICE_INFO

This structure holds a description of a module service. The service described is of the CSSM service type specified by the module usage type. This structure is defined by CSSM core services and pertains to add-in service providers of all service types. See the *CSSM Add-in Module Structure and Administration Specification*. for additional descriptions of these CSSM core structure and the definitions of flag values to be specified as values in these structures.

```
typedef struct cssm_serviceinfo {
    CSSM_STRING Description;    /* Service description */
    CSSM_SERVICE_TYPE Type;     /* Service type */
    CSSM_SERVICE_FLAGS Flags;   /* Service flags */
    uint32 NumberOfSubServices; /* Number of sub services in
                                        SubService List */
    union cssm_subservice_list {
        void *SubServiceList;
        CSSM_CSPSUBSERVICE_PTR CspSubServiceList;
        CSSM_DLSUBSERVICE_PTR  DlSubServiceList;
        CSSM_CLSUBSERVICE_PTR  ClSubServiceList;
        CSSM_TPSUBSERVICE_PTR  TpSubServiceList;
    } SubServiceList;
    void *Reserved;
} CSSM_SERVICE_INFO, *CSSM_SERVICE_INFO_PTR;
```

Definition

Description
A text description of the service.

Type
Specifies exactly one type of service structure, such as CSSM_SERVICE_CSP, CSSM_SERVICE_CL, and so on.

Flags
Characteristics of this service, such as whether it contains any embedded products.

NumberOfSubServices
The number of elements in the module SubServiceList.

SubServiceList
A list of descriptions of the encapsulated SubServices which are not of the basic service types.

CspSubServiceList
A list of descriptions of the encapsulated CSP SubServices.

DlSubServiceList
A list of descriptions of the encapsulated DL SubServices.

ClSubServiceList
A list of descriptions of the encapsulated CL SubServices.

TpSubServiceList
A list of descriptions of the encapsulated TP SubServices.

Reserved
This field is reserved for future use. It should always be set to NULL.

44.2.28 CSSM_MODULE_INFO

This structure aggregates all service descriptions about all service types of a module implementation. The structure is defined by CSSM core services and pertains to add-in service modules of all module types. See the *CSSM Add-in Module Structure and Administration Specification* for additional descriptions of these CSSM core structure and the definitions of flag values to be specified as values in these structures.

```
typedef struct cssm_moduleinfo {
    CSSM_VERSION Version;               /* Module version */
    CSSM_VERSION CompatibleCSSMVersion; /* CSSM version the
                                        module is written for*/
    CSSM_GUID_PTR InterfaceGUID; /* opt GUID defining supported
                                        interface */
    CSSM_STRING Description;            /* Module description */
    CSSM_STRING Vendor;                 /* Vendor name */
    CSSM_MODULE_FLAGS Flags;            /* Flags to describe and
                                        control module use */
    CSSM_KEY_PTR AppAuthenRootKeys;     /* Module-specific keys
                                        to authenticate apps */
    uint32 NumberOfAppAuthenRootKeys;   /* Number of module-specific
                                        root keys */
    CSSM_SERVICE_MASK ServiceMask;      /* Bit mask of supported
                                        services */
    uint32 NumberOfServices;            /* Number of services in
                                        ServiceList */
    CSSM_SERVICE_INFO_PTR ServiceList;  /* A list of service info
                                        structures */
    void *Reserved;
} CSSM_MODULE_INFO, *CSSM_MODULE_INFO_PTR;
```

Definition

Version
The major and minor version numbers of this add-in module.

CompatibleCSSMVersion
The version of CSSM that this module was written to.

InterfaceGUID
GUID describing the interface supported by the version of CSSM that this module was written to.

Description
A text description of this module and its functionality.

Vendor
The name and description of the module vendor.

Flags
Characteristics of this module, such as whether or not it is threadsafe.

AppAuthenRootKeys
Public root keys used by CSSM to verify an application's credentials when the service module has requested authentication based on module-specified root keys by setting the CSSM_MODULE_CALLER_AUTHENTOMODULE bit to true in its CSSM_MODULE_FLAGS mask. These keys should successfully authenticate only those applications that the service module wishes to recognize to receive the services the module has registered with CSSM during module installation.

NumberOfAppAuthenRootKeys
The number of public root keys in the AppAuthenRoot Keys list.

ServiceMask
A bit mask identifying the types of services available in this module.

NumberOfServices
The number of services for which information is provided. Multiple descriptions (as sub-services) can be provided for a single service category.

ServiceList
An array of pointers to the service information structures. This array contains NumberOfServices entries.

Reserved
This field is reserved for future use. It should always be set to NULL.

44.3 Cryptographic Operations

The manpages for Cryptographic Operations follow on the next page.

NAME

CSP_SignData

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_SignData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Signature)
```

DESCRIPTION

This function signs data using the private key.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

DataBufs (input)
A pointer to one or more CSSM_DATA structures containing the data to be signed.

DataBufCount (input)
The number of DataBufs to be signed.

Signature (output)
A pointer to the CSSM_DATA structure for the signature.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid context pointer.

CSSM_CSP_INVALID_DATA_POINTER
Invalid pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid data count.

CSSM_CSP_INVALID_CALLBACK
Invalid call back function.

CSSM_CSP_SIGN_UNKNOWN_ALGORITHM
Unknown algorithm.

CSSM_CSP_OPERATION_UNSUPPORTED
Service not supported.

CSSM_CSP_SIGN_FAILED
Sign failed.

CSSM_CSP_PRIKEY_NOT_FOUND
Cannot find the corresponding private key.

CSSM_CSP_PASSWORD_INCORRECT
Password incorrect.

CSSM_CSP_PASSWORD_NO_PARAM
No password or callback function provided.

CSSM_CSP_UNWRAP_FAILED
Unwrapped the private key failed.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSP_VerifyData, *CSP_SignDataInit*, *CSP_SignDataUpdate*, *CSP_SignDataFinal*

NAME

CSP_SignDataInit

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_SignDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context)
```

DESCRIPTION

This function initializes the staged sign data function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
Invalid passphrase attribute in the asymmetric context.

CSSM_CSP_INVALID_ATTR_KEY
Invalid key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
Key class is not private key class.

CSSM_CSP_KEY_USAGE_INCORRECT
Key usage does not allow signature.

CSSM_CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSP_SignData, CSP_SignDataUpdate, CSP_SignDataFinal

NAME

CSP_SignDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_SignDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the data for the staged sign data function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to one or more CSSM_DATA structures containing the data be signed.

DataBufCount (input)
The number of DataBufs to be signed.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM_CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

SEE ALSO

CSP_SignData, *CSP_SignDataInit*, *CSP_SignDataFinal*

NAME

CSP_SignDataFinal

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_SignDataFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Signature)
```

DESCRIPTION

This function completes the final stage of the sign data function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Signature (output)
A pointer to the CSSM_DATA structure for the signature.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_CALLBACK_FAILED
Passphrase callback function failed.

CSSM_CSP_PRIKEY_NOT_FOUND
Cannot find the corresponding private key.

CSSM_CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed.

CSSM_CSP_PASSPHRASE_INCORRECT
Passphrase incorrect.

CSSM_CSP_PRIKEY_ERROR
Error in getting the raw private key or private key storage error.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSP_SignData, *CSP_SignDataInit*, *CSP_SignDataUpdate*

NAME

CSP_VerifyData

SYNOPSIS

```
CSSM_BOOL CSSMSPI CSP_VerifyData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    const CSSM_DATA_PTR Signature)
```

DESCRIPTION

This function verifies the input data against the provided signature.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

DataBufs (input)
A pointer to one or more CSSM_DATA structures containing the data be verified.

DataBufCount (input)
The number of DataBufs to be verified.

Signature (input)
A pointer to a CSSM_DATA structure which contains the signature and the size of the signature.

RETURN VALUE

A CSSM_TRUE return value signifies the signature was successfully verified. When CSSM_FALSE is returned, either the signature was not successfully verified or an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_OPERATION_UNSUPPORTED
Verify service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_SIGNATURE
Invalid or missing signature.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
Key class is not public key class.

CSSM_CSP_KEY_USAGE_INCORRECT
Key usage does not allow verify.

CSSM_CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSP_SignData, *CSP_VerifyDataInit*, *CSP_VerifyDataUpdate*, *CSP_VerifyDataFinal*

NAME

CSP_VerifyDataInit

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_VerifyDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context)
```

DESCRIPTION

This function initializes the staged verify data function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
Key class is not public key class.

CSSM_CSP_KEY_USAGE_INCORRECT
Key usage does not allow verify.

CSSM_CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSP_VerifyDataUpdate, *CSP_VerifyDataFinal*, *CSP_VerifyData*

NAME

CSP_VerifyDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_VerifyDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the data to the staged verify data function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to one or more CSSM_DATA structures containing the data be verified.

DataBufCount (input)
The number of DataBufs to be verified.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

SEE ALSO

CSP_VerifyData, *CSP_VerifyDataInit*, *CSP_VerifyDataFinal*

NAME

CSP_VerifyDataFinal

SYNOPSIS

```
CSSM_BOOL CSSMSPI CSP_VerifyDataFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR Signature)
```

DESCRIPTION

This function finalizes the staged verify data function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Signature (input)
A pointer to a CSSM_DATA structure which contains the starting address for the signature to verify against and the length of the signature in bytes.

RETURN VALUE

A CSSM_TRUE return value signifies the signature successfully verified. When CSSM_FALSE is returned, either the signature was not successfully verified or an error has occurred; use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_INVALID_SIGNATURE
Invalid or missing signature.

CSSM_CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

SEE ALSO

CSP_VerifyData, *CSP_VerifyDataInit*, *CSP_VerifyDataUpdate*

NAME

CSP_DigestData

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_DigestData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Digest)
```

DESCRIPTION

This function computes a message digest for the supplied data.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

DataBufs (input)
A pointer to one or more CSSM_DATA structures containing the supplied data.

DataBufCount (input)
The number of DataBufs.

Digest (output)
A pointer to the CSSM_DATA structure for the message digest.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Digest service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, application has to free the memory in this case. If the output buffer pointer this is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSP_DigestDataInit, *CSP_DigestDataUpdate*, *CSP_DigestDataFinal*, *CSP_DigestDataClone*

NAME

CSP_DigestDataInit

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_DigestDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context)
```

DESCRIPTION

This function initializes the staged message digest function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

SEE ALSO

CSP_DigestData, *CSP_DigestDataUpdate*, *CSP_DigestDataClone*, *CSP_DigestDataFinal*

NAME

CSP_DigestDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_DigestDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the staged message digest function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to one or more CSSM_DATA structures containing the supplied data.

DataBufCount (input)
The number of DataBufs.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM_CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

SEE ALSO

CSP_DigestData, *CSP_DigestDataInit*, *CSP_DigestDataClone*, *CSP_DigestDataFinal*

NAME

CSP_DigestDataClone

SYNOPSIS

```
CSSM_CC_HANDLE CSSMSPI CSP_DigestDataClone
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE oldCCHandle,
    CSSM_CC_HANDLE newCCHandle)
```

DESCRIPTION

This function clones a given staged message digest context with its cryptographic attributes and intermediate result.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

oldCCHandle (input)
The old handle that describes the context of a staged message digest operation.

newCCHandle (output)
The new handle that describes the cloned context of a staged message digest operation.

RETURN VALUE

The pointer to a user-allocated CSSM_CC_HANDLE for holding the cloned context handle return from CSSM. If the pointer is NULL, an error has occurred; use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

Comments

When a digest context is cloned, a new context is created with data associated with the parent context. Changes made to the parent context after calling this function will not be reflected in the cloned context. The cloned context could be used with the CSP_DigestDataUpdate and CSP_DigestDataFinal functions.

SEE ALSO

CSP_DigestData, *CSP_DigestDataInit*, *CSP_DigestDataUpdate*, *CSP_DigestDataFinal*

NAME

CSP_DigestDataFinal

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_DigestDataFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Digest)
```

DESCRIPTION

This function finalizes the staged message digest function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Digest (output)
A pointer to the CSSM_DATA structure for the message digest.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_GET_STAGED_INFO_ERROR
Cannot find or get the staged information.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSP_DigestData, *CSP_DigestDataInit*, *CSP_DigestDataUpdate*, *CSP_DigestDataClone*

NAME

CSP_GenerateMac

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_GenerateMac
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function generates a message authentication code for the supplied data.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

DataBufs (input)
A pointer to one or more CSSM_DATA structures containing the supplied data.

DataBufCount (input)
The number of DataBufs.

Mac (output)
A pointer to the CSSM_DATA structure for the message authentication code.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
: Invalid input or output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
: The output buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
: Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
: Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
: Generate MAC service not supported.

CSSM_CSP_OPERATION_FAILED
: Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
: Supports only a single buffer of input.

CSSM_CSP_INVALID_ATTR_KEY
: Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
: Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
: Key class is not session key class.

CSSM_CSP_KEY_ALGID_MISMATCH
: The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
: Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
: Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
: Key size in bits unsupported.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSP_GenerateMacInit, *CSP_GenerateMacUpdate*, *CSP_GenerateMacFinal*

NAME

CSP_GenerateMacInit

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_GenerateMacInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context)
```

DESCRIPTION

This function initializes the staged message authentication code function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
: Key class is not session key class.

CSSM_CSP_KEY_ALGID_MISMATCH
: The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
: Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
: Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
: Key size in bits unsupported.

SEE ALSO

CSP_GenerateMac, *CSP_GenerateMacUpdate*, *CSP_GenerateMacFinal*

NAME

CSP_GenerateMacUpdate

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_GenerateMacUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the staged message authentication code function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to one or more CSSM_DATA structures containing the supplied data.

DataBufCount (input)
The number of DataBufs.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

SEE ALSO

CSP_GenerateMac, *CSP_GenerateMacInit*, *CSP_GenerateMacFinal*

NAME

CSP_GenerateMacFinal

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_GenerateMacFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function finalizes the staged message authentication code function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Mac (output)
A pointer to the CSSM_DATA structure for the message authentication code.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space, application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSP_GenerateMac, *CSP_GenerateMacInit*, *CSP_GenerateMacUpdate*

NAME

CSP_VerifyMac

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSP_VerifyMac
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount,
    CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function verifies a message authentication code for the supplied data.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs.

Mac (input)
A pointer to the CSSM_DATA structure containing the MAC to verify.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Verify MAC service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
Key class is not session key class.

CSSM_CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

SEE ALSO

CSSM_VerifyMacInit, *CSSM_VerifyMacUpdate*, *CSSM_VerifyMacFinal*

NAME

CSP_VerifyMacInit

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSP_VerifyMacInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context)
```

DESCRIPTION

This function initializes the staged message authentication code verification function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
: Key class is not session key class.

CSSM_CSP_KEY_ALGID_MISMATCH
: The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
: Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
: Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
: Key size in bits unsupported.

SEE ALSO

CSSM_VerifyMac, *CSSM_VerifyMacUpdate*, *CSSM_VerifyMacFinal*

NAME

CSP_VerifyMacUpdate

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSP_GenerateMacUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR DataBufs,
    uint32 DataBufCount)
```

DESCRIPTION

This function updates the staged message authentication code verification function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

DataBufs (input)
A pointer to a vector of CSSM_DATA structures that contain the data to be operated on.

DataBufCount (input)
The number of DataBufs.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

SEE ALSO

CSSM_VerifyMac, *CSSM_VerifyMacInit*, *CSSM_VerifyMacFinal*

NAME

CSP_VerifyMacFinal

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSP_VerifyMacFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR Mac)
```

DESCRIPTION

This function finalizes the staged message authentication code verification function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Mac (input)
A pointer to the CSSM_DATA structure containing the MAC to verify.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if the MAC verifies correctly, CSSM_FAIL otherwise.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid input CSSM_DATA buffer.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

SEE ALSO

CSSM_VerifyMac, *CSSM_VerifyMacInit*, *CSSM_VerifyMacUpdate*

NAME

CSP_QuerySize

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_QuerySize
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    CSSM_BOOL Encrypt,
    uint32 QuerySizeCount,
    CSSM_QUERY_SIZE_DATA_PTR DataBlock)
```

DESCRIPTION

This function queries for sizes of output data blocks for encryption and decryption operations. The Encrypt flag specifies the encryption or the decryption operation. The input sizes are specified in the DataBlock structures. The corresponding output size are returned in the DataBlock structures. Multiple input block sizes can be specified in a single call.

This function can also be used to query the output size requirements for the intermediate steps of a staged cryptographic operation (for example, CSP_EncryptDataUpdate and CSP_DecryptDataUpdate). There may be algorithm-specific and token-specific rules restricting the lengths of data following data update calls.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

Encrypt (input)
This parameter describes the SizeInputBlock in DataBlock is for encryption or decryption.

QuerySizeCount (input)
This parameter describes number of DataBlocks.

DataBlock (input/output)
Pointer to a CSSM_QUERY_SIZE_DATA structure which contains one SizeInputBlock and one SizeOutputBlock. The function returns the size of the output in bytes in SizeOutputBlock for the size of the input.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
: Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
: Context type and operation do not match.

CSSM_CSP_INVALID_POINTER
: Invalid output query size data pointer.

CSSM_CSP_INVALID_ALGORITHM
: Unknown algorithm.

CSSM_CSP_OPERATION_UNSUPPORTED
: Query size service not supported.

CSSM_CSP_OPERATION_FAILED
: Query size operation failed.

CSSM_CSP_INVALID_PADDING
: Unknown padding.

CSSM_CSP_INVALID_MODE
: Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
: Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_QUERY_SIZE_UNKNOWN
: Cannot determine size of output data blocks.

SEE ALSO

CSP_EncryptData, *CSP_EncryptDataUpdate*, *CSP_DecryptData*, *CSP_DecryptDataUpdate*, *CSP_SignData*, *CSP_VerifyData*, *CSP_DigestData*, *CSP_GenerateMac*

NAME

CSP_EncryptData

SYNOPSIS

```
CSSM_RETURN CSSM_SPI CSP_EncryptData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    uint32 *bytesEncrypted,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function encrypts the supplied data using information in the context. The CSP_QuerySize function can be used to estimate the output buffer size required.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

ClearBufs (input)
A pointer to one or more CSSM_DATA structures containing the clear data.

ClearBufCount (input)
The number of ClearBufs.

CipherBufs (output)
A pointer to one or more CSSM_DATA structures for the encrypted data.

CipherBufCount (input)
The number of CipherBufs.

bytesEncrypted (output)
A pointer to uint32 for the size of the encrypted data in bytes.

RemData (output)
A pointer to the CSSM_DATA structure for the last encrypted block containing padded data.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Encrypt data service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
Key class is not private or public key class for asymmetric context or is not session class for symmetric context.

CSSM_CSP_KEY_USAGE_INCORRECT
Key usage does not allow encryption.

CSSM_CSP_KEY_ALGID_MISMATCH
The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

CSSM_CSP_INVALID_PADDING
: Unknown padding.

CSSM_CSP_INVALID_MODE
: Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
: Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
: Cannot find the corresponding private key.

CSSM_CSP_PASSPHRASE_INVALID
: Passphrase length error or passphrase badly formed for asymmetric context.

CSSM_CSP_PASSPHRASE_INCORRECT
: Passphrase incorrect for asymmetric context.

CSSM_CSP_PRIKEY_ERROR
: Error in getting the raw private key or private key storage error for asymmetric context.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
: Init vector attribute data or length error for symmetric context.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place encryption can be done by supplying the same input and output buffers.

SEE ALSO

CSP_QuerySize, *CSP_DecryptData*, *CSP_EncryptDataInit*, *CSP_EncryptDataUpdate*, *CSP_EncryptDataFinal*

NAME

CSP_EncryptDataInit

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_EncryptDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context)
```

DESCRIPTION

This function initializes the staged encrypt function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
: Key class is not private or public key class for asymmetric context or is not session class for symmetric context.

CSSM_CSP_KEY_USAGE_INCORRECT
: Key usage does not allow encryption.

CSSM_CSP_KEY_ALGID_MISMATCH
: The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
: Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
: Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
: Key size in bits unsupported.

CSSM_CSP_INVALID_PADDING
: Unknown padding.

CSSM_CSP_INVALID_MODE
: Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
: Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
: Cannot find the corresponding private key for asymmetric context.

CSSM_CSP_PASSPHRASE_INVALID
: Passphrase length error or passphrase badly formed for asymmetric context.

CSSM_CSP_PASSPHRASE_INCORRECT
: Passphrase incorrect for asymmetric context.

CSSM_CSP_PRIKEY_ERROR
: Error in getting the raw private key or private key storage error for asymmetric context.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
: Init vector attribute data or length error for symmetric context

SEE ALSO

CSP_EncryptData, *CSP_EncryptDataUpdate*, *CSP_EncryptDataFinal*

NAME

CSP_EncryptDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_EncryptDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    uint32 *bytesEncrypted)
```

DESCRIPTION

This function updates the staged encrypt function. The CSP_QuerySize function can be used to estimate the output buffer size required for each update call. There may be algorithm-specific and token-specific rules restricting the lengths of data in CSP_EncryptUpdate calls.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

ClearBufs (input)
A pointer to one or more CSSM_DATA structures containing the clear data.

ClearBufCount (input)
The number of ClearBufs.

CipherBufs (output)
A pointer to one or more CSSM_DATA structures for the encrypted data.

CipherBufCount (input)
The number of CipherBufs.

bytesEncrypted (output)
A pointer to uint32 for the size of the encrypted data in bytes.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input or output data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
: Invalid input or output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
: The output buffer is not big enough.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
: Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
: Staged Cryptographic operation failed.

CSSM_CSP_MEMORY_ERROR
: Not enough memory to allocate.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
: Supports only a single buffer of input.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place encryption can be done by supplying the same input and output buffer.

SEE ALSO

CSP_QuerySize, *CSP_EncryptData*, *CSP_EncryptDataInit*, *CSP_EncryptDataFinal*

NAME

CSP_EncryptDataFinal

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_EncryptDataFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function finalizes the staged encrypt function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RemData (output)
A pointer to the CSSM_DATA structure for the last encrypted block containing padded data.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place encryption can be done by supplying the same input and output buffers.

SEE ALSO

CSP_EncryptData, *CSP_EncryptDataInit*, *CSP_EncryptDataUpdate*

NAME

CSP_DecryptData

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_DecryptData
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    uint32 *bytesDecrypted,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function decrypts the supplied encrypted data. The CSP_QuerySize function can be used to estimate the output buffer size required.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

CipherBufs (input)
A pointer to one or more CSSM_DATA structures containing the encrypted data.

CipherBufCount (input)
The number of CipherBufs.

ClearBufs (output)
A pointer to one or more CSSM_DATA structures for the decrypted data.

ClearBufCount (input)
The number of ClearBufs.

bytesDecrypted (output)
A pointer to uint32 for the size of the decrypted data in bytes.

RemData (output)
A pointer to the CSSM_DATA structure for the last decrypted block.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
: Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
: Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
: Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
: Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
: Invalid data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
: Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
: The output buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
: Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
: Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
: Decrypt data service not supported.

CSSM_CSP_OPERATION_FAILED
: Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
: Supports only a single buffer of input.

CSSM_CSP_INVALID_ATTR_KEY
: Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
: Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
: Key class is not private or public key class for asymmetric context or is not session class for symmetric context.

CSSM_CSP_KEY_USAGE_INCORRECT
: Key usage does not allow decryption.

CSSM_CSP_KEY_ALGID_MISMATCH
: The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
: Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
: Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
: Key size in bits unsupported.

CSSM_CSP_INVALID_PADDING
: Unknown padding.

CSSM_CSP_INVALID_MODE
: Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
: Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
: Cannot find the corresponding private key for asymmetric context.

CSSM_CSP_PASSPHRASE_INVALID
: Passphrase length error or passphrase badly formed for asymmetric context.

CSSM_CSP_PASSPHRASE_INCORRECT
: Passphrase incorrect for asymmetric context.

CSSM_CSP_PRIKEY_ERROR
: Error in getting the raw private key or private key storage error for asymmetric context.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
: Init vector attribute data or length error for symmetric context.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place decryption can be done by supplying the same input and output buffer.

SEE ALSO

CSP_QuerySize, *CSP_EncryptData*, *CSP_DecryptDataInit*, *CSP_DecryptDataUpdate*, *CSP_DecryptDataFinal*

NAME

CSP_DecryptDataInit

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSSM_CSP_DecryptDataInit
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context)
```

DESCRIPTION

This function initializes the staged decrypt function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_KEY
Invalid or missing key attribute in the context.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
: Key class is not private or public key class for asymmetric context or is not session class for symmetric context.

CSSM_CSP_KEY_USAGE_INCORRECT
: Key usage does not allow decryption.

CSSM_CSP_KEY_ALGID_MISMATCH
: The supplied key does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
: Key header and key data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
: Unknown key format.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
: Key size in bits unsupported.

CSSM_CSP_INVALID_PADDING
: Unknown padding.

CSSM_CSP_INVALID_MODE
: Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
: Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
: Cannot find the corresponding private key.

CSSM_CSP_PASSPHRASE_INVALID
: Passphrase length error or passphrase badly formed.

CSSM_CSP_PASSPHRASE_INCORRECT
: Passphrase incorrect.

CSSM_CSP_PRIKEY_ERROR
: Error in getting the raw private key or private key storage error.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
: Init vector attribute data or length error for symmetric context.

SEE ALSO

CSP_DecryptData, *CSP_DecryptDataUpdate*, *CSP_DecryptDataFinal*

NAME

CSP_DecryptDataUpdate

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_DecryptDataUpdate
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR CipherBufs,
    uint32 CipherBufCount,
    CSSM_DATA_PTR ClearBufs,
    uint32 ClearBufCount,
    uint32 *bytesDecrypted)
```

DESCRIPTION

This function updates the staged decrypt function. The CSP_QuerySize function can be used to estimate the output buffer size required for each update call. There may be algorithm-specific and token-specific rules restricting the lengths of data in CSP_DecryptUpdate calls.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

CipherBufs (input)
A pointer to one or more CSSM_DATA structures containing the encrypted data.

CipherBufCount (input)
The number of CipherBufs.

ClearBufs (output)
A pointer to one or more CSSM_DATA structures for the decrypted data. The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate spaces; application has to free the memory in this case. If this is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

ClearBufCount (input)
The number of ClearBufs.

bytesDecrypted (output)
A pointer to uint32 for the size of the decrypted data in bytes.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA_COUNT
Invalid input or output data count; data count cannot be 0.

CSSM_CSP_INVALID_DATA
Invalid input or output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

CSSM_CSP_VECTOROFBUFS_UNSUPPORTED
Supports only a single buffer of input.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place decryption can be done by supplying the same input and output buffers.

SEE ALSO

CSP_QuerySize, *CSP_DecryptData*, *CSP_DecryptDataInit*, *CSP_DecryptDataFinal*

NAME

CSP_DecryptDataFinal

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_DecryptDataFinal
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    CSSM_DATA_PTR RemData)
```

DESCRIPTION

This function finalizes the staged decrypt function.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

RemData(output)
A pointer to the CSSM_DATA structure for the last decrypted block.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_STAGED_OPERATION_UNSUPPORTED
Staged operation unsupported.

CSSM_CSP_STAGED_OPERATION_FAILED
Staged Cryptographic operation failed.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned. In-place decryption can be done by supplying the same input and output buffers.

SEE ALSO

CSP_DecryptData, *CSP_DecryptDataInit*, *CSP_DecryptDataUpdate*

NAME

CSP_GenerateKey

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_GenerateKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA_PTR KeyLabel,
    CSSM_KEY_PTR Key)
```

DESCRIPTION

This function generates a symmetric key. The CSP may cache keying material associated with the new symmetric key. When the symmetric key is no longer in active use, the application can invoke the CSSM_FreeKey interface to allow cached keying material associated with the symmetric key to be removed.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

KeyUsage (input/optional)
A bit mask specifying how the new key can be used.

KeyAttr (input/optional)
A bit mask specifying other attributes to be associated with the new key.

KeyLabel (input)
Pointer to a byte string that will be used as the label for the key.

Key (output)
Pointer to CSSM_KEY structure used to obtain the key. Upon function invocation, any values in the CSSM_Key structure should be ignored. All input values should be supplied in the cryptographic context, KeyUsage, KeyAttr, and KeyLabel input parameters.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid CSSM_DATA pointer for KeyLabel.

CSSM_CSP_INVALID_DATA
Invalid CSSM_DATA buffer for KeyLabel.

CSSM_CSP_INVALID_KEY_POINTER
Invalid or missing CSSM_KEY pointer.

CSSM_CSP_INVALID_KEY
Invalid CSSM_KEY buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output key buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Generate key service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_KEYUSAGE_MASK
Specified key usage mask is invalid.

CSSM_CSP_KEYUSAGE_MASK_UNSUPPORTED
Requested key usage mask unsupported.

CSSM_CSP_INVALID_KEYATTR_MASK
Specified key attribute mask is invalid.

CSSM_CSP_KEYATTR_MASK_UNSUPPORTED
Requested key attribute mask unsupported.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

CSSM_CSP_INVALID_ATTR_SEED
Invalid seed attribute in the context if caller provides the seed crypto data structure.

CSSM_CSP_CALLBACK_FAILED
Seed callback function failed if caller provides a seed callback function.

CSSM_CSP_INVALID_ATTR_SALT
Invalid salt attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_ALG_PARAMS
Invalid param attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_START_DATE
Invalid start date attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_END_DATE
Invalid end date if caller provides one.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSP_GenerateRandom, *CSP_GenerateKeyPair*

NAME

CSP_GenerateKeyPair

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_GenerateKeyPair
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    uint32 PublicKeyUsage,
    uint32 PublicKeyAttr,
    const CSSM_DATA_PTR PublicKeyLabel,
    CSSM_KEY_PTR PublicKey,
    uint32 PrivateKeyUsage,
    uint32 PrivateKeyAttr,
    const CSSM_DATA_PTR PrivateKeyLabel,
    CSSM_KEY_PTR PrivateKey)
```

DESCRIPTION

This function generates an asymmetric key pair. The CSP may cache keying material associated with the new asymmetric keypair. When one or both of the keys are no longer in active use, the application can invoke the CSSM_FreeKey interface to allow cached keying material associated with the key to be removed.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context. .

PublicKeyUsage (input/optional)
A bit mask specifying how the new public key can be used.

PublicKeyAttr (input/optional)
A bit mask specifying other attributes to be associated with the new public key.

PublicKeyLabel (input)
Pointer to a byte string that will be used as the label for the public key.

PublicKey (output)
Pointer to CSSM_KEY structure used to obtain the public key. Upon function invocation, any values in the CSSM_Key structure should be ignored. All input values should be supplied in the cryptographic context, PublicKeyUsage, PublicKeyAttr, and PublicKeyLabel input parameters.

PrivateKeyUsage (input/optional)
A bit mask specifying how the new private key can be used.

PrivateKeyAttr (input/optional)
A bit mask specifying other attributes to be associated with the new private key.

PrivateKeyLabel (input)
Pointer to a byte string that will be used as the label for the private key.

PrivateKey (output)
Pointer to CSSM_KEY structure used to obtain the private key. Upon function invocation, any values in the CSSM_Key structure should be ignored. All input values should be supplied in the cryptographic context, PublicKeyUsage, PublicKeyAttr, and PublicKeyLabel input parameters.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid CSSM_DATA pointer for PublicKeyLabel or PrivateKeyLabel.

CSSM_CSP_INVALID_DATA
Invalid CSSM_DATA buffer for PublicKeyLabel or PrivateKeyLabel.

CSSM_CSP_INVALID_KEY_POINTER
Invalid or missing CSSM_KEY pointer.

CSSM_CSP_INVALID_KEY
Invalid CSSM_KEY buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output key buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Generate key pair service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the context.

CSSM_CSP_CALLBACK_FAILED
Passphrase callback function failed.

CSSM_CSP_INVALID_KEYUSAGE_MASK
Specified key usage mask is invalid.

CSSM_CSP_KEYUSAGE_MASK_UNSUPPORTED
Requested key usage mask unsupported.

CSSM_CSP_INVALID_KEYATTR_MASK
Specified key attribute mask is invalid.

CSSM_CSP_KEYATTR_MASK_UNSUPPORTED
Requested key attribute mask unsupported.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported.

CSSM_CSP_INVALID_ATTR_ALG_PARAMS
Invalid param attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_START_DATE
Invalid start date attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_END_DATE
Invalid end date attribute if caller provides one.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

SEE ALSO

CSP_GenerateRandom, *CSP_GenerateKey*

NAME

CSP_GenerateRandom

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_GenerateRandom
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    CSSM_DATA_PTR RandomNumber)
```

DESCRIPTION

This function generates random data.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

RandomNumber (output)
Pointer to CSSM_DATA structure used to obtain the random number and the size of the random number in bytes.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid or missing output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Generate random service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_ATTR_SEED
Invalid seed attribute in the context if caller provides the seed crypto data structure.

CSSM_CSP_CALLBACK_FAILED
Seed callback function failed if caller provides a seed callback function.

CSSM_CSP_INVALID_ATTR_OUTPUT_SIZE
Invalid or missing output length attribute.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

NAME

CSP_FreeKey

SYNOPSIS

```
CSSM_RETURN CSSMAPI CSP_FreeKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_KEY_PTR KeyPtr)
```

DESCRIPTION

This function requests the cryptographic service provider to clean up any key material associated with the key. This function also releases the internal storage referenced by the KeyData field of the key structure, which can hold the actual key value. The key reference by KeyPtr can be a persistent key or a transient key. This function clears the cached copy of the key and has no effect on the long term persistence or transience of the key.

PARAMETERS

CSPHandle (input)
The handle that describes the module to perform this operation.

PublicKey (input)
The key whose associated keying material can be discarded at this time.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_KEY
Key not recognized by this CSP

CSSM_CSP_MEMORY_ERROR
Internal memory error

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented

NAME

CSP_ObtainPrivateKeyFromPublicKey

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_ObtainPrivateKeyFromPublicKey (
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_KEY_PTR PublicKey,
    CSSM_KEY_PTR Private_Key);
```

DESCRIPTION

Given a public key this function returns a reference to the private key. The private key and its associated passphrase can be used as an input to any function requiring a private key value.

PARAMETERS

CSPHandle (input)
The handle that describes the module to perform this operation.

PublicKey (input)
The public key corresponding to the private key being sought.

PrivateKey (output)
A reference to the private key corresponding to the public key.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CSP_PRIKEY_NOT_FOUND
Corresponding private key not found.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

NAME

CSP_WrapKey

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_WrapKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_CRYPTO_DATA_PTR PassPhrase,
    CSSM_KEY_PTR Key,
    CSSM_DATA_PTR DescriptiveData,
    CSSM_WRAP_KEY_PTR WrappedKey)
```

DESCRIPTION

This function wraps the supplied key using the context. The key can be a symmetric key or a reference to a private key. If the key is a symmetric key, then a symmetric context must be provided describing the wrapping algorithm. If the key is a private key, then an asymmetric context describing the wrapping algorithm, and a passphrase to unlock the referenced private key must be provided. If the specified wrapping algorithm is NULL, then the key is returned in raw format, if permitted and supported by the CSP. The CSP is responsible for incorporating all of the pertinent key attributes into the wrapped key, ensuring that the state of the key can be fully restored by the unwrap process.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up-calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle to the context that describes this cryptographic operation.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

PassPhrase (input)
The passphrase or a callback function to be used to obtain the passphrase that can be used by the CSP to unlock the private key before it is wrapped. This input is ignored when wrapping a symmetric, secret key.

Key (input)
A pointer to the target key to be wrapped. If a private key is to be wrapped, this is a reference to the private key. If a symmetric key is to be wrapped, the target key is that symmetric key.

DescriptiveData (input/optional)
A pointer to a CSSM_DATA structure containing additional descriptive data to be associated and included with the key during the wrapping operation. The caller and the wrapping algorithm incorporate knowledge of the structure of the descriptive data. If the wrapping algorithm does not accept additional descriptive data, then this parameter must be NULL. If the wrapping algorithm accepts descriptive data, the corresponding unwrapping algorithm can be used to extract the descriptive data and the key.

WrappedKey (output)
A pointer to a CSSM_KEY structure that returns the wrapped key.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match. The context has to be either symmetric context or asymmetric context.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Wrap key service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_KEY_POINTER
Invalid CSSM_KEY or CSSM_WRAP_KEYpointers.

CSSM_INVALID_SUBJECT_KEY
Invalid subject key (key to be wrapped).

CSSM_CSP_INVALID_CRYPTO_DATA_POINTER
Invalid or missing passphrase (parameter required if the subject key is a private key).

CSSM_CSP_CALLBACK_FAILED
Passphrase callback function failed for subject private key or for wrapping key in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
Cannot find the subject private key.

CSSM_CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed for either the passphrase parameter or passphrase in the asymmetric context.

CSSM_CSP_PRIKEY_ERROR
Error in getting the subject private key or subject private key storage error.

CSSM_CSP_INVALID_KEY
Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
Key class is not private or public key class for asymmetric context or is not session key class for symmetric context.

CSSM_CSP_KEY_ALGID_MISMATCH
The key in the context (key to be used for wrapping) does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
Key header and key data (for the wrapping key) is inconsistent.

CSSM_CSP_KEY_USAGE_INCORRECT
Key usage mask (for the wrapping key) does not allow wrap.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown key format (for the wrapping key).

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
Key size in bits unsupported (for the wrapping key).

CSSM_CSP_INVALID_PADDING
Unknown padding.

CSSM_CSP_INVALID_MODE
Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
Init vector attribute data or length error for symmetric context.

SEE ALSO

CSP_UnwrapKey

NAME

CSP_UnwrapKey

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_UnwrapKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_CRYPTO_DATA_PTR NewPassPhrase,
    const CSSM_KEY_PTR PublicKey
    const CSSM_WRAP_KEY_PTR WrappedKey,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA_PTR KeyLabel,
    CSSM_KEY_PTR UnwrappedKey,
    CSSM_DATA_PTR DescriptiveData)
```

DESCRIPTION

This function unwraps the wrapped key using the context. The wrapped key can be a symmetric key or a private key. If the key is a symmetric key, then a symmetric context must be provide describing the unwrapping algorithm. If the key is a private key, then an asymmetric context must be provide describing the unwrapping algorithm. Depending on the persistent object mode of the CSP and the storage mode specified by the key attribute value in the wrapped key header, the unwrapped key can be securely stored by the CSP and locked by the new passphrase. If the unwrapping algorithm is NULL and the wrapped key is actually a raw key (as indicated by its key attributes), then the key is imported into the CSP. Support for a NULL unwrapping algorithm, is at the option of the CSP. The CSP must recover the complete state of the unwrapped key based on the key attributes stored in the wrapped key.

PARAMETERS

CSPHandle (input)
: The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
: The handle that describes the context of this cryptographic operation.

Context (input)
: Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

PassPhrase (input)
: The passphrase or a callback function to be used to obtain the passphrase. If the unwrapped key is a private key and the persistent object mode is true, then the private key is unwrapped and securely stored by the CSP. The PassPhrase is used to secure the private key after it is unwrapped. It is assumed that a known public key is associated with the private key.

PublicKey (input)
: The public key corresponding to the private key being unwrapped.

WrappedKey (input)
: A pointer to the wrapped key. The wrapped key may be a symmetric key or the private key of a public/private keypair. The unwrapping method is specified as meta data within the wrapped key, and is not specified outside of the wrapped key.

KeyUsage (input/optional)
A bit mask specifying how the unwrapped key can be used.

KeyAttr (input/optional)
A bit mask specifying other attributes to be associated with the unwrapped key.

KeyLabel (input/optional)
Pointer to a byte string that will be used as the label for the unwrapped key.

UnwrappedKey (output)
A pointer to a CSSM_KEY structure that returns the unwrapped key.

DescriptiveData (output)
A pointer to a CSSM_DATA structure that returns any additional descriptive data that was associated with the key during the wrapping operation. It is assumed that the caller incorporated knowledge of the structure of this data. If no additional data is associated with the imported key, this output value is NULL.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_DATA_POINTER
Invalid output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_OPERATION_UNSUPPORTED
Unwrap key service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_KEYATTR
Specified key attribute is incorrect or unsupported.

CSSM_CSP_INVALID_KEY_POINTER
Invalid CSSM_KEY or CSSM_WRAP_KEYpointers.

CSSM_INVALID_SUBJECT_KEY
Invalid subject key (key to be unwrapped).

CSSM_CSP_INVALID_CRYPTO_DATA_POINTER
Invalid or missing passphrase (parameter required if the subject key is a private key).

CSSM_CSP_CALLBACK_FAILED
: Passphrase callback function failed for subject private key or for private key in the asymmetric context.

CSSM_CSP_PRIKEY_NOT_FOUND
: Cannot find the corresponding private key for either the subject private key or the private key in the asymmetric context.

CSSM_CSP_PASSPHRASE_INVALID
: Passphrase length error or passphrase badly formed for either the passphrase parameter or passphrase in the asymmetric context.

CSSM_CSP_PRIKEY_ERROR
: Error in getting the raw private key or private key storage error for either the subject private key or the private key in the asymmetric context.

CSSM_CSP_INVALID_KEY
: Invalid or missing key data in the context attribute.

CSSM_CSP_INVALID_KEYCLASS
: Key class is not private or public key class for asymmetric context or is not session key class for symmetric context.

CSSM_CSP_KEY_ALGID_MISMATCH
: The key in the context (key to be used for unwrapping) does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
: Key header and key data (for the unwrapping key) is inconsistent.

CSSM_CSP_KEY_USAGE_INCORRECT
: Key usage mask (for the unwrapping key) does not allow unwrap.

CSSM_CSP_KEY_FORMAT_INCORRECT
: Unknown key format (for the unwrapping key).

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
: Key size in bits unsupported (for the unwrapping key).

CSSM_CSP_INVALID_PADDING
: Unknown padding.

CSSM_CSP_INVALID_MODE
: Unknown algorithm mode for symmetric context.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
: Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_INVALID_ATTR_INIT_VECTOR
: Init vector attribute data or length error for symmetric context.

CSSM_CSP_INVALID_KEYATTR
: Specified key attribute is incorrect or unsupported.

SEE ALSO

CSP_WrapKey

NAME

CSP_DeriveKey

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_DeriveKey
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    const CSSM_KEY_PTR BaseKey,
    CSSM_DATA_PTR Param,
    uint32 KeyUsage,
    uint32 KeyAttr,
    const CSSM_DATA_PTR KeyLabel,
    CSSM_KEY_PTR DerivedKey)
```

DESCRIPTION

This function derives a new symmetric key using the context and information from the base key.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

BaseKey (input)
The base key used to derive the new key. The base key may be a public key, a private key, or a symmetric key.

Param (input/output)
This parameter varies depending on the derivation mechanism. Password based derivation algorithms use this parameter to return a cipher block chaining initialization vector. Concatenation algorithms will use this parameter to get the second item to concatenate.

KeyUsage (input/optional)
A bit mask specifying how the new key can be used.

KeyAttr (input/optional)
A bit mask specifying other attributes to be associated with the new key.

KeyLabel (input/optional)
Pointer to a byte string that will be used as the label for the derived key.

DerivedKey (output)
A pointer to a CSSM_KEY structure that returns the derived key.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output buffer is not big enough.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Derive key service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

CSSM_CSP_INVALID_SUBJECT_KEY
Invalid or missing BaseKey.

CSSM_CSP_INVALID_KEYUSAGE_MASK
Specified usage mask for the key being derived is invalid.

CSSM_CSP_KEYUSAGE_MASK_UNSUPPORTED
Requested usage mask for the key being derived is unsupported.

CSSM_CSP_INVALID_KEYATTR_MASK
Specified attribute mask for the key being derived is invalid.

CSSM_CSP_KEYATTR_MASK_UNSUPPORTED
Requested attribute mask for the key being derived is unsupported.

CSSM_CSP_KEY_USAGE_INCORRECT
Usage mask on BaseKey does not allow key derivation.

CSSM_CSP_INVALID_KEY
Invalid buffer specified for the DerivedKey parameter.

CSSM_CSP_NOT_ENOUGH_BUFFER
The output DerivedKey buffer is not big enough.

CSSM_CSP_KEY_ALGID_MISMATCH
The BaseKey does not match the operation.

CSSM_CSP_KEY_KEYHEADER_INCONSISTENT
BaseKey header and BaseKey data is inconsistent.

CSSM_CSP_KEY_FORMAT_INCORRECT
Unknown BaseKey format.

CSSM_CSP_INVALID_ATTR_SEED
Invalid seed attribute in the context if caller provides the seed crypto data structure.

CSSM_CSP_CALLBACK_FAILED
Seed callback function failed if caller provides a seed callback function.

CSSM_CSP_INVALID_ATTR_PASSPHRASE
Invalid or missing passphrase attribute in the asymmetric context.

CSSM_CSP_PASSPHRASE_INVALID
Passphrase length error or passphrase badly formed.

CSSM_CSP_INVALID_ATTR_SALT
Invalid salt attribute if caller provides one.

CSSM_CSP_INVALID_ATTR_INTERATION_COUNT
Invalid iteration count attribute or value.

CSSM_CSP_INVALID_KEY_SIZE_IN_BITS
The key size in bits for BaseKey or DerivedKey is unsupported.

NAME

CSP_GenerateAlgorithmParams

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_GenerateAlgorithmParams
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    uint32 ParamBits,
    CSSM_DATA_PTR Param)
```

DESCRIPTION

This function generates algorithm parameters for the specified context. These parameters include Diffie-Hellman key agreement parameters and DSA key generation parameters.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CCHandle (input)
The handle that describes the context of this cryptographic operation used to link to the CSP-managed information.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this context.

ParamBits (input)
Used to generate parameters for the algorithm (for example, Diffie-Hellman).

Param (output)
Pointer to CSSM_DATA structure used to obtain the key exchange parameter and the size of the key exchange parameter in bytes.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_CONTEXT
Context type and operation do not match.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_INVALID_DATA_POINTER
Invalid input or output CSSM_DATA pointer.

CSSM_CSP_INVALID_DATA
Invalid output CSSM_DATA buffer.

CSSM_CSP_INVALID_ALGORITHM
Unknown algorithm.

CSSM_CSP_OPERATION_UNSUPPORTED
Generate algorithm params not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

Comments

The output can be obtained either by filling the caller-supplied buffer or using the application's memory allocation functions to allocate space; application has to free the memory in this case. If the output buffer pointer is NULL, an error code CSSM_CSP_INVALID_DATA_POINTER is returned.

NAME

CSP_QueryKeySizeInBits

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_QueryKeySizeInBits
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_KEY_PTR Key,
    CSSM_KEY_SIZE_PTR KeySize)
```

DESCRIPTION

This function queries a crypto service provider for the effective and real size of a key in bits. The key can be specified alone or in the context of a cryptographic context. If specified alone, the CSP determines the effective bit size of the key based on the real bit size and any known constraints on the usage of that key. If a cryptographic context is provided, the effective bit size of the key is determined based on the assumption that the key would be used to perform the operation described by that cryptographic context.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform this function. If a NULL handle is specified, CSSM returns error.

CCHandle (input/optional)
A handle to the cryptographic context describing the operation for which the effective bit size of the key should be determined. If the context is specified, it must contain the key whose effective bit size is being queried. If the cryptographic context is not specified, then the key must be provided in the optional Key input parameter.

Key (input/optional)
A pointer to a CSSM_KEY structure containing the key for which size is to be determined. If the specific cryptographic context in which the key is to be used is not known the key must be specified alone in this parameter and the cryptographic context input parameter must be NULL. If the context is known and is specified by the CCHandle input parameter, then the key must be contained in the context structure and the Key input parameter must be NULL.

KeySize (output)
Pointer to a CSSM_KEYSIZE data structure to receive the size of the key in bits.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_CONTEXT_HANDLE
Invalid context handle.

CSSM_CSP_INVALID_CONTEXT_POINTER
Invalid CSSM_CONTEXT pointer.

CSSM_CSP_INVALID_KEY_POINTER
Key pointer is missing or invalid.

CSSM_CSP_INVALID_KEY
Invalid key buffer.

CSSM_CSP_INVALID_POINTER
Invalid output CSSM_KEY_SIZE pointer.

CSSM_CSP_OPERATION_UNSUPPORTED
Query key size in bits service not supported.

CSSM_CSP_OPERATION_FAILED
Cryptographic operation failed.

44.4 Cryptographic Sessions and Logon

The manpages for Cryptographic Sessions and Logon follow on the next page.

NAME

CSP_Login

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_Login
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_CRYPTO_DATA_PTR Password,
    const CSSM_DATA_PTR Reserved)
```

DESCRIPTION

Logs the user into the CSP, allowing for multiple login types and parallel operation notification.

PARAMETERS

CSPHandle (input)
Handle of the CSP to log into.

Password (input)
Password used to log into the token.

Reserved (input)
This field is reserved for future use. The value NULL should always be given. (May be used for multiple user support in the future.)

RETURN VALUE

CSSM_OK if login is successful, CSSM_FAIL is login fails. Use CSSM_GetError to determine the exact error.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_INVALID_PASSWORD
Invalid password.

CSSM_CSP_ALREADY_LOGGED_IN
User attempted to log in more than once.

CSSM_CSP_OPERATION_UNSUPPORTED
Login service not supported.

SEE ALSO

CSP_ChangeLoginPassword, *CSP_Logout*

NAME

CSP_Logout

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_Logout
    (CSSM_CSP_HANDLE CSPHandle)
```

DESCRIPTION

Terminates the login session associated with the specified CSP Handle.

PARAMETERS

CSPHandle (input)
Handle for the target CSP.

RETURN VALUE

CSSM_OK if successful, CSSM_FAIL if an error occurred. Use CSSM_GetError to determine the exact error.

ERRORS

CSSM_CSP_INVALID_CSP
Invalid CSP handle.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_NOT_LOGGED_IN
No login session existed.

CSSM_CSP_OPERATION_UNSUPPORTED
Log out service not supported.

SEE ALSO

CSP_Login, *CSP_ChangeLoginPassword*

NAME

CSP_ChangeLoginPassword

SYNOPSIS

```
CSSM_RETURN CSSMSPI CSP_ChangeLoginPassword
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_CRYPTO_DATA_PTR OldPassword,
    const CSSM_CRYPTO_DATA_PTR NewPassword)
```

DESCRIPTION

Changes the login password of the current login session from the old password to the new password. The requesting user must have a login session in process.

PARAMETERS

CSPHandle (input)
Handle of the CSP supporting the current login session.

OldPassword (input)
Current password used to log into the token.

NewPassword (input)
New password to be used for future logins by this user to this token.

RETURN VALUE

CSSM_OK if login is successful, CSSM_FAIL is login fails. Use CSSM_GetError to determine the exact error.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_INVALID_PASSWORD
Old password is invalid.

CSSM_CSP_OPERATION_UNSUPPORTED
Change login password service not supported.

SEE ALSO

CSP_Login, *CSP_Logout*

44.5 Extensibility Functions

The KRSP_PassThrough function is provided to allow KRSP developers to extend the key recovery functionality of the CSSM API. Because it is only exposed to CSSM as a function pointer, its name internal to the CSP can be assigned at the discretion of the CSP module developer. However, its parameter list and return value must match what is shown below. The error codes given in this section constitute the generic error codes which may be used by all CSPs to describe common error conditions.

CSP developers may also define their own module-specific error codes, as described in *CSSM Add-in Module Structure and Administration Specification*. The manpages for Extensibility Functions follow on the next page.

NAME

CSP_PassThrough

SYNOPSIS

```
void* CSSMSPI CSP_PassThrough
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CONTEXT_PTR Context,
    uint32 PassThroughId,
    const void * InData,)
```

DESCRIPTION

The CSP_PassThrough function is provided to allow CSP developers to extend the crypto functionality of the CSSM API.

PARAMETERS

CSPHandle (input)
Handle of the CSP supporting the passthrough function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

Context (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this custom context structure.

PassThroughId (input)
An identifier specifying the custom function to be performed.

InData (input)
A pointer to void structure containing the input data.

RETURN VALUE

A pointer to void structure contains the output.

ERRORS

CSSM_CSP_INVALID_CSP_HANDLE
Invalid CSP handle.

CSSM_CSP_INVALID_POINTER
Invalid pointer for input data.

CSSM_CSP_MEMORY_ERROR
Not enough memory to allocate.

CSSM_CSP_OPERATION_UNSUPPORTED
Service not supported.

CSSM_CSP_OPERATION_FAILED
Unable to perform custom function.

44.6 Module Management Functions

The CSP_GetCapabilities and CSP_EventNotify functions are used by the CSSM Core to interact with the CSP module. Because these functions are only exposed to CSSM as function pointers, their names internal to the CSP library can be assigned at the discretion of the CSP module developer. However, their parameter lists and return values must match what is shown below. The error codes given in this section constitute the generic error codes, which may be used by all CSP libraries to describe common error conditions. CSP module developers may also define their own module-specific error codes, as described in the *CSSM Add-in Module Structure and Administration Specification*.

NAME

CSP_GetCapabilities

SYNOPSIS

```
CSSM_CSPINFO_PTR CSSMSPI  CSP_GetCapabilities
    (CSSM_CSP_HANDLE CSPHandle,
    CSSM_BOOL CompleteCapabilitiesOnly,
    uint32 *CSPInfoCount)
```

DESCRIPTION

This function is called by the CSSM when the registry indicates that capabilities information for a CSP is dynamic.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CompleteCapabilitiesOnly (input)
Boolean flag that indicates whether all devices controlled by the CSP should be represented in the return list. If TRUE, all devices are listed regardless of availability. If FALSE, only devices that are available for use are listed.

CSPInfoCount (output)
The number of CSSM_CSPINFO structures returned. One structure should be returned for each device controlled by the CSP.

RETURN VALUE

The return value is an array of CSSM_CSPINFO structures, with the length returned in the CSPInfoCount parameter. If CSPInfoCount is zero, the return value will be NULL.

ERRORS

CSSM_INVALID_POINTER
Invalid pointer.

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_INVALID_GUID
Unknown GUID.

SEE ALSO

CSP_EventNotify

NAME

CSP_EventNotify

SYNOPSIS

```
CSSM_RETURN  CSSMSPI  CSP_EventNotify
    (CSSM_CSP_HANDLE CSPHandle,
    const CSSM_EVENT_TYPE Event,
    const uint32 Param)
```

DESCRIPTION

Called by the CSSM when an event that could impact the internal state of a CSP takes place.

PARAMETERS

CSPHandle (input)
The handle that describes the add-in cryptographic service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

Event (input)
One of the event types listed below.

Param (input)
This value will vary depending on the type of event. In the case where no parameter is required, this value will be zero.

RETURN VALUE

The return value from this function has varying effects based on the event type. In most cases the value CSSM_OK should be returned so indicate that the CSSM can continue. The value CSSM_FAIL should be returned in cases of fatal errors within the CSP.

Event Types:

CSSM_EVENT_ATTACH
An attach to the token is taking place. The CSP handle passed to the function is the new handle that will be returned to the application. This event will take place after the initial call to CSP_Initialize. Returning CSSM_FAIL results in a failure of the CSSM_CSP_Attach call.

CSSM_EVENT_DETACH
A detach from the token is taking place. The CSP handle passed to the function is a handle that will have been the subject of a previous CSSM_EVENT_ATTACH event. This event will take place immediately before the call to CSP_Uninitialize when the handle being detached is the only handle associated with that CSP. Returning CSSM_FAIL has no effect.

CSSM_EVENT_INFOATTACH
An attach to the token is taking place in order to get the capabilities list for the CSP. The CSP handle passed to the function is a temporary handle created for the specific purpose of calling CSP_GetCapabilities. This event will take place without a call to CSP_Initialize. When this event is received, only the minimal amount of initialization required to successfully perform a CSP_GetCapabilities call should be performed. Returning CSSM_FAIL results in a failure of the attach.

CSSM_EVENT_INFODETACH
A detach from the token is taking place. The CSP handle passed to the function is a handle that will have been the subject of a previous CSSM_EVENT_INFOATTACH event. This event will never be followed by a call to CSP_Uninitialize when the handle being detached is the only handle associated with that CSP. Returning CSSM_FAIL has no effect.

SEE ALSO

CSP_GetCapabilities, *CSSM_CSP_Attach*, *CSSM_CSP_Detach*

CAE Specification

Part 10:

CSSM Trust Policy Interface

The Open Group

Chapter 45

Introduction

45.1 CDSA Add-In Module Overview

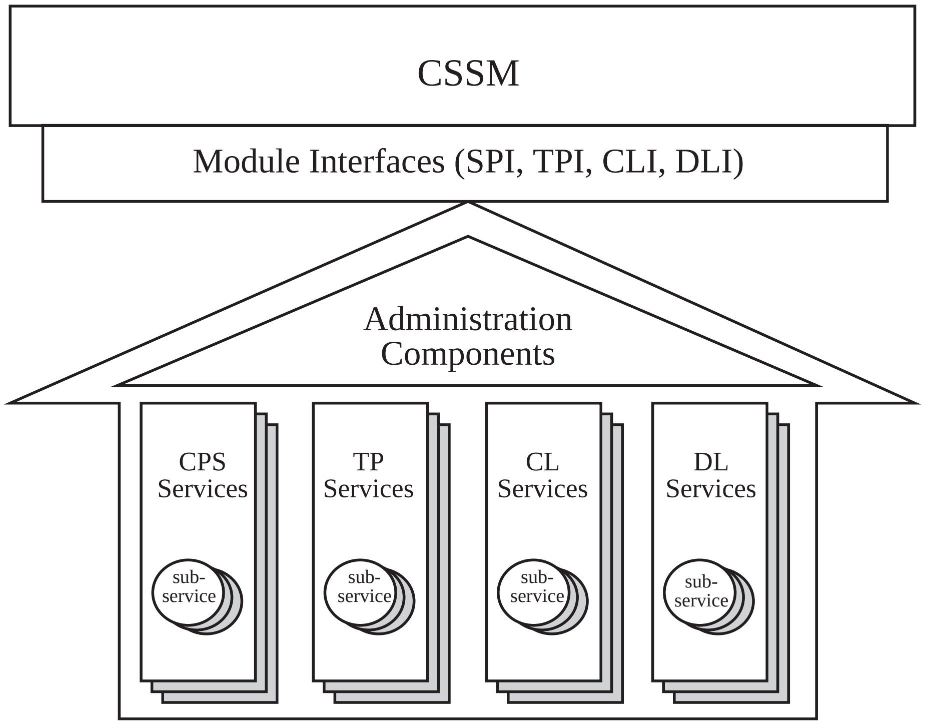

Figure 45-1 CDSA Add-In Module Structure

A CDSA add-in module is a dynamically-linkable library, composed of functions that implement some or all of the CSSM Module Interfaces. Add-in module functionality is partitioned into two areas:

- The provision of security services to applications
- Module administration

Add-in modules provide one or more categories of security services to applications. The service categories are Cryptographic Service Provider (CSP) services, Trust Policy (TP) services, Certificate Library (CL) services, and Data Storage Library (DL) services. Each security service contains one or more implementation instances, called sub-services. For a CSP service providing access to hardware tokens, a sub-service would represent a slot. For a DL service provider, a sub-service would represent a type of persistent storage. These sub-services each support the module interface for their respective service categories. This documentation-part describes the module interface functions in the trust policy service category. More information about CSP, CL and DL services can be found in the *CSSM Cryptographic Service Provider Interface Specification*, *CSSM Certificate Library Interface Specification* and in the *CSSM Data Storage Library Interface Specification* respectively.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions which allow CSSM to indicate events such as module attach and detach. In addition, as part of the attach operation, every module must be able to verify its own integrity, verify the integrity of CSSM, and register with CSSM. Detailed information about add-in module structure, administration, and interfaces can be found in the *CSSM Add-in Module Structure and Administration Specification*.

Information about the Common Data Security Architecture (CDSA) and Common Security Services Manager (CSSM) can be found in the *CSSM Application Programming Interface*.

45.2 Trust Policy Overview

Trust Policy modules implement policies defined by authorities and institutions. Policies define the level of trust required before certain actions can be performed. Three basic action categories exist for all certificate-based trust domains:

- Actions on certificates
- Actions on certificate revocation lists
- Domain-specific actions (such as enforcing business rules or access control policy)

The CSSM Trust Policy API defines the generic operations that each TP module supports. Each module may choose to implement the required subset of these operations for the policy it serves.

The CSSM API defines a pass-through function, which allows each module to provide additional functions, along with those defined by the CSSM Trust Policy API. When a TP function determines the trustworthiness of performing an action, it may invoke Certificate Library functions and Data storage Library functions to carry out the mechanics of the approved action. TP modules must be installed and registered with the CSSM Trust Policy Services Manager. Applications may query the Services Manager to retrieve properties of the TP module, as defined during installation.

45.2.1 Using Trust Policy Modules

An application determines the availability of a Trust Policy module by querying the CSSM Registry. When a new TP is installed on a system, it must be registered with CSSM. When a client requests that CSSM attach to a TP, CSSM returns a TP handle to the application which uniquely identifies the pairing of the application thread to the TP module instance. The application uses this handle to identify the TP in future function calls.

CSSM manages function tables provided by the TP module and the application. A function upcall table is used to register application memory allocation and de-allocation functions with CSSM. The Trust Policy module will have access to the upcall table. The Trust Policy module registers its function table with CSSM at library load time using *CSSM_RegisterServices*. See the *CSSM Add-in Module Structure and Administration Specification* for details of module installation and registration.

Chapter 46

Trust Policy Interface

46.1 Overview

A digital certificate is the binding of some identification to a public key in a particular domain. When a certificate is issued (created and signed) by the owner and authority of a domain, the binding between key and identity is validated by the digital signature on the certificate. The issuing authority also associates a level of trust with the certificate. The actions of the user, whose identity is bound to the certificate, are constrained by the trust policy governing the certificate's usage domain. A digital certificate is intended to be an unforgeable credential in cyberspace.

The use of digital certificates is the foundation on which the CDSA is designed. The CDSA assumes the concept of digital certificates in its broadest sense. Applications use the credential for:

- Identification
- Authentication
- Authorization

The applications interpret and manipulate the contents of certificates to achieve these ends, based on the real-world trust model they chose as their model for trust and security. The primary purpose of a Trust Policy (TP) module is to answer the question, "Is this certificate trusted for this action?" The CSSM Trust Policy API determines the generic operations that should be defined for certificate-based trust in every application domain. The specific semantics of each operation is defined by the:

- Application domain
- Trust model
- Policy statement for a domain
- Certificate type
- Real-world operation the user is trying to perform within the application domain
- The sources of trust (called anchors) and the sources of distrust in revocation lists

The trust model is expressed as an executable policy that is used by all applications that subscribe to that policy and the trust model it represents. As an infrastructure, CSSM is policy-neutral with respect to application-domain policies; it does not incorporate any single policy. For example, the verification procedure for a credit card certificate should be defined and implemented by the credit company issuing the certificate. Employee access to a lab housing a critical project should be defined by the company whose intellectual property is at risk. Rather than defining policies, CSSM provides the infrastructure for installing and managing policy-specific modules. This ensures complete extensibility of certificate-based trust on every platform hosting CSSM.

Policies define the credentials required for authorization to perform an action on another object. Certificates are the basic credentials representing a trust relationship among a set of two or more parties. When an organization issues certificates it defines its issuing procedure in a Certification Practice Statement (CPS). The statement identifies existing policies with which it is consistent.

The statement can also be the source of new policy definitions if the action and target object domains are not covered by an existing, published policy. An application domain can recognize multiple policies. A given policy can be recognized by multiple application domains.

Evaluation of trust depends on relationships among certificates. The trust domain can define accepted sources of trust, called anchors. Anchors can be mandated by fiat or can be computed by some other means. In contrast to the sources of trust, certificate revocation lists represent sources of distrust. Trust policies may consult these lists during the verification process

Different trust policies define different actions that an application may request. Some of these actions are common to every trust policy, and are operations on objects all trust models use. The objects believed to be common to all trust models are certificates and certificate revocation records. The basic operations on these objects are sign, verify, and revoke.

Based on this analysis, CSSM defines two categories of API calls that should be implemented by TP modules. The first category allows the TP module to validate operations relevant within an application domain (such as requesting authorization to make a $200 charge on a credit card certificate, and requesting access to the locked project lab). The second category enforces operations relevant within a trust model (for example, sign, verify, and revoke) on certificates and certificate revocation lists.

Application developers and trust domain authorities benefit from the ability to define and implement policy-based modules. Application developers are freed from the burden of implementing a policy description and certifying that their implementation conforms. Instead, the application needs only to build in a list of the authorities and certificate issuers it uses.

Domain authorities also benefit from an infrastructure that supports add-in Trust Policy modules. Authorities are ensured that applications using their module(s) adhere to the policies of the domain. Also, dynamic download of trust modules (possibly from remote systems) ensures timely and accurate propagation of policy changes. Individual functions within the module may combine local and remote processing. This flexibility allows the module developer to implement policies based on the ability to communicate with a remote authority system. This also allows the policy implementation to be decomposed in any convenient distributed manner.

Implementing a Trust Policy module may or may not be tightly coupled with one or more Certificate Library modules or one or more Data Storage Library modules. The trust policy embodies the semantics of the domain. The certificate library and the data storage library embody the syntax of a certificate format and operations on that format. A trust policy can be completely independent of certificate format, or it may be defined to operate with one or a small number of certificate formats. A trust policy implementation may invoke a certificate library module or data storage library modules to facilitate making policy based manipulations.

The Trust Policy API defines two categories of operation:

- Module installation and management
- Trust-based services

46.1.1 Trust Policy Services API

CSSM defines API calls for the following types of operations:

Creating Certificates. Client applications can request that a certificate be issued to the client. It is the responsibility of the trust policy module to determine whether the client the process of requesting and obtaining the certificate. The trust policy can include is trusted to be issued a certificate. If the client is authorized, the trust policy performs provide additional authorization information to the CA and can add information, on behalf of the client, to be included in the issued certificate.

Signing Certificates and Certificate Revocation Lists. Every system should be capable of being a Certificate Authority (CA), if so authorized. CAs are applications that issue and validate certificates and certificate revocation lists (CRLs). Issuing certificates and CRLs include initializing their attributes and digitally signing the result using the private key of the issuing authority. The private key used for signing is associated with the signer's certificate. The Trust Policy module must evaluate the trustworthiness of the signer's certificate before performing this operation. Some policies may require that multiple authorities sign a newly-issued certificate. If the TP trusts the signer's certificate, then the TP module may perform the cryptographic signaturing algorithm by invoking the signing function in a Certificate library module, or by directly invoking the data signing function in a CSP module. The certificate library functions that can be used to carry out some of the TP operations are documented in the *CSSM Certificate Library Interface Specification*.

Verifying Certificates and Certificate Revocation Lists. The TP module determines the trustworthiness of certificates and certificate revocation lists. The test focuses on the trustworthiness of the agent who signed the document. The TP module may need to perform operations on the certificate or CRL to determine trustworthiness. If these operations depend on the data format of the certificate or CRL, the TP module uses the services of a certificate library module to perform these checks. The TP module must determine if the certificate presented is trusted to perform actions defined by the TP module. An action for a TP module might be an employee's access to a lab system housing data for a critical project. The question of whether to allow the employee to access the system is asked through this function.

Revoking Certificates. When revoking a certificate, the identity of the revoking agent is presented in the form of another certificate. The TP module must determine trustworthiness of the revoking agent's certificate to perform revocation. If the requesting agent's certificate is trustworthy, the TP module carries out the operation directly by invoking a certificate library module to add a new revocation record to a CRL, marking the certificate as revoked. The CSSM API also defines a reason parameter that is passed to the TP module. The TP may use this parameter as part of its trust evaluation.

Pass-through Function. For operations not defined in the TPI, the pass- through function allows the TP module to provide support for these services to clients. These private services are identified by operation identifiers. TP module developers must provide documentation of these services.

46.1.2 Trust Policy Module Operations

Interface Name	Interface Description
TP_CertRequest	Determines trust in issuing a certificate to the caller and initiates a certificate request to a CA.
TP_CertRetrieve	Retrieves the certificate requested by TP_CertRequest.
TP_CertGroupVerify	Determines whether a group of one or more certificates is trustworthy. Policy identifiers are used to specify the policy domain(s) to be evaluated by the policy module.
TP_CertSign	Determines whether the signer's certificate is authorized to co-sign or notarize the target certificate. If so, The TP module carries out the operation. The *scope* of a signature may be used to identify which certificate field should be signed. An example is the case of multiple signatures on a certificate. Should signatures be applied to just the certificate, or to the certificate and all currently-existing signatures, as a notary public would do.
TP_CertRevoke	Determines whether the revoker's certificate is trusted to perform /sign the revocation. If so, the TP module carries out the operation by adding a new revocation record to the CRL.
TP_CertGroupConstruct	Construct a collection of certificates that forms a semantically related trust-relationship.
TP_CertGroupPrune	Remove from a collection of certificates those that do not participate in a semantically related trust-relationship outside of the local system.
TP_CrlVerify	Determines whether the CRL is trusted. This test may include verifying the correctness of the signature associated with the CRL, determining whether the CRL has been tampered with, and determining if the agent who signed the CRL is a trusted issuer of CRLs.
TP_CrlSign	Determines whether the certificate is trusted to sign the CRL. If so, the TP module carries out the operation.
TP_ApplyCrlToDb	Determines whether the memory-resident CRL is trusted and should be applied to a persistent database, which could result in designating certificates as revoked
TP_PassThrough	Executes TP module custom operations. This function accepts as input an operation ID and an arbitrary set of input parameters. The operation ID may specify any type of operation the TP wishes to export. Such operations may include queries or services specific to the domain represented by the TP module.

46.2 Data Structures

```
typedef uint32 CSSM_TP_HANDLE    /* Trust Policy Handle */
typedef uint32 CSSM_TP_ACTION
typedef CSSMAPI CSSMTPI
```

46.2.1 CSSM_DATA

The CSSM_DATA structure associates a length, in bytes, with an arbitrary block of contiguous memory. This memory must be allocated and freed using the memory management routines provided by the calling application via CSSM.

```
typedef struct cssm_data {
    uint32 Length;
    uint8* Data;
} CSSM_DATA, *CSSM_DATA_PTR
```

Definition

Length
The length, in bytes, of the memory block pointed to by Data.

Data
A pointer to a contiguous block of memory.

46.2.2 CSSM_OID

This structure stores object identifier for describing the data.

```
typedef CSSM_DATA CSSM_OID, *CSSM_OID_PTR;
```

46.2.3 CSSM_FIELD

This structure contains the tag/data pair for a single field of a certificate or CRL.

```
typedef struct cssm_field {
    CSSM_OID FieldOid;
    CSSM_DATA FieldValue;
}CSSM_FIELD, *CSSM_FIELD_PTR
```

Definition

FieldOid
The object identifier which uniquely identifies this certificate or CRL field.

FieldValue
The data contained in this certificate or CRL field.

46.2.4 CSSM_REVOKE_REASON

This structure represents the reason a certificate is being revoked.

```
typedef enum cssm_revoke_reason {
    CSSM_REVOKE_CUSTOM,
    CSSM_REVOKE_UNSPECIFIC,
    CSSM_REVOKE_KEYCOMPROMISE,
    CSSM_REVOKE_CACOMPROMISE,
    CSSM_REVOKE_AFFILIATIONCHANGED,
    CSSM_REVOKE_SUPERSEDED,
    CSSM_REVOKE_CESSATIONOFOPERATION,
    CSSM_REVOKE_CERTIFICATEHOLD,
    CSSM_REVOKE_CERTIFICATEHOLDRELEASE,
    CSSM_REVOKE_REMOVEFROMCRL
} CSSM_REVOKE_REASON
```

46.2.5 CSSM_CRL_TYPE

This structure represents the type and format used for revocation lists.

```
typedef enum cssm_crl_type {
    CSSM_CRL_TYPE_UNKNOWN,
    CSSM_CRL_TYPE_X_509v1,
    CSSM_CRL_TYPE_X_509v2
} CSSM_CRL_TYPE, *CSSM_CRL_TYPE_PTR;
```

46.2.6 CSSM_CRL_ENCODING

This structure represents the encoding format used for revocation lists.

```
typedef enum cssm_crl_encoding {
    CSSM_CRL_ENCODING_UNKNOWN,
    CSSM_CRL_ENCODING_CUSTOM,
    CSSM_CRL_ENCODING_BER,
    CSSM_CRL_ENCODING_DER,
    CSSM_CRL_ENCODING_BLOOM
} CSSM_CRL_ENCODING, *CSSM_CRL_ENCODING_PTR;
```

46.2.7 CSSM_DL_DB_HANDLE

This data structure holds a pair of handles, one for a data storage library and another for a data store opened and being managed by the data storage library.

```
typedef struct cssm_dl_db_handle {
    CSSM_DL_HANDLE DLHandle;
    CSSM_DB_HANDLE DBHandle;
} CSSM_DL_DB_HANDLE, *CSSM_DL_DB_HANDLE_PTR;
```

Definition

DLHandle
Handle of an attached module that provides DL services.

DBHandle
Handle of an open data store that is currently under the management of the DL module specified by the DLHandle.

46.2.8 CSSM_DL_DB_LIST

This data structure defines a list of handle pairs of (data storage library handle, data store handle).

```
typedef struct cssm_dl_db_list {
    uint32 NumHandles;
    CSSM_DL_DB_HANDLE_PTR DLDBHandle;
} CSSM_DL_DB_LIST, *CSSM_DL_DB_LIST_PTR;
```

Definition

NumHandles
Number of (data storage library handle, data store handle) pairs in the list.

DLDBHandle
List of (data storage library handle, data store handle) pairs.

46.2.9 CSSM_CERTGROUP

This structure contains a set of certificates. It is assumed that the certificates are related based on co-signaturing. The certificate group is a syntactic representation of a trust model. All certificates in the group must be of the same type.

```
typedef struct {
    CSSM_CERT_TYPE CertType; /* Certificate domain/type
                                                 identifier */
    CSSM_CERT_ENCODING  CertEncoding;  /* certificate encoding */
    uint32 NumCerts; /* number of elements in CertList array */
    CSSM_DATA_PTR CertList; /* List of opaque certificates */
    void *reserved;
} CSSM_CERTGROUP, *CSSM_CERTGROUP_PTR;
```

Definition

CertType
An identifier indicating how the certificate is formatted and the domain of interpretation.

CertEncoding
An indicator of the encoding applied to the certificates in the cert group.

NumCerts
Number of certificates in the group.

CertList
List of certificates.

reserved
Reserved for future use.

46.2.10 CSSM_EVIDENCE_FORM

This structure contains certificates, CRLs and other information used as audit trail evidence.

```
#define CSSM_EVIDENCE_FORM_UNSPECIFIC 0x0

#define CSSM_EVIDENCE_FORM_CERT 0x1

#define CSSM_EVIDENCE_FORM_CRL 0x2
```

```
typedef struct cssm_evidence {
    uint32 EvidenceForm;
                /* CSSM_EVIDENCE_FORM_CERT,CSSM_EVIDENCE_FORM_CRL */
    union cssm_format_type {
        CSSM_CERT_TYPE CertType;
        CSSM_CRL_TYPE CrlType
    } FormatType ;
    union cssm_format_encoding {
        CSSM_CERT_ENCODING CertEncoding;
        CSSM_CRL_ENCODING CrlEncoding
    } FormatEncoding ;
    CSSM_DATA_PTR Evidence; /* Evidence content */
} CSSM_EVIDENCE, *CSSM_EVIDENCE_PTR;
```

Definition

EvidenceForm
An identifier directing how to interpret the evidence format.

FormatType
Identifies the certificate type or the CRL type contained in the Evidence buffer.

FormatEncoding
Identifies the certificate encoding or the CRL encoding contained in the Evidence buffer.

Evidence
Buffer containing audit trail components.

46.2.11 CSSM_VERIFYCONTEXT

This data structure contains parameters useful in verifying certificate groups, certificate revocation lists and other forms of signed document

```
Typedef struct cssm_verify_context {
    CSSM_FIELD_PTR  PolicyIdentifiers,
    uint32 NumberofPolicyIdentifiers,
    CSSM_TP_STOP_ON VerificationAbortOn,
    CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    CSSM_DATA_PTR AnchorCerts,
    uint32 NumberofAnchorCerts,
    CSSM_FIELD_PTR VerifyScope,
    uint32 ScopeSize,
    CSSM_TP_ACTION Action,
    CSSM_NOTIFY_CALLBACK CallbackWithVerifiedCert,
    CSSM_DATA_PTR ActionData,
    CSSM_EVIDENCE_PTR *Evidence,
    uint32 *NumberOfEvidences;
} CSSM_VERIFYCONTEXT, *CSSM_VERIFYCONTEXT_PTR;
```

Definition

PolicyIdentifiers
The policy identifier is a OID-value pair. The CSSM_OID structure contains the name of the policy and the value is an optional, caller-specified input value for the TP module to use when applying the policy. The name space for policy identifiers is defined externally by the application domains served by the trust policy module.

NumberofPolicyIdentifiers
The number of Policy Identifiers provided in the PolicyIdentifiers parameter.

AnchorCerts A pointer to the CSSM_DATA structure containing one or more Certificates to be used in order to validate the Subject Certificate. These certificates can be root certificates, cross-certified certificates, and certificates belonging to locally designated sources of trust.

NumberofAnchorCerts
The number of anchor Certificates provided in the AnchorCerts parameter.

VerificationAbortOn
When a TP module verifies multiple conditions or multiple policies, the TP module can allow the caller to specify when to abort the verification process. If supported by the TP module, this selection can effect the evidence returned by the TP module to the caller. The default stopping condition is to stop evaluation according to the policy defined in the TP Module. The specifiable stopping conditions and their meaning are defined as follows:

CSSM_TP_STOP_ON	**Definition**
CSSM_STOP_ON_POLICY	Stop verification whenever the policy dictates it
CSSM_STOP_ON_NONE	Stop verification only after all conditions have been tested (ignoring the pass-fail status of each condition)
CSSM_STOP_ON_FIRST_PASS	Stop verification on the first condition that passes
CSSM_STOP_ON_FIRST_FAIL	Stop verification on the first condition that fails

The TP module may ignore the caller’s specified stopping condition and revert to the default of stopping according to the policy embedded in the module.

UserAuthentication
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on, depending on the context of the request. The required format for this credential is defined by the TP and recorded in the TPSubservice structure describing this module. If the supplied credential is insufficient, additional information can be obtained from the substructure field named MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If additional information is not required, this value can be NULL.

VerifyScope
A pointer to the CSSM_FIELD array containing the OID / Value pairs that are to be used to qualify the validity of the Certificate. The context of the validity checks will be evident from each OID / Value pairing. If VerifyScope is not specified, the TP Module must assume a default scope (portions of the Subject certificate) when performing the verification process.

ScopeSize
The number of entries in the verify scope list. If the verification scope is not specified, the input scope size must be zero.

Action
An application-specific and application-defined action to be performed under the authority of the input certificate. If no action is specified, the TP module defines a default action and performs verification assuming that action is being requested.

Note: It is also possible that a TP module verifies certificates for only one action.

CallbackWithVerifiedCert
A caller defined function to be invoked by the TP module once for each certificate examined in the verification process. The verified certificate is passed back to the caller via this function. The module invokes the callback with four input parameters. 1) module handle, 2) application specific handle, 3) reason code and 4) pointer to returned data parameter. The reason code will be CSSM_NOTIFY_CERT_VERIFIED and the data value will be a pointer to CSSM_DATA. Contained in the CSSM_DATA will be an opaque certificate. The callback function must free the CSSM_DATA structure and its contents. If the verification process completes in a single verify step, then no callbacks are made. If the callback function pointer is NULL, no callbacks are performed.

ActionData
A pointer to the CSSM_DATA structure containing the action-specific data or a reference to the action-specific data upon which the requested action should be performed. If no data is specified, and the specified action requires action data then the TP module defines one or more default data objects upon which the action or default action would be performed.

Evidence
A pointer to a list of CSSM_EVIDENCE objects containing an audit trail of evidence constructed by the TP module during the verification process. Typically this contains Certificates and CRLs that were used to establish the validity of the Subject Certificate, but other objects may be appropriate for other types of trust policies.

NumberOfEvidences
The number of entries in the Evidence list. The returned value is zero if no evidence is produced. Evidence may be produced even when verification fails. This evidence can describe why and how the operation failed to verify the subject certificate.

46.2.12 CSSM_TP_WRAPPEDPRODUCTINFO

This structure holds information describing any backend products used by the TP module to implement its services. This descriptive information is stored in the CSSM registry when the TP module is installed with CSSM. CSSM checks the integrity of the TP module description before using the information.

The descriptive information stored in this structures can be queried using the function *CSSM_GetModuleInfo*( ) and specifying the trust policy module GUID, service mask, subservice identifier, and level of information disclosure.

```
typedef struct cssm_tp_wrappedproductinfo {
    CSSM_VERSION StandardVersion;
    CSSM_STRING StandardDescription;  /* Description of standard
                                        product */
    CSSM_STRING_ProductVendor;        /* Vendor of wrapped
                                        product */
    uint32   ProductFlags;
}CSSM_TP_WRAPPEDPRODUCTINFO, *CSSM_TP_WRAPPEDPRODUCTINFO_PTR;
```

Definition

StandardVersion
Version number of the product behind this module.

StandardDescription
A string containing a descriptive name or title for this wrapped product.

ProductVendor
Name of the vendor who developed (and markets) the wrapped product.

ProductFlags
A bit mask describing attributes of the wrapped product.

46.2.13 CSSM_TPSUBSERVICE

Four structures are used to contain the attributes that describe a trust policy add-in module: the moduleinfo, the serviceinfo, the tp_wrappedproductinfo, and the tpsubservice structure. The first two structures are general and the attributes contained in them are applicable to all types of service modules. The last two structures are trust policy module-specific. This descriptive information is stored in the CSSM registry when the TP module is installed with CSSM. CSSM checks the integrity of the TP module description before using the information.

A trust policy module may implement multiple types of services and organize them as sub-services.

The descriptive information stored in these structures can be queried using the function *CSSM_GetModuleInfo*( ) and specifying the trust policy module GUID, service mask, subservice identifier, and level of information disclosure.

```
typedef struct cssm_tpsubservice {
    uint32 SubServiceId;
    char *Description;      /* Description of this subservice */
    CSSM_CERT_TYPE CertType;      /* cert types accepted by
                                            this module */
    CSSM_CERT_ENCODING CertEncoding;   /* Encoding of cert
                                         accepted by TP */
    CSSM_CALLER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
    uint32   NumberOfPolicyIdentifiers;
    CSSM_FIELD_PTR  PolicyIdentifiers;
    CSSM_TP_WRAPPEDPRODUCTINFO WrappedProduct;
} CSSM_TPSUBSERVICE, *CSSM_TPSUBSERVICE_PTR;
```

Definition

SubServiceId
A unique, identifying number for the sub-service described in this structure.

Description
A string containing a descriptive name or title for this sub-service.

CertType
A bitmask of the certificate types processed by the trust policy.

CertEncoding
A bitmask of the certificate encodings processed by the trust policy.

AuthenticationMechanism
An enumerated value defining the credential format accepted by the TP module. An authentication credential is required for some TP functions. Presented credentials must be of the required format.

NumberOfPolicyIdentifiers
The number of policies supported by this TP module.

PolicyIdentifiers
A list of the policies (represented by their identifiers) supported by this TP module. There must be NumberOfPolicyIdentifiers entries in this list.

WrappedProduct
A pointer to the wrapped product description.

46.2.14 CSSM_SPI_TP_FUNCS

This data structure contains function pointers to the routines that a TP module can support. The function prototypes are provided for compiler checking when assigning function pointers to the structure. This structure is used during the registration of the TP's services. The CSSM_MODULE_FUNCS data structure defined in the *CSSM Add-in Module Structure and Administration Specification* uses an opaque pointer to a function table ModuleServices. When the ServiceType is CSSM_USAGE_TP the function table pointer will refer to a CSSM_SPI_TP_FUNCS data structure.

```
typedef struct cssm_spi_tp_funcs {
    CSSM_DATA_PTR (CSSMTPI *CertRequest)  (
        CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_SUBSERVICE_UID CSPSubserviceUid,
        const CSSM_FIELD_PTR CertFields,
        uint32 NumberOfFields,
        const CSSM_FIELD_PTR  PolicyIdentifier,
        uint32 NumberOfPolicyIdentifiers,
        CSSM_CA_SERVICES MoreServiceRequests,
        const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
        sint32 *EstimatedTime,
        const CSSM_DATA_PTR ReferenceIdentifier);

    CSSM_DATA_PTR (CSSMTPI *CertRetrieve)  (
        CSSM_CL_HANDLE CLHandle,
        const CSSM_DATA_PTR ReferenceIdentifier,
        const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
        sint32 *EstimatedTime);

        CSSM_BOOL (CSSMTPI *CertGroupVerify) (CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_CSP_HANDLE CSPHandle,
        CSSM_DL_DB_LIST DBList,
        const CSSM_CERTGROUP_PTR CertToBeVerified,
        const CSSM_VERIFYCONTEXT_PTR VerifyContext);

    CSSM_DATA_PTR (CSSMTPI *CertSign) (CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_CC_HANDLE CCHandle,
        const CSSM_DL_DB_LIST_PTR DBList,
        const CSSM_DATA_PTR CertToBeSigned,
        const CSSM_CERTGROUP_PTR SignerCertGroup,
        const CSSM_VERIFYCONTEXT_PTR SignerVerifyContext,
        const CSSM_FIELD_PTR SignScope,
        uint32 ScopeSize);

    CSSM_DATA_PTR (CSSMTPI *CertRevoke) (
        CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_CSP_HANDLE CSPHandle,
        const CSSM_DL_DB_LIST_PTR DBList,
        const CSSM_DATA_PTR OldCrl,
        const CSSM_CERTGROUP_PTR CertGroupToBeRevoked,
        const CSSM_CERTGROUP_PTR RevokerCertGroup,
        const CSSM_VERIFYCONTEXT_PTR RevokerVerifyContext,
        CSSM_REVOKE_REASON Reason);

    CSSM_BOOL (CSSMTPI *CrlVerify) (
        CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_CSP_HANDLE CSPHandle,
        const CSSM_DL_DB_LIST_PTR DBList,
        const CSSM_DATA_PTR CrlToBeVerified,
        CSSM_CRL_TYPE CrlType,
        CSSM_CRL_ENCODING CrlEncoding,
        const CSSM_CERTGROUP_PTR SignerCertGroup,
        const CSSM_VERIFYCONTEXT_PTR VerifyContext);

    CSSM_DATA_PTR (CSSMTPI *CrlSign) (
        CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_CC_HANDLE CCHandle,
        const CSSM_DL_DB_LIST_PTR DBList,
        const CSSM_DATA_PTR CrlToBeSigned,
        CSSM_CRL_TYPE CrlType,
        CSSM_CRL_ENCODING CrlEncoding,
        const CSSM_CERTGROUP_PTR SignerCertGroup,
        const CSSM_VERIFYCONTEXT_PTR SignerVerifyContext,
        const CSSM_FIELD_PTR SignScope,
        uint32 ScopeSize);

    CSSM_RETURN (CSSMTPI *ApplyCrlToDb) (
        CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_CSP_HANDLE CSPHandle,
        const CSSM_DL_DB_LIST_PTR DBList,
        const CSSM_DATA_PTR CrlToBeApplied,
        CSSM_CRL_TYPE CrlType,
        CSSM_CRL_ENCODING CrlEncoding,
        const CSSM_CERTGROUP_PTR SignerCert,
        const CSSM_VERIFYCONTEXT_PTR SignerVerifyContext,

    CSSM_CERTGROUP_PTR (CSSMTPI *CertGroupConstruct) (
        CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_CSP_HANDLE CSPHandle,
        const CSSM_DL_DB_LIST_PTR DBList,
        CSSM_CERTGROUP_PTR CertGroupFrag);

    CSSM_CERTGROUP_PTR (CSSMTPI *CertGroupPrune) (
        CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        const CSSM_DL_DB_LIST_PTR DBList,
        CSSM_CERTGROUP_PTR OrderedCertGroup);

    void * (CSSMTPI *PassThrough) (
        CSSM_TP_HANDLE TPHandle,
        CSSM_CL_HANDLE CLHandle,
        CSSM_CSP_HANDLE CSPHandle,
        const CSSM_DL_DB_LIST_PTR DBList,
        uint32 PassThroughId,
        const void *InputParams);
} CSSM_SPI_TP_FUNCS, *CSSM_SPI_TP_FUNCS_PTR;
```

46.3 Trust Policy Operations

The manpages for Trust Policy Operations follow on the next page.

NAME

TP_CertRequest

SYNOPSIS

```
CSSM_DATA_PTR CSSMTPI TP_CertRequest
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_SUBSERVICE_UID CSPSubserviceUid,
    const CSSM_FIELD_PTR CertFields,
    uint32 NumberOfFields,
    const CSSM_FIELD_PTR  PolicyIdentifier,
    uint32 NumberOfPolicyIdentifiers,
    CSSM_CA_SERVICES MoreServiceRequests,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

This function determines whether or not the caller is authorized to create a new certificate. Trust is determined by the UserAuthentication information and the policy domains requested by the identifiers. If authorized the specified certificate library module should be used to create a new certificate.

The initial certificate values provided by the caller can be augmented with default values defined by the selected policy domains. The complete set of initial values must be forwarded to a certification authority for processing.

The CSPSubserviceUid uniquely identifies the cryptographic service provider that must store the private key associated with the new certificate.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). These values can be obtained from the companion function supported by the certificate library module.The estimate time defines the expected certificate creation time. This time may be substantial when certificate issuance requires offline authentication procedures by the CA process. In contrast, the estimated time can be zero, meaning the certificate can be obtained immediately. After the specified time has elapsed, the caller will use the reference identifier when calling CSSM_TP_CertRetrieve, to obtain the signed certificate. The reference identifier must persist across any number of application and library executions until this two step operation is completed by the CSSM_TP_CertRetrieve function. The reference identifier becomes invalid after successful completion or final failure of the CSSM_TP_CertRetrieve function. Successful completion of the CSSM_TP_CertRetrieve function returns a certificate to the caller. Failure returns a response stating that the original request can never be fulfilled.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CSPSubserviceUid (input)
The persistent ID identifying the add-in CSPmodule where the private key is to be stored. Optionally the CL module can use this CSP to perform additional cryptographic operations or may use another default CSP for that purpose.

CertFields (input)
A pointer to an array of OID/value pairs that identify the field values as initial values in the new certificate.

NumberOfFields (input)
The number of certificate field values being input. This number specifies the number of entries in the CertFields array.

PolicyIdentifier (input/optional)
The policy identifier to be enforced when creating the Certificate template. This identifies which certificate template should be initialized and controls initialization, including the specification of required fields, and default field values. If no policy identifier is provided as input, the TP module assumes a default policy and initializes the certificate template associated with that policy.

NumberOfPolicyIdentifiers (input)
The number of policy domains in which generated certificate template should be valid. This number specifies the number of entries in the PolicyIdentifier array.

MoreServiceRequests (input/optional)
A bit mask requesting additional certificate-creation-related services from the Certificate Authority issuing the certificate. For example, the caller can request backup or archive of the certificate's private key, publication of the certificate in a certificate directory service, and request out-of-band notification of the need to renew this certificate.

UserAuthentication (input/optional)
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on—depending on the context of the request. The required format for this credential is defined by the TP and recorded in the TPSubservice structure describing this module. If the supplied information provided is insufficient, additional information can be obtained from the substructure field named MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. For example, a pass-phrase may be requested from the end-user in order to authenticate the request. If additional information is not required, this parameter must be NULL.

EstimatedTime (output)
The number of seconds estimated before the signed certificate will be ready to be retrieved. A (default) value of zero indicates that the signed certificate can be retrieved immediately via the corresponding CL_CertRetrieve function call. When the certification process cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceIdentifier (output)
A reference identifier which uniquely identifies this specific request. The handle persists across application executions until it is terminated by the successful or failed completion of the CSSM_TP_CertRetrieve function.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the unsigned certificate template. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid Trust Policy Module Handle.

CSSM_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_TP_INVALID_FIELD_POINTER
Invalid pointer input.

CSSM_TP_INVALID_OID
Invalid attribute OID for this cert type.

CSSM_TP_MEMORY_ERROR
Not enough memory.

CSSM_TP_AUTHENTICATION_FAIL
Caller is not authorized for operation.

SEE ALSO

TP_CertRetrieve, *CSSM_CL_CertRequest*, *CSSM_CL_CertRetrieve*

NAME

TP_CertRetrieve

SYNOPSIS

```
CSSM_DATA_PTR CSSMTPI TP_CertRetrieve
    (CSSM_TP_HANDLE TPHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the certificate created in response to the TP_CertRequest function call. The reference handle identifies the corresponding CertRequest call. At completion of this operation, the private key associated with the new certificate must be stored in the local CSPspecified in the corresponding call to TP_CertRequest. The TP module, CL module, and the CA process provide secure handling (via key wrapping) of the private key until it is securely stored in the local CSP.

The caller may be required to provide additional authentication information to retrieve the certificate. The format of these credentials is defined by the Policy identifiers specified in the corresponding TP_CertRequest call and the CL module used to create the certificate.

It is possible that the certificate is not ready to be retrieved when this call is made. In that case, an EstimatedTime to complete certificate creation is returned with the reference identifier and a NULL certificate pointer. The reference identifier must persist until the request either succeeds or fails. The caller must use this reference identifier again attempting to retrieve the certificate after the newly specified estimated time has elapsed.

PARAMETERS

TPHandle (input)

The handle that describes the add-in trust policy library module used to perform this function.

ReferenceIdentifier (input)

A reference identifier which uniquely identifies the CSSM_TP_CertRequest call that initiated creation of the certificate returned by this function. The identifier persists across application executions until the CSSM_CL_CertRetrieve function completes (in success or failure).

UserAuthentication (input/optional)

A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on—depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information provided is insufficient, additional information can be provided by the substructure field names MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. For example, a pass-phrase may be requested from the end-user in order to authenticate the request. If additional information is not required, this parameter must be NULL.

EstimatedTime (output)

The number of seconds estimated before the signed Certificate will be returned. A (default) value of zero indicates that the signed Certificate has been returned as a result of this call. When the certification process cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the signed certificate. If the pointer is NULL, the calling application is expected to call back after the specified EstimatedTime. If the pointer is NULL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_TP_INVALID_REFERENCE
Invalid reference identifier.

SEE ALSO

TP_CertRequest, *CSSM_CL_CertRequest*, *CSSM_CL_CertUnsign*, *CSSM_CL_CertVerify*

NAME

TP_CertGroupVerify

SYNOPSIS

```
CSSM_BOOL CSSMTPI CSSM_TP_CertGroupVerify
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_CERTGROUP_PTR CertGroupToBeVerified,
    const CSSM_VERIFYCONTEXT_PTR VerifyContext);
```

DESCRIPTION

This function determines whether the certificate is trusted. The actions performed by this function differ based on the trust policy domain. The factors include practices, procedures and policies defined by the certificate issuer.

Typically certificate verification involves the verification of multiple certificates. The first certificate in the group is the target of the verification process. The other certificates in the group are used in the verification process to connect the target certificate with one or more anchors of trust. The supporting certificates can be contained in the provided certificate group or can be stored in the data stores specified in the DBList. This allows the trust policy module to construct a certificate group and perform verification in one operation. The data stores specified by DBList can also contain certificate revocation lists used in the verification process. It is also possible to provide a data store of anchor certificates. Typically the points of trust are few in number and are embedded in the caller or in the TPM during software manufacturing or at runtime

The caller can select to be notified incrementally as each certificate is verified. The CallbackWithVerifiedCert parameter (in the verifycontext) can specify a caller function to be invoked at the end of each certificate verification, returning the verified certificate for use by the caller.

Anchor certificates are a list of implicitly trusted certificates. These include root certificates, cross certified certificates, and locally defined sources of trust. These certificates form the basis to determine trust in the subject certificate.

A policy identifier can specify an additional set of conditions that must be satisfied by the subject certificate in order to meet the trust criteria. The name space for policy identifiers is defined by the application domains to which the policy applies. This is outside of CSSM. A list of policy identifiers can be specified and the stopping condition for evaluating that set of conditions.

The evaluation and verification process can produce a list of evidence. The evidence can be selected values from the certificates examined in the verification process, entire certificates from the process or other pertinent information that forms an audit trail of the verification process. This evidence is returned to the caller after all steps in the verification process have been completed.

If verification succeeds, the trust policy module may carry out the action on the specified data or may return approval for the action requiring the caller to perform the action. The caller must consult TP module documentation outside of this specification to determine all module-specific side effects of this operation.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the subject certificate and anchor certificates. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

CSPHandle (input/optional)
The handle that describes the add-in cryptographic service provider module used to perform this function. If no cryptographic service provider handle is specified, the TP module allocates a suitable CSP.

DBList (input/optional)
A list of certificate databases containing certificates that may be used to construct the trust structure of both the subject and signer certificate group.

CertGroupToBeVerified (input)
A group of one or more certificates to be verified. The first certificate in the group is the primary target certificate for verification. Use of the subsequent certificates during the verification process is specific to the trust domain.

VerifyContext (input)
A structure containing policy elements useful in verifying certificates and their use with respect to a security policy. Optional elements in the verify context left unspecified will cause the internal default values to be used. Default values are specified in the TP module vendor release documents.

Some elements in the verification context are optional while others are mandatory. Usage semantics guidelines are as follows:

PolicyIdentifiers (input/optional)
NumberofPolicyIdentifiers (input)
AnchorCerts (input/optional)
NumberofAnchorCerts (input)
VerificationAbortOn (input/optional)
VerifyScope (input/optional)
ScopeSize (input)
Action (input/optional)
CallbackWithVerifiedCert (input/optional)
ActionData (input/optional)
Evidence (output/optional)
NumberOfEvidences (output)

RETURN VALUE

A CSSM_TRUE return value signifies that the certificate can be trusted. When CSSM_FALSE is returned, either the certificate cannot be trusted or an error has occurred. This function can also return errors specific to CSP, CL and DL modules.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_INVALID_CSP_HANDLE
Invalid handle.

CSSM_TP_INVALID_CERT_GROUP
Invalid certificate group structure.

CSSM_TP_NOT_SIGNER
Signer certificate is not signer of subject.

CSSM_TP_NOT_TRUSTED
Signature can't be trusted.

CSSM_TP_CERT_VERIFY_FAIL
Unable to verify certificate.

CSSM_TP_INVALID_ACTION_DATA
Invalid action data specified for action.

CSSM_TP_VERIFY_ACTION_FAIL
Unable to determine trust for action.

CSSM_TP_INVALID_ANCHOR
An anchor certificate could not be identified.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

CSSM_CL_CertVerify, *CSSM_TP_CertSign*

NAME

TP_CertSign

SYNOPSIS

```
CSSM_DATA_PTR CSSMTPI TP_CertSign
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_DATA_PTR CertToBeSigned,
    const CSSM_CERTGROUP_PTR SignerCertGroup,
    const CSSM_VERIFYCONTEXT_PTR SignerVerifyContext,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize)
```

DESCRIPTION

The TP module first decides whether the signer certificate group is trusted to co-sign or notarize the certificate. The signer certificate is authenticated and checked for authority to perform the signing operation. Once trust is established, the TP signs the certificate template using the signer's certificate group and the SignScope to control the signing process.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input/optional)
The handle that describes the cryptographic context for signing the certificate. This context also identifies the cryptographic service provider to be used to perform the signing operation. If this handle is not provided by the caller, the trust policy module can assume a default signing algorithm and a default CSP, but the trust policy module may be unable to unlock the caller's private key without the caller's passphrase.If the trust policy module does not assume defaults or the default CSP is not available on the local system an error occurs.

DBList (input/optional)
A list of certificate databases containing certificates that may be used to construct the trust structure of the signer certificate group.

CertToBeSigned (input)
A pointer to the CSSM_DATA structure containing the certificate to be co-signed.

SignerCertGroup (input)
A group of one or more certificates that partially or fully represent the signer for this operation. The first certificate in the group is the target certificate used to perform the signing operation. The use of all subsequent certificates in the ordering is specific to the trust domain. For example, in a hierarchical trust model subsequent members are intermediate certificates of a certificate chain.

SignerVerifyContext (input)
A structure containing policy elements useful in verifying the signer's certificate with respect to the security policy. Optional elements in the verify context left unspecified will cause the internal default values to be used. Default values are specified in the TP module

vendor release documents.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the tags of the fields to be signed. A NULL input signs a default set of fields in the certificate.

ScopeSize (input)
The number of entries in the sign scope list.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the signed certificate. If the pointer is NULL, an error has occurred. This function can also return errors specific to CSP, CL, and DL modules.

ERRORS

CSSM_TP_INVALID_CERT_GROUP
Invalid certificate group structure.

CSSM_TP_CERTIFICATE_CANT_OPERATE
Signer certificate can't sign subject.

CSSM_TP_MEMORY_ERROR
Error in allocating memory.

CSSM_TP_CERT_SIGN_FAIL
Unable to sign certificate.

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

TP_CertVerify, *CSSM_CL_CertRequest*, *CSSM_CL_CertRetrieve*

NAME

TP_CertRevoke

SYNOPSIS

```
CSSM_DATA_PTR CSSMTPI TP_CertRevoke
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_DATA_PTR OldCrl,
    CSSM_CERTGROUP_PTR CertGroupToBeRevoked,
    CSSM_CERTGROUP_PTR RevokerCertGroup,
    const CSSM_VERIFYCONTEXT_PTR RevokerVerifyContext,
    CSSM_REVOKE_REASON Reason)
```

DESCRIPTION

The TP module determines whether the revoking certificate group can revoke the subject certificate group. The revoker certificate group is first authenticated and its applicability to perform this operation is determined. Once the trust is established, the TP revokes the subject certificate by adding it to the certificate revocation list. The revoker certificate and passphrase is used to sign the resultant CRL record.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input/optional)
The handle that describes the cryptographic context for signing the CRL record. This context also identifies the cryptographic service provider to be used to perform the signing operation. If this handle is not provided by the caller, the trust policy module can assume a default signing algorithm and a default CSP. If the trust policy module does not assume defaults or the default CSP is not available on the local system an error occurs.

DBList (input/optional)
A list of certificate databases containing certificates that may be used to construct the trust structure of the subject and revoker certificate group.

OldCrl (input/optional)
A pointer to the CSSM_DATA structure containing an existing certificate revocation list. If this input is NULL, a new list is created or the operation fails.

CertToBeRevoked (input)
A group of one or more certificates that partially or fully represent the certificate to be revoked by this operation. The first certificate in the group is the target certificate. The use of subsequent certificates is specific to the trust domain. For example, in a hierarchical trust model subsequent members are intermediate certificates of a certificate chain.

RevokerCertGroup (input)
A group of one or more certificates that partially or fully represent the revoking entity for this operation. The first certificate in the group is the target certificate representing the revoker. The use of subsequent certificates is specific to the trust domain.

RevokerVerifyContext (input)
A structure containing policy elements useful in verifying certificates and their use with respect to a security policy. Optional elements in the verify context left unspecified will cause the internal default values to be used. Default values are specified in the TP module vendor release documents. This context is used to verify the revoker certificate group.

Reason (input/optional)
The reason for revoking the subject certificate.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the updated certificate revocation list. If the pointer is NULL, an error has occurred. This function can also return errors specific to CSP, CL and DL modules.

ERRORS

CSSM_TP_INVALID_CRL
Invalid CRL.

CSSM_TP_INVALID_CERTIFICATE
Invalid certificate.

CSSM_TP_CERTIFICATE_CANT_OPERATE
Revoker certificate can't revoke subject.

CSSM_TP_MEMORY_ERROR
Error in allocating memory.

CSSM_TP_CERT_REVOKE_FAIL
Unable to revoke certificate.

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_INVALID_CSP_HANDLE
Invalid handle.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

CSSM_CL_CrlAddCert

NAME

TP_CrlVerify

SYNOPSIS

```
CSSM_BOOL CSSMTPI TP_CrlVerify
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_DATA_PTR CrlToBeVerified,
    CSSM_CRL_TYPE CrlType,
    CSSM_CRL_ENCODING CrlEncoding,
    const CSSM_CERTGROUP_PTR SignerCertGroup,
    const CSSM_VERIFYCONTEXT_PTR VerifyContext)
```

DESCRIPTION

This function verifies the integrity of the certificate revocation list and determines whether it is trusted. Some of the checks that may be performed include: verifying the signatures on the signer's certificate group, establishing the authorization of the signer to issue CRLs, verification of the signature on the CRL, verifying validity period of the CRL and the date the CRL was issued, and so on.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the certificates to be verified. If no certificate library module is specified, the TP module uses an assumed CL module, if required.

CSPHandle (input/optional)
The handle referencing a Cryptographic Service Provider to be used to verify signatures on the signer's certificate and on the CRL. The TP module is responsible for creating the cryptographic context structure required to perform the verification operation. If no CSP is specified, the TP module uses an assumed CSP to perform the operations.

DBList (input/optional)
A list of handle pairs specifying a data storage library module and a data store managed by that module. These data stores can be used to store or retrieve objects (such as certificate and CRLs) related to the signer's certificate. If no DL and DB handle pairs are specified, the TP module can use an assumed DL module and an assumed data store, if required.

CrlToBeVerified (input)
A pointer to the CSSM_DATA structure containing a signed certificate revocation list to be verified.

CrlType (input)
An indicator of the type of CRL contained in the CrlToBeVerified.

CrlEncoding (input)
An indicator of the encoding of CRL contained in the CrlToBeVerified.

SignerCertGroup (input)
A group of one or more certificates that partially or fully represent the signer of the certificate revocation list. The first certificate in the group is the target certificate representing the CRL signer . Use of subsequent certificates is specific to the trust domain.

For example, in a hierarchical trust model subsequent members are intermediate certificates of a certificate chain.

VerifyContext (input)
A structure containing policy elements useful in verifying certificates and their use with respect to a security policy. Optional elements in the verify context left unspecified will cause the internal default values to be used. Default values are specified in the TP module vendor release documents. This context is used to verify the CRL and the signer certificate group.

RETURN VALUE

A CSSM_TRUE return value means the certificate revocation list can be trusted. If CSSM_FALSE is returned, an error has occurred. This function can also return errors specific to CSP, CL and DL modules.

ERRORS

CSSM_TP_INVALID_CERTIFICATE
Invalid certificate.

CSSM_TP_NOT_SIGNER
Signer certificate is not signer of CRL.

CSSM_TP_NOT_TRUSTED
Certificate revocation list can't be trusted.

CSSM_TP_CRL_VERIFY_FAIL
Unable to verify certificate.

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

CSSM_CL_CrlVerify

NAME

TP_CrlSign

SYNOPSIS

```
CSSM_DATA_PTR CSSMTPI TP_CrlSign
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_DATA_PTR CrlToBeSigned,
    CSSM_CRL_TYPE CrlType,
    CSSM_CRL_ENCODING CrlEncoding,
    const CSSM_CERTGROUP_PTR SignerCertGroup,
    const CSSM_VERIFYCONTEXT_PTR SignerVerifyContext,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize)
```

DESCRIPTION

The TP module decides whether the signer certificate is trusted to sign the entire certificate revocation list. The signer certificate group is first authenticated and its applicability to perform this operation is determined. Once the trust is established, this operation signs the entire certificate revocation list. Individual records within the certificate revocation list were signed when they were added to the list.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input/optional)
The handle that describes the cryptographic context for signing the CRL. This context also identifies the cryptographic service provider to be used to perform the signing operation. If this handle is not provided by the caller, the trust policy module can assume a default signing algorithm and a default CSP. If the trust policy module does not assume defaults or the default CSP is not available on the local system an error occurs.

DBList (input/optional)
A list of certificate databases containing certificates that may be used to construct the trust structure of the signer certificate group.

CrlToBeSigned (input)
A pointer to the CSSM_DATA structure containing the certificate revocation list to be signed.

CrlType (input)
An indicator of the type of CRL contained in the CrlToBeSigned.

CrlEncoding (input)
An indicator of the encoding of CRL contained in the CrlToBeSigned.

SignerCertGroup (input)
A group of one or more certificates that partially or fully represent the signer for this operation. The first certificate in the group is the target certificate representing the signer. Use of subsequent certificates is specific to the trust domain. For example, in a hierarchical

trust model subsequent members are intermediate certificates of a certificate chain.

SignerVerifyContext (input)
A structure containing policy elements useful in verifying certificates and their use with respect to a security policy. Optional elements in the verify context left unspecified will cause the internal default values to be used. Default values are specified in the TP module vendor release documents.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the tags of the fields to be signed. A NULL input signs a default set of fields in the certificate revocation list.

ScopeSize (input)
The number of entries in the sign scope list.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the signed certificate revocation list. If the pointer is NULL, an error has occurred. This function can also return errors specific to CSP, CL and DL modules.

ERRORS

CSSM_TP_INVALID_CERTIFICATE
Invalid certificate.

CSSM_TP_CERTIFICATE_CANT_OPERATE
Signer certificate can't sign certificate revocation list.

CSSM_TP_MEMORY_ERROR
Error in allocating memory.

CSSM_TP_CRL_SIGN_FAIL
Unable to sign certificate revocation list.

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

CSSM_CL_CrlSign

NAME

TP_ApplyCrlToDb

SYNOPSIS

```
CSSM_RETURN CSSMTPI TP_ApplyCrlToDb
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    const CSSM_DATA_PTR CrlToBeApplied,
    CSSM_CRL_TYPE CrlType,
    CSSM_CRL_ENCODING CrlEncoding,
    const CSSM_CERTGROUP_PTR SignerCert,
    const CSSM_VERIFYCONTEXT_PTR SignerVerifyContext)
```

DESCRIPTION

This function first determines whether the memory-resident CRL is trusted. The CRL is authenticated, its signer is verified, and its authority to update the data sources is determined. If trust is established, this function updates persistent storage to reflect entries in the certificate revocation list. This results in designating persistent certificates as revoked.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module that can be used to manipulate the CRL as it is applied to the data store and to manipulate the certificates effected by the CRL, if required. If no certificate library module is specified, the TP module uses an assumed CL module, if required. If optional, the caller will set this value to 0.

CSPHandle (input/optional)
The handle referencing a Cryptographic Service Provider to be used to verify signatures on the CRL determining whether to trust the CRL and apply it to the data store. The TP module is responsible for creating the cryptographic context structures required to perform the verification operation. If no CSP is specified, the TP module uses an assumed CSP to perform these operations. If optional, the caller will set this value to 0.

DBList (input/optional)
A list of handle pairs specifying a data storage library module and a data store managed by that module. These data stores can contain certificates that might be effected by the CRL, they may contain CRLs, or both. If no DL and DB handle pairs are specified, the TP module must use an assumed DL module and an assumed data store for this operation. If optional, the caller will set this value to NULL.

CrlToBeApplied (input)
A pointer to the CSSM_DATA structure containing a certificate revocation list to be applied to the data store.

CrlType (input)
An indicator of the type of CRL contained in the CrlToBeApplied.

CrlEncoding (input)
An indicator of the encoding of CRL contained in the CrlToBeApplied.

SignerCert (input)
A group of one or more certificates that partially or fully represent the signer of the

certificate revocation list. The first certificate in the group is the target certificate representing the signer. Use of subsequent certificates is specific to the trust domain. For example, in a hierarchical trust model subsequent members are intermediate certificates of a certificate chain.

SignerVerifyContext (input)
A structure containing policy elements useful in verifying certificates and their use with respect to a security policy. Optional elements in the verify context left unspecified will cause the internal default values to be used. Default values are specified in the TP module vendor release documents. This context is used to verify the signer's authority to sign and issue certificate revocation lists.

RETURN VALUE

A CSSM_OK return value means the certificate revocation list has been used to update the revocation status of certificates in the specified database. If CSSM_FAIL is returned, an error has occurred. This function can also return errors specific to CSP, CL, and DL modules.

ERRORS

CSSM_TP_INVALID_CRL
Invalid certificate revocation list.

CSSM_TP_NOT_TRUSTED
Certificate revocation list can't be trusted.

CSSM_TP_APPLY_CRL_TO_DB_FAIL
Unable to apply certificate revocation list on database.

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

CSSM_CL_CrlGetFirstItem, *CSSM_CL_CrlGetNextItem*, *CSSM_DL_CertRevoke*

NAME

TP_CertGroupConstruct

SYNOPSIS

```
CSSM_CERTGROUP_PTR CSSMTPI TP_CertGroupConstruct

    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    CSSM_CERTGROUP_PTR CertGroupFrag)
```

DESCRIPTION

This function builds a collection of certificates that together make up a meaningful credential for a given trust domain. For example, in a hierarchical trust domain, a certificate group is a chain of certificates from an end entity to a top level certification authority. The constructed certificate group format (such as ordering) is implementation specific. However, the subject or end-entity is always the first certificate in the group.

A partially constructed certificate group is specified in CertGroupFrag. The first certificate is interpreted to be the subject or end-entity certificate. Subsequent certificates in the CertGroupFrag structure may be used during the construction of a certificate group in conjunction with certificates found in the data stores specified in DBList. The trust policy defines the certificates that will be included in the resulting set.

The constructed certificate group can be consistent locally or globally. Consistency can be limited to the local system if locally-defined points of trust are inserted into the group.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CSPHandle (input/optional)
A handle specifying the Cryptographic Service Provider to be used to verify certificates as the certificate group is constructed. If the a CSP handle is not specified, the trust policy module can assume a default CSP. If the module cannot assume a default, or the default CSP is not available on the local system, an error occurs.

DBList (input)
A list of certificate databases containing certificates that may be used to construct the trust structure of the subject certificate group.

CertGroupFrag (input)
The first certificate in the group represents the target certificate for which a group of semantically related certificates will be assembled. Subsequent intermediate certificates can be supplied by the caller. They need not be in any particular order.

RETURN VALUE

A CSSM_CERTGROUP_PTR return value contains a pointer to a valid certificate group. When NULL is returned an error has occurred. This function can also return errors specific to CL and DL modules.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_TP_INVALID_CERTIFICATE
Invalid certificate.

CSSM_TP_CERTGROUP_NOT_FOUND
Unable to construct meaningful cert group.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

TP_CertGroupPrune, *TP_CertVerify*

NAME

TP_CertGroupPrune

SYNOPSIS

```
CSSM_CERTGROUP_PTR CSSMTPI TP_CertGroupPrune

    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    CSSM_CERTGROUP_PTR OrderedCertGroup)
```

DESCRIPTION

This function removes any locally issued anchor certificates from a constructed certificate group. The resulting certificate group can be exported to external entities.

PARAMETERS

TPHandle input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

DBList (input)
A list of certificate databases containing certificates that contain anchor certificates restricted to have local scope. These certificates are candidates for removal from the subject certificate group.

OrderedCertGroup (input)
A group of semantically related certificates. (for example, the result of CSSM_TP_CertGroupConstruct)

RETURN VALUE

A CSSM_CERTGROUP_PTR return value contains a pointer to a certificate group without local anchor certificates. When NULL is returned an error has occurred. This function can also return errors specific to CL and DL modules.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
Invalid handle.

CSSM_TP_INVALID_CERTIFICATE
Invalid certificate.

CSSM_TP_INVALID_CERT_GROUP
Invalid CertGroup construction.

CSSM_MEMORY_ERROR
Internal memory error.

CSSM_FUNCTION_NOT_IMPLEMENTED
Function not implemented.

SEE ALSO

TP_CertGroupConstruct, *TP_CertVerify*

46.4 Extensibility Functions

The manpages for Extensibility Functions follow on the next page.

NAME

TP_PassThrough

SYNOPSIS

```
void * CSSMTPI TP_PassThrough
    (CSSM_TP_HANDLE TPHandle,
    CSSM_CL_HANDLE CLHandle,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_DL_DB_LIST_PTR DBList,
    uint32 PassThroughId,
    const void * InputParams)
```

DESCRIPTION

This function allows clients to call Trust Policy module-specific operations that have been exported. Such operations may include queries or services specific to the domain represented by the TP module.

PARAMETERS

TPHandle (input)
The handle that describes the add-in trust policy module used to perform this function.

CLHandle (input/optional)
The handle that describes the add-in certificate library module used to perform this function.

CSPHandle (input/optional)
The handle that describes the add-in cryptographic service provider module used to perform this function.

DBList (input/optional)
A list of certificate databases containing certificates that may be used by the pass through operation.

PassThroughId (input)
An identifier assigned by the TP module to indicate the exported function to perform.

InputParams (input/optional)
A pointer to the CSSM_DATA structure containing parameters to be interpreted in a function-specific manner by the TP module. If the passthrough function requires access to a private key located in the CSP referenced by CSPHandle, then the InputParams should contain a passphrase, callback or cryptographic context.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the output from the pass-through function. The output data must be interpreted by the calling application based on externally-defined information provided by the trust policy module vendor. If the pointer is NULL, an error has occurred. This function can also return errors specific to CSP, CL, and DL modules.

ERRORS

CSSM_INVALID_TP_HANDLE
Invalid handle.

CSSM_INVALID_CL_HANDLE
Invalid handle.

CSSM_INVALID_DL_HANDLE
Invalid handle.

CSSM_INVALID_DB_HANDLE
: Invalid handle.

CSSM_TP_INVALID_DATA_POINTER
: Invalid pointer for input data.

CSSM_TP_INVALID_ID
: Invalid pass through ID.

CSSM_TP_MEMORY_ERROR
: Error in allocating memory.

CSSM_TP_PASS_THROUGH_FAIL
: Unable to perform pass-through.

CSSM_FUNCTION_NOT_IMPLEMENTED
: Function not implemented.

CAE Specification

Part 11:
CSSM Certificate Library Interface

The Open Group

Chapter 47

Introduction

47.1 CSSM Add-In Module Overview

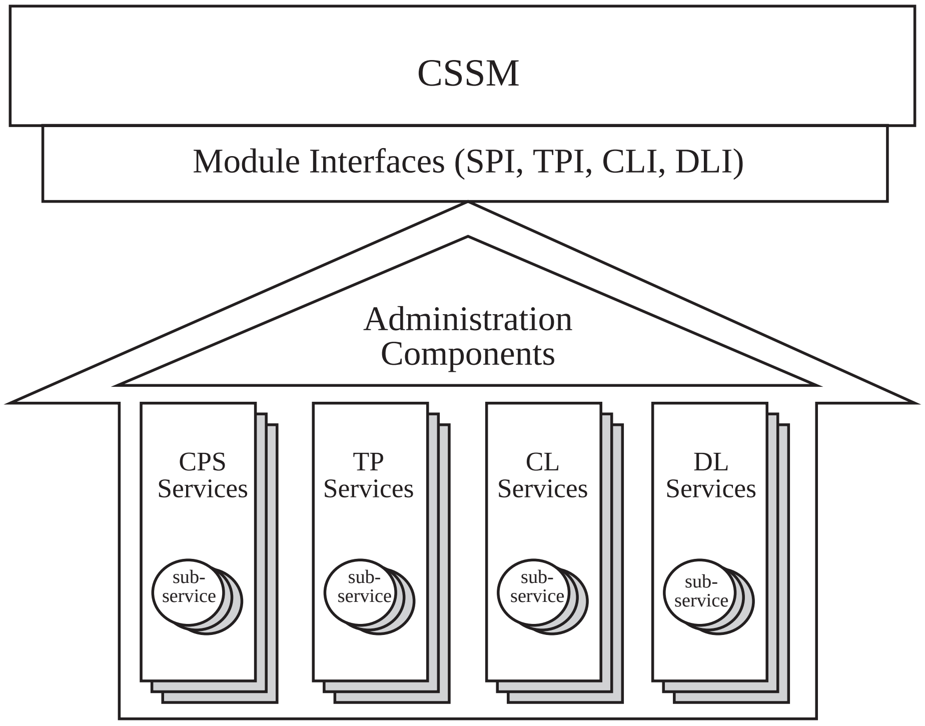

Figure 47-1 CDSA Add-In Module Structure

A CDSA add-in module is a dynamically-linkable library, composed of functions that implement some or all of the CSSM Module Interfaces. Add-in module functionality is partitioned into two areas:

- The provision of security services to applications
- Module administration

Add-in modules provide one or more categories of security services to applications. The service categories are Cryptographic Service Provider (CSP) services, Trust Policy (TP) services, Certificate Library (CL) services, and Data Storage Library (DL) services. Each security service contains one or more implementation instances, called sub-services. For a CSP service providing access to hardware tokens, a sub-service would represent a slot. For a CL service provider, a sub-service would represent a specific certificate format. These sub-services each support the module interface for their respective service categories. This documentation-part describes the module interface functions in the CL service category. More information about CSP services can be found in the *CSSM Cryptographic Service Provider Interface Specification*. More information about TP services can be found in the *CSSM Trust Policy Interface Specification*. More information about DL services can be found in the *CSSM Data Storage Library Interface Specification*.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions which allow CSSM to indicate events such as module attach and detach. In addition, as part of the attach operation, every module must be able to verify its own integrity, verify the integrity of CSSM, and register with CSSM. Detailed information about add-in module structure, administration, and interfaces can be found in the *CSSM Add-in Module Structure and Administration Specification*.

47.2 Certificate Library Overview

The primary purpose of a Certificate Library (CL) module is to perform syntactic operations on a specific certificate format, and its associated certificate revocation list (CRL) format. These manipulations encapsulate the complete life cycle of a certificate and the keypair associated with that certificate. Certificate and CRLs are related by the life cycle model and by the data formats used to represent them. For this reason, these objects should be manipulated by a single, cohesive library.

The Certificate Library encapsulates format-specific knowledge into a library which an application can access via CSSM. These libraries allow applications and add-in modules to interact with Certificate Authorities and to use certificates and CRLs for services such as signing, verification, creation and revocation without requiring knowledge of the certificate and CRL formats.

CSSM defines the general security API that all certificate libraries should provide to manipulate certificates and certificate revocation lists. The basic areas of functionality include:

- Certificate operations
- Certificate revocation list operations
- Extensibility functions

Each certificate library may implement some or all of these functions. The available functions are registered with CSSM when the module is attached. Each certificate library should be accompanied with documentation specifying supported functions, non-supported functions, and module-specific passthrough functions. It is the responsibility of the application developer to obtain and use this information when developing applications using a selected certificate library.

Certificate libraries manipulate memory-based objects only. The persistence of certificates, CRLs, and other security-related objects is an independent property of these objects. It is the responsibility of the application and/or the trust policy module to use data storage add-in modules to make objects persistent (if appropriate).

47.2.1 Certificate Life Cycle

The Certificate Library provides support for the certificate life cycle and for format- specific certificate or CRL manipulation, services which an application can access via CSSM. These libraries allow applications and add-in modules to create, sign, verify, revoke, renew, and recover certificates without requiring knowledge of certificate and CRL formats and encodings.

A certificate is a form of credential. Under current certificate models, such as X.509, SDSI, SPKI, and so on, a single certificate represents the identity of an entity and optionally associates authorizations with that entity. When a certificate is issued, the issuer includes a digital signature on the certificate. Verification of this signature is the mechanism used to establish trust in the identity and authorizations recorded in the certificate. Certificates can be signed by one or more other certificates. Root certificates are self-signed. The syntactic process of signing corresponds to establishing a trust relationship between the entities identified by the certificates.

The certificate life cycle is presented in Figure 47-2. It begins with the registration process. During registration, the authenticity of a user's identity is verified. This can be a two part process beginning with manual procedures requiring physical presence followed by backoffice procedures to register results for use by the automated system. The level of verification associated with the identity of the individual will depend on the Security Policy and Certificate Management Practice Statements that apply to the individual who will receive a certificate and the domain in which that certificate will be issued and used.

After registration, keying material is generated and a certificate is created. Once the private key material and public key certificate are issued to a user and backed up if appropriate, the active phase of the certificate management life cycle begins.

The active phase includes:

- Retrieval—retrieving a certificate from a remote repository such as an X.500 directory
- Verification—verifying the validity dates, signatures on a certificate and revocation status
- Revocation—asserting that a previously legitimate certificate is no longer a valid certificate
- Recovery—when an end-user can no longer access encryption keys (for example, because they have forgotten their password)
- Update—issuing a new public/private key pair when a legitimate pair has or will expire soon

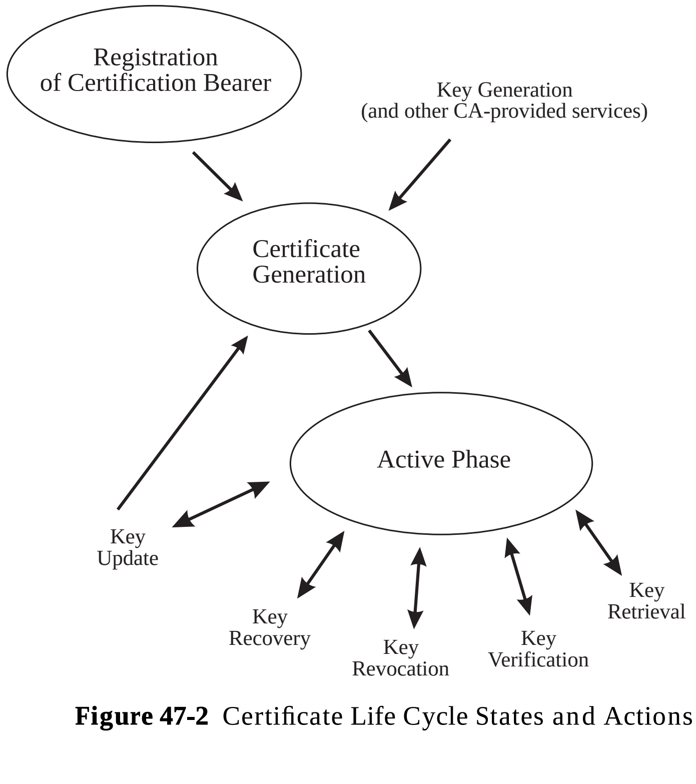

Figure 47-2 Certificate Life Cycle States and Actions

Chapter 48

Certificate Library Interface

48.1 Overview

The Certificate Library Interface (CLI) specifies the functions that a certificate library may make available to applications via CSSM in order to support certificate and certificate revocation list (CRL) formats. These functions mirror the CSSM API for certificates and certificate revocation lists. They include the basic areas of functionality expected of a certificate library: certificate operations, certificate revocation list operations, extensibility functions, and module management functions. The certificate library developer may choose to implement some or all of these CLI functions. The available functions will be made known to CSSM at attach time when it receives the certificate library's function table. In the function table, any unsupported function will have a NULL function pointer. It is the responsibility of the certificate library module developer to make its certificate format and general functionality known to application developers.

Certificate operations fall into four general areas:

- **Certificate Authority requests**—These operations include requesting certificate registration forms, requesting certificate issuance, and recovering certificates and their associated key pairs. The certificate library is expected to encapsulate the format and mechanisms required to communicate with the Certificate Authorities it supports.
- **Cryptographic operations**—These operations include signing a certificate and verifying the signature on a certificate. It is expected that the certificate library will determine the certificate fields to be signed or verified and will manage the interaction with a cryptographic service provider to perform the signing or verification.
- **Certificate field management**—Fields are added to a certificate when it is created. After the certificate is signed, the fields cannot be modified in any way. However, they can be queried for their values using the CSSM certificate interface.
- **Certificate format translation**—In the heterogeneous world of multiple certificate formats, CL modules may want to provide the service of translating between certificate formats. This translation would involve mapping the fields from one certificate format into another certificate format, while maintaining the original format for integrity verification purposes. For example, an X509V1 certificate may be exported to a SDSI format or imported into an X509V3 certificate, but the original data and signature must somehow be maintained. The supported import and export types are registered with CSSM as part of CL installation.

To support new certificate types and new uses of certificates, the sign and verify operations in the Certificate Library Interface support a scope parameter. The scope parameter enables an application to sign a portion of the certificate, namely the fields identified by the scope. This enables future certificate models, which are expected to allow field signing. CL modules that support existing certificate formats, such as X.509 Version 1, which sign and verify a pre-defined portion of the certificate, will ignore this parameter.

The CL module's certificate format is exposed via its fields. These fields will consist of tag/value pairs, where the tag is an object identifier (OID). These OIDs reference specific data types or data structures within the certificate or CRL. OIDs are defined by the certificate library developer at a granularity appropriate for the expected usage of the CL.

Operations on certificate revocation lists are comprised of Certificate Authority requests, cryptographic operations and field management operations on the CRL as a whole, and on individual revocation records. The entire CRL can be signed or verified. This will ensure the integrity of the CRL's contents as it is passed between systems. Individual revocation records may be signed when they are revoked and verified when they are queried, as determined by the CL Module. Certificates may be revoked and unrevoked by adding or removing them from the CRL at any time prior to its being signed. CRLs may be requested from the signing Certificate Authority. The contents of the CRL can be queried for all of its revocation records, specific certificates, or individual CRL fields.

A pass-through function is included in the Certificate Library Interface to allow certificate libraries to expose additional services beyond what is currently defined in the CSSM API. These services should be syntactic in nature, meaning that they should be dependent on the data format of the certificates and CRLs manipulated by the library. CSSM will pass an operation identifier and input parameters from the application to the appropriate certificate library. Within the CL_PassThrough function in the certificate library, the input parameters will be interpreted and the appropriate operation performed. The certificate library developer is responsible for making known to the application the identity and parameters of the supported pass-through operations.

48.1.1 Certificate Operations

This section provides a more detailed look at the functions that compose the certificate operations in the CLI. It gives a high-level overview of each function's expected operation, its parameter definitions where necessary, and potential differences between CL module implementations.

CL_CertRequest( )
: This function will submit a certificate creation request to a Certificate Authority (CA) process. A certificate template must be provided to specify the initial values for the certificate. As the certificate issuer, the CA process may add default field values prior to signing the new certificate. The private key associated with the certificate will be stored in the CSP module identified by the caller. This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimated time defines the expected certificate creation time, after which the caller must use CL_CertRetrieve, with the reference identifier, to obtain the signed certificate.

CL_CertRetrieve( )
: This function returns the certificate created in response to a CL_CertRequest function call. A reference identifier denotes the corresponding CL_CertRequest call. The caller may be required to provide additional authentication information to retrieve the certificate. This function returns the signed certificate and stores the associated private key (generated locally or remotely) in the CSP specified in CL_CertRequest.If the CA requires additional time prior to certificate retrieval, this function will return an updated EstimatedTime parameter.

CL_RegistrationFormRequest( )
: This function returns a blank registration form from a Registration Authority (RA) process. The RA process can be local or remote. The CL module incorporates knowledge of the name, location, and interface protocol for communication with the RA.

CL_CertMultiSignRequest( )
: This function submits a request to a Certificate Authority (CA) process to add one or more signatures to an existing certificate. The signing operation may be performed locally or remotely. The SignScope parameter defines the set of certificate fields that are to be

included in the signing process. This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimated time defines the expected signing time, after which the caller must use CL_CertMultiSignRetrieve, with the reference identifier, to obtain the multiply-signed certificate.

CL_CertMultiSignRetrieve( )
This function returns the multiply-signed certificate created in response to a CL_CertMultiSignRequest function call. A reference handle identifies the corresponding CL_CertMultiSignRequest. If the CA requires additional time prior to certificate retrieval, this function will return an updated EstimatedTime parameter.

CL_CertRecoveryRequest( )
This function submits a certificate recovery request to a Certificate Authority (CA) process (or other trusted backup facility) to prepare for the recovery of a set of certificates and their associated private keys. The caller can specify one or more certificate field values to limit the set of certificates selected for potential recovery. This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimated time defines the expected certificate recovery time, after which the caller must use CL_CertRecoveryRetrieve, with the reference identifier, to obtain the set of recovered certificates from the CA process.

CL_CertRecoveryRetrieve( )
This function obtains the set of certificates recovered in response to a CL_CertRecoveryRequest call. A reference identifier denotes the corresponding CL_CertRecoveryRequest. The caller may be required to provide additional authentication information to recover the certificates. If the CA requires additional time prior to certificate recovery, this function will return an updated EstimatedTime parameter.

CL_CertRecover( )
This function returns a certificate from a cache of certificates obtained by the CL_CertRecoveryRetrieve function. The certificate to be retrieved is specified by the CacheIndex parameter, which is a simple counter from one to the number of certificates in the cache. This function has no effect on the private key associated with the recovered certificate. Recovery of the private key can be performed using the function CL_CertKeyRecover.

CL_CertKeyRecover( )
This function recovers the private key associated with a certificate and securely stores that key in the specified cryptographic service provider. The key is retrieved from the cache specified by the CacheIndex parameter, which is a simple counter from one to the number of certificates in the cache. To selectively recover private keys from the cache, the function CL_CertRecover and be used to determine the appropriate CacheIndex to use when recovering the associated private key.

CL_CertAbortRecovery( )
This function terminates the iterative process of recovering certificates and their associated private keys from a cache of certificates. This function destroys all intermediate state and secret information used during the certificate and key recovery process, and must be called even if all certificates and their associated private keys have been recovered from the cache.

CL_CertVerify( )
This function will verify the signer certificate's signature on the subject certificate. The cryptographic context handle indicates the algorithm and parameters to be used for verification. If the certificate library module supports field signing, the VerifyScope parameter may be used to identify the fields that were signed.

CL_CertGetFirstFieldValue( )
: This function returns the first field in the certificate that matches the input OID. If the certificate contains more than one instance of the requested OID, the CL module will return a handle to be used to obtain the additional instances and a count of the total number of instances of this OID in the certificate. The application obtains the additional matching instances by repeated calls to CL_CertGetNextFieldValue.

CL_CertGetNextFieldValue( )
: This function returns the next field that matched the OID given in the CL_CertGetFirstFieldValue function. It will only be supported by certificate library modules that allow multiple instances of an OID in a single certificate.

CL_CertAbortQuery( )
: This function releases the handle that was assigned by the CL_CertGetFirstFieldValue function to identify the results of a certificate query. It will only be supported by certificate library modules that allow multiple instances of an OID in a single certificate.

CL_CertGetKeyInfo( )
: This function retrieves the public key information stored in the certificate. In most certificate formats this includes multiple fields, but it may not include all of the fields defined by the CSSM_KEY data structure. Each CL module is responsible for making known which portions of the CSSM_KEY data structure will be returned.

CL_CertGetAllFields( )
: This function returns a list of all the fields in the input certificate, as described by their OID/value pairs.

CL_CertImport( )
: This function translates a certificate from a foreign certificate type to the native certificate type manipulated by the CL module.

CL_CertExport( )
: This function translates a certificate from the native certificate type manipulated by the CL module into a foreign certificate type.

CL_CertDescribeFormat( )
: This function returns a list of object identifiers corresponding to the data objects composing the CL module's native certificate format.

48.1.2 Certificate Revocation List Operations

This section provides a more detailed look at the functions that compose the certificate revocation list operations in the CLI. This section gives a high-level overview of each function's expected operation, its parameter definitions where necessary, and potential differences between CL module implementations.

CL_CrlCreateTemplate( )
: This function creates a CRL in the CL module's native CRL format based on the OID/value pairs provided by the application. The CL module makes its supported OIDs available to the application via the CrlTemplate registered with CSSM and via the CL_CrlDescribeFormat function. The CL Module is responsible for indicating which fields are required to create a CRL, or which fields cannot be set using this function. The returned CRL will not be a valid CRL until it has been signed.

CL_CrlRequest( )
: This function submits a request to a Certificate Authority (CA) process to issue the most current version of a CRL of a specified name. This function returns a ReferenceIdentifier

and an EstimatedTime (specified in seconds). The estimated time defines the expected closing, signing and distribution time of the CRL, after which the caller must use the CL_CrlRetrieve, with the reference identifier, to obtain the CRL.

CL_CrlRetrieve( )
This function returns the CRL closed and issued in response to a CL_CrlRequest function call. A reference identifier denotes the corresponding CL_CrlRequest call. If the CA requires additional time prior to CRL retrieval, this function will return an updated EstimatedTime parameter.

CL_CrlSetFields( )
This function sets the fields of an existing CRL to new values, based on the OID/value pairs provided by the application. The CL Module is responsible for indicating any set of fields that must be or cannot be set using this function, and for specifying module-specific behavior such as overwriting existing fields, modifying extensions, or modifying CRL records. This operation is valid only if the CRL has not been closed by the process of signing the CRL (that is, execution of the function CSSM_CL_CrlSign). Once the CRL has been signed, fields cannot be changed.

CL_CrlAddCert( )
This function revokes the input certificate by adding a record representing the certificate to the CRL. The values for the new entry in the CRL are specified by a list of OID/value input pairs provided by the application. A CL module that supports field signing would use the revoker's certificate to sign the new record. The updated CRL is returned to the calling application. The CL Module is responsible for indicating any set of fields that must be or cannot be set using this function. This operation is valid only if the CRL has not been closed by the process of signing the CRL (that is, execution of the function CSSM_CL_CrlSign). Once the CRL has been signed, entries cannot be added or removed.

CL_CrlRemoveCert( )
This function reinstates the input certificate by removing the record representing the certificate from the CRL. The updated CRL is returned to the calling application. This operation is valid only if the CRL has not been closed by the process of signing the CRL (that is, execution of the function CSSM_CL_CrlSign). Once the CRL has been signed, entries cannot be added or removed.

CL_CrlSign( )
This function will create a digital signature for the entire CRL using the signer's certificate. The cryptographic context handle indicates the algorithm and parameters to be used for signing. The field or fields of the CRL that should be signed will depend on the implementation of the CL module. A CL module may choose to ignore the SignScope parameter if the fields to be signed are pre-defined. A CL module that supports field signing would sign the subset of fields specified by the SignScope parameter. The CL module may refuse to sign the CRL if a pre-defined set of fields do not contain valid data. Typically, this function will be used to sign the entire CRL prior to distributing it to other systems. The signature will be used to quickly detect tampering of the CRL. CRL queries may be performed on both signed and unsigned CRLs. Once the CRL has been signed it may not be modified. This means that entries cannot be added or removed from the CRL through application of the CSSM_CL_CrlAddCert or CSSM_CL_CrlRemoveCert operations. A signed CRl can be verified, applied to a data store, and searched for values.

CL_CrlVerify( )
This function will check the signer certificate's signature on the subject CRL to determine whether the CRL has been tampered with and whether the signer's certificate was actually used to sign the CRL. The cryptographic context handle indicates the algorithm and

parameters to be used for verification. If the certificate library supports field signing on a CRL, the VerifyScope may be used to identify the fields that were signed.

CL_IsCertInCrl( )
: This function searches the CRL for a record corresponding to the input certificate.

CL_CrlGetFirstFieldValue( )
: This function returns the first field in the CRL that matches the input OID. It is likely that the CRL will support multiple instances of an OID that represents a revoked certificate record. If an application requests an OID that has multiple instances within the CRL, a results handle and a count of the number of matching instances will be returned along with the first instance of the OID. The application uses the results handle to obtain the additional matching instances by repeated calls to CL_CrlGetNextFieldValue. For example, given the OID for "revocation record", this function would return the first revocation record in the CRL. The remaining revocation records could be obtained by successive calls to CL_CrlGetNextFieldValue.

CL_CrlGetNextFieldValue( )
: This function returns the next field associated with the input results handle (obtained via an initial call to CSSM_CL_CrlGetFirstFieldValue).

CL_CrlAbortQuery( )
: This function releases a handle that was assigned by the CL_CrlGetFirstFieldValue function to identify the results of a CRL query, and allows the CL to release all intermediate state information associated with the get operation.

CL_CrlDescribeFormat( )
: This function returns a list of the object identifiers that represent the fields in the certificate revocation list format supported by the CL module.

48.1.3 Extensibility Functions

CL_PassThrough( )
: This performs the CL module-specific function indicated by the operation ID. The operation ID specifies an operation that the CL has exported for use by an application or module. Such operations should be specific to the data format of the certificates and CRLs manipulated by the CL module.

48.2 Data Structures

This section describes the data structures which may be passed to or returned from a Certificate Library function. They will be used by applications to prepare data to be passed as input parameters into CSSM API function calls which will be passed without modification to the appropriate CL. The CL is then responsible for interpreting them and returning the appropriate data structure to the calling application via CSSM. These data structures are defined in the header file **<cssmtype.h>**, distributed with CSSM.

48.2.1 CSSM_CL_HANDLE

The CSSM_CL_HANDLE is used to identify the association between an application thread and an instance of a CL module. It is assigned when an application causes CSSM to attach to a Certificate Library. It is freed when an application causes CSSM to detach from a Certificate Library. The application uses the CSSM_CL_HANDLE with every CL function call to identify the targeted CL. The CL module uses the CSSM_CL_HANDLE to identify the appropriate application's memory management routines when allocating memory on the application's behalf.

```
typedef uint32 CSSM_CL_HANDLE
```

48.2.2 CSSM_CERT_TYPE

This variable specifies the type of certificate format supported by a certificate library and the types of certificates understood for import and export. They are expected to define such well-known certificate formats as X.509 Version 3 and SDSI as well as custom certificate formats. The list of enumerated values can be extended for new types by defining a label with an associated value greater than CSSM_CL_CUSTOM_CERT_TYPE.

```
typedef enum cssm_cert_type {
    CSSM_CERT_UNKNOWN =  0x00,
    CSSM_CERT_X_509v1 =  0x01,
    CSSM_CERT_X_509v2 =  0x02,
    CSSM_CERT_X_509v3 =  0x03,
    CSSM_CERT_PGP =      0x04,
    CSSM_CERT_SPKI =     0x05,
    CSSM_CERT_SDSIv1 =   0x06,
    CSSM_CERT_Intel =    0x08,
    CSSM_CERT_X_509_ATTRIBUTE = 0x09, /* X.509 attribute cert */
    CSSM_CERT_X9_ATTRIBUTE = 0x0A,    /* X9 attribute cert */
    CSSM_CERT_LAST =     0x7FFF,
} CSSM_CERT_TYPE, *CSSM_CERT_TYPE_PTR;

/* Applications wishing to define their own custom certificate
 * type should create a random uint32 whose value is greater than
 * the CSSM_CL_CUSTOM_CERT_TYPE */
#define CSSM_CL_CUSTOM_CERT_TYPE  0x08000
```

48.2.3 CSSM_CERT_ENCODING

This variable specifies the certificate encoding format supported by a certificate library.

```
typedef enum cssm_cert_encoding {
    CSSM_CERT_ENCODING_UNKNOWN =  0x00,
    CSSM_CERT_ENCODING_CUSTOM  =  0x01,
    CSSM_CERT_ENCODING_BER     =  0x02,
    CSSM_CERT_ENCODING_DER     =  0x03,
    CSSM_CERT_ENCODING_NDR     =  0x04,
} CSSM_CERT_ENCODING, *CSSM_CERT_ENCODING_PTR;
```

48.2.4 CSSM_CERT_BUNDLE_TYPE

This enumerated type lists the signed certificate aggregates that are considered to be certificate bundles.

```
typedef enum cssm_cert_bundle_type {
    CSSM_CERT_BUNDLE_UNKNOWN =  0x00,
    CSSM_CERT_BUNDLE_CUSTOM  =  0x01,
    CSSM_CERT_BUNDLE_PKCS7_SIGNED_DATA =  0x02,
    CSSM_CERT_BUNDLE_PKCS7_SIGNED_ENVELOPED_DATA =  0x03,
    CSSM_CERT_BUNDLE_PKCS12 =  0x04,
    CSSM_CERT_BUNDLE_PFX =  0x05,
    CSSM_CERT_BUNDLE_LAST = 0x7FFF
} CSSM_CERT_BUNDLE_TYPE;

/* Applications wishing to define their own custom certificate
 * BUNDLE type should create a random uint32 whose value
 * is greater than the CSSM_CL_CUSTOM_CERT_BUNDLE_TYPE */

#define CSSM_CL_CUSTOM_CERT_BUNDLE_TYPE  0x8000
```

48.2.5 CSSM_CERT_BUNDLE_ENCODING

This enumerated type lists the encoding methods applied to the signed certificate aggregates that are considered to be certificate bundles.

```
typedef enum cssm_cert_bundle_encoding {
    CSSM_CERT_BUNDLE_ENCODING_UNKNOWN =  0x00,
    CSSM_CERT_BUNDLE_ENCODING_CUSTOM  =  0x01,
    CSSM_CERT_BUNDLE_ENCODING_BER     =  0x02,
    CSSM_CERT_BUNDLE_ENCODING_DER     =  0x03
} CSSM_CERT_BUNDLE_ENCODING;
```

48.2.6 CSSM_CERT_BUNDLE_HEADER

This structure defines a bundle header, which describes the type and encoding of a certificate bundle.

```
typedef struct cssm_cert_bundle_header {
    CSSM_CERT_BUNDLE_TYPE BundleType;
    CSSM_CERT_BUNDLE_ENCODING BundleEncoding;
} CSSM_CERT_BUNDLE_HEADER, *CSSM_CERT_BUNDLE_HEADER_PTR;
```

Definition

BundleType
A descriptor which identifies the format of the certificate aggregate.

BundleEncoding
A descriptor which identifies the encoding of the certificate aggregate.

48.2.7 CSSM_CERT_BUNDLE

This structure defines a certificate bundle, which consists of a descriptive header and a pointer to the opaque bundle. The bundle itself is a signed opaque aggregate of certificates.

```
typedef struct cssm_cert_bundle {
    CSSM_CERT_BUNDLE_HEADER BundleHeader;
    CSSM_DATA Bundle;
} CSSM_CERT_BUNDLE, *CSSM_CERT_BUNDLE_PTR;
```

BundleHeader
Information describing the format and encoding of the bundle contents.

Bundle
A signed opaque aggregate of certificates.

48.2.8 CSSM_OID

The object identifier (OID) is used to hold an identifier for the data types and data structures which comprise the fields of a certificate or CRL. The underlying representation and meaning of the identifier is defined by the certificate library module. Popular representations include:

- A character string in a character set native to the platform
- A DER encoded X.509 OID that must be parsed
- An S-expression that must be evaluated
- An enumerated value that is defined in header files supplied by the CLM

```
typedef CSSM_DATA CSSM_OID, *CSSM_OID_PTR
```

48.2.9 CSSM_CRL_TYPE

This structure represents the type of format used for revocation lists.

```
typedef enum cssm_crl_type {
    CSSM_CRLTYPE_UNKNOWN,
    CSSM_CRLTYPE_X_509v1,
    CSSM_CRLTYPE_X_509v2,
} CSSM_CRL_TYPE, *CSSM_CRL_TYPE_PTR
```

48.2.10 CSSM_CRL_ENCODING

This structure represents the encoding format used for revocation lists.

```
typedef enum cssm_crl_encoding {
    CSSM_CRL_ENCODING_UNKNOWN,
    CSSM_CRL_ENCODING_CUSTOM,
    CSSM_CRL_ENCODING_BER,
    CSSM_CRL_ENCODING_DER,
    CSSM_CRL_ENCODING_BLOOM
} CSSM_CRL_ENCODING, *CSSM_CRL_ENCODING_PTR;
```

48.2.11 CSSM_FIELD

This structure contains the OID/value pair for any item that can be identified by an OID. A certificate library module uses this structure to hold an OID/value pair for fields in a certificate or CRL.

```
typedef struct cssm_field {
    CSSM_OID FieldOid;
    CSSM_DATA FieldValue;
}CSSM_FIELD, *CSSM_FIELD_PTR
```

Definition

FieldOid
The object identifier which identifies the certificate or CRL data type or data structure.

FieldValue
A CSSM_DATA type which contains the value of the specified OID in a contiguous block of memory.

48.2.12 CSSM_ESTIMATED_TIME_UNKNOWN

The value used by an authority or process to indicate that an estimated completion time cannot be determined.

```
#define CSSM_ESTIMATED_TIME_UNKNOWN -1
```

48.2.13 CSSM_CA_SERVICES

This bit mask defines the additional certificate-creation-related services that an issuing Certificate Authority (CA) can offer. Such services include (but are not limited to) archiving the certificate and keypair, publishing the certificate to one or more certificate directory services, and sending automatic, out-of-band notifications of the need to renew a certificate. A CA may offer any subset of these services. Additional services can be defined over time.

```
typedef uint32 CSSM_CA_SERVICES;
/*  bit masks for additional CA services at cert enroll  */
#define CSSM_CA_KEY_ARCHIVE  0x0001 /* archive cert and keys */
#define CSSM_CA_CERT_PUBLISH  0x0002 /* cert in directory
                                        service */
#define CSSM_CA_CERT_NOTIFY_RENEW 0x0004 /* notify at renewal
                                        time */
#define CSSM_CA_CERT_DIR_UPDATE 0x0008 /* multi-signed cert to
                                        dir svc */
#define CSSM_CA_CRL_DISTRIBUTE 0x0010 /* push CRL to everyone */
```

48.2.14 CSSM_CL_CA_CERT_CLASSINFO

This structure describes a class of certificates issued by a given CA.

```
typedef struct cssm_cl_ca_cert_classinfo {
   CSSM_STRING CertClassName; /* Name of the class of cert */
   CSSM_DATA CACert;   /* CA cert used to sign this cert class */
} CSSM_CL_CA_CERT_CLASSINFO, *CSSM_CL_CA_CERT_CLASSINFO_PTR;
```

Definition

CertClassName
The CA's description of the certificate class, including its name.

CACert
The CA's cert used to sign issued certificates of this cert class.

48.2.15 CSSM_CL_CA_PRODUCTINFO

This structure holds product information about a backend Certificate Authority (CA) that is accessible to the CL module. The CL module vendor is not required to provide this information, but may choose to do so. For example, a CL module that implements upstream protocols to a particular type of commercial CA can record information about that CA service in this structure.

```
typedef struct cssm_cl_ca_productinfo {
    CSSM_VERSION StandardVersion;    /* Version of standard this
                                        product conforms to */
    CSSM_STRING StandardDescription; /* Desc of standard this
                                        product conforms to */
    CSSM_VERSION ProductVersion;     /* Version of wrapped
                                        product/library */
    CSSM_STRING ProductDescription;  /* Description of wrapped
                                        product/library */
    CSSM_STRING ProductVendor;       /* Vendor of wrapped
                                        product/library */
    CSSM_NET_PROTOCOL NetworkProtocol;  /* The network protocol
                               supported by the CA service */
    CSSM_CERT_TYPE CertType;         /* Type of certs
                                        supported by CA */
    CSSM_CERT_ENCODING CertEncoding;  /* Cert encoding supported
                                         by CA */
    CSSM_CRL_TYPE CrlType;  /* CRL type supported by CA */

    CSSM_CRL_ENCODING CrlEncoding;  /* CRL encoding supported by CA */

    CSSM_CA_SERVICES AdditionalServiceFlags; /* Mask of
                   additional services a caller can request */
    uint32 NumberOfCertClasses;      /* Number of different
                     cert types or classes the CA can issue */
    CSSM_CL_CA_CERT_CLASSINFO_PTR CertClasses /* Information
                   about the cert classes issued by this CA */
} CSSM_CL_CA_PRODUCTINFO, *CSSM_CL_CA_PRODUCTINFO_PTR;
```

Definition

StandardVersion
If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription
If this product conforms to an industry standard, this is a description of that standard.

ProductVersion
Version number information for the actual product version used in this version of the DL module.

ProductDescription
A string describing the product.

ProductVendor
The name of the product vendor.

NetworkProtocol
The name of the network protocol.

CertType
An enumerated value specifying the certificate and type that the CA manages.

CertEncoding
An enumerated value specifying the certificate encoding that the CA manages

CrlType
An enumerated value specifying the CRL type that the CA manages

CrlEncoding
An enumerated value specifying the CRL encoding that the CA manages

AdditionalServiceFlags
A bit mask indicating the additional services a caller can request from a CA (as side effects and in conjunction with other service requests.

NumberOfCertClasses
The number of classes or levels of Certificates managed by this CA.

CertClasses
An array of information about the classes of certificates supported by this CA.

48.2.16 CSSM_CL_ENCODER_PRODUCTINFO

This structure holds product information about embedded products that a CL module uses to provide its services. The CL module vendor is not required to provide this information, but may choose to do so. For example, a CL module that manipulates X.509 certificates may embed a third party tool that parses, encodes, and decodes those certificates. The CL module vendor can describe such embedded products using this structure.

```
typedef struct cssm_cl_encoder_productinfo {
    CSSM_VERSION StandardVersion; /* Ver of standard the product
                                    conforms to */
    CSSM_STRING StandardDescription; /* Desc of standard this
                                    product conforms to */
    CSSM_VERSION ProductVersion;  /* Version of wrapped product
                                    or library */
    CSSM_STRING ProductDescription;  /* Description of wrapped
```

```
                                          product or library */
    CSSM_STRING ProductVendor;       /* Vendor of wrapped product
                                          or library */
    CSSM_CERT_TYPE CertType; /* Type of certs supported by encoder */

    CSSM_CRT_TYPE CrlType; /* Type of CRLs supported by encoder */

    uint32 ProductFlags;             /* Mask of selectable encoder
                                 features actually used by the CL */
} CSSM_CL_ENCODER_PRODUCTINFO, *CSSM_CL_ENCODER_PRODUCTINFO_PTR;
```

Definition

StandardVersion
If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription
If this product conforms to an industry standard, this is a description of that standard.

ProductVersion
Version number information for the actual product version used in this version of the DL module.

ProductDescription
A string describing the product.

ProductVendor
The name of the product vendor.

CertType
An enumerated value specifying the certificate type that the encoder processes (if limited to one type).

CrlType
An enumerated value specifying the CRL type that the encoder processes (if limited to one type).

ProductFlags
A bit mask indicating any selectable features of the embedded product that the CL module selected to use.

48.2.17 CSSM_CL_WRAPPEDPRODUCTINFO

This structure lists the set of embedded products and the CA service used by the CL module to implement its services. The CL module is not required to provide any of this information, but may choose to do so.

```
typedef struct cssm_cl_wrappedproductinfo {
         /* List of encode/decode/parse libraries embedded
                                             in the CL module */
    CSSM_CL_ENCODER_PRODUCTINFO_PTR EmbeddedEncoderProducts;
         /* library product description */
    uint32 NumberOfEncoderProducts;
         /* number of encode/decode/parse libraries used in CL */
         /* List of CAs accessible to the CL module */
    CSSM_CL_CA_PRODUCTINFO_PTR AccessibleCAProducts;
```

```
          /* CA product description*/
    uint32 NumberOfCAProducts;
          /* Number of accessible CAs */
} CSSM_CL_WRAPPEDPRODUCTINFO, *CSSM_CL_WRAPPEDPRODUCTINFO_PTR;
```

Definition

EmbeddedEncoderProducts
An array of structures that describe each embedded encoder product used in this CL module implementation.

NumberOfEncoderProducts
A count of the number of distinct embedded certificate encoder products used in the CL module implementation.

AccessibleCAProducts
An array of structures that describe each type of Certificate Authority accessible through this CL module implementation.

NumberOfCAProducts
A count of the number of distinct CA products described in the array AccessibleCAProducts.

48.2.18 CSSM_CLSUBSERVICE

This structure contains the static information that describes a certificate library sub- service. This information is stored in the CSSM registry when the CL module is installed with CSSM. CSSM checks the integrity of the CL module description before using the information. A certificate library module may implement multiple types of services and organize them as sub-services. For example, a CL module supporting X.509 encoded certificates may organize its implementation into three sub-services, one for X.509 version 1, a second for X.509 version 2, and a third for X.509 version 3. Most certificate library modules will implement exactly one sub-service.

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo( ) and specifying the certificate library module GUID.

```
typedef struct cssm_clsubservice {
    uint32 SubServiceId;
    CSSM_STRING Description;
    CSSM_CERT_TYPE CertType;
    CSSM_CERT_ENCODING CertEncoding;
    uint32 NumberOfBundleInfos;
    CSSM_CERT_BUNDLE_HEADER_PTR BundleInfo; /* first is default
                                                          value */
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
    uint32 NumberOfTemplateFields;
    CSSM_OID_PTR CertTemplate;
    uint32 NumberOfTranslationTypes;
    CSSM_CERT_TYPE_PTR CertTranslationTypes;
    CSSM_CL_WRAPPEDPRODUCTINFO WrappedProduct;
} CSSM_CLSUBSERVICE, *CSSM_CLSUBSERVICE_PTR;
```

Definition

SubServiceId
A unique, identifying number for the sub-service described in this structure.

Description
A string containing a description name or title for this sub-service.

CertType
An identifier for the type of certificate.

CertEncoding
An identifier for the certificate encoding format.

NumberOfBundleInfos
The number of distinct bundle type/encoding pairs supported by the certificate library module.

BundleInfo
A pointer to a list of bundle header structures. Each structure defines a bundle type and encoding supported by the certificate library module. The first bundle header is the default for the library.

AuthenticationMechanism
An enumerated value defining the credential format accepted by the CL module. Authentication credentials may be required when requesting certificate creation or other CL functions. Presented credentials must be of the required format.

NumberOfTemplateFields
The number of certificate template fields. This number also indicates the length of the CertTemplate array.

CertTemplate
A pointer to an array of tag/value pairs which identify the field values of a certificate.

NumberOfTranslationTypes
The number of certificate types that this certificate library add-in module can import and export. This number also indicates the length of the CertTranslationTypes array.

CertTranslationTypes
A pointer to an array of certificate types. This array indicates the certificate types that can be imported into and exported from this certificate library module's native certificate type.

WrappedProduct
Descriptions of the set of embedded products used by this module and the CA services available via this module.

48.2.19 Certificate Operations

This section describes the function prototypes and error codes expected for the certificate functions in the CLI. The functions will be exposed to CSSM via a function table, so the function names may vary at the discretion of the certificate library developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications. The error codes given in this section constitute the generic error codes that are defined by CSSM for use by all certificate libraries in describing common error conditions. A certificate library may also define and return vendor-specific error codes. Applications must consult vendor supplied documentation for the specification and description of any error codes defined outside of this specification.

NAME

CL_CertRequest

SYNOPSIS

```
CSSM_RETURN CSSMCLI CL_CertRequest
    (CSSM_CL_HANDLE CLHandle,
    CSSM_SUBSERVICE_UID CSPSubserviceUid,
    const CSSM_FIELD_PTRSubjectCertTemplate,
    uint32 NumberOfFields,
    const CSSM_DATA_PTR CACert,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize,
    const CSSM_NET_ADDRESS_PTR CALocation,
    CSSM_CA_SERVICES MoreServiceRequests,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

This function submits a certificate creation request to a Certificate Authority (CA) process identified by the SignerCert. The CA process may be local or remote. The CL module incorporates knowledge of the name, location, and interface protocol for communicating with the CA. The certificate fields provide the initial values for the certificate. The CA can add other default values known only to the CA.

As the certificate issuer, the CA process signs the new certificate. If the signer's certificate is not specified in this function, the CA assumes a default signing certificate it uses to issue certificates. The SignScope defines the set of certificate fields to be included in the signing process. The signing operation may be performed locally or remotely. The caller specifies the CSP to be used for storing the private key. This same CSP may optionally be used for cryptographic operations. The CL module is responsible for creating and destroying all cryptographic contexts required to perform these operations.

The caller can request additional certificate-creation-related services from the CA. These requests are designated by the MoreServiceRequests bit mask. CSSM-defined bit masks allow the caller to request certificate and key archival, certificate registration with a directory service, certificate renewal notification, and other services. CAs are not required to provide such services. The CL module works with the CA process to provide the requested services.

The caller is required to provide authentication information so the CA process can determine whether the caller is authorized to request a certificate. The specific format of the credential is specified by the CL module. The caller can query the CL Module Info structure to obtain this information.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimate time defines the expected certificate creation time. This time may be substantial when certificate issuance requires offline authentication procedures by the CA process. In contrast, the estimated time can be zero, meaning the certificate can be obtained immediately. After the specified time has elapsed, the caller must use the CL module interface CSSM_CL_CertRetrieve, with the reference identifier, to obtain the signed certificate.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CSPSubserviceUid (input)
The identifier which uniquely describes the add-in CSP module subservice where the private key is to be stored. Optionally, the CL module can use this CSP to perform additional cryptographic operations or may use another default CSP for that purpose.

SubjectCertTemplate (input)
A pointer to an array of OID / Value pairs providing the initial values for the certificate.

NumberOfFields (input)
The number of certificate field values being input. This number specifies the number of entries in the SubjectCertTemplate array.

CACert (input / optional)
A pointer to the CSSM_DATA structure containing the desired Certification Authority's signing certificate. If the CACert is NULL, the CL module or the CA process can provide a default signing certificate.

SignScope (input / optional)
A pointer to the CSSM_FIELD array containing the OID / value pairs specifying the certificate fields to be signed. When the input value is NULL, the CL assumes and includes a default set of certificate fields in the signing process.

ScopeSize (input)
The number of entries in the sign scope list. If no signing scope is specified, then the scope size must be zero.

CALocation (input / optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on the SignerCert input parameter or can assume a default CA process location. If SignerCert is not specified and a default cannot be assumed, the request cannot be initiated and the operation fails.

MoreServiceRequests (input / optional)
A bit mask requesting additional certificate-creation-related services from the Certificate Authority issuing the certificate. CSSM-defined bit masks allow the caller to request backup or archival of the certificate's private key, publication of the certificate in a certificate directory service, request out-of-band notification of the need to renew this certificate, or request other services.

UserAuthentication (input / optional)
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on, depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information is insufficient, additional information can be provided by the substructure field MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If authentication information is not required, this parameter must be NULL.

EstimatedTime (output)
The number of seconds estimated before the signed certificate will be ready to be retrieved. A (default) value of zero indicates that the signed certificate can be retrieved immediately via the corresponding CL_CertRetrieve function call. When the certification process cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceHandle (output)
A reference identifier which uniquely identifies this specific request. The identifier persists across application executions until it is terminated by the successful or failed completion of the CSSM_CL_CertRetrieve function.

RETURN VALUE

A CSSM_OK return value signifies the requested operation has proceeded and that CL_CertRetrieve should be called (after the specified amount of time) in order to retrieve the results. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_INVALID_SIGNER_CERTIFICATE
Revoked or expired signer certificate.

CSSM_CL_INVALID_SCOPE
Invalid scope.

CSSM_AUTHENTICATION_FAIL
Invalid/unauthorized credential.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_CERT_REQUEST_FAIL
Unable to submit certificate creation request.

SEE ALSO

CL_CertRetrieve, *CL_CertVerify*

NAME

CL_CertRetrieve

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CertRetrieve
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_NET_ADDRESS_PTR CALocation,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the certificate created in response to the CL_CertRequest function call. The reference identifier denotes the corresponding CertRequest call. The signing operation, performed by the Certificate Authority (CA) process, may have been performed locally or remotely. In either case, the private key associated with the certificate is stored in the local CSP specified by the caller. The CL module and the CA process provide secure handling (via key wrapping) of the private key until it is securely stored in the local CSP. The CL module incorporates knowledge of the name, location, and interface protocol for communicating with the CA.

The caller may be required to provide additional authentication information to retrieve the certificate. The format of these credentials is defined by the CL module and recorded in the CLSubservice structure, which can be queried by the caller.

This function returns the signed certificate and stores the associated private key in the CSP specified in CL_CertRequest. It is possible that the certificate is not ready to be retrieved when this call is made. In that case, an EstimatedTime to complete certificate creation is returned with a NULL certificate pointer. The caller reuses the reference identifier to retrieve the certificate after the estimated time to completion has elapsed.

PARAMETERS

CLHandle (input)
: The handle that describes the add-in certificate library module used to perform this function.

ReferenceIdentifier (input)
: A reference identifier which uniquely identifies the CSSM_CL_CertRequest call that initiated creation of the certificate returned by this function. The identifier persists across application executions until the CSSM_CL_CertRetrieve function completes (in success or failure).

CALocation (input/optional)
: A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on state information associated with the ReferenceIdentifier or can assume a default CA process location. If insufficient state is associated with the ReferenceIdentifier and a default cannot be assumed, the retrieval cannot be completed and the operation fails.

UserAuthentication (input/optional)
: A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on, depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information is insufficient, additional information can be provided

by the substructure field *MoreAuthenticationData*. This field contains an immediate data value or a callback function to collect additional information from the user. If authentication information is not required, this parameter must be NULL.

EstimatedTime (output)

The number of seconds estimated before the signed Certificate will be returned. A (default) value of zero indicates that the signed Certificate has been returned as a result of this call. When the certification process cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the signed certificate. If the pointer is NULL, the calling application is expected to call back after the specified EstimatedTime. If the pointer is NULL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_IDENTIFIER
Invalid reference identifier.

CSSM_AUTHENTICATION_FAIL
Invalid/unauthorized credential for operation.

CSSM_CL_CERT_SIGN_FAIL
Unable to sign certificate.

CSSM_CL_EXTRA_SERVICE_FAIL
Unable to perform additional certificate-creation-related services.

CSSM_CL_PRIVATE_KEY_STORE_FAIL
Unable to store private key in CSP.

CSSM_CL_MEMORY_ERROR
Not enough memory.

SEE ALSO

CL_CertRequest, *CL_CertUnsign*, *CL_CertVerify*

NAME

CL_RegistrationFormRequest

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_RegistrationFormRequest
    (CSSM_CL_HANDLE CLHandle)
```

DESCRIPTION

This function returns a blank registration form from a Registration Authority (RA) process. The RA process can be local or remote. The CL module incorporates knowledge of the name, location, and interface protocol for communicating with the RA.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the blank registration form. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_UNABLE_TO_RETRIEVE_FORM
Unable to retrieve the registration form.

SEE ALSO

CL_CertRequest

NAME

CL_RegistrationFormSubmit

SYNOPSIS

```
CSSM_USER_AUTHENTICATION_PTR CSSMCLI CL_RegistrationFormSubmit
    (CSSM_CL_Handle CLHandle,
    const CSSM_DATA_PTR RegistrationForm,
    const CSSM_NET_ADDRESS_ADDR RALocation,
    const CSSM_NET_ADDRESS_ADDR CALocation)
```

DESCRIPTION

The completed registration form is submitted to a Registration Authority requesting approval for certificate generation by a Certification Authority. An authentication credential is returned. This credential can be used as the input authentication credential in a certificate request call.

PARAMETERS

CLHandle (input)
A handle for the module that will perform the operation.

RegistrationForm (input)
A pointer to the CSSM_DATA structure containing the completed registration form to be submitted to the Registration Authority and Certification Authority.

RALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the RA process. If the input is NULL, the module can assume a default RA process location. If a default cannot be assumed, the request cannot be initiated and the operation fails.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module or the Registration Authority can assume a default CA process location. If a default cannot be assumed, the request cannot be initiated and the operation fails.

RETURN VALUE

A pointer to a CSSM_USER_AUTHENTICATION credential. When NULL is returned, an error occurred or the registration form was rejected by the RA or the CA. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_RA
Unknown or unreachable Registration Authority.

CSSM_CL_NO_DEFAULT_RA
No default Registration Authority.

CSSM_CL_RA_REJECTED_FORM
RA rejected the registration form.

CSSM_CL_CA_REJECTED_FORM
CA rejected the registration form.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_FORM_SUBMIT_FAIL
Unable to submit the registration form.

SEE ALSO

CL_RegistrationFormRequest

NAME

CL_CertMultiSignRequest

SYNOPSIS

```
CSSM_RETURN CSSMCLI CL_CertMultiSignRequest
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR SubjectCert,
    const CSSM_DATA_PTR CACert,
    uint32 NumberOfCACerts,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize,
    const CSSM_NET_ADDRESS_PTR CALocation,
    CSSM_CA_SERVICES MoreServiceRequests,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    sint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

This function submits a request to a Certificate Authority (CA) process to add one or more signatures to an existing certificate. This could be a notary public service or a simple multiple signature facility. The CA process may be local or remote. The CL module incorporates knowledge of the name, location, and interface protocol for communicating with the CA.

The CA process performs the signaturing operation once for each specified signer certificate. The signing operation may be performed locally or remotely. The CA must have access to the private keys associated with the signer certificates. If no signer's certificate is specified, the CA can assume one or more default signing certificates it uses for a multi-signing service. If no defaults are defined, the CA can reject the request.

The CL module selects and uses a default CSP to perform any required cryptographic operations. The CL module is responsible for creating and destroying all cryptographic contexts required to perform these operations.

The SignScope defines the set of certificate fields in the Subject Cert that are to be included in the signing process.

The caller can request additional signing-related services from the CA. These requests are designated by the MoreServiceRequests bit mask. CSSM-defined bit masks allow the caller to request full notary public services, and re-publishing of the new multiply-signed certificate with all directory services holding a copy of the old certificate. CAs are not required to provide such services. The CL module works with the CA process to provide the requested services.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimate time defines the expected signing time. This time may be substantial when the multiple signature model requires off-line procedures (such as a notary public). In contrast, the estimated time can be zero, meaning the multiply-signed certificate can be obtained immediately. After the specified time has elapsed, the caller must use the CL module interface CSSM_CL_CertMultiSignRetrieve, with the reference identifier, to obtain the multiply-signed certificate.

PARAMETERS

CLHandle (input)
: The handle that describes the add-in certificate library module used to perform this function.

SubjectCert (input)
: A pointer to the CSSM_DATA structure containing the certificate to be signed multiple times.

CACerts (input/optional)
: A pointer to an array of one or more CSSM_DATA structures containing the signing certificates of the desired Certification Authorities. If CACerts is NULL, the CL module or the CA process can provide a default set of signing certificates.

NumberOfCACerts (input)
: The number of CA signing certificates presented in the CACerts array. If no CA certificates are specified, the value of this parameter must be zero.

SignScope (input/optional)
: A pointer to the CSSM_FIELD array containing the OID/value pairs specifying the certificate fields to be included in the signature calculation. When the input value is NULL, the CL assumes and includes a default set of certificate fields in the signing process.

ScopeSize (input)
: The number of entries in the sign scope list. If no signing scope is specified, then the scope size must be zero.

CALocation (input/optional)
: A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on the SignerCert input parameter or can assume a default CA process location. If SignerCert is not specified and a default cannot be assumed, the request cannot be initiated and the operation fails.

MoreServiceRequests (input/optional)
: A bit mask requesting additional signing-related services from the Certificate Authority performing this function.

UserAuthentication (input/optional)
: A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, an so on, depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information is insufficient, additional information can be provided by the substructure field MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If authentication information is not required, this parameter must be NULL.

EstimatedTime (output)
: The number of seconds estimated before the multiply-signed certificate will be ready to be retrieved. A (default) value of zero indicates that the certificate can be retrieved immediately via the corresponding CL_CertRetrieve function call. When the signing authority cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceHandle (output)
: A reference identifier which uniquely identifies this specific request. The identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_MultiSignRetrieve function.

RETURN VALUE

A CSSM_OK return value signifies the requested operation has proceeded and that CL_CertMultiSignRetrieve should be called (after the specified amount of time) in order to retrieve the results. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_INVALID_SIGNER_CERTIFICATE
Revoked or expired signer certificate.

CSSM_CL_INVALID_SCOPE
Invalid scope.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_SIGN_REQUEST_FAIL
Unable to submit certificate signing request.

SEE ALSO

CL_CertMultiSignRetrieve

NAME

CL_CertMultiSignRetrieve

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CertMultiSignRetrieve
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_NET_ADDRESS_PTR CALocation,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the multiply-signed certificate created in response to the CL_CertMultiSignRequest function call. The reference identifier denotes the corresponding CL_CertMultiSignRequest call.

It is possible that the certificate is not ready to be retrieved when this call is made. In that case, an EstimatedTime to complete the signing process is returned with a NULL certificate pointer. The caller reuses the reference identifier to retrieve the certificate after the estimated time to completion has elapsed.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ReferenceIdentifier (input)
A reference identifier which uniquely identifies the CSSM_CL_CertMultiSignRequest call that initiated the multiple signing request. This identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_MultiSignRetrieve function.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on state information associated with the ReferenceIdentifier or can assume a default CA process location. If insufficient state is associated with the ReferenceIdentifier and a default cannot be assumed, the retrieval cannot be completed and the operation fails.

EstimatedTime (output)
The number of seconds estimated before the multiply-signed Certificate will be returned. A (default) value of zero indicates that the certificate has been returned as a result of this call. When the signing authority cannot estimate the time required to sign the certificate, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the multiply-signed certificate. If the pointer is NULL, the calling application is expected to call back after the specified EstimatedTime. If the pointer is NULL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_IDENTIFIER
: Invalid reference identifier.

CSSM_CL_CERT_SIGN_FAIL
: Unable to sign certificate.

CSSM_CL_EXTRA_SERVICE_FAIL
: Unable to perform additional signing-related services.

CSSM_CL_MEMORY_ERROR
: Not enough memory.

SEE ALSO

CL_CertMultiSignRequest, *CL_CertVerify*

NAME

CL_CertRecoveryRequest

SYNOPSIS

```
CSSM_RETURN CSSMCLI CL_CertRecoveryRequest
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR CACert,
    const CSSM_NET_ADDRESS_PTR CALocation,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    const CSSM_FIELD_PTR SelectedCertFieldValues,
    const uint32 NumberOfFieldValues,
    uint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

This function submits a certificate recovery request to a Certificate Authority (CA) process (or other trusted backup facility) to prepare for the recovery of a set of certificates and their associated private keys. The CA process is identified by the CA certificate, available to applications via a query to the CSSM registry. The caller can specify one or more certificate field values to limit the set of certificates selected for potential recovery. The recovery facility process may be local or remote. The CL module incorporates knowledge of the name, location, and interface protocol for communicating with the CA.

The caller is required to provide authentication information so the CA process can determine whether the caller is authorized to recover a certificate. The specific format of the credential is specified by the CL module. The caller can query the CL Module Info structure to obtain this information. Additional authentication information may also be required. It can be provided in the substructure field named MoreAuthenticationData.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimate time defines the expected certificate recovery time. This time may be substantial when many certificates are being recovered or manual procedures are required. In contrast, the estimated time can be zero, meaning the set of recovered certificates can be obtained immediately. After the specified time has elapsed, the caller must use the CL module interface CSSM_CL_CertRecoveryRetrieve, with the reference identifier, to obtain the set of recovered certificates from the CA process.

PARAMETERS

CLHandle (input)
: The handle that describes the add-in certificate library module used to perform this function.

CACert (input/optional)
: A pointer to the CSSM_DATA structure containing the certificate of the issuer of the certificate to be recovered. This certificate identifies the CA to be contacted for certificate recovery. If the CACert is NULL, the CL module can attempt certificate recovery from a default CA process.

CALocation (input/optional)
: A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on the SignerCert input parameter or can assume a default CA process location. If SignerCert is not specified and a default cannot be assumed, the request cannot be initiated and the operation fails.

UserAuthentication (input/optional)
: A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on, depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information is insufficient, additional information can be provided by the substructure field MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If authentication information is not required, this parameter must be NULL.

SelectedCertFieldValues (input/optional)
: An array of one or more field values that must be matched as part of the process of selecting certificates for recovery. If no certificate field values are specified, then the all of the caller's certificates (known to this CL module) will be selected for possible recovery.

NumberOfFieldValues (input)
: The number of selected certificate field values listed in the array SelectedCertFieldValues. If no certificate field values are specified, then this value must be zero.

EstimatedTime (output)
: The number of seconds estimated before the set of recovered certificates will be ready to be retrieved. A (default) value of zero indicates that the recovered certificates can be retrieved immediately via the corresponding CL_CertRecoveryRetrieve function call. When the recovery process cannot estimate the time required to prepare the recovered certificates, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceIdentifier (output)
: A reference identifier which uniquely identifies this specific request. The handle must be used in all subsequent calls to retrieve the set of recovered certificates. The identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_CertRecoveryRetrieve function.

RETURN VALUE

A CSSM_OK return value signifies the requested operation has proceeded and that CL_CertRecoveryRetrieve should be called (after the specified amount of time) in order to retrieve the results. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
: Invalid Certificate Library Handle.

CSSM_CL_INVALID_CSP_HANDLE
: Invalid CSP Handle.

CSSM_CL_INVALID_DATA_POINTER
: Invalid pointer input.

CSSM_AUTHENTICATION_FAIL
: Invalid/unauthorized credential.

CSSM_CL_MEMORY_ERROR
: Not enough memory.

CSSM_CL_CERT_REQUEST_FAIL
: Unable to submit certificate recovery request.

SEE ALSO

CL_CertRecoveryRetrieve, *CL_CertRecover*, *CL_CertRecoverKey*, *CL_CertAbortRecovery*

NAME

CL_CertRecoveryRetrieve

SYNOPSIS

```
CSSM_RETURN CSSMCLI CL_CertRecoveryRetrieve
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_NET_ADDRESS_PTR CALocation,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    CSSM_HANDLE CacheHandle,
    uint32 *NumberOfRetrievedCerts,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the set of certificates recovered in response to the CL_CertRecoveryRequest function call. The reference identifier denotes the corresponding CertRecoveryRequest call.

The caller may be required to provide additional authentication information to recover the certificates. The format of these credentials is defined by the CL module and recorded in the CLSubservice structure, which can be queried by the caller.

This function obtains the set of recovered certificates and their associated private keys. It returns a cache handle to reference the returned set. The cache handle is used when retrieving individual certificates and keys using the CSSM_CL_CertRecover function.

It is possible that the recovered certificates are not ready to be retrieved when CSSM_CL_CertRecoveryRetrieve is called. In that case, an EstimatedTime to complete certificate recovery is returned with a NULL cache handle. The caller reuses the reference identifier to retrieve the recovered certificates after the estimated time to completion has elapsed.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ReferenceIdentifier (input)
A reference identifier which uniquely identifies the CSSM_CL_CertRecoveryRequest call that initiated recovery of the set of certificates obtained by this function. The identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_CertRecoveryRetrieve function.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on state information associated with the ReferenceIdentifier or can assume a default CA process location. If insufficient state is associated with the ReferenceIdentifier and a default cannot be assumed, the retrieval cannot be completed and the operation fails.

UserAuthentication (input/optional)
A pointer to the CSSM_USER_AUTHENTICATION structure containing the authentication information to be used in association with this request. The authentication information may be a pass-phrase, a PIN, a completed registration form, a Certificate to facilitate a signing operation, and so on, depending on the context of the request. The required format for this credential is defined by the CL and recorded in the CLSubservice structure describing this module. If the supplied information is insufficient, additional information can be provided

by the substructure field MoreAuthenticationData. This field contains an immediate data value or a callback function to collect additional information from the user. If authentication information is not required, this parameter must be NULL.

CacheHandle (output)
A reference handle which uniquely identifies the cache of recovered certificates and their associated private keys. If the certificate retrieval process has not been completed, the returned cache handle is zero. A non-zero cache handle can be used in the CSSM_CL_CertRecover function to complete the recovery of an individual certificate and its private key. The handle is not persistent. It used is terminated by calling CSSM_CL_CertAbortRecovery or by termination of the caller process.

NumberOfRetrievedCerts (output)
The number of certificates in the cache.

EstimatedTime (output)
The number of seconds estimated before the set of recovered certificates will be returned. A (default) value of zero indicates that the set has been returned as a result of this call. When the recovery process cannot estimate the time required to prepare the recovered certificates, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A CSSM_RETURN value indicating whether the operation obtained a set of recovered certificates. If the result is CSSM_FAIL, and a NULL cache handle and a positive EstimatedTime are returned, then the calling application is expected to call this function again after the specified EstimatedTime. If the result is CSSM_FAIL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_IDENTIFIER
Invalid reference identifier.

CSSM_AUTHENTICATION_FAIL
Invalid /unauthorized credential for operation.

CSSM_CL_MEMORY_ERROR
Not enough memory.

SEE ALSO

CL_CertRecoveryRequest, *CL_CertRecover*, *CL_CertRecoverKey*, *CL_CertAbortRecovery*

NAME

CL_CertRecover

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CertRecover
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CacheHandle,
    const uint32 CacheIndex)
```

DESCRIPTION

This function returns a certificate from a cache of certificates retrieved by the CSSM_CL_CertRecoveryRetrieve function. The cache contains a set of certificates in unspecified order. The certificate to be retrieved is specified by the CacheIndex parameter, which is a simple counter from one to the number of certificates in the cache. The selected certificate is returned as a result of the function call.

This function has no effect on the private key associated with the recovered certificate. Recovery of the private key can be performed using the function CSSM_CL_CertKeyRecover.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CacheHandle (input)
A reference handle which uniquely identifies the cache of retrieved, recovered certificates and their associated private keys.

CacheIndex (input)
An index value that selects a certificate from the cache of retrieved, recovered certificates and associated keys. The value must be less than or equal to the number of certificates in the cache.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the recovered certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_HANDLE
Invalid cache handle.

CSSM_CL_INVALID_INDEX
Cache index value is out of range.

CSSM_CL_MEMORY_ERROR
Not enough memory.

SEE ALSO

CL_CertRecoveryRequest, *CL_CertRecoveryRetrieve*, *CL_CertRecoverKey*, *CL_CertAbortRecovery*

NAME

CL_CertKeyRecover

SYNOPSIS

```
CSSM_RETURN CSSMCLI CL_CertKeyRecover
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CacheHandle,
    const uint32 CacheIndex,
    CSSM_CSP_HANDLE CSPHandle,
    const CSSM_CRYPTO_DATA_PTR PassPhrase)
```

DESCRIPTION

This function recovers the private key associated with a certificate and securely stores that key in the specified cryptographic service provider. The key (and its associated certificate) are among a set of certificates and private keys contained in the cache specified by the CacheHandle.

Cache entries are in unspecified order. The private key to be retrieved is specified by the CacheIndex parameter, which is a simple counter from one to the number of certificates in the cache.

The recovery process associates the private key with the public key contained in the certificate, securely stores the private key in the specified cryptographic service provider, and associates the new PassPhrase with the recovered, stored private key.

To selectively recover private keys from the cache, the function CSSM_CL_CertRecover can be used to review recovered certificates and determine the appropriate CacheIndex to use when recovering the associated private key.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CacheHandle (input)
A reference handle which uniquely identifies the cache of retrieved, recovered certificates and their associated private keys.

CacheIndex (input)
An index value that selects a certificate from the cache of retrieved, recovered certificates and associated keys. The value must be less than or equal to the number of certificates in the cache.

CSPHandle (input)
The handle that describes the add-in CSP module where the private key is to be stored. Optionally, the CL module can use this CSP to perform additional cryptographic operations or may use another default CSP for that purpose.

PassPhrase (input)
A pointer to the CSSM_CRYPTO_DATA structure containing the new passphrase to be associated with the recovered certificate and private key. The passphrase can be specified by immediate data in this parameter or a callback function to request a passphrase from the caller's process.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_CL_INVALID_HANDLE
Invalid cache handle.

CSSM_CL_INVALID_INDEX
Cache index value is out of range.

CSSM_CL_PRIVATE_KEY_STORE_FAIL
Unable to store private key in CSP.

CSSM_CL_MEMORY_ERROR
Not enough memory.

SEE ALSO

CL_CertRecoveryRequest, *CL_CertRecoveryRetrieve*, *CL_CertRecover*, *CL_CertAbortRecovery*

NAME

CL_CertAbortRecovery

SYNOPSIS

```
CSSM_RETURN CSSMCLI CL_CertAbortRecovery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE CacheHandle)
```

DESCRIPTION

This function terminates the iterative process of recovering certificates and their associated private keys from a cache of certificates. This function must be called even if all certificates and their associated private keys are recovered from the cache. This function destroys all intermediate state and secret information used during the certificate and key recovery process. At completion of this function, the specified cache handle is invalid and the operations CSSM_CL_CertRecover and CSSM_CL_CertRecoverKey can no be invoked using this handle.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CacheHandle (input)
A handle which identifies the cache of retrieved, recovered certificates and their associated private keys.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_HANDLE
Invalid cache handle.

CSSM_CL_ABORT_RECOVERY_FAIL
Unable to abort the recovery process.

SEE ALSO

CL_CertRecoveryRequest, *CL_CertRecoveryRetrieve*, *CL_CertRecover*, *CL_CertRecoverKey*

NAME

CL_CertVerify

SYNOPSIS

```
CSSM_BOOL CSSMCLI CL_CertVerify
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR CertToBeVerified,
    const CSSM_DATA_PTR SignerCert,
    const CSSM_FIELD_PTR VerifyScope,
    uint32 ScopeSize)
```

DESCRIPTION

This function verifies that the signed certificate has not been altered since it was signed by the designated signer. Only one signature is verified by this function. If the certificate to be verified includes multiple signatures, this function must be applied once for each signature to be verified. This function verifies a digital signature over the certificate fields specified by VerifyScope. If the verification scope fields are not specified, the function performs verification using a pre-selected set of fields in the certificate.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

CertToBeVerified (input)
A pointer to the CSSM_DATA structure containing a certificate containing at least one signature for verification. An unsigned certificate template cannot be verified.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate used to sign the subject certificate. This certificate provides the public key to use in the verification process and if the certificate being verified contains multiple signatures, the signer's certificate indicates which signature is to be verified.

VerifyScope (input/optional)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be used in verifying the signature (that is, the fields that were used to calculate the signature). If the verify scope is null, the certificate library module assumes that its default set of certificate fields were used to calculate the signature and those same fields are used in the verification process.

ScopeSize (input)
The number of entries in the verify scope list. If the verification scope is not specified, the input value for scope size must be zero.

RETURN VALUE

CSSM_TRUE if the certificate signature verified. CSSM_FALSE if the certificate signature did not verify or an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Cryptographic Context Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_INVALID_CONTEXT
Invalid context for the requested operation.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_INVALID_SCOPE
Invalid scope.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CERT_VERIFY_FAIL
Unable to verify certificate.

SEE ALSO

CL_CertSign

NAME

CL_CertGetFirstFieldValue

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CertGetFirstFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Cert,
    const CSSM_OID_PTR CertField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields)
```

DESCRIPTION

This function returns the value of the designated certificate field. If more than one field matches the CertField OID, the first matching field will be returned. The number of matching fields is an output parameter, as is the ResultsHandle to be used to retrieve the remaining matching fields.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate.

CertField (input)
A pointer to an object identifier that identifies the field value to be extracted from the Cert.

ResultsHandle (output)
A pointer to the CSSM_HANDLE that should be used to obtain any additional matching fields.

NumberOfMatchedFields (output)
The number of fields that match the CertField OID.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the value of the requested field. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_UNKNOWN_TAG
Unknown field tag.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_NO_FIELD_VALUES
No field values for this results handle.

CSSM_CL_CERT_GET_FIELD_VALUE_FAIL
Unable to get field value.

SEE ALSO

CL_CertGetNextFieldValue, *CL_CertAbortQuery*, *CL_CertGetAllFields*, *CL_CertDescribeFormat*

NAME

CL_CertGetNextFieldValue

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CertGetNextFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function returns the value of a certificate field, when that field occurs multiple times in a certificate. Certificates with repeated fields (such as multiple signatures) have multiple field values corresponding to a single OID. A call to the function CSSM_CL_CertGetFirstFieldValue initiates the process and returns a results handle identifying the certificate from which values are being obtained and the OID corresponding to those values. The CSSM_CL_CertGetNextFieldValue function can be called repeatedly to obtain these values one at a time.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a certificate query.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the value of the requested field. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_RESULTS_HANDLE
Invalid results handle.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_NO_FIELD_VALUES
No field values for this results handle.

CSSM_CL_CERT_GET_FIELD_VALUE_FAIL
Unable to get field value.

SEE ALSO

CL_CertGetFirstFieldValue, CL_CertAbortQuery

NAME

CL_CertAbortQuery

SYNOPSIS

```
CSSM_RETURN CSSMCLI CL_CertAbortQuery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function terminates the get operation initiated by CSSM_CL_CertGetFirstFieldValue and allows the CL to release all intermediate state information associated with the query. This function should be called even if all values retrieved by the call to CSSM_CL_CertGetFirstFieldValue are obtained by repeated calls to CSSM_CL_CertGetNextFieldValue.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a certificate query.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_RESULTS_HANDLE
Invalid results handle.

CSSM_CL_CERT_ABORT_QUERY_FAIL
Unable to abort query.

SEE ALSO

CL_CertGetFirstFieldValue, *CL_CertGetNextFieldValue*

NAME

CL_CertGetKeyInfo

SYNOPSIS

```
CSSM_KEY_PTR CSSMCLI CL_CertGetKeyInfo
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Cert)
```

DESCRIPTION

This function obtains information about the certificate's public key. Ideally, this information comprises the key fields the application needs to create a cryptographic context that uses this certificate's key.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate from which to extract the public key information.

RETURN VALUE

A pointer to the CSSM_KEY structure containing the public key and possibly other key information. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_UNKNOWN_TAG
Unknown field tag.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_CERT_GET_KEY_INFO_FAIL
Unable to get key information.

SEE ALSO

CL_CertGetFirstFieldValue

NAME

CL_CertGetAllFields

SYNOPSIS

```
CSSM_FIELD_PTR CSSMCLI CL_CertGetAllFields
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Cert,
    uint32 *NumberOfFields)
```

DESCRIPTION

This function returns a list of the fields in the input certificate, as described by their OID/value pairs.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate whose fields will be returned.

NumberOfFields (output)
The length of the output CSSM_FIELD array.

RETURN VALUE

A pointer to an array of CSSM_FIELD structures that describe the contents of the certificate using OID/value pairs. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid handle.

CSSM_CL_INVALID_POINTER
Invalid pointer.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_CERT_GET_ALL_FIELDS_FAIL
Unable to return the list of fields.

SEE ALSO

CL_CertGetFirstFieldValue, *CL_CertDescribeFormat*

NAME

CL_CertGroupToSignedBundle

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CertGroupToSignedBundle
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CERTGROUP_PTR CertGroupToBundle,
    const CSSM_DATA_PTR SignerCert,
    const CSSM_CERT_BUNDLE_HEADER_PTR BundleInfo);
```

DESCRIPTION

This function accepts as input a certificate group (as an array of individual certificates) and returns a certificate bundle (a codified and signed aggregation of the certificates in the group). The certificate group will first be encoded according to the BundleInfo input by the user. If BundleInfo is NULL, the library will perform a default encoding for its default bundle type. If possible, the certificate group ordering will be maintained in this certificate aggregate encoding. After encoding, the certificate aggregate will be signed using the input context and signer certificate. The CL module embeds knowledge of the signing scope for the bundle types it supports. The signature is then associated with the certificate aggregate according to the bundle type and encoding rules and is returned as a bundle to the calling application.

PARAMETERS

CLHandle (input)
The handle of the add-in module to perform this operation.

CCHandle (input)
The handle of the cryptographic context to control the signing operation. The operation will fail if a signature is required for this type of bundle and the cryptographic context is not valid.

CertGroupToBundle (input)
An array of individual, encoded certificates. All of the certificates in this list will be included in the resulting certificate bundle.

SignerCert (input/optional)
If signing is required for this type of certificate bundle, this is the certificate to be used to sign the bundle. If a signing certificate is required but not specified, then the module will assume a default certificate. If a signature is not required for this certificate bundle type, this parameter will be ignored.

BundleInfo (input/optional)
A structure containing the type and encoding of the bundle to be created. If the type and the encoding are not specified, then the module will assume a default bundle type and bundle encoding.

RETURN VALUE

The function returns a pointer to a signed certificate bundle containing all of the certificates in the certificate group. The bundle is of the type and encoding requested by the caller or is the default defined by the library module if the BundleInfo was not specified by the caller. If the pointer is NULL, an error has occurred. Use *CSSM_GetError* to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CC_HANDLE
: Invalid context handle.

CSSM_CL_INVALID_BUNDLE_INFO
: Unknown bundle type or encoding.

CSSM_CL_INVALID_SIGNER_CERTIFICATE
: Revoked or expired signer certificate.

CSSM_CL_MEMORY_ERROR
: Error allocating memory.

CSSM_CL_CERGROUPTOBUNDLE_FAIL
: Unable to create the signed bundle.

SEE ALSO

CL_CertGroupFromVerifiedBundle

NAME

CL_CertGroupFromVerifiedBundle

SYNOPSIS

```
CSSM_BOOL CSSMCLI CL_CertGroupFromVerifiedBundle
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_CERT_BUNDLE_PTR CertBundle,
    const CSSM_DATA_PTR SignerCert,
    CSSM_CERTGROUP_PTR *CertGroup);
```

DESCRIPTION

This function accepts as input a certificate bundle (a codified and signed aggregation of the certificates in the group), verifies the signature of the bundle (if a signature is present) and returns a certificate group (as an array of individual certificates) including every certificate contained in the bundle. The signature on the certificate aggregate is verified using the cryptographic context and possibly using the input signer certificate. The CL module embeds the knowledge of the verification scope for the bundle types that it supports. A CL module's supported bundle types and encodings are available to applications by querying the CSSM registry. The type and encoding of the certificate bundle must be specified with the input bundle. If signature verification is successful, the certificate aggregate will be parsed into a certificate group whose order corresponds to the certificate aggregate ordering. This certificate group will then be returned to the calling application.

PARAMETERS

CLHandle (input)
The handle of the add-in module to perform this operation.

CCHandle (input)
The handle of the cryptographic context to control the verification operation.

CertBundle (input)
A structure containing a reference to a signed, encoded bundle of certificates, and to descriptors of the type and encoding of the bundle. The bundled certificates are to be separated into a certificate group (list of individual encoded certificates). If the bundle type and bundle encoding are not specified, the add-in module may either attempt to decode the bundle assuming a default type and encoding or may immediately fail.

SignerCert (input/optional)
The certificate to be used to verify the signature on the certificate bundle. If the bundle is signed but this field is not specified, then the module will assume a default certificate for verification.

CertGroup (output)
A pointer to the certificate group, represented as an array of individual, encoded certificates. The group contains all of the certificates contained in the certificate bundle.

RETURN VALUE

A CSSM_BOOL value corresponding to the result of the verification process. If a signature is required for this type of bundle and signature verification fails, the function returns CSSM_FALSE. If signature verification is required and succeeds, the function returns CSSM_TRUE and attempts to create a certificate group containing all certificates in the bundle. If the group cannot be created, the CertGroup is set to NULL and an error code is set. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle

CSSM_CL_INVALID_CC_HANDLE
Invalid context handle

CSSM_CL_INVALID_BUNDLE_INFO
Unknown bundle type or encoding

CSSM_CL_INVALID_SIGNER_CERTIFICATE
Revoked or expired signer certificate

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input

CSSM_CL_MEMORY_ERROR
Error allocating memory

CSSM_CL_CERGROUPFROMBUNDLE_FAIL
Unable to create the cert group

SEE ALSO

CL_CertGroupToSignedBundle

NAME

CL_CertImport

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CertImport
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CERT_TYPE ForeignCertType,
    CSSM_CERT_ENCODING ForeignCertEncoding,
    const CSSM_DATA_PTR ForeignCert)
```

DESCRIPTION

This function imports a certificate from the specified foreign format into the native format of the specified certificate library. The set of ForeignCertTypes supported for import is at the discretion of the certificate library and documented for each module as part of the CSSM_CLSUBSERVICE structure available from the CSSM Registry.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ForeignCertType (input)
A unique value that identifies the type of the certificate being imported.

ForeignCertEncoding (input)
A unique value that identifies the encoding of the certificate being imported.

ForeignCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be imported into the certificate library modules native certificate type.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the native-type certificate imported from the foreign certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_CERT_IMPORT_FAIL
Unable to import certificate.

SEE ALSO

CL_CertExport

NAME

CL_CertExport

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CertExport
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CERT_TYPE TargetCertType,
    CSSM_CERT_ENCODING TargetCertEncoding,
    const CSSM_DATA_PTR NativeCert)
```

DESCRIPTION

This function exports a certificate from the native format of the specified certificate library into the specified target certificate format. The set of TargetCertTypes supported for export is at the discretion of the certificate library and is documented for each module as part of the CSSM_CLSUBSERVICE structure available from the CSSM Registry.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

TargetCertType (input)
A unique value that identifies the target type of the certificate being exported.

TargetCertEncoding (input)
A unique value that identifies the target encoding of the certificate being exported.

NativeCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be exported.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the target-type certificate exported from the native certificate. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_CERT_EXPORT_FAIL
Unable to export certificate.

SEE ALSO

CL_CertImport

NAME

CL_CertDescribeFormat

SYNOPSIS

```
CSSM_OID_PTR CSSMCLI CL_CertDescribeFormat
    (CSSM_CL_HANDLE CLHandle,
     uint32 *NumberOfFields)
```

DESCRIPTION

This function returns a list of the object identifiers used to describe the certificate format supported by the specified CL.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

NumberOfFields (output)
The length of the returned array of OIDs.

RETURN VALUE

A pointer to the array of CSSM_OIDs which represent the supported certificate format. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid handle.

CSSM_CL_INVALID_POINTER
Invalid pointer.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_CERT_DESCRIBE_FORMAT_FAIL
Unable to return the list of OIDs.

SEE ALSO

CSSM_CL_CertGetAllFields, *CSSM_CL_CertGetFirstFieldValue*

48.3 Certificate Revocation List Operations

This section describes the function prototypes and error codes supported by a Certificate Library module for operations on certificate revocation lists (CRLs). The functions will be exposed to CSSM via a function table, so the function names may vary at the discretion of the certificate library developer. However, the function parameter list and return type must match the prototypes given in this section in order to be used by applications. The error codes given in this section constitute the generic error codes that are defined by CSSM for use by all certificate libraries in describing common error conditions. A certificate library may also define and return vendor-specific error codes. The error codes defined by CSSM are considered to be comprehensive and few if any vendor-specific codes should be required. Applications must consult vendor supplied documentation for the specification and description of any error codes defined outside of this specification.

NAME

CL_CrlCreateTemplate

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CrlCreateTemplate
    (CSSM_CL_HANDLE CLHandle
    const CSSM_FIELD_PTR CrlTemplate,
    uint32 NumberOfFields)
```

DESCRIPTION

This function creates an unsigned, memory-resident CRL. Fields in the CRL are initialized with the descriptive data specified by the OID/value input pairs. The specified OID/value pairs can initialize all or a subset of the general attribute fields in the new CRL, though the module developer may specify a set of fields that must be or cannot be set using this operation. Subsequent values may be set using the CSSM_CL_CrlSetFields operation.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CrlTemplate (input)
Any array of field OID/value pairs containing the values to initialize the CRL attribute fields.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlTemplate input parameter.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the new CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_MEMORY_ERROR
Not enough memory to allocate for the CRL.

CSSM_CL_CRL_CREATE_FAIL
Unable to create CRL.

SEE ALSO

CSSM_CL_CrlSetFields, *CSSM_CL_CrlAddCert*, *CSSM_CL_CrlSign*, *CSSM_CL_CertGetFirstFieldValue*

NAME

CL_CrlSetFields

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CrlSetFields
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_FIELD_PTR CrlTemplate,
    uint32 NumberOfFields,
    const CSSM_DATA_PTR OldCrl);
```

DESCRIPTION

This function will set the fields of the input CRL to the new values, specified by the input OID/value pairs. The module developer may specify a set of fields that must be or cannot be set using this operation. This operation is valid only if the CRL has not been closed by the process of signing the CRL (that is, execution of the function CSSM_CL_CrlSign). Once the CRL has been signed, fields cannot be changed.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CrlTemplate (input)
Any array of field OID/value pairs containing the values to initialize the CRL attribute fields.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlTemplate input parameter.

OldCrl (input)
The CRL to be updated with the new attribute values. The CRL must be unsigned and available for update.

RETURN VALUE

A pointer to the modified, unsigned CRL If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_FIELD_POINTER
Invalid pointer input.

CSSM_CL_INVALID_TEMPLATE
Invalid template for this CRL type.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_CRL_SET_FAIL
Unable to set CRL field values.

SEE ALSO

CSSM_CL_CrlCreateTemplate, *CSSM_CL_CrlAddCert*, *CSSM_CL_CrlSign*, *CSSM_CL_CertGetFirstFieldValue*

NAME

CL_CrlRequest

SYNOPSIS

```
CSSM_RETURN CSSMCLI CL_CrlRequest
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_FIELD_PTR CrlIdentifier,
    const CSSM_DATA_PTR CACert,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize,
    const CSSM_NET_ADDRESS_PTR CALocation,
    CSSM_CA_SERVICES MoreServiceRequests,
    sint32 *EstimatedTime,
    const CSSM_DATA_PTR ReferenceIdentifier)
```

DESCRIPTION

This function submits a request to a Certificate Authority (CA) process to issue the most current version of a CRL of a specified name. The SignerCert input parameter indicates which CA process should receive the request. The selected CA process may be local or remote.

When all prerequisite conditions have been satisfied, such as some minimum time has elapsed since the last version of the requested CRL was issued, the CA process closes out the CRL, signs it and can distribute it to all interested and requesting parties. The CA must have access to the private keys associated with the signer's certificate to sign the CRL. If no signer's certificate is specified, the CL module can assume a default CA process from which it always acquires CRLs. If no defaults are known to the CL module, the CL module can reject the request.

The CL module selects and uses a default CSP for any required cryptographic operations. The CL module and the CA process are responsible for creating and destroying all cryptographic contexts required to perform this service.

The SignScope defines the set of CRL fields that are to be included in the signing process.

The caller can request additional CRL-related services from the CA. These requests are designated by the MoreServiceRequests bit mask. CSSM-defined bit masks allow the caller to request immediate distribution of the latest CRL to any and all interested parties. CAs are not required to provide these additional services. The CL module works with the CA process to provide the requested.

This function returns a ReferenceIdentifier and an EstimatedTime (specified in seconds). The estimate time defines the expected closing, signing and distribution time. This time may be substantial when closing a CRL requires off-line procedures or the service model mandates a minimum time between distributions. In contrast, the estimated time can be zero, meaning the CRL can be obtained immediately. After the specified time has elapsed, the caller must use the CL module interface CSSM_CL_CrlRetrieve, with the reference identifier, to obtain the CRL.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CrlIdentifier (input)
A pointer to an OID-value pair that uniquely identifies (names) the CRL being requests from the CA.

CACert (input/optional)
A pointer to the CSSM_DATA structure containing the desired Certification Authority's

signing certificate to be used when issuing the CRL. If the CACert is NULL, the CL module or the CA process can provide a default signing certificate for issuing the CRL.

SignScope (input/optional)
A pointer to the CSSM_FIELD array containing the OID/value pairs specifying the CRL fields to be included in the signature calculation. When the input value is NULL, the CA assumes and includes a default set of CRL fields in the signing process.

ScopeSize (input)
The number of entries in the sign scope list. If no signing scope is specified, then the scope size must be zero.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on the SignerCert input parameter or can assume a default CA process location. If SignerCert is not specified and a default cannot be assumed, the request cannot be initiated and the operation fails.

MoreServiceRequests (input/optional)
A bit mask requesting additional CRL-related services from the Certificate Authority performing this function.

EstimatedTime (output)
The number of seconds estimated before the CRL will be ready to be retrieved. A (default) value of zero indicates that the CRL can be retrieved immediately via the corresponding CL_CrlRetrieve function call. When the certification process cannot estimate the time required to prepare the CRL, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

ReferenceIdentifier (output)
A reference identifier which uniquely identifies this specific request. The identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_CrlRetrieve function.

RETURN VALUE

A CSSM_OK return value signifies the requested operation has proceeded and that CL_CrlRetrieve should be called (after the specified amount of time) in order to retrieve the results. If CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized certificate format.

CSSM_CL_INVALID_SIGNER_CERTIFICATE
Revoked or expired signer certificate.

CSSM_CL_INVALID_SCOPE
Invalid scope.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_SIGN_REQUEST_FAIL
Unable to submit certificate signing request.

SEE ALSO

CL_CrlRetrieve

NAME

CL_CrlRetrieve

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CrlRetrieve
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR ReferenceIdentifier,
    const CSSM_NET_ADDRESS_PTR CALocation,
    sint32 *EstimatedTime)
```

DESCRIPTION

This function returns the CRL closed and issued in response to the CL_CrlRequest function call. The reference identifier denotes the corresponding call.

It is possible that the CRL is not ready to be retrieved when this call is made. In that case, an EstimatedTime to complete the CRL issuing process is returned with the reference identifier and a NULL certificate pointer. The caller must attempt to retrieve the CRL again after the estimated time to completion has elapsed.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ReferenceIdentifier (input)
A reference identifier which uniquely identifies the CSSM_CL_CrlRequest call that initiated the CRL issuing request. The identifier persists across application executions until it is terminated by successful or failed completion of the CSSM_CL_CrlRetrieve function.

CALocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the CA process. If the input is NULL, the module can determine a CA process and its location based on state information associated with the ReferenceIdentifier or can assume a default CA process location. If insufficient state is associated with the ReferenceIdentifier and a default cannot be assumed, the retrieval cannot be completed and the operation fails.

EstimatedTime (output)
The number of seconds estimated before the CRL will be returned. A (default) value of zero indicates that the CRL has been returned as a result of this call. When the certification process cannot estimate the time required to prepare the CRL, the output value for estimated time is CSSM_ESTIMATED_TIME_UNKNOWN.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the CRL. If the pointer is NULL, the calling application is expected to call back after the specified EstimatedTime. If the pointer is NULL and EstimatedTime is zero, an error has occurred. If the EstimatedTime is CSSM_ESTIMATED_TIME_UNKNOWN, the call back time is not defined and the application must periodically poll for completion. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_IDENTIFIER
Invalid reference identifier.

CSSM_CL_CERT_SIGN_FAIL
: Unable to sign CRL.

CSSM_CL_EXTRA_SERVICE_FAIL
: Unable to perform additional CRL-related services.

CSSM_CL_MEMORY_ERROR
: Not enough memory.

SEE ALSO

CL_CrlRequest

NAME

CL_CrlAddCert

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CrlAddCert
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR Cert,
    const CSSM_DATA_PTR RevokerCert,
    const CSSM_FIELD_PTR CrlEntryFields,
    uint32 NumberOfFields,
    const CSSM_DATA_PTR OldCrl)
```

DESCRIPTION

This function revokes the input certificate by adding a record representing the certificate to the CRL. The values for the new entry in the CRL are specified by the a list of OID/value input pairs. The module developer may specify that a certain set of fields must be or cannot be set by use of the list of OID/value pairs. If the CRL format supports the signing of individual records, the revoker's certificate is used to sign the new CRL entry. The operation is valid only if the CRL has not been closed by the process of signing the CRL (that is, execution of the function CSSM_CL_CrlSign). Once the CRL has been signed, entries cannot be added or removed.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be revoked.

RevokerCert (input)
A pointer to the CSSM_DATA structure containing the revoker's certificate.

CrlEntryFields (input)
An array of OID/value pairs specifying the initial values for descriptive data fields of the new CRL entry.

NumberOfFields (input)
The number of OID/value pairs specified in the CrlEntryFields input parameter.

OldCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to which the newly revoked certificate will be added.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the updated CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Context Handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
: Invalid Certificate.

CSSM_CL_INVALID_CRL
: Invalid CRL.

CSSM_CL_MEMORY_ERROR
: Not enough memory to allocate the CRL.

CSSM_CL_CRL_ADD_CERT_FAIL
: Unable to add certificate to CRL.

SEE ALSO

CL_CrlRemoveCert

NAME

CL_CrlRemoveCert

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CrlRemoveCert
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Cert,
    const CSSM_DATA_PTR OldCrl)
```

DESCRIPTION

This function reinstates a certificate by removing it from the specified CRL. The operation is valid only if the CRL has not be closed by the process of signing the CRL (that is, execution of the function CSSM_CL_CrlSign). Once the CRL has been signed, entries cannot be added or removed.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be unrevoked.

OldCrl (input)
A pointer to the CSSM_DATA structure containing the CRL from which the certificate is to be removed.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the updated CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
Invalid Certificate.

CSSM_CL_INVALID_CRL
Invalid CRL.

CSSM_CL_MEMORY_ERROR
Not enough memory to allocate the CRL.

CSSM_CL_CRL_REMOVE_CERT_FAIL
Unable to remove certificate from CRL.

SEE ALSO

CL_CrlAddCert

NAME

CL_CrlSign

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CrlSign
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR UnsignedCrl,
    const CSSM_DATA_PTR SignerCert,
    const CSSM_FIELD_PTR SignScope,
    uint32 ScopeSize)
```

DESCRIPTION

This function signs, in accordance with the specified cryptographic context, the fields of the CRL indicated in the SignScope parameter. Once the CRL has been signed it may not be modified. This means that entries cannot be added or removed from the CRL through application of the CSSM_CL_CrlAddCert or CSSM_CL_CrlRemoveCert operations. A signed CRL can be verified, applied to a data store, and searched for values.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

UnsignedCrl (input)
A pointer to the CSSM_DATA structure containing the CRL to be signed.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate to be used to sign the CRL.

SignScope (input)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be signed. A null input signs all the fields in the CRL.

ScopeSize (input)
The number of entries in the sign scope list.

RETURN VALUE

A pointer to the CSSM_DATA structure containing the signed CRL. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Context Handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
Invalid Certificate.

CSSM_CL_INVALID_CRL_PTR
Invalid CRL pointer.

CSSM_CL_INVALID_SCOPE_PTR
SignScope pointer is invalid.

CSSM_CL_MEMORY_ERROR
Not enough memory to allocate the CRL.

CSSM_CL_CRL_SIGN_FAIL
Unable to sign CRL.

SEE ALSO

CL_CrlVerify

NAME

CL_CrlVerify

SYNOPSIS

```
CSSM_BOOL CSSMCLI CL_CrlVerify
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    const CSSM_DATA_PTR CrlToBeVerified,
    const CSSM_DATA_PTR SignerCert,
    const CSSM_FIELD_PTR VerifyScope,
    uint32 ScopeSize)
```

DESCRIPTION

This function verifies that the signed CRL has not been altered since it was signed by the designated signer. It does this by verifying the digital signature over the fields specified by the VerifyScope parameter.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input)
The handle that describes the context of this cryptographic operation.

CrlToBeVerified (input)
A pointer to the CSSM_DATA structure containing the CRL to be verified.

SignerCert (input)
A pointer to the CSSM_DATA structure containing the certificate used to sign the CRL.

VerifyScope (input)
A pointer to the CSSM_FIELD array containing the tag/value pairs of the fields to be verified. A null input verifies all the fields in the CRL.

ScopeSize (input)
The number of entries in the verify scope list.

RETURN VALUE

A CSSM_TRUE return value signifies that the certificate revocation list verifies successfully. When CSSM_FALSE is returned, either the CRL verified unsuccessfully or an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Context Handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
Invalid Certificate.

CSSM_CL_INVALID_CRL_PTR
Invalid CRL pointer.

CSSM_CL_INVALID_SCOPE_PTR
VerifyScope pointer is invalid.

CSSM_CL_MEMORY_ERROR
Not enough memory to allocate the CRL.

CSSM_CL_CRL_VERIFY_FAIL
Unable to verify CRL.

SEE ALSO

CL_CrlSign

NAME

CL_IsCertInCrl

SYNOPSIS

```
CSSM_BOOL CSSMCLI CL_IsCertInCrl
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Cert,
    const CSSM_DATA_PTR Crl)
```

DESCRIPTION

This function searches the CRL for a record corresponding to the certificate. The operation will fail if neither the CRL or the revocation records have been signed. If a signature exists, the application is responsible for verifying that the signature was created by a trusted party.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Cert (input)
A pointer to the CSSM_DATA structure containing the certificate to be located.

Crl (input)
A pointer to the CSSM_DATA structure containing the CRL to be searched.

RETURN VALUE

A CSSM_TRUE return value signifies that the certificate is in the CRL. When CSSM_FALSE is returned, either the certificate is not in the CRL or an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid CL handle.

CSSM_CL_INVALID_CERTIFICATE_PTR
Invalid Certificate.

CSSM_CL_INVALID_CRL_PTR
Invalid CRL pointer.

NAME

CL_CrlGetFirstFieldValue

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CrlGetFirstFieldValue
    (CSSM_CL_HANDLE CLHandle,
    const CSSM_DATA_PTR Crl,
    const CSSM_OID_PTR CrlField,
    CSSM_HANDLE_PTR ResultsHandle,
    uint32 *NumberOfMatchedFields)
```

DESCRIPTION

This function returns the value of the designated CRL field. If more than one field matches the CrlField OID, the first matching field will be returned. The number of matching fields is an output parameter, as is the ResultsHandle to be used to retrieve the remaining matching fields.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

Crl (input)
A pointer to the CSSM_DATA structure that contains the CRL from which the field is to be retrieved.

CrlField (input)
A pointer to an object identifier that identifies the field value to be extracted from the Crl.

ResultsHandle (output)
A pointer to the CSSM_HANDLE, which should be used to obtain any additional matching fields.

NumberOfMatchedFields (output)
The number of fields that match the CrlField OID.

RETURN VALUE

Returns a pointer to a CSSM_DATA structure containing the first field that matched the CrlField. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNKNOWN_FORMAT
Unrecognized CRL format.

CSSM_CL_UNKNOWN_TAG
Unknown field tag.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_NO_FIELD_VALUES
No field values for this results handle.

CSSM_CL_CRL_GET_FIELD_VALUE_FAIL
Unable to get field value.

SEE ALSO

CL_CrlGetNextFieldValue, *CL_CrlAbortQuery*

NAME

CL_CrlGetNextFieldValue

SYNOPSIS

```
CSSM_DATA_PTR CSSMCLI CL_CrlGetNextFieldValue
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function returns the value of a CRL field, when that field occurs multiple times in a CRL. CRLs with repeated fields (such as revocation records) have multiple field values corresponding to a single OID. A call to the function CSSM_CL_CrlGetFirstFieldValue initiates the process and returns a results handle identifying the CRL from which values are being obtained and the OID corresponding to those values. The CSSM_CL_CrlGetNextFieldValue function can be called repeatedly to obtain these values one at a time.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a CRL query.

RETURN VALUE

Returns a pointer to a CSSM_DATA structure containing the next field in the CRL, which matched the CrlField specified in the CL_CrlGetFirstFieldValue function. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_RESULTS_HANDLE
Invalid results handle.

CSSM_CL_MEMORY_ERROR
Not enough memory.

CSSM_CL_NO_FIELD_VALUES
No field values for this results handle.

CSSM_CL_CRL_GET_FIELD_VALUE_FAIL
Unable to get field value.

SEE ALSO

CL_CrlGetFirstFieldValue, CL_CrlAbortQuery

NAME

CL_CrlAbortQuery

SYNOPSIS

```
CSSM_RETURN CSSMCLI CL_CrlAbortQuery
    (CSSM_CL_HANDLE CLHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function terminates the query initiated by CL_CrlGetFirstFieldValue and allows the CL to release all intermediate state information associated with the get operation.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

ResultsHandle (input)
The handle that identifies the results of a CRL query.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_RESULTS_HANDLE
Invalid results handle.

CSSM_CL_CRL_ABORT_QUERY_FAIL
Unable to abort query.

SEE ALSO

CL_CrlGetFirtsFieldValue, CL_CrlGetNextFieldValue

NAME

CL_CrlDescribeFormat

SYNOPSIS

```
CSSM_OID_PTR CSSMCLI CL_CrlDescribeFormat
    (CSSM_CL_HANDLE CLHandle,
     uint32 *NumberOfFields)
```

DESCRIPTION

This function returns a list of the object identifiers used to describe the CRL format supported by the specified CL.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

NumberOfFields (output)
The length of the output array.

RETURN VALUE

A pointer to the array of CSSM_OID structures which are supported for CRL operations in the specified CL module. If the return pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid handle.

CSSM_CL_INVALID_POINTER
Invalid pointer.

CSSM_CL_MEMORY_ERROR
Error allocating memory.

CSSM_CL_CRL_DESCRIBE_FORMAT_FAIL
Unable to return the list of fields.

48.4 Extensibility Functions

The CL_PassThrough function is provided to allow CL developers to extend the certificate and CRL format-specific functionality of the CSSM API. Because it is only exposed to CSSM as a function pointer, its name internal to the certificate library can be assigned at the discretion of the CL module developer. However, its parameter list and return value must match what is shown below. The error codes given in this section constitute the generic error codes, which may be used by all certificate libraries to describe common error conditions. Certificate library developers may also define their own module-specific error codes, as described in Section 3.5.2.

NAME

CL_PassThrough

SYNOPSIS

```
void * CSSMCLI CL_PassThrough
    (CSSM_CL_HANDLE CLHandle,
    CSSM_CC_HANDLE CCHandle,
    uint32 PassThroughId,
    const void * InputParams)
```

DESCRIPTION

This function allows applications to call certificate library module-specific operations. Such operations may include queries or services that are specific to the domain represented by the CL module.

PARAMETERS

CLHandle (input)
The handle that describes the add-in certificate library module used to perform this function.

CCHandle (input/optional)
The handle that describes the context of the cryptographic operation.

PassThroughId (input)
An identifier assigned by the CL module to indicate the function to perform.

InputParams (input)
A pointer to a module implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested CL module.

RETURN VALUE

A pointer to a module implementation-specific structure containing the output from the pass-through function. The output data must be interpreted by the calling application based on externally available information. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_CL_INVALID_CL_HANDLE
Invalid Certificate Library Handle.

CSSM_CL_INVALID_CC_HANDLE
Invalid Cryptographic Context Handle.

CSSM_CL_INVALID_DATA_POINTER
Invalid pointer input.

CSSM_CL_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_CL_PASS_THROUGH_FAIL
Unable to perform pass through.

CAE Specification

Part 12:

CSSM Data Storage Library Interface

The Open Group

Chapter 49

Introduction

49.1 CDSA Add-In Module Overview

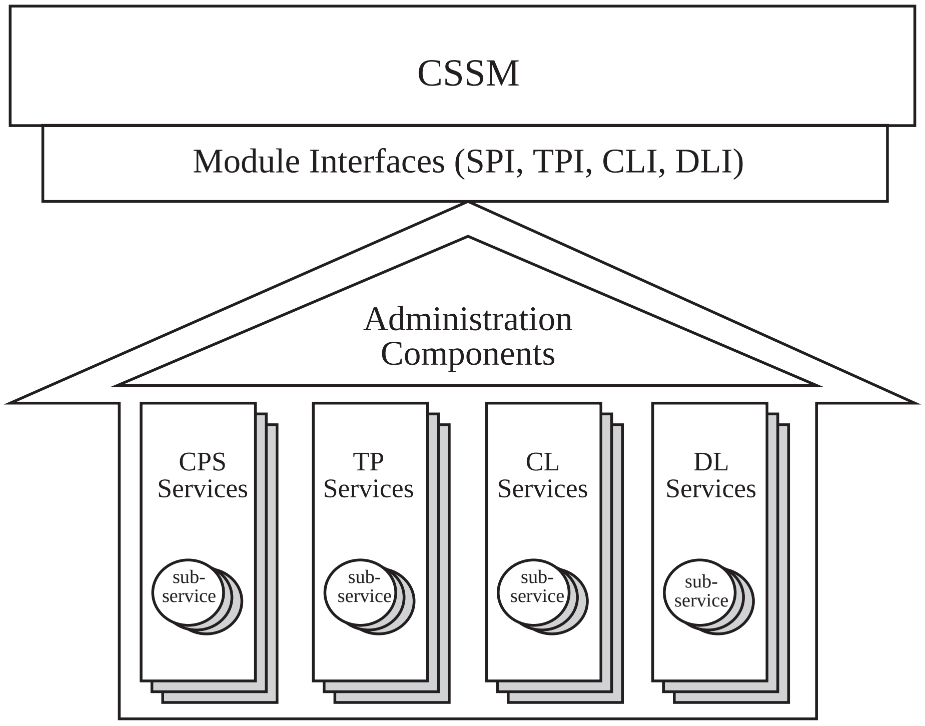

Figure 49-1 CDSA Add-In Module Structure

A CDSA add-in module is a dynamically-linkable library, composed of functions that implement some or all of the CSSM Module Interfaces. Add-in module functionality is partitioned into two areas:

- The provision of security services to applications
- Module administration

Add-in modules provide one or more categories of security services to applications. The service categories are Cryptographic Service Provider (CSP) services, Trust Policy (TP) services, Certificate Library (CL) services, and Data Storage Library (DL) services. Each security service contains one or more implementation instances, called sub-services. For a CSP service providing access to hardware tokens, a sub-service would represent a slot. For a DL service provider, a sub-service would represent a type of persistent storage. These sub-services each support the module interface for their respective service categories. This documentation-part describes the module interface functions in the DL service category. More information about CSP services can be found in the *CSSM Cryptographic Service Provider Interface Specification*. More information about TP services can be found in the *CSSM Trust Policy Interface Specification*. More information about CL services can be found in the *CSSM Certificate Library Interface Specification*.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions which allow CSSM to indicate events such as module attach and detach. In addition, as part of the attach operation, every module must be able to verify its own integrity, verify the integrity of CSSM, and register with CSSM. Detailed information about add-in module structure, administration, and interfaces can be found in the *CSSM Add-in Module Structure and Administration Specification*.

49.2 Data Storage Library Overview

A module with data storage library (DL) services provides access to persistent data stores of certificates, certificate revocation lists (CRLs), keys, policies and other security-related objects. Stable storage can be provided by a:

- Commercially-available database management system product
- Directory service
- Custom hardware-based storage device
- Native file system

The implementation of DL operations should be semantically free. For example, a DL operation which inserts a trusted X.509 certificate into a data store should not be responsible for verifying the trust on that certificate. The semantic interpretation of security objects should be implemented in TP services, layered services, and applications.

A pass-through function is defined in the DL API. This mechanism allows each DL to provide additional functions to store and retrieve certificates, CRLs and other security-related objects. Pass-through functions may be used to increase functionality or enhance performance.

Chapter 50

Data Storage Library Interface

50.1 Overview

The primary purpose of a data storage library (DL) is to provide access to persistent stores of security-related objects. The DL provides access to these data stores by translating calls from the DLI into the native interface of the data store. The native interface of the data store may be that of a database management system package, a directory service, a custom storage device, or a traditional local or remote file system. Applications are able to obtain information about the available DL services by using the CSSM_GetModuleInfo function to query the CSSM registry. The information about the DL service includes:

- Vendor information. Information about the module vendor, a text description of the DL and the module version number.
- Types of supported data stores. The module may support one or more types of persistent data stores as separate sub-services. For each type of data store, the DL provides information on the supported query operators and optionally provides specific information on the accessible data stores.

The data storage library may chose to provide information about the data stores that it has access to. Applications can obtain information about these data stores by using the CSSM_GetModuleInfo function call. The information about the data store includes:

- Types of persistent security objects. The types of security objects which may be stored include certificates, certificate revocation lists (CRLs), keys, policy objects, and generic data objects. A single data store can contain a single object type in one format, a single object type in multiple formats, or multiple object types.
- Attributes of persistent security objects. The stored security object may have attributes which must be included by the calling application on data insertion and which are returned by the DL on data retrieval.
- Data store indexes. These indexes are high-performance query paths constructed as part of data store creation and maintained by the data store.
- Secure Access Mechanisms. A data store may restrict a user’s ability to perform certain actions on the data store or on the data store’s contents. This structure exposes the mechanism required to authenticate to the data store.
- Record Integrity Capabilities. Some data stores will insure the integrity of the data store’s contents. To insure the integrity of the data store’s contents, the data store is expected to sign and verify each record.
- Data store location. The persistent repository can be local or remote.

To build indexes or to satisfy an application’s request for record retrieval, the data store may need to parse the stored security objects. If the application has invoked CSSM_DL_SetDbRecordParsingFunctions for a given security object type, those functions will be used to parse that security object as the need arises. If the application has not explicitly set record parsing functions, the default add-in modules set by the data store creator will be used for parsing.

To ensure a minimal level of interoperability among applications and DL modules, CSSM requires that all DL modules recognize and support two pre-defined attribute names for all record types. All applications can use these strings as valid attribute names even if no value is stored in association with the attribute name.

Secured access to the data store and to the data store's contents may be enforced by the data storage library, the data store or both. The partitioning of authentication responsibility is exposed via the DL and DB authentication mechanisms available from the CSSM registry.

Data stores may be added to a data storage library in one of three ways:

- Using DL_DbCreate. This creates and opens a new, empty data store with the specified schema.
- Using DL_DbImport with information and data. If the specified data store does not exist, a new data store is created with the specified schema and the exported data records.
- Using DL_DbImport with information only. In this case, the data store's native format is the same as that managed by the DL service. Importing its information makes it accessible via this DL service.

In all cases, it is the responsibility of the DL service to update the CSSM registry with information about the new data store. This can be accomplished by making use of the CSSM_GetModuleInfo and CSSM_SetModuleInfo functions.

50.1.1 Categories of Operations

The data storage library SPI defines four categories of operations:

- Data storage library operations
- Data store operations
- Data record operations
- Extensibility operations

Data storage library operations are used to control access to the data storage library. They include:

- Authentication to the DL Module. A user may be required to present valid credentials to the data storage library prior to accessing any of the data stores embedded in the DL module. The DL module will be responsible for insuring that the user does not exceed his/her access privileges.

  The data store functions operate on a data store as a single unit. These operations include:

- Opening and closing data stores. A DL service manages the mapping of logical data store names to the storage mechanisms it uses to provide persistence. The caller uses logical names to reference persistent data stores. The open operation prepares an existing data store for future access by the caller. The close operation terminates current access to the data store by the caller.
- Creating and deleting data stores. A DL creates a new, empty data store and opens it for future access by the caller. An existing data store may be deleted. Deletion discards all data contained in the data store.
- Importing and exporting data stores. Occasionally a data store must be moved from one system to another or a DL service may need to provide access to an existing data store. The import and export operations may be used in conjunction to support the transfer of an entire data store. The export operation prepares a snapshot of a data store. (Export does not delete

the data store it snapshots.) The import operation accepts a snapshot (generated by the export operation) and includes it in a new or existing data store managed by a DL. Alternately, the import operation may be used independently to register an existing data store with a DL.

The data record operations operate on a single record of a data store. They include:

- Adding new data objects. A DL adds a persistent copy of data object to an open data store. This operation may or may not include the creation of index entries. The mechanisms used to store and retrieve persistent data objects are private to the implementation of a DL module.
- Deleting data objects. A DL removes single data object from the data store.
- Retrieving data objects. A DL provides a search mechanism for selectively retrieving a copy of persistent security objects. Selection is based on a selection criterion.

Data store extensibility operations include:

- Pass-through for unique, module-specific operations. A pass-through function is included in the data storage library interface to allow data store libraries to expose additional services beyond what is currently defined in the CSSM API. CSSM passes an operation identifier and input parameters from the application to the appropriate data storage library. Within the DL_PassThrough function in the data storage library, the input parameters are interpreted and the appropriate operation performed. The data storage library developer is responsible for making known to the application the identity and parameters of the supported pass-through operations.

50.1.2 Data Storage Library Operations

CSSM_RETURN CSSMDLI DL_Authenticate( )
  Authenticates a user’s ability to use this DL for accessing the underlying data stores.

50.1.3 Data Store Operations

CSSM_DB_HANDLE CSSMDLI DL_DbOpen( )
  For authorized users, this opens a data store with the specified logical name in the requested access mode. Returns a handle to the data store.

CSSM_RETURN CSSMDLI DL_DbClose( )
  Closes a data store.

CSSM_DB_HANDLE CSSMDLI DL_DbCreate( )
  Creates and opens a new, empty data store with the specified logical name and the specified schema. As a side effect, the DL updates the CSSM Registry to expose information about the new data store.

CSSM_RETURN CSSMDLI DL_DbDelete( )
  For authorized users, this deletes all records from the specified data store and removes current state information associated with that data store.

CSSM_RETURN CSSMDLI DL_DbImport( )
  Accepts as input a flag for what to import, a filename, a logical name, and a schema for a data store. If information about the data store is being imported, then the DL updates its list of accessible data stores to include this new data store with the specified schema. If the contents of the data store are being imported, then the file contains an exported copy of an existing data store. The data records contained in the file must be in the native format of a data store. The DL imports all security objects in the file (such as certificates and CRLs),

creating a new data record for each. If the specified logical name is that of an existing data store, the new records will be added to the data store. Otherwise, a new data store will be created with the specified schema to hold the new records.

Note: This mechanism can be used to copy data stores among systems or to restore a persistent data store from a backup copy. It could also be used to import data stores that were created and managed by other DLs, but this is not the typical implementation and use of this interface.

CSSM_RETURN CSSMDLI DL_DbExport( )
Accepts as input the logical name of a data store and the name of a target output file. The specified data store contains persistent data records. A representation of the schema for the data store being exported is written to the file along with a copy of each data record in the data store.

Note: This mechanism can be used to copy data stores among systems or to create a backup of persistent data stores.

CSSM_RETURN CSSMDLI DL_DbSetRecordParsingFunctions( )
Sets the functions to be used for parsing the specified type of security object.

CSSM_DB_RECORD_PARSING_FNTABLE_PTR CSSMDLI
DL_DbGetRecordParsingFunctions( ) Returns the function pointers in use for parsing the specified type of security object.

char * CSSMDLI DL_GetDbNameFromHandle( )
Retrieves the data source name corresponding to an opened database handle.

CSSM_NAME_LIST_PTR CSSMDLI DL_GetDbNames( )
Returns the names of data stores accessible via this DL module.

CSSM_RETURN CSSMDLI DL_FreeNameList( )
Frees the list of data stores returned by a call to DL_GetDbNames.

50.1.4 Data Record Operations

CSSM_DB_UNIQUE_RECORD_PTR CSSMDLI DL_DataInsert( )
Accepts as input a handle to a data store, the type of the security object, the attributes of the object and the object itself. The data object and its attributes are made persistent in the specified data store. This may or may not include the creation of index entries, and so on. The DL module will return to the calling application a unique identifier for the input record which may be used to rapidly retrieve the security object. The mechanisms used to store and retrieve persistent security objects is private to the implementation of the Data Storage Library.

CSSM_RETURN CSSMDLI DL_DataDelete( )
Accepts as input a handle to a data store and a unique identifier of the security object. The object is removed from the data store. If the object is not found in the specified data store, or if the user does not have deletion permissions, the operation fails.

CSSM_RETURN CSSMDLI DL_DataModify( )
Accepts as input a handle to a data store, the type of the security object, a unique record identifier, the attributes to be modified and the data to be modified. If present, the attributes are added, replaced, or deleted depending on the input attribute values and the current contents of the record attributes. If present, the record data is replaced with the input data. An indicator of the operation's success or failure is returned. Either all or none of the requested modifications will have occurred.

CSSM_DATA_PTR CSSMDLI DL_DataGetFirst( )

Accepts as input a handle to a data store, a query, and a list of the attributes to be retrieved. The query is composed of the type of data record to be retrieved, a selection predicate, and any limits or flags on the query. Selection predicates are represented as a set of (relational operator, attribute) pairs that are connected by a conjunctive operator. Query limits provide a mechanism for the user to specify upper bounds on the search time and/or the number of records retrieved. Not all DL modules will support query limits. The specified data store is searched for data objects of the specified type that match the selection criteria. This function returns the first data object matching the criteria together with the requested attributes and a unique identifier for use in future references. If additional objects matched, a selection handle is returned that may be used to retrieve the subsequent objects. A data storage library may limit the number of concurrently managed selection handles to exactly one. The library developer must document all such restrictions and application developers should proceed accordingly.

CSSM_DATA_PTR CSSMDLI DL_DataGetNext( )

Accepts as input a selection results handle that was returned by an invocation of the function CSSM_DL_DataGetFirst and a list of the attributes to be returned. If there are no more records to retrieve, the EndOfDataStore flag is set to CSSM_TRUE and the function returns NULL. Otherwise, a DL module returns the next data record, the requested attributes, and a unique identifier, from the set specified by the selection results handle. A data storage library may limit the number of concurrently-managed selection result handles to exactly one. The library developer must document such restrictions, and application developers should proceed accordingly.

CSSM_RETURN CSSMDLI DL_DataAbortQuery( )

Cancels the query initiated by CSSM_DL_DataGetFirst function and resets the selection results handle.

CSSM_RETURN CSSMDLI DL_DataGetFromUniqueRecordId( )

Accepts as input a unique record identifier and a list of the attributes to be retrieved. The specified data store is searched for the record corresponding to the unique record identifier. This function returns the data object together with the requested attributes.

CSSM_RETURN CSSMDLI DL_FreeUniqueRecord( )

Frees the memory associated with the input unique record structure.

50.1.5 Extensibility Functions

void * CSSMDLI DL_PassThrough( )

Accepts as input an operation ID and a set of arbitrary input parameters. The operation ID may specify any type of operation a DL wishes to export for use by an application or by another module. Such operations may include queries or services that are specific to certain types of security objects or specific types of data stores managed by a DL module. It is the responsibility of the DL developer to make information on the availability and usage of passthrough operations available to application developers.

50.2 Data Storage Data Structures

50.2.1 CSSM_DL_HANDLE

A unique identifier for an attached module that provides data storage library services.

```
typedef uint32 CSSM_DL_HANDLE  /* Data Storage Library Handle */
```

50.2.2 CSSM_DB_HANDLE

A unique identifier for an open data store.

```
typedef uint32 CSSM_DB_HANDLE    /* Data Store Handle */
```

50.2.3 CSSM_DL_DB_HANDLE

This data structure holds a pair of handles, one for a data storage library and another for a data store opened and being managed by the data storage library.

```
typedef struct cssm_dl_db_handle {
    CSSM_DL_HANDLE DLHandle;
    CSSM_DB_HANDLE DBHandle;
} CSSM_DL_DB_HANDLE, *CSSM_DL_DB_HANDLE_PTR;
```

Definition

DLHandle
Handle of an attached module that provides DL services.

DBHandle
Handle of an open data store that is currently under the management of the DL module specified by the DLHandle.

50.2.4 CSSM_DL_DB_LIST

This data structure defines a list of handle pairs of (data storage library handle, data store handle).

```
typedef struct cssm_dl_db_list {
    uint32 NumHandles;
    CSSM_DL_DB_HANDLE_PTR DLDBHandle;
} CSSM_DL_DB_LIST, *CSSM_DL_DB_LIST_PTR;
```

Definition

NumHandles
Number of (data storage library handle, data store handle) pairs in the list.

DLDBHandle
List of (data storage library handle, data store handle) pairs.

50.2.5 CSSM_DB_ATTRIBUTE_NAME_FORMAT

This enumerated list defines the two formats used to represent an attribute name. The name can be represented by a character string in the native string encoding of the platform or the name can be represented by an opaque OID structure that is interpreted by the DL module.

```
typedef enum cssm_db_attribute_name_format {
    CSSM_DB_ATTRIBUTE_NAME_AS_STRING = 0,
    CSSM_DB_ATTRIBUTE_NAME_AS_OID = 1,
} CSSM_DB_ATTRIBUTE_NAME_FORMAT, *CSSM_DB_ATTRIBUTE_NAME_FORMAT_PTR;
```

50.2.6 CSSM_DB_ATTRIBUTE_FORMAT

This enumerated list defines the formats for attribute values. Many data storage library modules manage only one attribute format, CSSM_DB_ATTRIBUTE_FORMAT_STRING. These format indicators can be set by application caller or by the data storage library module. When a caller selects an format option that cannot be supported by the library module, the module can ignore the caller-declared format.

```
typedef enum cssm_db_attribute_format {
    CSSM_DB_ATTRIBUTE_FORMAT_STRING = 0,
    CSSM_DB_ATTRIBUTE_FORMAT_INTEGER = 1,
    CSSM_DB_ATTRIBUTE_FORMAT_REAL = 2,
    CSSM_DB_ATTRIBUTE_FORMAT_TIME = 3,
    CSSM_DB_ATTRIBUTE_FORMAT_MONEY = 4,
    CSSM_DB_ATTRIBUTE_FORMAT_BLOB = 5,
} CSSM_DB_ATTRIBUTE_FORMAT, *CSSM_DB_ATTRIBUTE_FORMAT;
```

50.2.7 CSSM_DB_ATTRIBUTE_INFO

This data structure describes an attribute of a persistent record. The description is part of the schema information describing the structure of records in a data store. The description includes the format of the attribute name and the attribute name itself. The attribute name implies the underlying data type of a value that may be assigned to that attribute.

```
typedef struct cssm_db_attribute_info {
    CSSM_DB_ATTRIBUTE_NAME_FORMAT AttributeNameFormat;
    union cssm_db_attribute_label {
        char * AttributeName;  /* e.g., "record label" */
        CSSM_OID AttributeID;  /* e.g., CSSMOID_RECORDLABEL */
    } Label;
    CSSM_DB_ATTRIBUTE_FORMAT AttributeFormat;
} CSSM_DB_ATTRIBUTE_INFO, *CSSM_DB_ATTRIBUTE_INFO_PTR;
```

Definition

AttributeNameFormat
Indicates which of the two formats was selected to represent" the attribute name.

AttributeName
A character string representation of the attribute name.

AttributeID
A DER-encoded OID representation of the attribute name.

AttributeFormat
Indicates the format of the attribute.The Data Storage Library may not support more than one format, typically CSSM_DB_ATTRIBUTE_FORMAT_STRING. In this case, the library module can ignore any format specification provided by the caller.

50.2.8 CSSM_DB_ATTRIBUTE_DATA

This data structure holds an attribute value that can be stored in an attribute field of a persistent record. The structure contains a value for the data item and a reference to the meta information (typing information and schema information) associated with the attribute.

```
typedef struct cssm_db_attribute_data {
    CSSM_DB_ATTRIBUTE_INFO Info;
    CSSM_DATA Value;
} CSSM_DB_ATTRIBUTE_DATA, *CSSM_DB_ATTRIBUTE_DATA_PTR;
```

Definition

Info
A reference to the meta-information/schema describing this attribute in relationship to the data store at large.

Value
The data-present value assigned to the attribute.

To ensure a minimal level of interoperability among applications and DL modules, CSSM requires that all DL modules recognize and support two pre-defined attribute names for all record types:

- PrintName: a printable or viewable string name associated with the record
- Alias: an arbitrary value associated with the record. The value can be non-printable.

Applications that create new data stores and define the associated schema are encouraged to define these attributes as part of the schema. If the data store creator does not define these attributes, the DL module must add these attributes with the following minimum storage size requirements

- PrintName: the associated value is a string of maximum length 16 characters
- Alias: the associated value is an arbitrary data type of maximum length 8 bytes

Applications are encouraged to provide values for these attributes when creating data store records, but values are not required. All applications can use these strings as valid attribute names even if no value is stored in association with this attribute name. When no value is associated with a pre-defined attribute name, it is possible for a DL module that encapsulates a data store schema to return one of the following:

- A module-defined default value
- A value selected from a database-key attribute in the data store
- A NULL value

The CSSM_DB_ATTRIBUTE_DATA structure for the pre-defined attribute name "PrintName" contains the following values:

```
  {  AttributeNameFormat = CSSM_DB_ATTRIBUTE_NAME_AS_STRING
     AttributeName = "PrintName"
     Value = <a value in a CSSM_DATA structure>  }
```

50.2.9 CSSM_DB_RECORDTYPE

This enumerated list defines the categories of persistent security-related objects that can be managed by a data storage library module. These categories are in one-to-one correspondence with types of records that can be managed by a data store.

```
typedef enum cssm_db_recordtype {
    CSSM_DL_DB_RECORD_GENERIC = 0,
    CSSM_DL_DB_RECORD_CERT = 1,
    CSSM_DL_DB_RECORD_CRL = 2,
    CSSM_DL_DB_RECORD_KEY = 3,
    CSSM_DL_DB_RECORD_POLICY = 4,
} CSSM_DB_RECORDTYPE;
```

50.2.10 CSSM_DB_CERTRECORD_SEMANTICS

These bit masks define a list of usage semantics for how certificates may be used. It is anticipated that additional sets of bit masks will be defined listing the usage semantics of how other record types can be used, such as CRL record semantics, key record semantics, policy record semantics, and so on.

```
#define CSSM_DB_CERT_USE_TRUSTED 0x00000001
                         /* application-defined as trusted */
#define CSSM_DB_CERT_USE_SYSTEM  0x00000002
                         /* the CSSM system cert */
#define CSSM_DB_CERT_USE_OWNER   0x00000004
                         /* private key owned by system user*/
#define CSSM_DB_CERT_USE_REVOKED 0x00000008
                         /* revoked cert - used w\ CRL APIs */
#define CSSM_DB_CERT_USE_SIGNING 0x00000010
                         /* use cert for signing only */
#define CSSM_DB_CERT_USE_PRIVACY 0x00000020
                         /* use cert for confidentiality only */
```

Record semantic designations are advisory only. For example, the designation CSSM_DB_CERT_USE_OWNER suggests that the private key associated with the public key contained in the certificate is local to the system. This statement was probably true when the certificate was created. Various actions could make this assertion false. The private key could have expired, been revoked, or be stored in a portable cryptographic storage device that is not currently resident on the system. The validity of the advisory designation CSSM_DB_CERT_USE_TRUSTED should be verified using standard certificate verification procedures. Although these designators are advisory, application or trust policies can choose to use this information if it is useful for their purpose. For example, a trust policy can define how advisory designations can be used when full policy evaluation requires connection to a remote facility that is currently inaccessible.

Management practices for record semantic designators define the agent and the time when a data store record can be assigned a particular designator value. Reasonable usage is described as follows:

Designation Value	**Assigning Time**	**Assigning Agents**
CSSM_DB_CERT_USE_TRUSTED	Local record creation time Remote record creation time Reset at any time	Sys Admin App App / Record Owner
CSSM_DB_CERT_USE_SYSTEM	Local record creation time Should not be reset	Sys Admin App
CSSM_DB_CERT_USE_OWNER	Local record creation time Reset at any time	App / Record Owner
CSSM_DB_CERT_USE_REVOKED	Set once only	Sys Admin App App / Record Owner
CSSM_DB_CERT_SIGNING	Local record creation time	Remote Authority Local Authority Record Owner
CSSM_DB_CERT_PRIVACY	Local record creation time	Remote Authority Local Authority Record Owner

50.2.11 CSSM_DB_RECORD_ATTRIBUTE_INFO

This structure contains the meta information or schema information about all of the attributes in a particular record type. The description specifies the record type, the number of attributes in the record type, and a type information for each attribute. This description includes the CSSM pre-defined attributes named "PrintName" and "Alias".

```
typedef struct cssm_db_record_attribute_info {
    CSSM_DB_RECORDTYPE DataRecordType;
    uint32 NumberOfAttributes;
    CSSM_DB_ATTRIBUTE_INFO_PTR AttributeInfo;
} CSSM_DB_RECORD_ATTRIBUTE_INFO, *CSSM_DB_RECORD_ATTRIBUTE_INFO_PTR;
```

Definition

DataRecordType
A CSSM_DB_RECORDTYPE.

NumberOfAttributes
The number of attributes in a record of the specified type.

AttributeInfo
A list of pointers to the type (schema) information for each of the attributes.

50.2.12 CSSM_DB_RECORD_ATTRIBUTE_DATA

This structure aggregates the actual data values for all of the attributes in a single record.

```
typedef struct cssm_db_record_attribute_data {
    CSSM_DB_RECORDTYPE DataRecordType;
    uint32 SemanticInformation;
    uint32 NumberOfAttributes;
    CSSM_DB_ATTRIBUTE_DATA_PTR AttributeData;
} CSSM_DB_RECORD_ATTRIBUTE_DATA, *CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR;
```

Definition

DataRecordType
A CSSM_DB_RECORDTYPE.

SemanticInformation
A bit mask of type CSSM_XXXRECORD_SEMANTICS defining how the record can be used. Currently these bit masks are defined only for certificate records (CSSM_CERTRECORD_SEMANTICS). For all other record types, a bit mask of zero must be used or a set of semantically meaningful masks must be defined.

NumberOfAttributes
The number of attributes in the record of the specified type.

AttributeData
A list of attribute name/value pairs. If no stored value is associated with this attribute, the attribute data pointer is NULL.

50.2.13 CSSM_DB_RECORD_PARSING_FNTABLE

This structure defines the three prototypes for functions that can parse the opaque data object stored in a record. It is used in the CSSM_DbSetRecordParsingFunctions function to override the default parsing module for a given record type. The DL module developer designates the default parsing module for each record type stored in the data store.

```
typedef struct cssm_db_record_parsing_fntable {
    CSSM_DATA_PTR (CSSMAPI *RecordGetFirstFieldValue)
        (CSSM_HANDLE Handle,
        const CSSM_DATA_PTR Data,
        const CSSM_OID_PTR DataField,
        CSSM_HANDLE_PTR ResultsHandle,
        uint32 *NumberOfMatchedFields);
    CSSM_DATA_PTR (CSSMAPI *RecordGetNextFieldValue)
        (CSSM_HANDLE Handle,
        CSSM_HANDLE ResultsHandle);
    CSSM_RETURN (CSSMAPI *RecordAbortQuery)
        (CSSM_HANDLE Handle,
        CSSM_HANDLE ResultsHandle);
} CSSM_DB_RECORD_PARSING_FNTABLE, *CSSM_DB_RECORD_PARSING_FNTABLE_PTR;
```

Definition

**RecordGetFirstFieldValue*
A function to retrieve the value of a field in the opaque object. The field is specified by attribute name. The results handle holds the state information required to retrieve subsequent values having the same attribute name.

**RecordGetNextFieldValue*
A function to retrieve subsequent values having the same attribute name from a record parsed by the first function in this table.

**RecordAbortQuery*
Stop subsequent retrieval of values having the same attribute name from within the opaque object.

50.2.14 CSSM_DB_PARSING_MODULE_INFO

This structure aggregates the persistent subservice ID of a default parsing module with the record type that it parses. A parsing module can parse multiple record types. The same ID would be repeated with each record type parsed by the module.

```
typedef struct cssm_db_parsing_module_info {
    CSSM_DB_RECORDTYPE  RecordType;
    CSSM_SUBSERVICE_UID   ModuleSubserviceUid;
} CSSM_DB_PARSING_MODULE_INFO, *CSSM_DB_PARSING_MODULE_INFO_PTR;
```

Definition

RecordType
The type of record parsed by the module specified by GUID.

ModuleSubserviceUid
A persistent subservice ID identifying the default parsing module for the specified record type.

50.2.15 CSSM_DB_INDEX_TYPE

This enumerated list defines two types of indexes: indexes with unique values (that is, primary database keys) and indexes with non-unique values. These values are used when creating a new data store and defining the schema for that data store.

```
typedef enum cssm_db_index_type {
    CSSM_DB_INDEX_UNIQUE = 0,
    CSSM_DB_INDEX_NONUNIQUE = 1
} CSSM_DB_INDEX_TYPE;
```

50.2.16 CSSM_DB_INDEXED_DATA_LOCATION

This enumerated list defines where within a record the indexed data values reside. Indexes can be constructed on attributes or on fields within the opaque object in the record. However, the logical location of the index value between these two categories may be unknown by the user of this enum.

```
typedef enum cssm_db_index_data_location {
    CSSM_DB_INDEX_ON_UNKNOWN = 0,
    CSSM_DB_INDEX_ON_ATTRIBUTE = 1,
    CSSM_DB_INDEX_ON_RECORD = 2
} CSSM_DB_INDEX_DATA_LOCATION;
```

50.2.17 CSSM_DB_INDEX_INFO

This structure contains the meta information or schema description of an index defined on an attribute. The description includes the type of index (for example, unique key or non-unique key), the logical location of the indexed attribute in the CSSM record (for example, an attribute or a field within the opaque object in the record), and the meta information on the attribute itself.

```
typedef struct cssm_db_index_info {
    CSSM_DB_INDEX_TYPE IndexType;
    CSSM_DB_INDEXED_DATA_LOCATION IndexedDataLocation;
    CSSM_DB_ATTRIBUTE_INFO Info;
} CSSM_DB_INDEX_INFO, *CSSM_DB_INDEX_INFO_PTR;
```

Definition

IndexType
A CSSM_DB_INDEX_TYPE.

IndexedDataLocation
A CSSM_DB_INDEXED_DATA_LOCATION.

Info
The meta information description of the attribute being indexed.

50.2.18 CSSM_DB_UNIQUE_RECORD

This structure contains an index descriptor and a module-defined value. The index descriptor may be used by the module to enhance the performance when locating the record. The module-defined value must uniquely identify the record. For a DBMS, this may be the record data. For a PKCS #11 DL, this may be an object handle. Alternately, the DL may have a module-specific scheme for identifying data which has been inserted or retrieved.

```
typedef struct cssm_db_unique_record {
    CSSM_DB_INDEX_INFO RecordLocator;
    CSSM_DATA RecordIdentifier;
} CSSM_DB_UNIQUE_RECORD, *CSSM_DB_UNIQUE_RECORD_PTR;
```

Definition

RecordLocator
The information describing how to locate the record efficiently.

RecordIdentifier
A module-specific identifier which will allow the DL to locate this record.

50.2.19 CSSM_DB_RECORD_INDEX_INFO

This structure contains the meta information or schema description of the set of indexes defined on a single record type. The description includes the type of the record, the number of indexes and the meta information describing each index. The data store creator can specify an index over a CSSM pre-defined attribute. When no index has been defined, the DL module has the option to add an index over a CSSM pre-defined attribute or any other attribute defined by the data store creator.

```
typedef struct cssm_db_record_index_info {
    CSSM_DB_RECORDTYPE DataRecordType;
    uint32 NumberOfIndexes;
    CSSM_DB_INDEX_INFO_PTR IndexInfo;
} CSSM_DB_RECORD_INDEX_INFO, *CSSM_DB_RECORD_INDEX_INFO_PTR;
```

Definition

DataRecordType
A CSSM_DB_RECORDTYPE.

NumberOfIndexes
The number of indexes defined on the records of the given type.

IndexInfo
An array containing a description of each index defined over the specified record type.

50.2.20 CSSM_DB_ACCESS_TYPE

This bitmask describes a user's desired level of access to a data store.

```
typedef uint32 CSSM_DB_ACCESS_TYPE, *CSSM_DB_ACCESS_TYPE_PTR;

#define CSSM_DB_ACCESS_READ         0x00001
#define CSSM_DB_ACCESS_WRITE        0x00002
#define CSSM_DB_ACCESS_PRIVILEGED   0x00004
                                        /* versus user mode */
#define CSSM_DB_ACCESS_ASYNCHRONOUS 0x00008
                                        /* versus synchronous */
```

Definition

ReadAccess
A boolean indicating that the user requests read access.

WriteAccess
A boolean indicating that the user requests write access.

PrivilegedMode
A boolean indicating that the user requests privileged operations, such as modifying data store access rights.

Asynchronous
A boolean indicating that the user requests asynchronous access.

50.2.21 CSSM_DBINFO

This structure contains the meta-information about an entire data store. The description includes the types of records stored in the data store, the attribute schema for each record type, the index schema for all indexes over records in the data store, the type of authentication mechanism used to gain access to the data store, and other miscellaneous information used by the DL module to manage the data store in a secure manner.

```
typedef struct cssm_dbinfo {
/* meta information about each record type in this data store
including meta information about record attributes and indexes */
    uint32 NumberOfRecordTypes;
    CSSM_DB_PARSING_MODULE_INFO_PTR DefaultParsingModules;
    CSSM_DB_RECORD_ATTRIBUTE_INFO_PTR RecordAttributeNames;
    CSSM_DB_RECORD_INDEX_INFO_PTR RecordIndexes;

  /* access restrictions for opening this data store */
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;

  /* transparent integrity checking options for this data store */
    CSSM_BOOL RecordSigningImplemented;
    CSSM_DATA SigningCertificate;
    CSSM_SUBSERVICE_UID SigningCspSubserviceUid;

    /* additional information */
    CSSM_BOOL IsLocal;
    char *AccessPath;           /* URL, dir path, etc. */
    void *Reserved;
} CSSM_DBINFO, *CSSM_DBINFO_PTR;
```

Definition

NumberOfRecordTypes
The number of distinct record types stored in this data store.

DefaultParsingModules
A pointer to a list of (record-type, GUID) pairs which define the default parsing module for each record type.

RecordAttributeNames
The meta (schema) information about the attributes associated with each record type that can be stored in this data store.

RecordIndexes
The meta (schema) information about the indexes that are defined over each of the record types that can be stored in this data store.

AuthenticationMechanism
Defines the authentication mechanism required when accessing this data store.

RecordSigningImplemented
A flag indicating whether or not the DL module provides record integrity service based on digital signaturing of the data store records.

SigningCertificate
The certificate used to sign data store records when the transparent record integrity option

is in effect.

SigningCspSubserviceUid
The persistent service ID for the cryptographic service provider to be used to sign data store records when the transparent record integrity option is in effect.

IsLocal
Indicates whether the physical data store is local.

AccessPath
A character string describing the access path to the data store, such as an URL, a file system path name, a remote directory service name, and so on.

Reserved
Reserved for future use.

50.2.22 CSSM_DB_OPERATOR

These are the logical operators which can be used when specifying a selection predicate.

```
typedef enum cssm_db_operator {
    CSSM_DB_EQUAL = 0,
    CSSM_DB_NOT_EQUAL = 1,
    CSSM_DB_APPROX_EQUAL = 2,
    CSSM_DB_LESS_THAN = 3,
    CSSM_DB_GREATER_THAN = 4,
    CSSM_DB_EQUALS_INITIAL_SUBSTRING = 5,
    CSSM_DB_EQUALS_ANY_SUBSTRING = 6,
    CSSM_DB_EQUALS_FINAL_SUBSTRING = 7,
    CSSM_DB_EXISTS = 8
} CSSM_DB_OPERATOR, *CSSM_DB_OPERATOR_PTR;
```

50.2.23 CSSM_DB_CONJUNCTIVE

These are the conjunctive operations which can be used when specifying a selection criterion.

```
typedef enum cssm_db_conjunctive{
    CSSM_DB_NONE = 0,
    CSSM_DB_AND = 1,
    CSSM_DB_OR = 2
} CSSM_DB_CONJUNCTIVE, *CSSM_DB_CONJUNCTIVE_PTR;
```

50.2.24 CSSM_SELECTION_PREDICATE

This structure defines the selection predicate to be used for data store queries.

```
typedef struct cssm_selection_predicate {
    CSSM_DB_OPERATOR DbOperator;
    CSSM_DB_ATTRIBUTE_DATA Attribute;
} CSSM_SELECTION_PREDICATE, *CSSM_SELECTION_PREDICATE_PTR;
```

Definition

DbOperator
: The relational operator to be used when comparing a value to the values stored in the specified attribute in the data store.

Attribute
: The meta information about the attribute to be searched and the attribute value to be used for comparison with values in the data store.

50.2.25 CSSM_QUERY_LIMITS

This structure defines the time and space limits a caller can set to control early termination of the execution of a data store query. The constant values CSSM_QUERY_TIMELIMIT_NONE and CSM_QUERY_SIZELIMIT_NONE should be used to specify no limit on the resources used in processing the query. These limits are advisory. Not all data storage library modules recognize and act upon the query limits set by a caller.

```
#define CSSM_QUERY_TIMELIMIT_NONE 0
#define CSSM_QUERY_SIZELIMIT_NONE 0

typedef struct cssm_query_limits {
    uint32 TimeLimit;     /* in seconds */
    uint32 SizeLimit;     /* max. number of records to return */
} CSSM_QUERY_LIMITS, *CSSM_QUERY_LIMITS_PTR;
```

Definition

TimeLimit
: Defines the maximum number of seconds of resource time that should be expended performing a query operation. The constant value CSSM_QUERY_TIMELIMIT_NONE means no time limit is specified.

SizeLimit
: Defines the maximum number of records that should be retrieved in response to a single query. The constant value CSSM_QUERY_SIZELIMIT_NONE means no space limit is specified.

50.2.26 CSSM_QUERY_FLAGS

These flags may be used by the application to request query-related operation, such as the format of the returned data.

```
typedef uint32 CSSM_QUERY_FLAGS

#define CSSM_QUERY_RETURN_DATA  0x1
                   /* On  = Return the data record
                   Off = Return a reference to the data record */
```

50.2.27 CSSM_QUERY

This structure holds a complete specification of a query to select records from a data store.

```
typedef struct cssm_query {
    CSSM_DB_RECORDTYPE RecordType;
    CSSM_DB_CONJUNCTIVE Conjunctive;
    uint32 NumSelectionPredicates;
    CSSM_SELECTION_PREDICATE_PTR SelectionPredicate;
    CSSM_QUERY_LIMITS QueryLimits;
    CSSM_QUERY_FLAGS QueryFlags;
} CSSM_QUERY, *CSSM_QUERY_PTR;
```

Definition

RecordType
Specifies the type of record to be retrieved from the data store.

Conjunctive
The conjunctive operator to be used in constructing the selection predicate for the query.

NumSelectionPredicates
The number of selection predicates to be connected by the specified conjunctive operator to form the query.

SelectionPredicate
The list of selection predicates to be combined by the conjunctive operator to form the data store query.

QueryLimits
Defines the time and space limits for processing the selection query. The constant values CSSM_QUERY_TIMELIMIT_NONE and CSM_QUERY_SIZELIMIT_NONE should be used to specify no limit on the resources used in processing the query.

QueryFlags
Query-related requests from the application.

50.2.28 CSSM_DLTYPE

This enumerated list defines the types of underlying data management systems that can be used by the DL module to provide services. It is the option of the DL module to disclose this information. It is anticipated that other underlying data servers will be added to this list over time.

```
typedef enum cssm_dltype {
    CSSM_DL_UNKNOWN = 0,
    CSSM_DL_CUSTOM = 1,
    CSSM_DL_LDAP = 2,
    CSSM_DL_ODBC = 3,
    CSSM_DL_PKCS11 = 4,
    CSSM_DL_FFS = 5, /* flat file system  or fast file system */
    CSSM_DL_MEMORY = 6,
    CSSM_DL_REMOTEDIR = 7
} CSSM_DLTYPE;
```

50.2.29 CSSM_DL_PKCS11_ATTRIBUTES

Each type of DL module can define it own set of type-specific attributes. This structure contains the attributes that are specific to a PKCS#11 compliant data storage device.

```
typedef void *CSSM_DL_CUSTOM_ATTRIBUTES;
typedef void *CSSM_DL_LDAP_ATTRIBUTES;
typedef void *CSSM_DL_ODBC_ATTRIBUTES;
typedef void *CSSM_DL_FFS_ATTRIBUTES;

typedef struct cssm_dl_pkcs11_attributes {
    uint32 DeviceAccessFlags;
} *CSSM_DL_PKCS11_ATTRIBUTE, *CSSM_DL_PKCS11_ATTRIBUTE_PTR;
```

Definition

DeviceAccessFlags
Specifies the PKCS#11-specific access modes applicable for accessing persistent objects in a PKCS#11 data store.

50.2.30 CSSM_DB_DATASTORES_UNKNOWN

Not all DL modules can maintain a summary of managed data stores. In this case, the DL module reports its number of data stores as CSSM_DB_DATASTORES_UNKNOWN. Data stores can (and probably do) exist, but the DL module cannot provide a list of them.

```
#define CSSM_DB_DATASTORES_UNKNOWN (0xFFFFFFFF)
```

50.2.31 CSSM_DL_WRAPPEDPRODUCT_INFO

This structure holds product information about all backend data store services used by the DL module. The DL module vendor is not required to provide this information, but may choose to do so. For example, a DL module that uses a commercial database management system can record information about that product in this structure. Another example is a DL module that supports certificate storage through an X.500 certificate directory server. The DL module can describe the X.500 directory service in this structure.

```
typedef struct cssm_dl_wrappedproductinfo {
    CSSM_VERSION StandardVersion;    /* Ver of standard the
                                        product conforms to */
    CSSM_STRING StandardDescription; /* Descr of standard the
                                        product conforms to */
    CSSM_VERSION ProductVersion;     /* Version of wrapped
                                        product or library */
    CSSM_STRING StandardDescription; /* Desc of standard this
                                        product conforms to */
    CSSM_STRING ProductVendor;       /* Vendor of wrapped product
                                        or library */
    CSSM_NET_PROTOCOL NetworkProtocol;  /* The network protocol
                         supported by remote storage service */
    uint32 ProductFlags;             /* Mask of selectable DB
                      service features actually used by the DL */
} CSSM_DL_WRAPPEDPRODUCT_INFO, *CSSM_DL_WRAPPEDPRODUCT_INFO_PTR;
```

Definition

StandardVersion
If this product conforms to an industry standard, this is the version number of that standard.

StandardDescription
If this product conforms to an industry standard, this is a description of that standard.

ProductVersion
Version number information for the actual product version used in this version of the DL module.

ProductDescription
A string describing the product.

ProductVendor
The name of the product vendor.

NetworkProtocol
The name of the network protocol.

ProductFlags
A bit mask enumerating selectable features of the data base service that the DL module uses in its implementation.

50.2.32 CSSM_NAME_LIST

The CSSM_NAME_LIST structure is used to return the logical names of the data stores that a DL module can access.

```
typedef struct cssm_name_list {
    uint32 NumStrings;
    char** String;
} CSSM_NAME_LIST, *CSSM_NAME_LIST_PTR;
```

Definition

NumStrings
Number of strings in the array pointed to by String.

String
A pointer to an array of strings.

50.2.33 CSSM_DLSUBSERVICE

This structure contains the static information that describes a data storage library sub-service. This information is securely stored in the CSSM registry when the DL module is installed with CSSM. A data storage library module may implement multiple types of services and organize them as sub-services. For example, a DL module supporting two types of remote directory services may organize its implementation into two sub-services, one for an X.509 certificate directory and a second for custom enterprise policy data store. Most data storage library modules will implement exactly one sub-service.

Not all DL modules can maintain a summary of managed data stores. In this case, the DL module reports its number of data stores as CSSM_DB_DATASTORES_UNKNOWN. Data stores can (and probably do) exist, but the DL module cannot provide a list of them.

```
#define CSSM_DB_DATASTORES_UNKNOWN (-1)
```

The descriptive information stored in these structures can be queried using the function *CSSM_GetModuleInfo*( ) and specifying the data storage library module GUID.

```
typedef struct cssm_dlsubservice {
    uint32 SubServiceId;
    CSSM_STRING Description;
    CSSM_DLTYPE Type;
    union cssm_dlsubservice_attributes {
        CSSM_DL_CUSTOM_ATTRIBUTES CustomAttributes;
        CSSM_DL_LDAP_ATTRIBUTES LdapAttributes;
        CSSM_DL_ODBC_ATTRIBUTES OdbcAttributes;
        CSSM_DL_PKCS11_ATTRIBUTES Pkcs11Attributes;
        CSSM_DL_FFS_ATTRIBUTES FfsAttributes;
    } Attributes;
    CSSM_DL_WRAPPEDPRODUCT_INFO WrappedProduct;
    CSSM_USER_AUTHENTICATION_MECHANISM AuthenticationMechanism;

    /* meta information about the query support */
    /* provided by the module */
    uint32 NumberOfRelOperatorTypes;
    CSSM_DB_OPERATOR_PTR RelOperatorTypes;
    uint32 NumberOfConjOperatorTypes;
    CSSM_DB_CONJUNCTIVE_PTR ConjOperatorTypes;
    CSSM_BOOL QueryLimitsSupported;

    /* meta information about the encapsulated */
    /* data stores (if known) */
    sint32 NumberOfDataStores;
    CSSM_NAME_LIST_PTR DataStoreNames;
    CSSM_DBINFO_PTR DataStoreInfo;

    /* additional information */
    void *Reserved;
} CSSM_DLSUBSERVICE, *CSSM_DLSUBSERVICE_PTR;
```

Definition

SubServiceId
A unique, identifying number for the sub-service described in this structure.

Description
A string containing a description name or title for this sub-service.

Type
An identifier for the type of underlying data store the DL module uses to provide persistent storage.

Attributes
A structure containing attributes that define additional parameter values specific to the DL module type.

WrappedProduct
Descriptions of the backend data store services used by this module.

AuthenticationMechanism
Defines the authentication mechanism required when using this DL module. This authentication mechanism is distinct from the authentication mechanism (specified in a DBInfo structure) required to access a specific data store.

NumberOfRelOperatorTypes
The number of distinct relational operator the DL module accepts in selection queries for retrieving records from its managed data stores.

RelOperatorTypes
The list of specific relational operators that can be used to formulate selection predicates for queries on a data store. The list contains NumberOfRelOperatorTypes operators.

NumberOfConjOperatorTypes
The number of distinct conjunctive operator the DL module accepts in selection queries for retrieving records from its managed data stores.

ConjOperatorTypes
A list of specific conjunctive operators that can be used to formulate selection predicates for queries on a data store. The list contains NumberOfConjOperatorTypes operators.

QueryLimitsSupported
A boolean indicating whether query limits are effective when the DL module executes a query.

NumberOfDataStores
The number of data stores managed by the DL module. This information may not be known by the DL module, in which case this value will equal CSSM_DB_DATASTORES_UNKNOWN.

DataStoreNames
A list of names of the data stores managed by the DL module. This information may not be known by the DL module and hence may not be available. The list contains NumberOfDataStores entries.

DataStoreInfo
A list of pointers to information about each data store managed by the DL module. This information may not be known in advance by the DL module and hence may not be available through this structure. The list contains NumberOfDataStores entries.

Reserved
Reserved for future use.

50.3 Data Storage Library Operations

The manpages for Data Storage Library Operations follow on the next page.

NAME

DL_Authenticate

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_Authenticate
    (const CSSM_DL_HANDLE DLHandle,
    const CSSM_DB_HANDLE DBHandle,
    const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
    const CSSM_DB_USER_AUTHENTICATION_PTR UserAuthentication)
```

DESCRIPTION

This function allows the caller to provide authentication credentials to the DL module at a time other than data store creation, deletion, open, import, and export. AccessRequest defines the type of access to be associated with the caller. If the authentication credential applies to access and use of a DL module in general, then the data store handle specified in the DLDBHandle must be NULL. When the authorization credential is to apply to a specific data store, the handle for that data store must be specified in the DLDBHandle pair.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module used to perform this function.

DBHandle (input/optional)
The handle that describes the data store to which access is being requested. If the authentication request is authentication to the DL module in general, then the data store handle must be NULL.

AccessRequest (input)
An indicator of the requested access mode for the data store or DL module in general.

UserAuthentication (input)
The caller's credential as required for obtaining authorized access to the data store or to the DL module in general.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_INVALID_ACCESS_MODE
Unrecognized access type.

CSSM_INVALID_AUTHENTICATION
Unrecognized or invalid authentication credential.

50.4 Data Store Operations

The manpages for Data Store Operations follow on the next page.

NAME

DL_DbOpen

SYNOPSIS

```
CSSM_DB_HANDLE CSSMDLI DbOpen
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS_PTR DbLocation,
    const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
    const CSSM_DB_USER_AUTHENTICATION_PTR UserAuthentication,
    const void *OpenParameters)
```

DESCRIPTION

This function opens the data store with the specified logical name under the specified access mode. If no DbName is provided, the default data store will be opened. If user authentication credentials are required, they must be provided. Also, additional open parameters may be required to open a given data store and are supplied in the OpenParameters.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbName (input)
A pointer to the string containing the logical name of the data store.

DbLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbName (for existing data stores) or can assume a default storage service process location. If the DbName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

AccessRequest (input)
An indicator of the requested access mode for the data store, such as read-only or read-write.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

RETURN VALUE

Returns the CSSM_DB_HANDLE of the opened data store. If the handle is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_DATASTORE_NOT_EXISTS
The data store with the logical name does not exist.

CSSM_DL_INVALID_AUTHENTICATION
Caller is not authorized for specified access mode.

CSSM_DL_DB_OPEN_FAIL
Open caused an exception.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

DL_DbClose

NAME

DL_DbClose

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_DbClose
    (CSSM_DL_DB_HANDLE DLDBHandle)
```

DESCRIPTION

This function closes an open data store.

PARAMETERS

DLDBHandle (input)
A handle structure containing the DL handle for the attached DL module and the DB handle for an open data store managed by the DL. This specifies the open data store to be closed.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_DB_CLOSE_FAIL
Close caused an exception.

SEE ALSO

DL_DbOpen

NAME

DL_DbCreate

SYNOPSIS

```
CSSM_DB_HANDLE CSSMDLI DL_DbCreate
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS_PTR DbLocation,
    const CSSM_DBINFO_PTR DBInfo,
    const CSSM_DB_ACCESS_TYPE_PTR AccessRequest,
    const CSSM_DB_USER_AUTHENTICATION_PTR UserAuthentication,
    const void *OpenParameters)
```

DESCRIPTION

This function creates and opens a new data store. The name of the new data store is specified by the input parameter DbName. The record schema for the data store is specified in the DBINFO structure. The newly created data store is opened under the specified access mode. If user authentication credentials are required, they must be provided. Also, additional open parameters may be required and are supplied in the OpenParameters.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module used to perform this function.

DbName (input)
The logical name for the new data store.

DbLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbName (for existing data stores) or can assume a default storage service process location. If the DbName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

DBInfo (input)
A pointer to a structure describing the format/schema of each record type that will be stored in the new data store. If the schema definition does not specify the CSSM pre-defined attribute name "PrintName" and "Alias", these attributes are added by the DL module with the minimum associated storage size.

AccessRequest (input)
An indicator of the requested access mode for the data store, such as read-only or read-write.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

OpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the data store.

RETURN VALUE

A handle to the newly created, open data store. When NULL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_AUTHENTICATION
Caller is not authorized for the operation.

CSSM_DL_INVALID_DBINFO
Invalid meta information for the schema.

CSSM_DL_DB_CREATE_FAIL
Create caused an exception.

CSSM_REGISTRY_ERROR
Unable to add-update registry entry.

CSSM_DL_INVALID_CSP_HANDLE
Invalid default CSP handle (integrity signing).

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

DL_DbOpen, *DL_DbClose*, *DL_DbDelete*

NAME

DL_DbDelete

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_DbDelete
    (CSSM_DL_HANDLE DLHandle,
    const char *DbName,
    const CSSM_NET_ADDRESS_PTR DbLocation,
    const CSSM_DB_USER_AUTHENTICATION_PTR UserAuthentication)
```

DESCRIPTION

This function deletes all records from the specified data store and removes all state information associated with that data store.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbName (input)
A pointer to the string containing the logical name of the data store.

DbLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbName (for existing data stores) or can assume a default storage service process location. If the DbName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

UserAuthentication (input/optional)
The caller's credential as required for obtaining access (and consequently deletion capability) to the data store. If no credentials are required for the specified data store, then user authentication must be NULL.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_INVALID_AUTHENTICATION
Caller is not authorized for operation.

CSSM_REGISTRY_ERROR
Unable to update registry entry.

CSSM_DL_DB_DELETE_FAIL
Delete caused an exception.

SEE ALSO

DL_DbCreate

NAME

DL_DbImport

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_DbImport
    (CSSM_DL_HANDLE DLHandle,
    const char *DbDestinationName,
    const CSSM_NET_ADDRESS_PTR DbDestinationLocation,
    const char *DbSourceName,
    const CSSM_NET_ADDRESS_PTR DbSourceLocation,
    const CSSM_DBINFO_PTR DBInfo,
    const CSSM_BOOL InfoOnly,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    const void *DestinationOpenParameters,
    const void *SourceOpenParameters)
```

DESCRIPTION

This function makes the contents of the source data store available from the destination data source. This may involve registering the source data store with this DL module or the transfer of records from the source to the destination.

If INFO_ONLY is TRUE, information about an existing data store is registered with the DL module but no records are imported. The DL module will update the CSSM registry with the DbDestinationName and DBInfo to inform applications that this data store is available. This method may be used to make existing data stores available via the CSSM DL interface.

If INFO_ONLY is FALSE, this function creates a new data store, or adds to an existing data store, by importing records from the specified data source. It is assumed that the data source contains records exported from a data store using the function CSSM_DL_DbExport.

The DbDestinationName specifies the name of a new or existing data store. If a new data store is being created, the DBInfo structure provides the meta information (schema) for the new data store. This structure describes the record attributes and the index schema for the new data store. If the data store already exists, then the existing meta information (schema) is used. (Dynamic schema evolution is not supported.)

Typically, user authentication is required to create a new data store or to write to an existing data store. An authentication credential is presented to the DL module in the form required by the module. The required form is documented in the capabilities and feature descriptions for this module. The resulting data store is not opened as a result of this operation.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbDestinationName (input)
The name of the data store which will contain the imported records.

DbDestinationLocation (input/optional) A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbDestinationName or can assume a default storage service process location. If the DbDestinationName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

DbSourceName (input)
The name of the data source from which to obtain the records to be imported.

DbSourceLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbSourceName or can assume a default storage service process location. If the DbSourceName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

DBInfo (input/optional)
A data structure containing a detailed description of the meta information (schema) for the new data store. If a new data store is being created, then the caller must specify the meta information (schema), or the data source must include the meta information required for proper import of the records. If meta information is supplied by the caller and specified in the data source, then the meta information provided by the caller overrides the meta information recorded in the data source. If the data store exists and records are being added, then this pointer must be NULL. The existing meta information will be used and the schema cannot be evolved.

InfoOnly (input)
A boolean value indicating what to import. If TRUE, import only the DBInfo, which describes the a data store. If FALSE, import both the DBInfo and all of the records exported from a data store.

UserAuthentication (input/optional)
The caller's credential as required for authorization to create a data store. If the DL module requires no additional credentials to create a new data store, then user authentication can be NULL.

DestinationOpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the source data store.

SourceOpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the destination data store.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully and the new data store was created. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_PTR
NULL source or destination names.

CSSM_REGISTRY_ERROR
Unable to add/update registry entry.

CSSM_DL_DB_IMPORT_FAIL
DB exception doing import function.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

DL_DbExport

NAME

DL_DbExport

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_DbExport
    (CSSM_DL_HANDLE DLHandle,
    const char *DbDestinationName,
    const CSSM_NET_ADDRESS_PTR DbDestinationLocation,
    const char *DbSourceName,
    const CSSM_NET_ADDRESS_PTR DbSourceLocation,
    const CSSM_BOOL InfoOnly,
    const CSSM_USER_AUTHENTICATION_PTR UserAuthentication,
    const void *DestinationOpenParameters,
    const void *SourceOpenParameters)
```

DESCRIPTION

This function exports a copy of the data store records from the source data store to data container that can be used as the input data source for the CSSM_DL_DbImport function. The DL module may require additional user authentication to determine authorization to snapshot a copy of an existing data store.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbDestinationName (input)
The name of the destination data container which will contain a copy of the source data store's records.

DbDestinationLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbDestinationName or can assume a default storage service process location. If the DbDestinationName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

DbSourceName (input)
The name of the data store from which the records are to be exported.

DbSourceLocation (input/optional)
A pointer to a network address directly or indirectly identifying the location of the storage service process. If the input is NULL, the module can determine a storage service process and its location based on the DbSourceName or can assume a default storage service process location. If the DbSourceName does not distinguish the storage service process and a default cannot be assumed, the service cannot be performed and the operation fails.

InfoOnly (input)
A boolean value indicating what to export. If TRUE, export only the DBInfo, which describes the a data store. If FALSE, export both the DBInfo and all of the records in the specified data store.

UserAuthentication (input/optional)
The caller's credential as required for authorization to snapshot/copy a data store. If the DL module requires no additional credentials to perform this operation, then user authentication can be NULL.

DestinationOpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the source data store.

SourceOpenParameters (input/optional)
A pointer to a module-specific set of parameters required to open the destination data store.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_PTR
NULL source or destination names.

CSSM_DL_DB_EXPORT_FAIL
DB exception doing export function.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

DL_DbImport

NAME

DL_DbSetRecordParsingFunctions

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_DbSetRecordParsingFunctions
    (CSSM_DL_HANDLE DLHandle,
    const char* DbName,
    const CSSM_DB_RECORDTYPE RecordType,
    const CSSM_DB_RECORD_PARSING_FNTABLE_PTR FunctionTable)
```

DESCRIPTION

This function sets the records parsing function table, overriding the default parsing module, for records of the specified type, in the specified data store. Three record parsing functions can be specified in the table. The functions can be implemented to parse multiple record types. In this case, multiple calls to DbSetRecordParsingFunctions must be made, once for each record type that should be parsed using these functions. The DL module uses these functions to parse the opaque data object stored in a data store record. If no parsing function table has been set for a given record type, then the default parsing module is invoked for that record type.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.
The name of the data store with which to associate the parsing functions.

RecordType (input)
One of the record types parsed by the functions specified in the function table.

FunctionTable (input)
The function table referencing the three parsing functions to be used with the data store specified by DbName.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL Handle.

CSSM_DL_INVALID_DB_NAME
Invalid DB Name.

CSSM_DL_MEMORY_ERROR
Error allocating memory.

SEE ALSO

DL_GetRecordParsingFunctions

NAME

DL_DbGetRecordParsingFunctions

SYNOPSIS

```
CSSM_DB_RECORD_PARSING_FNTABLE_PTR CSSMDLI DL_DbGetRecordParsingFunctions
    (CSSM_DL_HANDLE DLHandle,
    const char* DbName,
    const CSSM_DB_RECORDTYPE RecordType)
```

DESCRIPTION

This function gets the records parsing function table, that operates on records of the specified type, in the specified data store. Three record parsing functions can be returned in the table. The functions can be implemented to parse multiple record types. In this case, multiple calls to DbGetRecordParsingFunctions must be made, once for each record type whose parsing functions are required by the caller. The DL module uses these functions to parse the opaque data object stored in a data store record. If no parsing function table has been set for a given record type, then a NULL value is returned.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

DbName (input)
The name of the data store with which the parsing functions are associated.

RecordType (input)
The record type whose parsing functions are requested by the caller.

RETURN VALUE

A function table for the parsing function appropriate to the specified record type. When CSSM_NULL is returned, either no function table has been set for the specified record type or an error has occurred. Use CSSM_GetError to obtain the error code and determine the reason for the NULL result.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL Handle.

CSSM_DL_INVALID_DB_NAME
Invalid DB Name.

CSSM_DL_MEMORY_ERROR
Error allocating memory.

SEE ALSO

DL_SetRecordParsingFunctions

NAME

DL_GetDbNames

SYNOPSIS

```
CSSM_NAME_LIST_PTR CSSMDLI DL_GetDbNames
    (CSSM_DL_HANDLE DLHandle)
```

DESCRIPTION

This function returns a list of the logical data store names that the specified DL module can access and a count of the number of logical names in that list.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

RETURN VALUE

Returns a pointer to a CSSM_NAME_LIST structure which contains a list of data store names. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_MEMORY_ERROR
Error allocating memory.

CSSM_DL_NO_DATA_SOURCES
No known data store names.

CSSM_DL_GET_DB_NAMES_FAIL
Get DB Names failed.

CSSM_DL_INVALID_DL_HANDLE
Invalid DL Handle.

SEE ALSO

DL_GetDbNameFromHandle, *DL_FreeNameList*

NAME

DL_GetDbNameFromHandle

SYNOPSIS

```
char * CSSMDLI DL_GetDbNameFromHandle
    (CSSM_DL_DB_HANDLE DLDBHandle)
```

DESCRIPTION

This function retrieves the data source name corresponding to an opened database handle.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module used to perform this function and the open data store whose name is being requested.

RETURN VALUE

Returns a string which contains a data store name. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_MEMORY_ERROR
Error allocating memory.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB Handle.

CSSM_DL_INVALID_DL_HANDLE
Invalid DL Handle.

SEE ALSO

DL_GetDbNames

NAME

DL_FreeNameList

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_FreeNameList
    (CSSM_DL_HANDLE DLHandle,
    CSSM_NAME_LIST_PTR NameList)
```

DESCRIPTION

This function frees the list of the logical data store names that was returned by DL_GetDbNames.

PARAMETERS

DLHandle (input)
The handle that describes the add-in data storage library module to be used to perform this function.

NameList (input)
A pointer to the CSSM_NAME_LIST.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_MEMORY_ERROR
Error allocating memory.

CSSM_DL_INVALID_PTR
Invalid pointer to the name list.

CSSM_DL_INVALID_DL_HANDLE
Invalid DL Handle.

SEE ALSO

DL_GetDbNames

50.5 Data Record Operations

The manpages for Data Record Operations follow on the next page.

NAME

DL_DataInsert

SYNOPSIS

```
CSSM_DB_UNIQUE_RECORD_PTR CSSMDLI DL_DataInsert
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_DB_RECORDTYPE RecordType,
    const CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    const CSSM_DATA_PTR Data)
```

DESCRIPTION

This function creates a new persistent data record of the specified type by inserting it into the specified data store. The values contained in the new data record are specified by the Attributes and the Data. The attribute value list contains zero or more attribute values. The DL module may require initial values for the CSSM pre-defined attributes. The DL modules can assume default values for any unspecified attribute values or can return an error condition when DLM-required attributes values are not specified by the caller. The Data is an opaque object to be stored in the new data record.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store in which to insert the new data record.

RecordType (input)
Indicates the type of data record being added to the data store

Attributes (input/optional)
A list of structures containing the attribute values to be stored in that attribute and the meta information (schema) describing those attributes. The list contains at most one entry per attribute in the specified record type. The DL module can assume default values for those attributes that are not assigned values by the caller or may return an error. If the specified record type does not contain any attributes, this parameter must be NULL.

Data (input/optional)
A pointer to the CSSM_DATA structure which contains the opaque data object to be stored in the new data record. If the specified record type does not contain an opaque data object, this parameter must be NULL.

RETURN VALUE

A pointer to a CSSM_DB_UNIQUE_RECORD_POINTER containing a unique identifier associated with the new record. This unique identifier structure can be used in future references to this record using this DLDBHandle pairing. It may not be valid for other DLHandles targeted to this DL module or to other DBHandles targeted to this data store. When NULL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid Data Store handle.

CSSM_DL_INVALID_RECORDTYPE
Invalid record type for this data store.

CSSM_DL_INVALID_ATTRIBUTE
Invalid attribute for this record type in this data store.

CSSM_DL_MISSING_VALUE
Missing attribute or data value for this record type.

CSSM_DL_DATA_INSERT_FAIL
Add caused an exception.

SEE ALSO

DL_DataDelete

NAME

DL_DataDelete

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_DataDelete
    (CSSM_DL__DB_HANDLE DLDBHandle,
    const CSSM_DB_UNIQUE_RECORD_PTR UniqueRecordIdentifier)
```

DESCRIPTION

This function removes from the specified data store, the data record specified by the unique record identifier.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store from which to delete the specified data record.

UniqueRecordIdentifier (input)
A pointer to a CSSM_DB_UNIQUE_RECORD identifier containing unique identification of the data record to be deleted from the data store. Once the associated record has been deleted, this unique record identifier cannot be used in future references.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid Data Storage handle.

CSSM_DL_INVALID_RECORD_IDENTIFIER
Invalid data pointer.

CSSM_DL_DATA_DELETE_FAIL
Delete caused an exception.

SEE ALSO

CSSM_DL_DataInsert

NAME

DL_DataModify

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_DataModify
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_DB_RECORDTYPE RecordType,
    const CSSM_DB_UNIQUE_RECORD_PTR UniqueRecordIdentifier,
    const CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR AttributesToBeModified,
    const CSSM_DATA_PTR DataToBeModified)
```

DESCRIPTION

This function modifies the persistent data record identified by the UniqueRecordIdentifier. The modifications are specified by the Attributes and Data parameters. For each attribute in the Attributes list, the attribute is added if does not exist, or replaced if it does exist. If a Data value is specified, the record data value should be replaced. To remove a record or attribute, set the value to NULL.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store in which the data record resides.

RecordType (input)
Indicates the type of data record being modified.

UniqueRecordIdentifier (input)
A pointer to a CSSM_DB_UNIQUE_RECORD identifier containing unique identification of the data record to be modified.

AttributesToBeModified (input/optional)
A list containing the names of the attributes to be modified and their new values. For each attribute in the Attributes list, the attribute is added if does not exist, or replaced if it does exist. If the attribute value is NULL, the attribute is deleted. If the Attributes parameter is NULL, no attribute modification occurs.

DataToBeModified (input/optional)
A pointer to the CSSM_DATA structure which contains the opaque data object to be stored in the data record. If this parameter is NULL, no Data modification occurs.

RETURN VALUE

A CSSM_OK return value signifies that the function completed successfully. When CSSM_FAIL is returned, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid Data Store handle.

CSSM_DL_INVALID_RECORDTYPE
Invalid record type for this data store.

CSSM_DL_INVALID_ATTRIBUTE
Invalid attribute for this record type in this data store.

CSSM_DL_DATA_MODIFY_FAIL
Modify caused an exception.

SEE ALSO

DL_DataInsert, *DL_DataDelete*

NAME

DL_DataGetFirst

SYNOPSIS

```
CSSM_DB_UNIQUE_RECORD_PTR CSSMDLI DL_DataGetFirst
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_QUERY_PTR Query,
    CSSM_HANDLE_PTR  ResultsHandle,
    CSSM_BOOL  *EndOfDataStore,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR  Data)
```

DESCRIPTION

This function retrieves the first data record in the data store that matches the selection criteria. The selection criteria (including selection predicate and comparison values) is specified in the Query structure. The DL module can use internally managed indexing structures to enhance the performance of the retrieval operation. This function returns the first record satisfying the query in the list of Attributes and the opaque Data object. This function also returns a flag indicating whether any records satisfied the query and, if so, a results handle to be used when retrieving subsequent records satisfying the query. If the query selection criteria specifies time or space limits for executing the query, those limits also apply to retrieval of the additional selected data records retrieved using the CSSM_DL_DataGetNext function. Finally, this function returns a unique record identifier associated with the retrieved record. This structure can be used in future references to the retrieved data record.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store to search for records satisfying the query.

Query (input/optional)
The query structure specifying the selection predicate(s) used to query the data store. The structure contains meta information about the search fields and the relational and conjunctive operators forming the selection predicate. The comparison values to be used in the search are specified in the Attributes and Data parameter. The CSSM pre-defined attribute names "PrintName" and "Alias" are valid in any query, regardless of the stored value for those attributes. If no query is specified, the DL module can return the first record in the data store (that is, perform sequential retrieval) or return an error.

ResultsHandle (output)
This handle should be used to retrieve subsequent records that satisfied this query.

EndOfDataStore (output)
A flag indicating whether additional unretrieved records satisfied the query. If TRUE, then additional records satisfying the query can be retrieved using CSSM_DL_DataGetNext. If FALSE, then all records satisfying the query have been retrieved.

Attributes (input/output)
The calling application specifies the names of the attributes to be retrieved. The DL module fills in these attributes' values for the retrieved record. If the Attributes pointer is NULL, the DL module should not return the record's attributes.

Data (output)
The opaque object stored in the retrieved record. If the Data pointer is NULL, the DL module should not return the record's data.

RETURN VALUE

If successful and EndOfDataStore is FALSE, this function returns a pointer to a CSSM_UNIQUE_RECORD structure containing a unique identifier associated with the retrieved record. This unique identifier structure can be used in future references to this record using this DLDBHandle pairing. It may not be valid for other DLHandles targeted to this DL module or to other DBHandles targeted to this data store. If the pointer is NULL and EndOfDataStore is TRUE, then a normal termination condition has occurred. If the pointer is NULL and EndOfDataStore is FALSE, then an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_INVALID_SELECTION_PRED
Invalid selection predicate.

CSSM_DL_NO_DATA_FOUND
No data records match the selection predicate.

CSSM_DL_DATA_GETFIRST_FAIL
An exception occurred when processing the query.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

DL_DataGetNext, *DL_DataAbortQuery*

NAME

DL_DataGetNext

SYNOPSIS

```
CSSM_DB_UNIQUE_RECORD_PTR CSSMDLI DL_DataGetNext
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_HANDLE ResultsHandle,
    CSSM_BOOL *EndOfDataStore,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR Data)
```

DESCRIPTION

This function returns the next data record referenced by the ResultsHandle. The ResultsHandle references a set of records selected by an invocation of the DataGetFirst function. The record values are returned in the Attributes and Data parameters. A flag indicates whether any additional records satisfying the original query remained to be retrieved. The function also returns a unique record identifier for the returned record.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store from which records were selected by the initiating query.

ResultsHandle (output)
The handle identifying a set of records retrieved by a query executed by the DataGetFirst function.

EndOfDataStore (output)
A flag indicating whether additional unretrieved records satisfied the query. If TRUE, then additional records satisfying the query can be retrieved using CSSM_DL_DataGetNext. If FALSE, then all records satisfying the query have been retrieved.

Attributes (input/output)
The calling application specifies the names of the attributes to be retrieved. The DL module fills in these attributes' values for the retrieved record. If the Attributes pointer is NULL, the DL module should not return the record's attributes.

Data (output)
The opaque object stored in the retrieved record. If the Data pointer is NULL, the DL module should not return the record's data.

RETURN VALUE

If successful and EndOfDataStore is FALSE, this function returns a pointer to a CSSM_UNIQUE_RECORD structure containing a unique identifier associated with the retrieved record. This unique identifier structure can be used in future references to this record using this DLDBHandle pairing. It may not be valid for other DLHandles targeted to this DL module or to other DBHandles targeted to this data store. If the pointer is NULL and EndOfDataStore is TRUE, then a normal termination condition has occurred. If the pointer is NULL and EndOfDataStore is FALSE, then an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library handle.

CSSM_DL_INVALID_DB_HANDLE
: Invalid Data Store handle.

CSSM_DL_INVALID_RESULTS_HANDLE
: Invalid query handle.

CSSM_DL_NO_MORE_RECORDS
: No more records for this selection handle.

CSSM_DL_DATA_GETNEXT_FAIL
: Opening the records caused an exception.

CSSM_DL_MEMORY_ERROR
: Error in allocating memory.

SEE ALSO

DL_DataGetFirst, *DL_DataAbortQuery*

NAME

DL_DataAbortQuery

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_DataAbortQuery
    (CSSM_DL_DB_HANDLE DLDBHandle,
    CSSM_HANDLE ResultsHandle)
```

DESCRIPTION

This function terminates the query initiated by DL_DataGetFirst, and allows a DL to release all intermediate state information associated with the query.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store from which records were selected by the initiating query.

ResultsHandle (input)
The selection handle returned from the initial query function.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid data storage library Handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid data store handle.

CSSM_DL_INVALID_RESULTS_HANDLE
Invalid results handle.

CSSM_DL_DATA_ABORT_QUERY_FAIL
Unable to abort query.

SEE ALSO

DL_DataGetFirst, *DL_DataGetNext*

NAME

DL_DataGetFromUniqueRecordId

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_DataGetFromUniqueRecordId
    (CSSM_DL_DB_HANDLE DLDBHandle,
    const CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord,
    CSSM_DB_RECORD_ATTRIBUTE_DATA_PTR Attributes,
    CSSM_DATA_PTR  Data)
```

DESCRIPTION

This function retrieves the data record and attributes associated with this unique record identifier. The DL module can use indexing structure identified in the UniqueRecord to enhance the performance of the retrieval operation.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store to search for the data record.

UniqueRecord (input)
The pointer to a unique record structure returned from a DL_DataInsert, DL_DataGetFirst, or DL_DataGetNext operation.

Attributes (input/output)
The calling application specifies the names of the attributes to be retrieved. The DL module fills in these attributes' values for the retrieved record. If the Attributes pointer is NULL, the DL module should not return the record's attributes.

Data (output)
The opaque object stored in the retrieved record. If the Data pointer is NULL, the DL module should not return the record's data.

RETURN VALUE

CSSM_OK if the function was successful. CSSM_FAIL if an error condition occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_NO_DATA_FOUND
No data records match the unique record id.

CSSM_DL_DATA_GETFROMUNIQUEID_FAIL
An exception occurred when processing the query.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

SEE ALSO

DL_DataInsert, *DL_DataGetFirst*, *DL_DataGetNext*

NAME

DL_FreeUniqueRecord

SYNOPSIS

```
CSSM_RETURN CSSMDLI DL_FreeUniqueRecord
    (CSSM_DB_UNIQUE_RECORD_PTR UniqueRecord)
```

DESCRIPTION

This function frees the memory associated with the data store unique record structure.

PARAMETERS

UniqueRecord (input)
The pointer to the memory that describes the data store unique record structure.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful, and returns an error code if an error has occurred.

ERRORS

CSSM_INVALID_UNIQUE_RECORD_POINTER
Invalid data store unique record pointer.

SEE ALSO

DL_DataInsert, *DL_DataGetFirst*, *DL_DataGetNext*

50.6 Extensibility Functions

The manpages for Extensibility Functions follow on the next page.

NAME

DL_PassThrough

SYNOPSIS

```
void * CSSMDLI DL_PassThrough
    (CSSM_DL_DB_HANDLE DLDBHandle,
    uint32 PassThroughId,
    const void * InputParams)
```

DESCRIPTION

This function allows applications to call data storage library module-specific operations that have been exported. Such operations may include queries or services that are specific to the domain represented by a DL module.

PARAMETERS

DLDBHandle (input)
The handle pair that describes the add-in data storage library module to be used to perform this function and the open data store upon which the function is to be performed.

PassThroughId (input)
An identifier assigned by a DL module to indicate the exported function to be performed.

InputParams (input)
A pointer to a module implementation-specific structure containing parameters to be interpreted in a function-specific manner by the requested DL module.

RETURN VALUE

A pointer to a module implementation-specific structure containing the output from the pass-through function. The output data must be interpreted by the calling application based on externally-available information. If the pointer is NULL, an error has occurred. Use CSSM_GetError to obtain the error code.

ERRORS

CSSM_DL_INVALID_DL_HANDLE
Invalid DL handle.

CSSM_DL_INVALID_DB_HANDLE
Invalid DB handle.

CSSM_DL_INVALID_PASSTHROUGH_ID
Invalid passthrough ID.

CSSM_DL_INVALID_PTR
Invalid pointer.

CSSM_DL_PASS_THROUGH_FAIL
DB exception doing passthrough function.

CSSM_DL_MEMORY_ERROR
Error in allocating memory.

CAE Specification

Part 13:

CSSM Key Recovery Interface

The Open Group

Chapter 51

Introduction

51.1 CDSA Add-In Module Overview

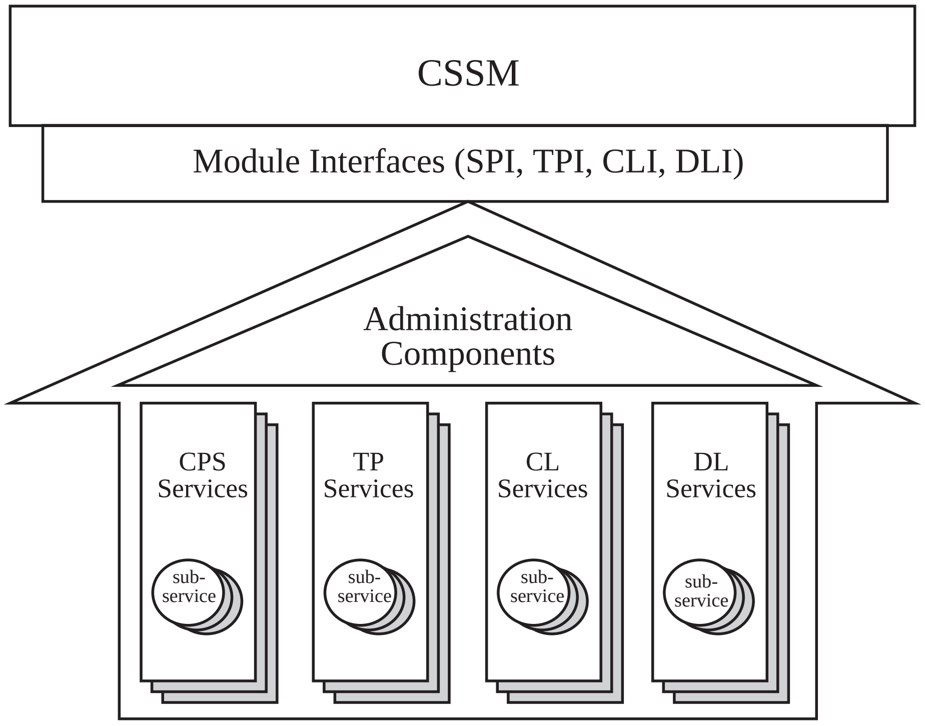

Figure 51-1 CDSA Add-In Module Structure

A CDSA add-in module is a dynamically-linkable library, composed of functions that implement some or all of the CSSM Module Interfaces. Add-in module functionality is partitioned into two areas:

- The provision of security services to applications
- Module administration

Add-in modules provide one or more categories of security services to applications. The service categories are Cryptographic Service Provider (CSP) services, Key Recovery Service Provider (KRSP) services, Trust Policy (TP) services, Certificate Library (CL) services, and Data Storage Library (DL) services. Each security service contains one or more implementation instances, called sub-services. For a CSP service providing access to hardware tokens, a sub-service would represent a slot. For a DL service provider, a sub-service would represent a type of persistent storage. These sub-services each support the module interface for their respective service categories. This documentation-part describes the module interface functions in the KRSP service category. More information about CSP services can be found in the *CSSM Cryptographic Service Provider Interface Specification*. More information about DL services can be found in the *CSSM Data Storage Library Interface Specification*. More information about TP services can be found in the *CSSM Trust Policy Interface Specification*. More information about CL services can be found in the *CSSM Certificate Library Interface Specification*.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions which allow CSSM to indicate events such as module attach and detach. In addition, as part of the attach operation, every module must be able to verify its own integrity, verify the integrity of CSSM, and register with CSSM. Detailed information about add-in module structure, administration, and interfaces can be found in the *CSSM Add-in Module Structure and Administration Specification*.

51.2 Key Recovery Overview

Key recovery mechanisms serve many useful purposes. They may be used by individuals to recover lost or corrupted keys; they may be used by enterprises to deter corporate insiders from using encryption to bypass the corporate security policy regarding the flow of proprietary information. Corporations may also use key recovery mechanisms to recover employee keys in certain situations; for example, in the employee's absence. The use of key recovery mechanisms in web based transactional scenarios can serve as an additional technique of non-repudiation and audit, that may be admissible in a court of law. Finally, key recovery mechanisms may be used by jurisdictional law enforcement bodies to access the contents of confidentiality protected communications and stored data. Thus, there appears to be multiple incentives for the incorporation as well as adoption of key-recovery mechanisms in local and distributed encryption based systems.

51.2.1 Key Recovery Nomenclature

Denning and Brandstad [Key Escrow], present a taxonomy of key escrow systems. Here, a different scheme of nomenclature was adopted in order to exhibit some of the finer nuances of key recovery schemes. The term *key recovery* encompasses mechanisms that allow authorized parties to retrieve the cryptographic keys used for data confidentiality, with the ultimate goal of recovery of encrypted data. The remainder of this section will discuss the various types of key recovery mechanisms, the phases of key recovery, and the policies with respect to key recovery.

51.2.2 Key Recovery Types

There are two classes of key recovery mechanisms based on the way keys are held to enable key recovery:

- Key escrow—techniques based on the paradigm that the government or a trusted party called an escrow agent, holds the actual user keys or portions thereof.
- Key encapsulation—techniques based on the paradigm that a cryptographically encapsulated form of the key is made available to parties that require key recovery. The technique ensures that only certain trusted third parties called recovery agents can perform the unwrap operation to retrieve the key material buried inside.

There can also be hybrid schemes that use escrow mechanisms in addition to encapsulation mechanisms.

An orthogonal way to classify key recovery mechanisms is based on the nature of the key:

- Long-term, private keys
- Ephemeral keys

Both types can be escrowed or encapsulated. Since escrow schemes involve the actual archival of keys, they typically deal with long-term keys, in order to avoid the proliferation problem that arises when trying to archive the myriad ephemeral keys. Key encapsulation techniques, on the other hand, usually operate on the ephemeral keys.

For a large class of key recovery (escrow as well as encapsulation) schemes, there are a set of *key recovery fields* that accompany an enciphered message or file. These key recovery fields may be used by the appropriate authorized parties to recover the decryption key and or the plaintext. Typically, the key recovery fields comprise information regarding the key escrow or recovery agent(s) that can perform the recovery operation; they also contain other pieces of information to enable recovery.

In a key escrow scheme for long-term private keys, the "escrowed" keys are used to recover the ephemeral data confidentiality keys. In such a scheme, the key recovery fields may comprise the identity of the escrow agent(s), identifying information for the escrowed key, and the bulk encryption key wrapped in the recipient’s public key (which is part of an escrowed key pair); thus the key recovery fields include the key exchange block in this case. In a key escrow scheme where bulk encryption keys are archived, the key recovery fields may comprise information to identify the escrow agent(s), and the escrowed key for that enciphered message.

In a typical key encapsulation scheme for ephemeral bulk encryption keys, the key recovery fields are distinct from the key exchange block, (if any.) The key recovery fields identify the recovery agent(s), and contain the bulk encryption key encapsulated using the public keys of the recovery agent(s).

The key recovery fields are generated by the party performing the data encryption, and associated with the enciphered data. To ensure the integrity of the key recovery fields, and its association with the encrypted data, it may be required for processing by the party performing the data decryption. The processing mechanism ensures that successful data decryption cannot occur unless the integrity of the key recovery fields are maintained at the receiving end. In schemes where the key recovery fields contain the key exchange block, decryption cannot occur at the receiving end unless the key recovery fields are processed to obtain the decryption key; thus the integrity of the key recovery fields are automatically verified. In schemes where the key recovery fields are separate from the key exchange block, additional processing must be done to ensure that decryption of the ciphertext occurs only after the integrity of the key recovery fields are verified.

51.2.3 Lifetime of Key Recovery Fields

Cryptographic products fall into one of two fundamental classes: *archived-ciphertext products*, and *transient-ciphertext products*. When the product allows either the generator or the receiver of ciphertext to archive the ciphertext, the product is classified as an archived-ciphertext product. On the other hand, when the product does not allow the generator or receiver of ciphertext to archive the ciphertext, it is classified as a transient-ciphertext product.

In both cases, since it is a not meaningful to archive key recovery fields without archiving the associated ciphertext, the lifetimes of key recovery fields should never be greater than the lifetime of the associated ciphertext.

It is important to note that the lifetime of key recovery fields should never be greater than the lifetime of the associated ciphertext. This is somewhat obvious, since recovery of the key is only meaningful if the key can be used to recover the plaintext from the ciphertext. Hence, when archived-ciphertext products are key recovery enabled, the key recovery fields are typically archived as long as the ciphertext. Similarly, when transient-ciphertext products are key recovery enabled, the key recovery fields are associated with the ciphertext for the duration of its lifetime. It is not meaningful to archive key recovery fields without archiving the associated ciphertext.

51.2.4 Key Recovery Policy

Key recovery policies are mandatory policies that may be derived from enterprise-based or jurisdiction-based rules on the use of cryptographic products for data confidentiality. Political jurisdictions may choose to define key recovery policies for cryptographic products based on export, import, or use controls. Enterprises may define internal and external domains, and may mandate key recovery policies on the cryptographic products within their own domain.

Key recovery policies come in two flavors: *key recovery enablement policies* and *key recovery inter-operability policies*. Key recovery enablement policies specify the exact cryptographic protocol suites (algorithms, modes, key lengths and so on) and perhaps usage scenarios, where key recovery enablement is mandated. Furthermore, these policies may also define the number of bits of the cryptographic key that may be left out of the key recovery enablement operation; this is typically referred to as the *workfactor*. Key recovery inter-operability policies specify to what degree a key-recovery-enabled cryptographic product is allowed to interoperate with other cryptographic products.

51.2.5 Operational Scenarios for Key Recovery

There are three basic operational scenarios for key recovery:

- Enterprise key recovery
- Law enforcement key recovery
- Individual key recovery

Enterprise key recovery allows enterprises to enforce stricter monitoring of the use of cryptography, and the recovery of enciphered data when the need arises. The user in this scenario is the enterprise employee. Enterprise key recovery is based on a mandatory key recovery policy; however, this policy is set (typically through administrative means) by the organization or enterprise at the time of installation of a recovery-enabled cryptographic product. The enterprise key recovery policy should not be modifiable or by-passable by the individual using the cryptographic product. Enterprise key recovery mechanisms may use special, enterprise-authorized escrow or recovery agents.

In the law enforcement scenario, key recovery is mandated by the jurisdictional law enforcement authorities in the interest of national security and law enforcement. The user in this scenario is the private citizen in the jurisdiction where the product is being used. For a specific cryptographic product, the key recovery policies for multiple jurisdictions may apply simultaneously. The policies (if any) of the jurisdiction(s) of manufacture of the product, as well as the jurisdiction of installation and use, need to be applied to the product such that the most restrictive combination of the multiple policies is used. Thus, law enforcement key recovery is based on mandatory key recovery policies; these policies are logically bound to the cryptographic product at the time the product is shipped. There may be some mechanism for vendor-controlled updates of such law enforcement key recovery policies in existing products; however, organizations and end users of the product are not able to modify this policy at their discretion. The escrow or recovery agents used for this scenario of key recovery need to be strictly controlled in most cases, to ensure that these agents meet the eligibility criteria for the relevant political jurisdiction where the product is being used.

Individual key recovery is user-discretionary in nature, and is performed for the purpose of recovery of enciphered data by the owner of the data, if the cryptographic keys are lost or corrupted. The user in this scenario is the traditional "end-user" of the software product. Since this is a non-mandatory key recovery scenario, it is not based on any policy that is enforced by the cryptographic product; rather, the product may allow the user to specify when individual key recovery enablement is to be performed. There are few restrictions on the use of specific escrow or recovery agents.

Key recovery enabled cryptographic products must be designed so that the key recovery enablement operation is mandatory and noncircumventable in the law enforcement and enterprise scenarios, and discretionary for the individual scenario. The escrow and recovery agent(s) that are used for law enforcement and enterprise scenarios must be tightly controlled so that the agents are validated to belong to a set of authorized or approved agents. In the law enforcement and enterprise scenarios, the key recovery process typically needs to be performed without the knowledge and cooperation of the parties involved in the cryptographic association.

The components of the key recovery fields also varies somewhat between the three scenarios. In the law enforcement scenario, the key recovery fields must contain identification information for the escrow or recovery agent(s); whereas for the enterprise and individual scenarios, the agent identification information is not so critical, since this information may be available from the context of the recovery enablement operation. For the individual scenario, there needs to be a strong user authentication component in the key recovery fields, to allow the owner of the key recovery fields to authenticate themselves to the agents; however, for the enterprise and law enforcement scenarios, the authorization credentials checked by the agents may be in the form of legal documents, or enterprise-authorization documents for key recovery, that may not be tied to any authentication component in the key recovery fields. For the law enforcement and enterprise scenarios, the key recovery fields may contain recovery information for both the generator and receiver of the enciphered data; in the individual scenario, only the information of the generator of the enciphered data is typically included (at the discretion of the generating party.)

51.3 Key Recovery in the Common Data Security Architecture

The Common Data Security Architecture (CDSA) defines an open infrastructure for security services. Within the four layer architecture, the Common Security Services Manager (CSSM) is the central layer that manages the range of security service options available to applications. CSSM allows applications to dynamically select:

- Categories of security services
- Mechanisms that perform desired security services
- Implementations of selected security mechanisms

CSSM acts as a broker between applications requesting security services and dynamically loadable security service modules. The CSSM application programming interface (CSSM-API) defines the interface for accessing security services. The CSSM service provider interface (CSSM-SPI) defines the interface for service providers who develop plug-able security service products.

CSSM is extensible in that it also provides dynamic loading of module managers that provide elective categories of security services. Key recovery is an important security service for applications and institutions that choose to use it. CSSM accommodates key recovery as an elective category of security service.

A complete architectural description of CDSA and CSSM is contained in the *Common Data Security Architecture (CDSA) Specification*.

Chapter 52

Key Recovery Service Provider Interface

52.1 Overview

A CDSA add-in module is a dynamically-linkable library, composed of functions that implement some or all of the CSSM Module Interfaces. Add-in module functionality is partitioned into two areas:

- The provision of security services to applications
- Module administration

Add-in modules provide one or more categories of security services to applications. The service categories are Cryptographic Service Provider (CSP) services, Trust Policy (TP) services, Certificate Library (CL) services, and Data Storage Library (DL) services. Each security service contains one or more implementation instances, called sub-services. For a CSP service providing access to hardware tokens, a sub-service would represent a slot. For a CL service provider, a sub-service would represent a specific certificate format. These sub-services each support the module interface for their respective service categories. This documentation-part describes the module interface functions in the KR service category.

Each module, regardless of the security services it offers, has the same set of administrative responsibilities. Every module must expose functions which allow CSSM to indicate events such as module attach and detach. In addition, as part of the attach operation, every module must be able to verify its own integrity, verify the integrity of CSSM, and register with CSSM. Detailed information about add-in module structure, administration, and interfaces can be found in the *CSSM Add-in Module Structure and Administration Specification*.

52.1.1 Key Recovery Phases

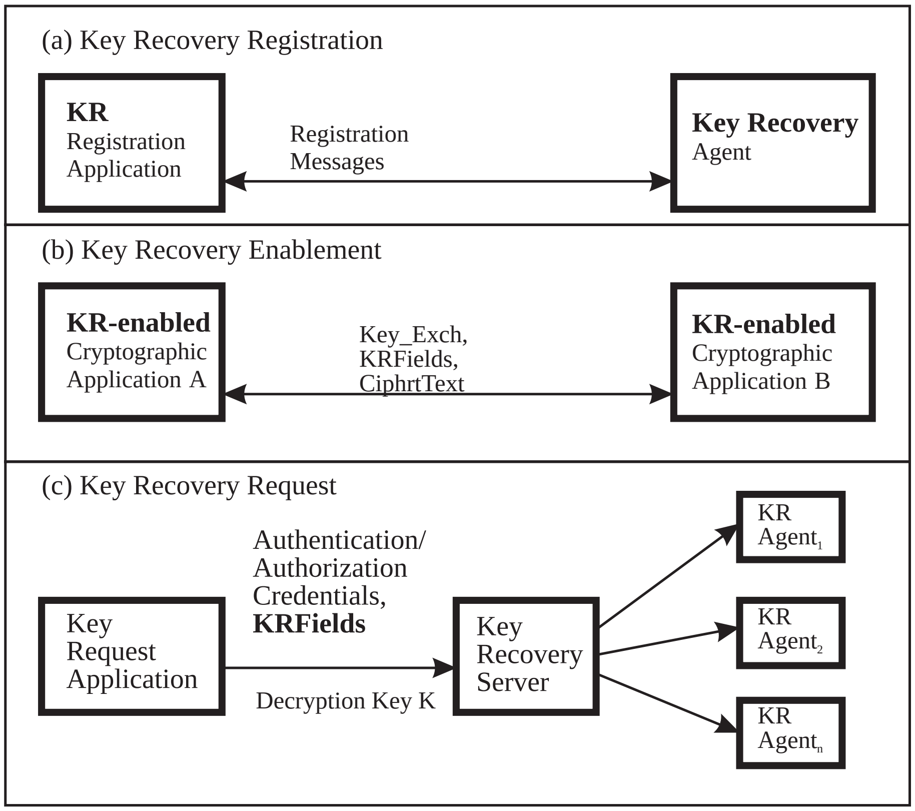
(a) Key Recovery Registration
KR Registration Application
Registration Messages
Key Recovery Agent
(b) Key Recovery Enablement
KR-enabled Cryptographic Application A
Key_Exch, KRFields, CiphrtText
KR-enabled Cryptographic Application B
(c) Key Recovery Request
Authentication/ Authorization Credentials, **KRFields**
Key Request Application
Decryption Key K
Key Recovery Server
KR Agent$_1$
KR Agent$_2$
KR Agent$_n$

Figure 52-1 Key Recovery Phases

The process of cryptographic key recovery involves three major phases. First, there is an optional *key recovery registration* phase where the parties that desire key recovery perform some initialization operations with the escrow or recovery agents; these operations include obtaining a user public key certificate (for an escrowed key pair) from an escrow agent, or obtaining a public key certificate from a recovery agent . Next, parties that are involved in cryptographic associations have to perform operations to enable key recovery (such as the generation of key recovery fields, and so on)—this is typically called the *key recovery enablement* phase. Finally, authorized parties that desire to recover the data keys, do so with the help of a recovery server and one or more escrow agents or recovery agents—this is the *key recovery request* phase.

Figure 52-1 illustrates the three phases of key recovery. In Figure 52-1(a), a key recovery client registers with a recovery agent prior to engaging in cryptographic communication. In Figure 52-1(b), two key-recovery-enabled cryptographic applications are communicating using a key encapsulation mechanism; the key recovery fields are passed along with the ciphertext and key exchange block, to enable subsequent key recovery. The key recovery request phase is illustrated in Figure 52-1(c), where the key recovery fields are provided as input to the key recovery server along with the authorization credentials of the client requesting service. The key recovery server interacts with one or more local or remote key recovery agents to reconstruct the secret key that can be used to decrypt the ciphertext.

It is envisioned that governments or organizations will operate their own recovery server hosts independently, and that key recovery servers may support a single or multiple key recovery mechanisms. There are a number of important issues specific to the implementation and operation of the key recovery servers, such as vulnerability and liability. The focus of this documentation-part is a framework based approach to implementing the key recovery operations pertinent to end parties that use encryption for data confidentiality. The issues with respect to the key recovery server and agents will not be discussed further here.

52.1.2 Key Recovery Registration Operations

CSSM_RETURN CSSMKRSPI KRSP_RegistrationRequest( )
: Performs a recovery registration request operation. A callback may be supplied to allow the registration operation to query for additional input information, if necessary. The result of the registration request operation is a reference handle that may be used to invoke the KRSP_RegistrationRetrieve function.

CSSM_RETURN CSSMKRSPI KRSP_RegistrationRetrieve( )
: Completes a recovery registration operation. The result of the registration operation is returned in the form of a key recovery profile.

52.1.3 Key Recovery Enablement Operations

CSSM_RETURN CSSMKRSPI KRSP_GenerateRecoveryFields( )
: Accepts as input the key recovery context handle, the session based recovery parameters and the cryptographic context handle, and several other parameters of relevance to the KRSP, and outputs a buffer of the appropriate mechanism-specific key recovery fields in a format defined and interpreted by the specific KRSP involved. It returns a cryptographic context handle, which now be used for the encryption APIs in the cryptographic framework.

CSSM_RETURN CSSMKRSPI KRSP_ProcessRecoveryFields( )
: Accepts as input the key recovery context handle, the cryptographic context handle, several other parameters of relevance to a KRSP, and the unparsed buffer of key recovery fields. It returns with a cryptographic context handle which can then be used for the decryption APIs in the cryptographic framework.

52.1.4 Key Recovery Request Operations

CSSM_RETURN CSSMKRSPI KRSP_RecoveryRequest( )
: Performs a recovery request operation for one or more recoverable keys. A callback may be supplied to allow the recovery request operation to query for additional input information, if necessary. The result of the recovery request operation is a results handle that may be used to obtain each recovered key and its associated meta information using the KRSP_GetRecoveredObject function.

CSSM_RETURN CSSMKRSPI KRSP_RecoveryRetrieve( )
: Completes a recovery request operation for one or more recoverable keys. The result of the recovery operation is a results handle that may be used to obtain each recovered key and its meta information using the KRSP_GetRecoveredObject function.

CSSM_RETURN CSSMKRSPI KRSP_GetRecoveredObject( )
: Retrieves a single recovered key and its associated meta information.

CSSM_RETURN CSSMKRSPI KRSP_RecoveryRequestAbort( )
: Terminates a recovery request operation and releases any state information associated with it

52.1.5 Privileged Context Functions

CSSM_RETURN CSSMKRSPI KRSP_PassPrivFunc( )
: Returns a private CSSM callback function that the service provider can use to exempt itself from recursive screening by its own key recovery policy.

52.1.6 Extensibility Functions

CSSM_RETURN CSSMKRSPI KRSP_PassThrough( )
: Accepts as input an operation ID and an arbitrary set of input parameters. The operation ID may specify any type of operation the KR wishes to export. Such operations may include queries or services specific to the key recovery mechanism implemented by the KR module.

52.2 Data Structures

52.2.1 CSSM_KR_HANDLE

This data structure represents the key recovery module handle. The handle value is a unique pairing between a key recovery module and an application that has attached that module. KR handles can be returned to an application as a result of the CSSM_ModuleAttach function.

```
typedef uint32 CSSM_KRSP_HANDLE
                      /* Key Recovery Service Provider Handle */
```

52.2.2 CSSM_KR_NAME

This data structure contains a typed name. The namespace type specifies what kind of name is contained in the third parameter.

```
typedef struct cssm_kr_name {
    uint8 type; /* namespace type */
    uint8 length; /* name string length */
    char *name; /* name string */
} CSSM_KR_NAME
```

Definition

type
: The type of the key recovery name space.

length
: The length of the name (in bytes).

name
: The name represented in a string.

52.2.3 CSSM_KR_PROFILE

This data structure encapsulates the key recovery profile for a given user and a given key recovery mechanism.

```
typedef struct cssm_kr_profile {
    CSSM_KR_NAME UserName; /* name of the user */
    CSSM_DATA_PTR UserCertificate /* public key certificate of
                                    the user */
    uint8 LE_KRANum; /* number of KRA cert chains in the
                                    following list */
    CSSM_CERT_LIST_PTR LE_KRACertChainList; /* list of Law
                        enforcement KRA certificate chains*/
    uint8 ENT_KRANum; /* number of KRA cert chains in the
                                    following list */
    CSSM_CERT_LIST_PTR ENT_KRACertChainList; /* list of
                         Enterprise KRA certificate chains*/
    CSSM_DATA_PTR ENTAuthenticationInfo; /* authentication
                   information for enterprise key recovery */
    uint8 INDIV_KRANum; /* number of KRA cert chains in the
                                    following list */
    CSSM_CERT_LIST_PTR INDIV_KRACertChainList; /* list of
                         Individual KRA certificate chains*/
    CSSM_DATA_PTR INDIVAuthenticationInfo; /* authentication
                   information for individual key recovery */
    uint32 KRFlags; /* flag values to be interpreted by KRSP */
    CSSM_DATA_PTR Extensions; /* reserved for extensions
                                    specific to KRSPs */
} CSSM_KR_PROFILE, *CSSM_KR_PROFILE_PTR;
```

Definition

UserName
The user's name.

UserCertificate
The user's certificate chain, used for identity and authentication when performing policy evaluation.

LE_KRANum
The number of LE Key Recovery agents in the following list.

LE_KRACertChainList
A list of certificates chains one per Key Recovery Agent authorized for LE key recovery.

ENT_KRANum
The number of ENT Key Recovery agents in the following list.

ENT_KRACertChainList
A list of certificates chains one per Key Recovery Agent authorized for ENT key recovery.

ENTAuthenticationInfo
Authentication information to be used for ENT key recovery.

INDIV_KRANum
The number of INDIV Key Recovery agents in the following list.

INDIV_KRACertChainList
A list of certificates chains one per Key Recovery Agent authorized for INDIV key recovery.

INDIVAuthenticationInfo
Authentication information to be used for INDIV key recovery.

KRFlags
A bit mask specifying the user's selected service options specific to the selected key recovery service module.

Extensions
Reserved for future use.

52.2.4 CSSM_CERT_LIST

This data structure encapsulates a generic list of items.

```
typedef struct cssm_cert_list {
    uint32 NumberCerts;
    CSSM_DATA_PTR CertList;
} CSSM_CERT_LIST, *CSSM_CERT_LIST_PTR;
```

Definition

NumberCerts
Count of the number of certs in the list.

CertList
Pointer to a list of certificate items.

52.2.5 CSSM_CONTEXT_ATTRIBUTE Extensions

The key recovery context creation operations return key recovery context handles that are represented as cryptographic context handles. In order to use to the CSSM_CONTEXT data structure to implement key recovery contexts, the CSSM_CONTEXT will be used to hold new types of attributes, as shown below:

```
typedef struct cssm_context_attribute {
    uint32 AttributeType;
    uint32 AttributeLength;
    union cssm_context_attribute_value {
      char *String;
      uint32 Uint32;
      CSSM_CRYPTO_DATA_PTR Crypto;
      CSSM_KEY_PTR Key;
      CSSM_DATA_PTR Data;
      CSSM_DATE_PTR Date;
      CSSM_RANGE_PTR Range;
      CSSM_KR_PROFILE_PTR KRProfile;
    } Attribute;
} CSSM_CONTEXT_ATTRIBUTE, *CSSM_CONTEXT_ATTRIBUTE_PTR;
```

All but the last member of the union above are part of the core CSSM API Specification. The descriptions of these basic fields and members are in the *CSSM Application Programming Interface*. The KRProfile member of the union has been added specifically to support key recovery contexts, and is described below.

Definition

KRProfile
A pointer to the key recovery profile structure that defines the user parameters with respect to the key recovery process.

52.2.6 CSSM_ATTRIBUTE_TYPE Additions

Several new attribute types were defined to support the key recovery context attributes. The following definitions are added to the enumerated type CSSM_ATTRIBUTE_TYPE:

```
CSSM_ATTRIBUTE_KRPROFILE_LOCAL   = CSSM_ATTRIBUTE_LAST+1,
            /* local entity profile */
CSSM_ATTRIBUTE_KRPROFILE_REMOTE  = CSSM_ATTRIBUTE_LAST+2,
            /* remote entity profile */
```

52.2.7 CSSM_KRSUBSERVICE

Two structures are used to contain all of the static information that describes a key recovery add-in module: the krinfo structure and the krsubservice structure. This descriptive information is securely stored in the CSSM registry when the KR module is installed with CSSM. A key recovery module may implement multiple types of services and organize them as sub-services. For example, a KR module supporting two an encapsulation mechanism and an escrow mechanism may organize its implementation as two subservices.

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo( ) and specifying the key recovery module GUID.

```
typedef struct cssm_krsubservice {
    uint32 SubServiceId;
    char *Description;      /* Description of this sub service */
    CSSM_CALLER_AUTHENTICATION_MECHANISM AuthenticationMechanism;
} CSSM_KRSUBSERVICE, *CSSM_KRSUBSERVICE_PTR;
```

Definition

SubServiceId
A unique, identifying number for the sub-service described in this structure.

Description
A string containing a descriptive name or title for this sub-service.

AuthenticationMechanism
An enumerated value defining the credential format accepted by the KR module. When an authentication credential is required by a KR function, the presented credentials must be of the required format.

52.2.8 CSSM_KRINFO

Two structures are used to contain all of the static information that describes a key recovery add-in module: the krinfo structure and the krsubservice structure. This descriptive information is securely stored in the CSSM registry when the KR module is installed with CSSM. A key recovery module may implement multiple types of services and organize them as sub-services. For example, a KR module supporting two an encapsulation mechanism and an escrow mechanism may organize its implementation as two subservices.

The descriptive information stored in these structures can be queried using the function CSSM_GetModuleInfo( ) and specifying the key recovery module GUID.

```
typedef struct cssm_krinfo {
    CSSM_VERSION Version;  /* major and minor version number */
    char *Description;     /* Detailed description of this KR */
    char *Vendor;          /* KRSP Vendor name */
    char *Jurisdiction;    /* Home jurisdiction of the
                                            KRSP installation */
    uint32 NumberSubService;
    CSSM_KRSUBSERVICE_PTR SubService;
} CSSM_KRINFO, *CSSM_KRINFO_PTR;
```

Definition

Version
The major and minor version number of the add-in module.

Description
A character string containing a general description of this key recovery module.

Vendor
A character string containing the name of the vendor who implemented and manufactured this key recovery module.

Jurisdiction
A character string describing the geographical region where the key recovery module is installed.

NumberOfSubServices
The number of sub-services implemented by this key recovery module. Every KR module implements at least one sub-service.

Subservices
A pointer to an array of sub-service structures. Each structure contains detailed information about that sub-service.

52.2.9 CSSM_PRIV_FUNC_PTR

The callback function provided by the CSSM to allow a cryptographic context to be made privileged with respect to key recovery policy override operations.

```
typedef CSSM_RETURN (*CSSM_PRIV_FUNC_PTR)
                              (CSSM_CC_HANDLE hContext,
                              CSSM_BOOL Priv);
```

Definition

hContext
The context whose privilege state is to be modified.

Priv
Flag value denoting whether the privilege should be acquired (Priv = TRUE) or dropped (Priv = FALSE)

52.3 Key Recovery Registration Operations

The manpages for Key Recovery Registration Operations follow on the next page.

NAME

KRSP_RegistrationRequest

SYNOPSIS

```
CSSM_RETURN CSSMKRSPI KRSP_RegistrationRequest
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KRRegistrationContextHandle,
    const CSSM_CONTEXT_PTR KRRegistrationContext,
    CSSM_DATA_PTR KRInData,
    CSSM_CRYPTO_DATA_PTR UserCallback,
    uint8 KRFlags,
    uint32 *EstimatedTime
    CSSM_HANDLE_PTR ReferenceHandle )
```

DESCRIPTION

This function performs a key recovery registration operation. The KRInData contains known input parameters for the recovery registration operation. A UserCallback function may be supplied to allow the registration operation to interact with the user interface, if necessary. When this operation is successful, a ReferenceHandle and an EstimatedTime parameter are returned; the ReferenceHandle is to be used to invoke the KRSP_RegistrationRetrieve function, after the EstimatedTime in seconds.

PARAMETERS

KRSPHandle (input)
The handle that describes the add-in key recovery service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

KRRegistrationContextHandle (input)
The handle that describes the context of this key recovery operation used to link to the KRSP-managed information.

KRRegistrationContext (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this key recovery context.

KRInData (input)
Input data for key recovery registration.

UserCallback (input)
A callback function that may be used to collect further information from the user interface.

KRFlags (input)
Flag values for recovery registration. Defined values are:

- KR_INDIV—signifies that the registration is for the IND scenario.
- KR_ENT—signifies that the registration is for the ENT scenario.
- KR_LE—signifies that the registration is for the LE scenario.

EstimatedTime (output)
The estimated time after which the CSSM_KR_RegistrationRetrieve call should be invoked to obtain registration results.

ReferenceHandle (output)
The handle to use to invoke the CSSM_KR_RegistrationRetrieve function.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_KR_INVALID_HANDLE
Invalid registration handle.

CSSM_KR_INVALID_POINTER
Invalid pointer.

CSSM_MEMORY_ERROR
Memory error.

NAME

KRSP_RegistrationRetrieve

SYNOPSIS

```
CSSM_RETURN CSSMKRSPI KRSP_RegistrationRetrieve
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE_PTR ReferenceHandle,
    uint32 *EstimatedTime,
    CSSM_KR_PROFILE_PTR KRProfile)
```

DESCRIPTION

This function completes a key recovery registration operation. The results of a successful registration operation are returned through the KRProfile parameter, which may be used with the profile management API functions.

If the results are not available when this function is invoked, the KRProfile parameter is set to NULL, and the EstimatedTime parameter indicates when this operation should be repeated with the same ReferenceHandle.

PARAMETERS

KRSPHandle (input)
The handle that describes the add-in key recovery service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

ReferenceHandle (input)
The handle to the key recovery registration request that is to be completed.

EstimatedTime (output)
The estimated time after which this call should be repeated to obtain registration results. This is set to a non-zero value only when the KRProfile parameter is NULL.

KRProfile (input/output)
Key recovery profile that is filled in by the registration operation.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_KR_INVALID_HANDLE
Invalid reference handle.

CSSM_MEMORY_ERROR
Memory error.

52.4 Key Recovery Enablement Operations

The manpages for Key Recovery Enablement Operations follow on the next page.

NAME

KRSP_GenerateRecoveryFields

SYNOPSIS

```
CSSM_RETURN CSSMKRSPI KRSP_GenerateRecoveryFields
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KREnablementContextHandle,
    const CSSM_CONTEXT_PTR KREnablementContext,
    CSSM_CC_HANDLE CryptoContextHandle,
    const CSSM_CONTEXT_PTR CryptoContext,
    CSSM_DATA_PTR KRSPOptions,
    uint32 KRFlags,
    CSSM_DATA_PTR KRFields)
```

DESCRIPTION

This function generates the key recovery fields for a cryptographic association given the key recovery context, and the cryptographic context containing the key that is to be made recoverable. The session attribute and the flags are interpreted by the KRSP. A set of key recovery fields (KRFields) is returned if the function is successful. The KRFlags parameter may be used to fine tune the contents of the KRFields produced by this operation.

PARAMETERS

KRSPHandle (input)
The handle that describes the add-in key recovery service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

KREnablementContextHandle (input)
The handle that describes the context of this key recovery operation used to link to the KRSP-managed information.

KREnablementContext (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this key recovery context.

CryptoContextHandle (input)
The handle that describes the cryptographic context used to link to the CSP-managed information.

CryptoContext (input)
Pointer to CSSM_CONTEXT structure that describes the attributes of the cryptographic context.

KRSPOptions (input)
The key recovery service provider specific options. These options are uninterpreted by the SKMF, but passed on to the KRSP.

KRFlags (input)
Flag values for key recovery fields generation. Defined values are:

- KR_INDIV—signifies that the individual key recovery fields should be generated.
- KR_ENT—signifies that the enterprise key recovery fields should be generated.
- KR_LE_MAN—signifies that the law enforcement key recovery fields pertaining to the manufacturing jurisdiction should be generated.
- KR_LE_USE—signifies that the law enforcement key recovery fields pertaining to the jurisdiction of use should be generated.

- KR_OPTIMIZE—signifies that performance optimization options are to be adopted by a KRSP while implementing this operation.
- KR_DROP_WORKFACTOR—signifies that the key recovery fields should be generated without using the key size work factor.

KRFields (output)
The key recovery fields in the form of a data blob.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error if an error has occurred.

ERRORS

CSSM_KR_INVALID_CC_HANDLE
Invalid crypto context handle.

CSSM_KR_INVALID_KRC_HANDLE
Invalid key recovery context handle.

CSSM_KR_INVALID_OPTIONS
Invalid recovery options.

CSSM_MEMORY_ERROR
Memory error.

NAME

KRSP_ProcessRecoveryFields

SYNOPSIS

```
CSSM_RETURN CSSMKRSPI KRSP_ProcessRecoveryFields
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KREnablementContextHandle,
    const CSSM_CONTEXT_PTR KREnablementContext,
    CSSM_CC_HANDLE CryptoContextHandle,
    const CSSM_CONTEXT_PTR CryptoContext,
    CSSM_DATA_PTR KRSPOptions,
    uint32 KRFlags,
    CSSM_DATA_PTR KRFields)
```

DESCRIPTION

This call processed a set of key recovery fields given the key recovery context, and the cryptographic context for the encryption operation, and returns a non-NULL cryptographic context handle if the processing was successful. The returned handle may be used for the decrypt API calls of the CSSM.

PARAMETERS

KRSPHandle (input)
The handle that describes the add-in key recovery service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

KREnablementContextHandle (input)
The handle that describes the context of this key recovery operation used to link to the KRSP-managed information.

KREnablementContext (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this key recovery context.

CryptoContextHandle (input)
The handle that describes the cryptographic context used to link to the CSP-managed information.

CryptoContext (input)
Pointer to CSSM_CONTEXT structure that describes the attributes of the cryptographic context.

KRSPOptions (input)
The key recovery service provider specific options. These options are uninterpreted by the SKMF, but passed on to the KRSP.

KRFlags (input)
Flag values for key recovery fields generation. Defined values are:

- KR_ENT—signifies that only the enterprise key recovery fields should be processed.
- KR_LE—signifies that only the law enforcement key recovery fields should be processed.
- KR_ALL—signifies that LE, and ENT key recovery fields should be processed.
- KR_OPTIMIZE—signifies that available optimization options are to be adopted.

KRFields (input)
The key recovery fields to be processed in the form of a data blob.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if the processing operation is successful and returns an error if an error has occurred.

ERRORS

CSSM_KR_INVALID_CC_HANDLE
Invalid crypto context handle.

CSSM_KR_INVALID_KRC_HANDLE
Invalid key recovery context handle.

CSSM_KR_INVALID_OPTIONS
Invalid recovery options.

CSSM_MEMORY_ERROR
Memory error.

52.5 Key Recovery Request Operations

The manpages for Key Recovery Request Operations follow on the next page.

NAME

KRSP_RecoveryRequest

SYNOPSIS

```
CSSM_RETURN CSSMKRSPI KRSP_RecoveryRequest
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_CC_HANDLE KRRequestContextHandle,
    const CSSM_CONTEXT_PTR KRRequestContext,
    CSSM_DATA_PTR KRInData,
    CSSM_CRYPTO_DATA_PTR UserCallback,
    uint32 *EstimatedTime,
    CSSM_HANDLE_PTR ReferenceHandle)
```

DESCRIPTION

This function performs a key recovery request operation. The KRInData contains known input parameters for the recovery request operation. A UserCallback function may be supplied to allow the recovery operation to interact with the user interface, if necessary. If the recovery request operation is successful, a ReferenceHandle and an EstimatedTime parameter is returned; the ReferenceHandle is to be used to invoke the KRSP_RecoveryRetrieve function, after the EstimatedTime in seconds.

PARAMETERS

KRSPHandle (input)
The handle that describes the add-in key recovery service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

KRRequestContextHandle (input)
The handle that describes the context of this key recovery operation used to link to the KRSP-managed information.

KRRequestContext (input)
Pointer to CSSM_CONTEXT structure that describes the attributes with this key recovery context.

KRInData (input)
Input data for key recovery requests. For encapsulation schemes, the key recovery fields are included in this parameter.

UserCallback (input)
A callback function that may be used to collect further information from the user interface.

EstimatedTime (output)
The estimated time after which the CSSM_KR_RecoveryRetrieve call should be invoked to obtain recovery results.

ReferenceHandle (output)
Handle returned when recovery request is successful. This handle may be used to invoke the CSSM_KR_RecoveryRetrieve function.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid handle.

CSSM_KR_INVALID_HANDLE
Invalid recovery context handle.

CSSM_KR_INVALID_RECOVERY_CONTEXT
Invalid context value.

CSSM_KR_INVALID_POINTER
Invalid pointer.

CSSM_MEMORY_ERROR
Memory error.

NAME

KRSP_RecoveryRetrieve

SYNOPSIS

```
CSSM_RETURN CSSMKRSPI KRSP_RecoveryRetrieve
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE ReferenceHandle,
    uint32 *EstimatedTime,
    CSSM_HANDLE_PTR CacheHandle,
    uint32 *NumberOfResults)
```

DESCRIPTION

This function completes a key recovery request operation. The ReferenceHandle parameter indicates which outstanding recovery request is to be completed. The results of a successful recovery operation are referenced by the CacheHandle parameter, which may be used with the KRSP_GetRecoveredObject function to retrieve the recovered keys.

If the results are not available at the time this function is invoked, the CacheHandle is NULL, and the EstimatedTime parameter indicates when this operation should be repeated with the same ReferenceHandle.

PARAMETERS

KRSPHandle (input)
The handle that describes the add-in key recovery service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

ReferenceHandle (input)
Handle that indicates which key recovery request operation is to be completed.

EstimatedTime (output)
The estimated time after which this call should be repeated to obtain recovery results. This is set to a non-zero value only when the ResultsHandle parameter is NULL.

CacheHandle (output)
Handle returned when recovery operation is successful. This handle may be used to get individual keys using the KRSP_GetRecoveredObject function. This handle is NULL, if EstimatedTime parameter is not zero.

NumberOfResults (output)
The number of recovered key objects that may be obtained using the ResultsHandle.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid KR handle.

CSSM_KR_INVALID_HANDLE
Invalid reference handle.

CSSM_MEMORY_ERROR
Memory error.

CSSM_KR_FAIL
Function failed.

NAME

KRSP_GetRecoveredObject

SYNOPSIS

```
CSSM_RETURN CSSMKRSPI KRSP_GetRecoveredObject
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE CacheHandle,
    uint32 IndexInResults,
    CSSM_CSP_HANDLE CSPHandle,
    CSSM_CRYPTO_DATA_PTR PassPhrase,
    CSSM_KEY_PTR RecoveredKey,
    uint32 Flags,
    CSSM_DATA_PTR OtherInfo)
```

DESCRIPTION

This function is used to step through the results of a recovery request operation in order to retrieve a single recovered key at a time along with its associated meta information. The cache handle returned from a successful KRSP_RecoveryRetrieve operation is used . When multiple keys are recovered by a single recovery request operation, the index parameter indicates which item to retrieve through this function.

The RecoveredKey parameter serves as an input template for the key to be returned. If a private key is to be returned by this operation, the PassPhrase parameter is used to inject the private key into the CSP indicated by the RecoveredKey template; the corresponding public key is returned in the RecoveredKey parameter. Subsequently, the PassPhrase and the public key may be used to reference the private key when operations using the private key are required. The OtherInfo parameter may be used to return other meta data associated with the recovered key.

PARAMETERS

KRSPHandle (input)
The handle that describes the add-in key recovery service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

CacheHandle (input)
The handle returned from a successful CSSM_KR_RecoveryRequest operation.

IndexInResults (input)
The index into the results that are referenced by the ResultsHandle parameter.

CSPHandle (input/optional)
This parameter identifies the CSP that the recovered key should be injected into. It may be set to NULL if the key is to be returned in raw form to the caller.

PassPhrase (input)
This parameter is only relevant if the recovered key is a private key. It is used to protect the private key when it is inserted into the CSP specified by the RecoveredKey template.

RecoveredKey (output)
This parameter returns the recovered key.

Flags (input)
Flag values relevant for recovery of a key. Possible values are: CERT_RETRIEVE—if the recovered key is a private key, return the corresponding public key certificate in the OtherInfo parameter.

OtherInfo (output)
This parameter is used if there are additional information associated with the recovered key

(such as the public key certificate when recovering a private key) that is to be returned.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid KR Handle.

CSSM_KR_INVALID_CSP_HANDLE
Invalid CSP Handle.

CSSM_KR_INVALID_HANDLE
Invalid cache handle.

CSSM_KR_INVALID_INDEX
Cache index value is out of range.

CSSM_KR_PRIVATE_KEY_STORE_FAIL
Unable to store private key in CSP.

CSSM_MEMORY_ERROR
Not enough memory.

NAME

KRSP_RecoveryRequestAbort

SYNOPSIS

```
CSSM_RETURN CSSMKRSPI KRSP_RecoveryRequestAbort
    (CSSM_KRSP_HANDLE KRSPHandle,
    CSSM_HANDLE ResultsHandle )
```

DESCRIPTION

This function terminates a recovery request operation and releases any state information related to the recovery request.

PARAMETERS

KRSPHandle (input)
The handle that describes the add-in key recovery service provider module used to perform up calls to CSSM for the memory functions managed by CSSM.

ResultsHandle (input)
The handle returned from a successful KRSP_RecoveryRequest operation.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_INVALID_KR_HANDLE
Invalid KR handle.

CSSM_KR_INVALID_HANDLE
Invalid cache handle.

52.6 Privileged Context Operations

The manpages for Privileged Context Operations follow on the next page.

NAME

KRSP_PassPrivFunc

SYNOPSIS

```
CSSM_RETURN CSSMKRSPI KRSP_PassPrivFunc
    (CSSM_PRIV_FUNC_PTR CSSM_SetContextPriv);
```

DESCRIPTION

This function is used to provide the KRSP with the CSSM_SetContextPriv callback function. This callback is implemented by the CSSM and allows the setting or dropping of the privilege state flag for a given cryptographic context. This is used by the KRSP to make a cryptographic context privileged with respect to key recovery policy enforcement decisions, so that the KRSP itself is allowed to bypass the key recovery policy controls. The KRSP is expected to reset the privilege state flag as soon as the need for the privilege is over.

PARAMETERS

CSSM_SetContextPriv (input)
The callback that is used by the KRSP to set or drop the privilege state flag for a given cryptographic context.

RETURN VALUE

A CSSM return value. This function returns CSSM_OK if successful and returns an error code if an error has occurred.

ERRORS

CSSM_MEMORY_ERROR
Not enough memory.

52.7 Extensibility Functions

The KRSP_PassThrough function is provided to allow KRSP developers to extend the key recovery functionality of the CSSM API. Because it is only exposed to CSSM as a function pointer, its name internal to the KRSP can be assigned at the discretion of the KRSP module developer. However, its parameter list and return value must match what is shown below. The error codes given in this section constitute the generic error codes which may be used by all KRSPs to describe common error conditions. KRSP developers may also define their own module-specific error codes, as described in the *CSSM Add-in Module Structure and Administration Specification*.

NAME

KRSP_PassThrough

SYNOPSIS

```
void* CSSMKRSPI KRSP_PassThrough
    (CSSM_KR_HANDLE KRHandle,
    uint32 PassThroughId,
    const void * InData);
```

DESCRIPTION

The KRSP_PassThrough function is provided to allow KRSP developers to extend the key recovery functionality of the CSSM API.

PARAMETERS

KRHandle (input)
The handle that describes the context of this key recovery operation.

PassThroughId (input)
An identifier specifying the custom function to be performed.

InData (input)
A pointer to a void structure containing the input data.

RETURN VALUE

A pointer to a void structure contains the output.

ERRORS

CSSM_KR_INVALID_HANDLE
Invalid handle.

CSSM_KR_INVALID_POINTER
Invalid pointer for input data.

CSSM_MEMORY_ERROR
Not enough memory to allocate.

CSSM_KR_UNSUPPORTED_OPERATION
Add-in does not support this function.

CSSM_KR_PASS_THROUGH_FAILED
Unable to perform custom function.

Glossary

Asymmetric algorithms
Cryptographic algorithms using one key to encrypt, and a second key to decrypt. They are often called public-key algorithms. One key is called the public key, and the other is called the private key or secret key. RSA (Rivest-Shamir-Adelman) is the most commonly used public-key algorithm. It can be used for encryption and for signing.

Cryptographic Service Providers (CSPs)
Modules that provide secure key storage and cryptographic functions. The modules may be software only or hardware with software drivers. The cryptographic functions provided may include:

- Bulk encryption and decryption
- Digital signing
- Cryptographic hash
- Random number generation
- Key exchange

Certification Authority (CA)
An entity that guarantees or sponsors a certificate. For example, a credit card company signs a cardholder's certificate to assure that the cardholder is who he or she claims to be. The credit card company is a certificate authority. Certificate authorities issue, verify, and revoke certificates.

Certificate
See Digital certificate.

Certificate chain
The hierarchical chain of all the other certificates used to sign the current certificate. This includes the Certificate Authority (CA) who signs the certificate, the CA who signed that CA's certificate, and so on. There is no limit to the depth of the certificate chain.

Certificate signing
The Certificate Authority (CA) can sign certificates it issues or cosign certificates issued by another CA. In a general signing model, an object signs an arbitrary set of one or more objects. Hence, any number of signers can attest to an arbitrary set of objects. The arbitrary objects could be, for example, pieces of a document for libraries of executable code.

Certificate validity date
A start date and a stop date for the validity of the certificate. If a certificate expires, the Certificate Authority (CA) may issue a new certificate.

Common Data Security Architecture (CDSA)
A set of layered security services that address communications and data security problems in the emerging Internet and Intranet application space. The CDSA consists of three basic layers:

- A set of system security services
- The Common Security Services Manager (CSSM)
- Add-in Security Modules (CSPs, TPs, CLs, DLs)

Common Security Services Manager (CSSM)
The central layer of the Common Data Security Architecture (CDSA) that defines six key service components:

- Cryptographic Services Manager
- Trust Policy Services Manager
- Certificate Library Services Manager
- Data Storage Library Services Manager
- Integrity Services Manager
- Security Context Manager

The CSSM binds together all the security services required by PC applications. In particular, it facilitates linking digital certificates to cryptographic actions and trust protocols.

Cryptographic algorithm
A method or defined mathematical process for implementing a cryptography operation. A cryptographic algorithm may specify the procedure for encrypting and decrypting a byte stream, digitally signing an object, computing the hash of an object, generating a random number.

Cryptoki
The name of the PKCS#11 version 1.0 standard published by RSA Laboratories. The standard specifies the interface for accessing cryptographic services performed by a removable device. For additional information see *http://www.rsa.com*.

Digital certificate
The binding of some identification to a public key in a particular domain, as attested to directly or indirectly by the digital signature of the owner of that domain. A digital certificate is an unforgeable credential in cyberspace. The certificate is issued by a trusted authority, covered by that party's digital signature. The certificate may attest to the certificate holder's identity, or may authorize certain actions by the certificate holder. A certificate may include multiple signatures and may attest to multiple objects or multiple actions.

Digital signature
A data block that was created by applying a cryptographic signing algorithm to some other data using a secret key. Digital signatures may be used to:

- Authenticate the source of a message, data, or document
- Verify that the contents of a message hasn't been modified since it was signed by the sender
- Verify that a public key belongs to a particular person

Typical digital signing algorithms include MD5 with RSA encryption, and DSS, the Digital Signature Standard defined by NIST FIPS Pub 186.

Hash algorithm
A cryptographic algorithm used to compress a variable-size input stream into a unique, fixed-size output value. The function is one-way, meaning the input value cannot be derived from the output value. A cryptographically strong hash algorithm is collision-free, meaning unique input values produce unique output values. Hashing is typically used in digital signing algorithms. Example hash algorithms include MD and MD2 from RSA Data Security. MD5, also from RSA Data Security, hashes a variable-size input stream into a 128-bit output value. SHA, a Secure Hash Algorithm published by the U.S. Government, produces a 160-bit hash value from a variable-size input stream.

Hypertext Transfer Protocol (HTTP)
The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol which is widely used for data transfer over the Internet. More information about HTTP is available at *http://www.w3.org/Protocols/* and at *http://www.ics.uci.edu/pub/ietf/http/*.

JAVA
JAVA is an object-oriented language for development of platform-independent applications. JAVA runtime defines a sandbox paradigm to provide a secure JAVA execution environment. Additional information can be found at *http://www.javasoft.com*.

Leaf Certificate
The certificate in a certificate chain that has not been used to sign another certificate in that chain. The leaf certificate is signed directly or transitively by all other certificates in the chain.

Meta-information
Descriptive information specified by an add-in service module and stored in the CSSM registry. This information advertises the add-in modules services. CSSM supports application queries for this information. The information my change at runtime.

Message digest
The digital fingerprint of an input stream. A cryptographic hash function is applied to an input message arbitrary length and returns a fixed-size output, which is called the digest value.

Nonce
A non-repeating value, usually but not necessarily random.

Owned certificate
A certificate whose associated private key resides in a local CSP. Digital signature algorithms require the private key when signing data. A system may supply certificates it owns along with signed data to enable other to verify the signature. A system uses certificates that it does not own to verify signatures created by others.

PolicyMaker
PolicyMaker is a language for evaluating trust policy expressions. Additional information can be found at:

- *ftp://ftp.research.att.com/dist/mab/policymaker.ps*
- Matt Blaze, Joan Feigenbaum, Jack Lacy, "Decentralized Trust Management" Proceedings of the Symposium on Security and Privacy, IEEE Computer Society and Press, Los Alamitos, 1996, pp. 164-173

Pretty Good Privacy (PGP)
PGP is a widely available software package providing data encryption and decryption using the IDEA cryptographic algorithms. To date,PGP facilities have been applied to securing data files and electronic mail communications. Additional information can be found at http://www.pgp.com

Private key
The cryptographic key used to decipher or sign messages in public-key cryptography. This key is kept secret by its owner.

Public key
The cryptographic key used to encrypt messages in public-key cryptography. The public key is available to multiple users (for example, the public).

Random number generators
A function that generates cryptographically strong random numbers that cannot be easily guessed by an attacker. Random numbers are often used to generate session keys.

Root certificate
The prime certificate, such as the official certificate of a corporation or government entity. The root certificate is positioned at the top of the certificate hierarchy in its domain, and it guarantees the other certificates in its certificate chain. The root certificate's public key is the foundation of signature verification in its domain.

Secret key
A cryptographic key used with symmetric algorithms, usually to provide confidentiality.

Secure Electronic Transaction (SET)
A specification designed to utilize technology for authenticating the parties involved in payment card purchases on any type of online network, including the Internet. SET focuses on maintaining confidentiality of information, ensuring message integrity, and authenticating the parties involved in a transaction. More information about SET is available at:

- *http://www.visa.com/cgi-bin/vee/nt/ecomm/main.html?2+0*
- *http://www.visa.com/nt/ecomm/set/set_bk1.zip*

Secure MIME (S/MIME)
MIME is a mechanism for specifying and describing the format of Internet message bodies also known as attachments to electronic mail. S / MIME provides a method to send and receive secure MIME messages. In order to validate the keys of a message sent to it, an S / MIME agent needs to certify that the encryption key is valid. Additional information can be found at:

- *http://ds.internic.net/rfc/rfc1521.txt*
- *http://ds.internic.net/internet-drafts/draft-dusse-smime-msg-04.txt*
- *http://ds.internic.net/internet-drafts/draft-dusse-smime-cert-03.txt*
- *http://www.imc.org/draft-dusse-smime-msg*
- *http://www.rsa.com/smime*

Secure Sockets Layer (SSL)
SSL (also known as Above Transport Layer Security (TLS)) is a security protocol that prevents eavesdropping, tampering, or message forgery over the Internet. An SSL service negotiates a secure session between two communicating endpoints. Basic facilities include certificate-based authentication, end-to-end data integrity and optional data privacy. Additional information can be found at *http://search.netscape.com/newsref/std/SSL.html* and *http://search.netscape.com/newsref/ssl/3-SPEC.html*. SSL has been submitted to the IETF as an Internet Draft for Transport Layer Security (TLS). More information about TLS can be found at *ftp://ftp.ietf.org/internet-drafts/draft-ietf-tls-protocol-03.txt*.

Security Context
A control structure that retains state information shared between a cryptographic service provider and the application agent requesting service from the CSP. Only one context can be active for an application at any given time, but the application is free to switch among contexts at will, or as required. A security context specifies CSP and application-specific values, such as required key length and desired hash functions.

Security-relevant event
An event where a CSP-provided function is performed, an add-in security module is loaded, or a breach of system security is detected.

Session key
A cryptographic key used to encrypt and decrypt data. The key is shared by two or more communicating parties, who use the key to ensure privacy of the exchanged data.

Signature
See Digital signature.

Signature chain
The hierarchical chain of signers, from the root certificate to the leaf certificate, in a certificate chain.

Symmetric algorithms
Cryptographic algorithms that use a single secret key for encryption and decryption. Both the sender and receiver must know the secret key. Well-known symmetric functions include DES (Data Encryption Standard) and IDEA. DES was endorsed by the U.S. Government as a standard in 1977. It's an encryption block cipher that operates on 64-bit blocks with a 56-bit key. It is designed to be implemented in hardware, and works well for bulk encryption. IDEA (International Data Encryption Algorithm) uses a 128-bit key.

Token
The logical view of a cryptographic device, as defined by a CSP's interface. A token can be hardware, a physical object, or software. A token contains information about its owner in digital form, and about the services it provides for electronic-commerce and other communication applications. A token is a secure device. It may provide a limited or a broad range of cryptographic functions.

Examples of hardware tokens are SmartCards and PMCIA cards.

Verification
The process of comparing two message digests. One message digest is generated by the message sender and included in the message. The message recipient computes the digest again. If the message digests are exactly the same, it shows or proves there was no tampering of the message contents by a third party (between the sender and the receiver). A process performed to check the integrity of a message, to determine the sender of a message, or both. Different algorithms are used to support different modes of verification. A typical procedure supporting integrity verification is the combination of a one-way hash function and a reversible digital signaturing algorithm. A one-way hash of the message is computed. The hash value is signed by encrypting it with a private key. The message and the encrypted hash value are sent to a receiver. The recipient recomputes the one-way hash, decrypts the signed hash value, and compares it with the computed hash. If the values match then the message has not been message has not been tampered since it was signed. The identity of a sender can be verified by a challenge-response protocol. The recipient sends the message sender a random challenge value. The original sender uses its private key to sign the challenge value and returns the result to the receiver. The receiver uses the corresponding public key to verify the signature over the challenge value. If the signature verifies the sender is the holder of the private key. If the receiver can reliably associate the corresponding public key with the named/known entity, then the identity of the sender is said to have been verified.

Web of trust
A trust network among people who know and communicate with each other. Digital certificates are used to represent entities in the web of trust. Any pair of entities can determine the extent of trust between the two, based on their relationship in the web.

Index

AddInAuthenticate **636**
Asymmetric algorithms 1045
Certificate 1045
Certificate chain 1045
Certificate signing 1045
Certificate validity date 1045
Certification Authority (CA) 1045
CL_CertAbortQuery **912**
CL_CertAbortRecovery **906**
CL_CertDescribeFormat **921**
CL_CertExport **920**
CL_CertGetAllFields **914**
CL_CertGetFirstFieldValue **909**
CL_CertGetKeyInfo **913**
CL_CertGetNextFieldValue **911**
CL_CertGroupFromVerifiedBundle **917**
CL_CertGroupToSignedBundle **915**
CL_CertImport **919**
CL_CertKeyRecover **904**
CL_CertMultiSignRequest **893**
CL_CertMultiSignRetrieve **896**
CL_CertRecover **903**
CL_CertRecoveryRequest **898**
CL_CertRecoveryRetrieve **901**
CL_CertRequest **885**
CL_CertRetrieve **888**
CL_CertVerify **907**
CL_CrlAbortQuery **942**
CL_CrlAddCert **931**
CL_CrlCreateTemplate **923**
CL_CrlDescribeFormat **943**
CL_CrlGetFirstFieldValue **939**
CL_CrlGetNextFieldValue **941**
CL_CrlRemoveCert **933**
CL_CrlRequest **926**
CL_CrlRetrieve **929**
CL_CrlSetFields **924**
CL_CrlSign **934**
CL_CrlVerify **936**
CL_IsCertInCrl **938**
CL_PassThrough **945**
CL_RegistrationFormRequest **890**
CL_RegistrationFormSubmit **891**
Common Data Security Architecture (CDSA) 1045
Common Security Services Manager (CSSM) 1046
Cryptographic algorithm 1046
Cryptographic Service Providers (CSPs) 1045
Cryptoki 1046
CSP_ChangeLoginPassword **811**
CSP_DecryptData **776**
CSP_DecryptDataFinal **783**
CSP_DecryptDataInit **779**
CSP_DecryptDataUpdate **781**
CSP_DeriveKey **801**
CSP_DigestData **741**
CSP_DigestDataClone **746**
CSP_DigestDataFinal **748**
CSP_DigestDataInit **743**
CSP_DigestDataUpdate **744**
CSP_EncryptData **767**
CSP_EncryptDataFinal **774**
CSP_EncryptDataInit **770**
CSP_EncryptDataUpdate **772**
CSP_EventNotify **816**
CSP_FreeKey **793**
CSP_GenerateAlgorithmParams **804**
CSP_GenerateKey **785**
CSP_GenerateKeyPair **788**
CSP_GenerateMac **750**
CSP_GenerateMacFinal **756**
CSP_GenerateMacInit **752**
CSP_GenerateMacUpdate **754**
CSP_GenerateRandom **791**
CSP_GetCapabilities **815**
CSP_Login **809**
CSP_Logout **810**
CSP_ObtainPrivateKeyFromPublicKey **794**
CSP_PassThrough **813**
CSP_QueryKeySizeInBits **806**
CSP_QuerySize **765**
CSP_SignData **726**
CSP_SignDataFinal **732**
CSP_SignDataInit **728**
CSP_SignDataUpdate **730**
CSP_UnwrapKey **798**
CSP_VerifyData **734**
CSP_VerifyDataFinal **740**
CSP_VerifyDataInit **736**
CSP_VerifyDataUpdate **738**
CSP_VerifyMac **758**
CSP_VerifyMacFinal **764**
CSP_VerifyMacInit **760**

CSP_VerifyMacUpdate..**762**
CSP_WrapKey..**795**
CSSM_ClearError..**406**, 669
CSSM_CL_CertAbortQuery...............................**315**
CSSM_CL_CertAbortRecovery **309**
CSSM_CL_CertDescribeFormat..........................**325**
CSSM_CL_CertExport..**324**
CSSM_CL_CertGetAllFields...............................**317**
CSSM_CL_CertGetFirstFieldValue....................**312**
CSSM_CL_CertGetKeyInfo.................................**316**
CSSM_CL_CertGetNextFieldValue...................**314**
CSSM_CL_CertGroupFromVerifiedBundle......**320**
CSSM_CL_CertGroupToSignedBundle............**318**
CSSM_CL_CertImport **322**
CSSM_CL_CertKeyRecover...............................**307**
CSSM_CL_CertMultiSignRequest.....................**296**
CSSM_CL_CertMultiSignRetrieve **299**
CSSM_CL_CertRecover **306**
CSSM_CL_CertRecoveryRequest **301**
CSSM_CL_CertRecoveryRetrieve **304**
CSSM_CL_CertRequest **288**
CSSM_CL_CertRetrieve.......................................**291**
CSSM_CL_CertVerify...**310**
CSSM_CL_CrlAbortQuery..................................**345**
CSSM_CL_CrlAddCert..**334**
CSSM_CL_CrlCreateTemplate **327**
CSSM_CL_CrlDescribeFormat...........................**346**
CSSM_CL_CrlGetFirstFieldValue......................**342**
CSSM_CL_CrlGetNextFieldValue.....................**344**
CSSM_CL_CrlRemoveCert **336**
CSSM_CL_CrlRequest..**329**
CSSM_CL_CrlRetrieve **332**
CSSM_CL_CrlSetFields..**328**
CSSM_CL_CrlSign **337**
CSSM_CL_CrlVerify **339**
CSSM_CL_IsCertInCrl **341**
CSSM_CL_PassThrough **348**
CSSM_CL_RegistrationFormRequest................**293**
CSSM_CL_RegistrationFormSubmit.................**294**
CSSM_CompareGuids **414**
CSSM_CSP_ChangeLoginPassword **152**
CSSM_CSP_CreateAsymmetricContext **135**
CSSM_CSP_CreateDeriveKeyContext **137**
CSSM_CSP_CreateDigestContext **132**
CSSM_CSP_CreateKeyGenContext **139**
CSSM_CSP_CreateMacContext **133**
CSSM_CSP_CreatePassThroughContext **141**
CSSM_CSP_CreateRandomGenContext **134**
CSSM_CSP_CreateSignatureContext **129**
CSSM_CSP_CreateSymmetricContext **130**
CSSM_CSP_Login **150**
CSSM_CSP_Logout **151**
CSSM_CSP_PassThrough **238**
CSSM_DecryptData **201**
CSSM_DecryptDataFinal....................................**208**
CSSM_DecryptDataInit.......................................**204**
CSSM_DecryptDataUpdate **206**
CSSM_DeleteContext **145**
CSSM_DeleteContextAttributes **148**
CSSM_DeliverModuleManagerEvent...............**602**
CSSM_DeregisterManagerServices **600**
CSSM_DeregisterServices..................................**665**
CSSM_DeriveKey **227**
CSSM_DestroyError...**408**
CSSM_DigestData **169**
CSSM_DigestDataClone **174**
CSSM_DigestDataFinal.......................................**175**
CSSM_DigestDataInit..**171**
CSSM_DigestDataUpdate **172**
CSSM_DL_Authenticate **380**
CSSM_DL_DataAbortQuery..............................**396**
CSSM_DL_DataDelete **389**
CSSM_DL_DataGetFirst **392**
CSSM_DL_DataGetFromUniqueRecordId **397**
CSSM_DL_DataGetNext **394**
CSSM_DL_DataInsert..**387**
CSSM_DL_DataModify.......................................**390**
CSSM_DL_DbClose **370**
CSSM_DL_DbCreate **371**
CSSM_DL_DbDelete..**373**
CSSM_DL_DbExport...**378**
CSSM_DL_DbGetRecordParsingFunctions......**382**
CSSM_DL_DbImport...**375**
CSSM_DL_DbOpen **368**
CSSM_DL_DbSetRecordParsingFunctions.......**381**
CSSM_DL_FreeNameList **385**
CSSM_DL_FreeUniqueRecord **398**
CSSM_DL_GetDbNameFromHandle **384**
CSSM_DL_GetDbNames **383**
CSSM_DL_PassThrough **400**
CSSM_EncryptData **192**
CSSM_EncryptDataFinal **199**
CSSM_EncryptDataInit.......................................**195**
CSSM_EncryptDataUpdate................................**197**
CSSM_Free **103**
CSSM_FreeContext **143**
CSSM_FreeInfo...**80**
CSSM_FreeKey...**230**
CSSM_FreeList **102**
CSSM_FreeModuleInfo................................**96**, 663
CSSM_GenerateAlgorithmParams **231**
CSSM_GenerateKey...**212**

CSSM_GenerateKeyPair **215**
CSSM_GenerateMac **177**
CSSM_GenerateMacFinal **182**
CSSM_GenerateMacInit **179**
CSSM_GenerateMacUpdate **181**
CSSM_GenerateRandom **218**
CSSM_GetAPIMemoryFunctions **104**
CSSM_GetContext **142**
CSSM_GetContextAttribute **146**
CSSM_GetError **404**, 667
CSSM_GetGUIDUsage **97**
CSSM_GetHandleInfo **666**
CSSM_GetHandleUsage **98**
CSSM_GetInfo **79**
CSSM_GetModuleGUIDFromHandle **99**
CSSM_GetModuleInfo **93**, 660
CSSM_GetModuleManagerInfo **596**
CSSM_GetSubserviceUIDFromHandle **100**
CSSM_Init **78**
CSSM_InitError **407**
CSSM_IsCLError **410**
CSSM_IsCSPError **413**
CSSM_IsCSSMError **409**
CSSM_IsDLError **411**
CSSM_IsTPError **412**
CSSM_KRPolicyInfo **446**
CSSM_KR_CreateRecoveryEnablement
  Context **444**
CSSM_KR_CreateRecoveryRegistration
  Context **443**
CSSM_KR_CreateRecoveryRequestContext **445**
CSSM_KR_GenerateRecoveryFields **452**
CSSM_KR_GetRecoveredObject **460**
CSSM_KR_PassThrough **464**
CSSM_KR_ProcessRecoveryFields **454**
CSSM_KR_RecoveryRequest **456**
CSSM_KR_RecoveryRequestAbort **462**
CSSM_KR_RecoveryRetrieve **458**
CSSM_KR_RegistrationRequest..........................**448**
CSSM_KR_RegistrationRetrieve **450**
CSSM_KR_SetEnterpriseRecoveryPolicy..........**441**
CSSM_ListAttachedModuleManagers **597**
CSSM_ListModules **92**
CSSM_Load **81**
CSSM_ModuleAttach **89**, 657
CSSM_ModuleDetach **91**, 659
CSSM_ModuleInstall.......................................**86**, 654
CSSM_ModuleManagerInstall **592**
CSSM_ModuleManagerUninstall **594**
CSSM_ModuleUninstall **88**, 656
CSSM_ObtainPrivateKeyFromPublicKey **220**
CSSM_QueryKeySizeInBits.................................**210**
CSSM_QuerySize..**190**
CSSM_RegisterManagerServices **599**
CSSM_RegisterServices..**664**
CSSM_RequestCssmExemption. **82**
CSSM_RetrieveCounter **235**
CSSM_RetrieveUniqueId **234**
CSSM_SetContext **144**
CSSM_SetError **405**, 668
CSSM_SetModuleInfo **95**, 662
CSSM_SignData **154**
CSSM_SignDataFinal..**161**
CSSM_SignDataInit **157**
CSSM_SignDataUpdate **159**
CSSM_TP_ApplyCrlToDb **264**
CSSM_TP_CertGroupConstruct **267**
CSSM_TP_CertGroupPrune................................**269**
CSSM_TP_CertGroupVerify................................**253**
CSSM_TP_CertRequest **248**
CSSM_TP_CertRetrieve **251**
CSSM_TP_CertRevoke **258**
CSSM_TP_CertSign **256**
CSSM_TP_CrlSign **262**
CSSM_TP_CrlVerify..**260**
CSSM_TP_PassThrough..**272**
CSSM_UnwrapKey **224**
CSSM_UpdateContextAttributes. **147**
CSSM_VerifyComponents **84**
CSSM_VerifyData..**163**
CSSM_VerifyDataFinal...**168**
CSSM_VerifyDataInit **165**
CSSM_VerifyDataUpdate **167**
CSSM_VerifyDevice **236**
CSSM_VerifyMac...**184**
CSSM_VerifyMacFinal **189**
CSSM_VerifyMacInit **186**
CSSM_VerifyMacUpdate.....................................**188**
CSSM_WrapKey **221**
DeregisterDispatchTable **589**
Digital certificate 1046
Digital signature 1046
DL_Authenticate **974**
DL_DataAbortQuery **1003**
DL_DataDelete **996**
DL_DataGetFirst **999**
DL_DataGetFromUniqueRecordId **1004**
DL_DataGetNext **1001**
DL_DataInsert **994**
DL_DataModify **997**
DL_DbClose..**978**
DL_DbCreate..**979**

DL_DbDelete **981**
DL_DbExport **986**
DL_DbGetRecordParsingFunctions **989**
DL_DbImport **983**
DL_DbOpen **976**
DL_DbSetRecordParsingFunctions **988**
DL_FreeNameList **992**
DL_FreeUniqueRecord **1005**
DL_GetDbNameFromHandle **991**
DL_GetDbNames **990**
DL_PassThrough **1007**
EventNotify **634**
EventNotifyManager **590**
Hash algorithm 1046
Hypertext Transfer Protocol (HTTP) 1047
Initialize **585**, 632
ISL_CheckAddressWithinModule **523**
ISL_ContinueVerification **487**
ISL_CopyCertificateChain **503**
ISL_CreateCertificateAttributeEnumerator **507**
ISL_CreateCertificateChain **502**
ISL_CreateManifestSectionAttribute
  Enumerator **516**
ISL_CreateManifestSectionEnumerator **493**
ISL_CreateSignatureAttributeEnumerator **497**
ISL_CreateVerifiedSignatureRoot **490**
ISL_CreateVerifiedSignatureRootWith
  Certificate **491**
ISL_FindCertificateAttribute **506**
ISL_FindManifestSection **492**
ISL_FindManifestSectionAttribute **515**
ISL_FindRegistryAttribute **480**
ISL_FindSignatureAttribute **496**
ISL_GetCertficateChain **486**
ISL_GetLibHandle **524**
ISL_GetManifestSignatureRoot **511**
ISL_GetModuleManifestSection **519**
ISL_GetNextCertificateAttribute **508**
ISL_GetNextManifestSection **494**
ISL_GetNextManifestSectionAttribute **517**
ISL_GetNextSignatureAttribute **498**
ISL_GetReturnAddress **522**
ISL_LocateProcedureAddress **521**
ISL_RecycleCertificateAttributeEnumerator **509**
ISL_RecycleCertificateChain **504**
ISL_RecycleManifestSectionAttribute
  Enumerator **518**
ISL_RecycleManifestSectionEnumerator **495**
ISL_RecycleSignatureAttributeEnumerator **499**
ISL_RecycleVerifiedModuleCredentials **488**
ISL_RecycleVerifiedSignatureRoot **500**
ISL_SelfCheck **482**
ISL_VerifyAndLoadModule **512**
ISL_VerifyAndLoadModuleAndCredentials **483**
ISL_VerifyData **514**
ISL_VerifyLoadedModule **513**
ISL_VerifyLoadedModuleAndCredentials **485**
JAVA 1047
KRSP_GenerateRecoveryFields **1030**
KRSP_GetRecoveredObject **1038**
KRSP_PassPrivFunc **1042**
KRSP_PassThrough **1044**
KRSP_ProcessRecoveryFields **1032**
KRSP_RecoveryRequest **1035**
KRSP_RecoveryRequestAbort **1040**
KRSP_RecoveryRetrieve **1037**
KRSP_RegistrationRequest **1026**
KRSP_RegistrationRetrieve **1028**
Leaf Certificate 1047
Message digest 1047
Meta-information 1047
ModuleManagerAuthenticate **587**
Nonce 1047
Owned certificate 1047
PolicyMaker 1047
Pretty Good Privacy (PGP) 1047
Private key 1047
Public key 1047
Random number generators 1048
RegisterDispatchTable **588**
Root certificate 1048
Secret key 1048
Secure Electronic Transaction (SET) 1048
Secure MIME (S/MIME) 1048
Secure Sockets Layer (SSL) 1048
Security Context 1048
Security-relevant event 1048
Session key 1049
Signature 1049
Signature chain 1049
Symmetric algorithms 1049
Terminate **586**, 633
Token 1049
TP_ApplyCrlToDb **854**
TP_CertGroupConstruct **856**
TP_CertGroupPrune **858**
TP_CertGroupVerify **843**
TP_CertRequest **838**
TP_CertRetrieve **841**
TP_CertRevoke **848**
TP_CertSign **846**
TP_CrlSign **852**

TP_CrlVerify **850**
TP_PassThrough **861**
Verification 1049
Web of trust 1049